Übungsskript
Einführung in die Informatik 1

Stefan Berktold
www.stecrz.de

2019 / 2020

PDF-Lösung

EIDI-Übungsskript
Stand: 14.11.2020*

*Version vom **10.06.2020**

VORWORT & HINWEISE

Allgemeine Information

Dieses Skript wurde über einen Zeitraum von fünf Jahren speziell zur Vorbereitung auf die Klausur im Modul „*Einführung in die Informatik 1*“ (IN0001 EIDI) an der TU München erstellt, kann aber zum Teil auch für das begleitende „*Praktikum: Grundlagen der Programmierung*“ (IN0002 PGdP) genutzt werden. Für andere Module (bspw. EIDI für andere Fachrichtungen) ist dieses Skript **nicht geeignet**, denn obwohl im Block I alle Java-Grundlagen wiederholt werden, liegt der Fokus klar auf EIDI-spezifischen Themen. Enthalten sind vorwiegend Übungsaufgaben, jedoch auch Kurzzusammenfassungen einiger wesentlicher Konzepte.

Java-Grundkenntnisse

Dieses Skript ist kein Java-Tutorial. Wenn du Programmierung im Allgemeinen lernen willst, dann würde ich dir stattdessen zu Büchern oder Online-Kursen raten, je nachdem womit du am besten lernst. Auch im PGdP lernst du Java, wobei das Tempo in den vergangenen Jahren stark gesteigert wurde, d. h. einem Anfänger würde ich ehrlicherweise zu anderen (zusätzlichen) Quellen raten. Ich persönlich habe Java mit dem Buch „*Java lernen BlueJ (4. Auflage)*“ (Kap. 1 – 7), dann in der Schule, dann über die EIDI-Vorlesung gelernt. Heute würde ich eine der zahlreichen YouTube-Videoserien vorziehen.

Das Skript

Die Klausur besteht zum Großteil *nicht* aus Schema-F-Aufgaben, d. h. es ist *nicht* sinnvoll, Lösungsvorschläge abzuschreiben oder auswendig zu lernen. Die beste Vorbereitung ist die Bearbeitung von Übungsaufgaben, also Programmieren. Ich habe versucht dieses Skript so zu gestalten, dass du im Hinblick auf die Klausur möglichst viel lernst. Das Skript enthält daher neben Aufgaben, die in ähnlicher Form in der Klausur vorkommen könnten, auch eine Vielzahl an Übungen, die absichtlich nicht klausurähnlich verfasst sind. An vielen Stellen gibt es so viele Teilaufgaben, dass du getrost mit einer anderen Aufgabe weitermachen kannst, wenn du dich sicher genug fühlst.

Ich habe außerdem ein **Sternchensystem** eingeführt, das jeder Übung eine Relevanz von 0 (☆☆☆) bis 3 (★★★) zuordnet, sodass du einen Anhaltspunkt hast, in welche Themen du mehr Zeit investieren solltest, und welche Aufgaben du eher überspringen kannst. Eine Aufgabe hat umso mehr Sterne, je *wichtiger* ich persönlich es finde, diese Aufgabe zu bearbeiten. Mehr Sternchen bedeuten *nicht* zwangsläufig, dass die Aufgabe eher in der Klausur drankommt und spiegeln außerdem *nicht* unbedingt die Schwierigkeit einer Aufgabe wider.

Die Klausur

Du solltest im Hinterkopf behalten, dass die EIDI-Klausur in erster Linie von der PGdP-Übungsleitung erstellt wird, wobei der Dozent i. d. R. (meines Wissens) eher wenig an der Erstellung der Aufgaben beteiligt ist und die Korrektur vorwiegend von Tutoren (also anderen Studenten wie mir) durchgeführt wird. Neben der Bearbeitung des Skripts empfehle ich dir unbedingt die Bearbeitung von Altklausuren! Da ich diese nicht weitergeben darf, sind sie nicht im Skript enthalten, finden sich aber teilweise auf Portalen wie Studydrive oder Unistuff. PGdP-Aufgaben sind ebenfalls sinnvoll, kosten aber viel Zeit, da deren Umfang üblicherweise wesentlich größer ist als der von Klausuraufgaben.

Der Verfasser

Kann man mir überhaupt glauben?

Skepsis ist gut, also: Ich habe selbst Informatik an der TUM studiert und EIDI bzw. PGdP im ersten Semester (2015/16) mit 1,0 bestanden, wobei ich schon während des Semesters viel Zeit in die Beantwortung von Fragen anderer gesteckt habe (btw: gute Klausurvorbereitung). Anschließend habe ich mich weiter auf diese Veranstaltungen konzentriert, d. h. ich war als Tutor im PGdP tätig (2016/17 + 2017/18), habe mehrfach das EIDI-Repetitorium an der TUM gehalten (2017 + 2018 + 2019), Crashkurse gegeben (2018 + 2019 + 2020) und so manch anderes. Insgesamt habe ich das PGdP bislang fünfmal (WS 2015 - 2019) in vollem Umfang bearbeitet und daher ein – wie ich denke – relativ gutes Bild über die wichtigsten Themen (siehe nächste Seite).

Woher stammen die Skript-Aufgaben?

Mit der Erstellung dieses Skripts habe ich Ende 2016 begonnen, woraufhin dieses stetig erweitert und an die jährlich wechselnde Übungsleitung und Dozenten angepasst wurde. Einige (entsprechend gekennzeichnete) Kapitel sind deshalb nicht mehr klausurrelevant, aber für kommende Semester trotzdem noch enthalten (bspw. Zahlenbasen, werden auch in anderen LVs behandelt). Ich habe dieses Skript in vollständiger Eigenarbeit erstellt, d. h. alle Aufgaben und Grafiken wurden von mir selbst gestaltet. Das Skript steht nicht in Zusammenarbeit mit der TU München und erhebt deshalb auch keinen Anspruch auf Vollständigkeit. Fehler vorbehalten.

Verwendung zu nicht-privaten Zwecken

Du kannst dieses Skript oder einzelne Auszüge gerne auch zu Lehrzwecken verwenden (bspw. als Tutor im EIDI-Repetitorium der TUM). Ich würde dich jedoch darum bitten, die Verwendung immer entsprechend zu kennzeichnen (namentliche Erwähnung bzw. Website-Link). Danke!

SUPPORT 😇

Wie du dir vielleicht vorstellen kannst, ist der Zeitaufwand für die Konzeptionierung sinnvoller Übungsaufgaben, bei denen insbesondere auch diverse Sonderfälle aufgezeigt werden (welche in der Klausur leider genauso abgeprüft werden wie die Normalfälle), immens – ganz zu schweigen von der Erstellung von Grafiken und Lösungsvorschlägen/-erklärung (welche ich auch zu einigen Klausuren erstellt habe)... Aus diesem Grund war das Skript bislang nur käuflich zu erwerben. Leider wird meine Zeit nicht ausreichen, um das Skript optimal für kommendes Semester (2020/21) anzupassen, auch wenn die meisten Inhalte identisch bleiben werden. Da der wesentliche Sinn des Skripts aber immer war, ein möglichst gutes Hilfsmittel zur Klausurvorbereitung zu sein, habe ich mich dazu entschieden, es dir kostenlos bereitzustellen. Für die Beantwortung kleinerer Fragen zum Skript bin ich weiterhin unter *stefan@stecrz.de* erreichbar.

Wenn ich dir mit diesem Skript bei deiner Vorbereitung helfen konnte, würde ich mich sehr freuen, wenn du mich das im Gegenzug wissen lässt. Entscheide z. B. mit einer kleinen Spende über *paypal.me/stecrz* (oder obige E-Mail) selbst, wie viel dir dieses Skript wert ist. Auch ich bin Student, deshalb verstehe ich natürlich auch, wenn dein Konto nur 1-2 Euro übrig hat. Und falls nicht: Ich freue mich genauso über ein einfaches *Danke*! 😇

LÖSUNGSVORSCHLÄGE

https://shop.stecrz.de/download/5641

KLAUSURAUFBAU

Die Klausur enthält typischerweise 7 oder 8 Aufgaben aus den folgenden Themengebieten:

1. **Gemischte Fragen** zu Java
2. Programmieraufgabe zu **Arrays / Strings** / Zahlen (ggf. als Lückentext) (Kapitel 7 und 18)
3. **Kontrollflussdiagramm oder Syntaxbaum** zeichnen (Block II)
4. **Polymorphie** (Block V)
5. Programmieraufgabe zu **Objektorientierung**, bspw. Iteratoren (Block IV)
6. Programmieraufgabe zu **Rekursion** (Block III)
7. Programmieraufgabe zu **Algorithmen**, bspw. Binärbäume (Block VI)
8. Programmieraufgabe zu **Streams** (Kapitel 19)
9. Programmieraufgabe zu **Threads** (Kapitel 21)

Die Reihenfolge kann variieren, wobei die Aufgaben i. d. R. nach Schwierigkeit sortiert sind, weshalb ich empfehle, der Reihe nach vorzugehen, allerdings nicht zu viel Zeit bei einer Aufgabe zu verlieren. Das bezieht sich vor allem auf die „Gemischte Fragen"-Aufgabe, welche i. d. R. aus mehreren kleinen Verständnisfragen im Multiple-Choice-Format besteht. In diesen Aufgabenblock fallen ggf. auch Auswertungen (→ Kap. 4) und reguläre Ausdrücke (→ Kap. 8). Man kann diese Fragen entweder relativ schnell oder gar nicht lösen, da hier zumeist bestimmte Sonderfälle abgeprüft werden. Fragen in diesem Stil sind über das gesamte Skript in Wahr/Falsch-Aufgaben verteilt, da sie keinem bestimmten Thema zugeordnet sind, z. B. gültige Variablennamen (→ Kap. 2). Zum ersten Block im Skript (Kapitel 1 bis 6) gibt es keine eigene Klausuraufgabe (abgesehen von der MC-Aufgabe) – hier werden lediglich die Grundkonzepte von Java behandelt.

Eine Klausuraufgabe deckt oft mehrere Themen ab! So sind bspw. „Rekursion" und „Algorithmen" Themen, die bisher häufig durch eine prozedurale (d. h. nicht-objektorientierte) Programmieraufgabe abgeprüft wurden (bspw. binäre Suche, Mergesort, ...). Rekursion findet sich aber auch in vielen Datenstrukturen wieder (bspw. können Listen rekursiv definierte Klassen sein), d. h. es ist gut möglich, dass die Objektorientierungsaufgabe gleichzeitig das Thema Rekursion abdeckt. Eine klare Thementrennung ist deshalb nicht möglich (abgesehen von Block II).

Eine der Programmieraufgaben ist oft als Lückentext gestellt. Begriffsdefinition werden in der Klausur so gut wie nie abgefragt. Der Theorieanteil ist extrem gering bis nicht vorhanden. Dennoch findest du im Skript an einigen Stellen Unterteilungen und Erklärungen von Begriffen, welche entweder notwendig sind (z. B. „Was heißt *rekursiv*?") oder dein Verständnis stärken sollen (z. B. verschiedene Rekursionsarten). Lerne sie aber bitte nicht auswendig! In der Klausur wirst du *keine* Fragen der Form „Welchen Wertebereich hat `int`?", „Welche Arten von Ausdrücken und Anweisungen gibt es?" oder „Beschreibe alle Java-Modifier." finden.

In der letzten Klausur (06.07.2020) wurden erstmals auch die Themengebiete MiniJava-VM, Netzwerkprogrammierung und grafische Benutzeroberflächen (GUIs) abgeprüft, was zuvor nie der Fall war, obwohl sie schon seit mehreren Semestern Teil der Vorlesung sind. Möglicherweise hängt das damit zusammen, dass diese Klausur COVID-19-bedingt online durchgeführt wurde. Da das Skript seitdem nicht aktualisiert wurde, deckt es diese Themengebiete nicht ab.

Grundsätzlich besteht viel Spielraum für die Klausurgestaltung. In manchen Klausuren waren bspw. Polymorphie-Aufgaben eher einfacher, die Aufgaben zu Rekursion dafür umso schwieriger, in anderen Klausuren war es genau andersherum.

THEMENÜBERSICHT

1. GRUNDLEGENDE BEGRIFFE

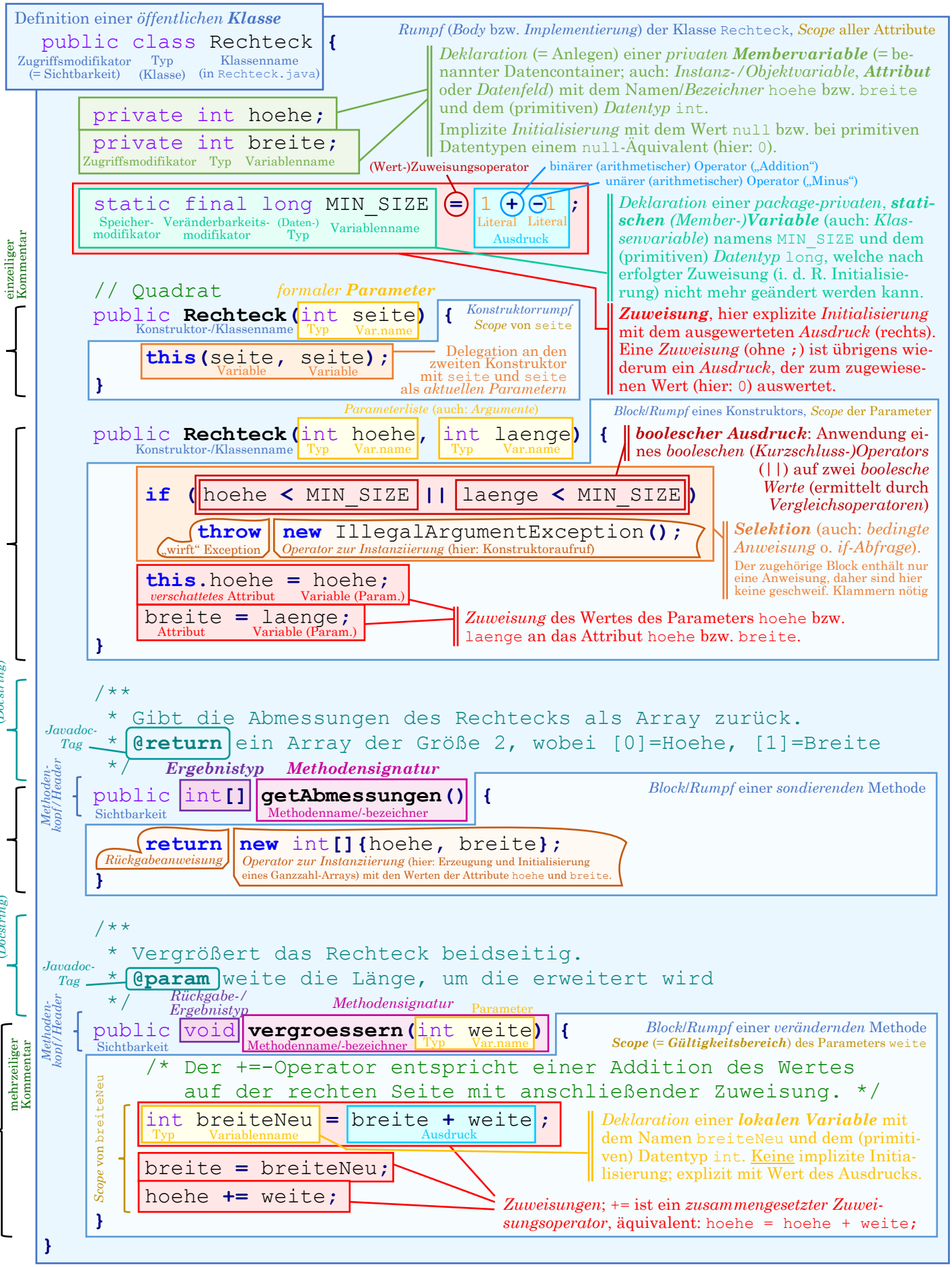
Markieren Sie in folgendem *Quelltext* alle

(1) *Variablenbezeichner* und *-deklarationen*,
(2) *Schlüsselwörter*, darunter insb. (a) *Typen*, (b) *Operatoren* und (c) *Modifier*,
(3) *Ausdrücke (expressions)*, insb. *Zuweisungen (assignment statements)*,
(4) *Literale* und
(5) *Kontrollstrukturen* (Selektion, Iteration, ...).

Trennen Sie unterschiedliche Variablenarten (*lokale Variable*, *Attribute*, ...) und unterstreichen Sie die *Methodenköpfe* (*header*). Kennzeichnen Sie einzelne *Blöcke* und überlegen Sie sich den Gültigkeitsbereich (*scope*) aller Variablen. Kommt *Überladung (overloading)*, *Überschreibung (overriding)* oder *Verschattung* (*hiding*) vor?

```
public class Rechteck {

    private int hoehe;
    private int breite;
    static final long MIN_SIZE = 1 + -1;

    // Quadrat
    public Rechteck(int seite) {
        this(seite, seite);
    }

    public Rechteck(int hoehe, int laenge) {
        if (hoehe < MIN_SIZE || laenge < MIN_SIZE)
            throw new IllegalArgumentException();
        this.hoehe = hoehe;
        breite = laenge;
    }

    /**
     * Gibt die Abmessungen des Rechtecks als Array zurück.
     * @return ein Array der Größe 2, wobei [0]=Hoehe, [1]=Breite
     */
    public int[] getAbmessungen() {
        return new int[]{hoehe, breite};
    }

    /**
     * Vergrößert das Rechteck beidseitig.
     * @param weite die Länge, um die erweitert wird
     */
    public void vergroessern(int weite) {
        /* Der +=-Operator entspricht einer Addition des Wertes
           auf der rechten Seite mit anschließender Zuweisung. */
        int breiteNeu = breite + weite;
        breite = breiteNeu;
        hoehe += weite;
    }
}
```

Lösungsvorschlag:



2. Primitive Datentypen & Typecasting

Im Folgenden sind alle in Java definierten *primitiven Datentypen* und deren Wertebereiche gelistet. Neben diesen existieren noch die sog. *Referenzdatentypen*. Variablen eines solchen Typs (jede Klasse, bspw. `String` oder einem *Wrapper*-Typ) speichern im Gegensatz zu Variablen primitiven Typs als *Wert* nicht direkt eine Zahl (z. B. `17.0`), sondern einen *Verweis auf ein Objekt*, also die Speicheradresse – eine Referenz/Zeiger/Pointer (grafisch häufig dargestellt als Pfeil).

	Typ	**Größe** (Bit)	**Wertebereich**	Suffix	***Wrapper**klasse* (in `java.lang`)
boole-scher Wert	`boolean`	undefiniert	`false` / `true`	-	`Boolean`
Ganze Zahlen	`byte`	8	$-2^7 \dots 2^7-1$ -128 ... 127	-	`Byte`
	`short`	16	$-2^{15} \dots 2^{15}-1$ -32768 ... 32767	-	`Short`
	`int`	32	$-2^{31} \dots 2^{31}-1$ -2147483648 ... 2147483647	-	`Integer`
	`long`	64	$-2^{63} \dots 2^{63}-1$ ca. $-9 \cdot 10^{18} \dots 9 \cdot 10^{18}$	`l` / `L`	`Long`
Gleitkommazahlen (nach IEEE 754)	`float`	32 » Vorzeichen: 1 » Mantisse: 23 » Exponent: 8	ca. $\pm 1{,}4 \cdot 10^{-45} \dots \pm 3{,}4 \cdot 10^{38}$ und `NaN`, `±Infinity`, `±0` (Exponent: -126 ... 127)	`f` / `F`	`Float`
	`double`	64 » Vorzeichen: 1 » Mantisse: 52 » Exponent: 11	ca. $\pm 4{,}9 \cdot 10^{-324} \dots \pm 1{,}7 \cdot 10^{308}$ und `NaN`, `±Infinity`, `±0` (Exponent: -1022 ... 1023)	(`d` / `D`)	`Double`
Unicode-Zeichen (UTF-16)	`char`	16	0 ... $2^{16}-1$ 0 ... 65535 (z. B. 65 ≙ `'A'`)* `'\u0000'` ... `'\uFFFF'`	-	`Character`

* Für uns interessant sind i. d. R. nur die ersten 128 Zeichen (ASCII), vgl. *www.torsten-horn.de/techdocs/ascii.htm*

Präfix	**Bedeutung** (Alphabet)
... für ganze Zahlen:	
`0b` / `0B`	Binärsystem (`0-1`)
`0`	Oktalsystem (`0-7`)
(keiner)	Dezimalsystem (`0-9`)
`0x` / `0X`	Hexadezimalsystem (`0-F`)
... für Gleitkommazahlen:	
(keiner)	Dezimalsystem (`0-9`)
`0x` / `0X`	Hexadezimalsystem (`0-F`)

Zur Ankündigung/Einleitung des optionalen **Exponenten** in Gleitkommazahlen verwendet man (nach der Fließkommazahl) das **Präfix** `e` / `E`.

Es bestimmt eine *Verschiebung des Kommas* um x Stellen (Multiplikation mit 10^x), wobei der Exponent x **ganzzahlig** ist (z. B. `1.1e2` für `110.0`).

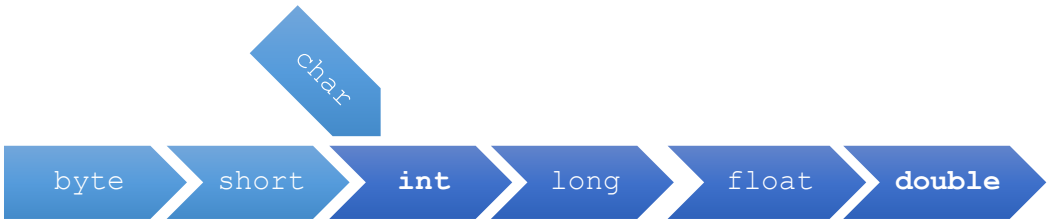
Für Gleitkommazahlen im Hexadezimalsystem ist der Exponent obligatorisch! Er entspricht nun einer Multiplikation der Vorkommazahl (und nur dieser) mit 2^x und wird eingeleitet mit dem Präfix `p` / `P` (da `e` / `E` eine gültige Hexadezimalziffer ist).

VARIABLENNAMEN …

… dürfen lediglich aus Buchstaben, Ziffern, Unterstrichen und Währungssymbolen bestehen
… dürfen nicht mit einer Ziffer beginnen
… dürfen nicht reservierten Schlüsselwörtern (`new`, `class`, `while`, `boolean`, ...) entsprechen
… sind case-sensitive (es wird zwischen Groß- und Kleinschreibung unterschieden)

TYPUMWANDLUNG

Implizites Typecasting bezeichnet in Java den Vorgang der automatischen *Typverallgemeinerung/-erweiterung*. So ergibt die Division `7/2` bspw. `3` (`int` geteilt durch `int` ergibt `int`), während bei `7/2.0` die `7` implizit (d. h. automatisch, ohne es explizit zu fordern) zur Gleitkommazahl `7.0` wird, sodass die Division `3.5` ergibt. Die Typerweiterung ist nötig, weil Operatoren (hier die Division) nur zwischen Werten desselben Typs definiert sind. Aber warum wird die `7` zu `7.0` geändert, schließlich wäre es auch vorstellbar, die `2.0` zu `2` zu ändern, oder? Dafür gibt es eine vordefinierte Ordnung; der mögliche Wertebereich wird größer:



Beispiele:
```
int x = 7; double y = x/2;   // y speichert den Wert 3.0 (erst int-Division)
int z = 'A';                 // z speichert den Wert 65
```

Achtung: Der Cast von `int` oder `long` zu `float` bzw. von `long` zu `double` wird zwar automatisch durchgeführt, kann aufgrund der eingeschränkten Genauigkeit von Gleitkommazahlen (Mantissenbits) allerdings zu Informationsverlust führen.

Beispiel: `int x = 16777217; float y = x;` → `y == 16777216`

Explizites Typecasting bezeichnet die explizite *Typeinschränkung* und zeigt sich in Java i. d. R. durch einen Cast der Form `(Datentyp)Wert`. Bei primitiven Datentypen entspricht das der Einschränkung des Wertebereichs (im Gegensatz zur Erweiterung bei implizitem Typecasting). Möchte man eine Gleitkommazahl `x` bspw. mathematisch korrekt auf eine Ganzzahl des Typs `int` runden, so kann man dafür die Java-Funktion `Math.round(double)` verwenden. Da diese aber leider einen `long` zurückgibt, muss dieser anschließend noch explizit zu `int` gecastet werden, da ein impliziter Typecast von `long` nach `int` nicht möglich ist:

Beispiele:
```
double a = 2.7; int b = (int)a;      // b speichert den Wert 2
int c = (int) Math.round(a);         // c speichert den Wert 3
char d = 'C'; d = (char) (d + 2);    // d speichert den Wert 'E'
```

Der Cast im letzten Beispiel ist nötig, weil die Addition `d+2` einen `int` ergibt. Warum? `char` plus `int` geht nicht, weil Operatoren nur zwischen Werten desselben Typs funktionieren, daher erfolgt ein impliziter Upcast von `char 'C'` zu `int 67` und es ergibt sich der `int`-Wert *69*. Einen `int`-Wert kann man nicht implizit in einer `char`-Variable speichern, daher ist der explizite `(char)`-Cast nötig. Vergisst man diesen, so kompiliert das Programm nicht (*Compiler-Fehler*).

Typecasting bei Referenzdatentypen (Vorgriff auf Kapitel 12 und 17, Polymorphie):

Implizites und explizites Typecasting lässt sich analog dazu auch auf Referenzdatentypen (Klassen) übertragen, wobei die Vererbungshierarchie der Klassen vorgibt, in welche „Richtung" implizit gecastet werden kann und in welche ein expliziter Cast nötig ist. Damit ein Cast aber überhaupt möglich ist, müssen die beiden Klassen (Typ der Variable, die gecastet werden soll, und Cast-Klasse) in einer (ggf. indirekten) Vererbungsrelation stehen, d. h. bspw. ein Cast zwischen einer Klasse `Baum` und `Boolean` oder einem Array und `String` wäre in keinem Fall möglich. Das folgende Beispiel soll zeigen, wann Typecasting nötig ist (siehe Var. `f`) und wann es zwar möglich ist (kein Compiler-Fehler), aber zu Problemen führen kann (siehe Var. `i`):

```
1  class Tier {}
2  class Hund extends Tier {}
3  class Maus extends Tier {}
4  Tier a = new Hund();   // möglich wegen implizitem Cast
5  Hund b = new Hund();   // möglich, ohne Cast
6  Tier c = b;            // möglich wegen implizitem Cast
7  Tier d = (Tier) b;     // möglich, überflüssiger expliziter Cast
8  Tier e = (Hund) b;     // möglich, überflüssiger expl. Cast, dann impl. Cast
9  Hund f = a;            // Compiler-Fehler, statischer Typ von a ist Tier
10 Hund g = (Hund) a;     // möglich wegen explizitem Cast
11 Hund h = g;            // möglich, ohne Cast
12 a = new Maus();        // möglich wegen implizitem Cast
13 Hund i = (Hund) a;     // Laufzeitfehler, dynamischer Typ von a ist Maus
```

SONSTIGE FAKTEN ZU PRIMITIVEN DATENTYPEN

- Mit Unicode-Literalen kann man ganz normal rechnen: `'A' + 2 == 'C'`
- Die Genauigkeit von Gleitkommazahlen ist beschränkt: `0.1 + 0.1 + 0.1 != 0.3`
- Die Division durch Null (`0`) löst nur bei Ganzzahltypen eine `ArithmeticException` aus.
- Auszug besonderer Operationen bei Gleitkommazahlen:

```
±0.0  /  ±0.0        ==     NaN
  ±x  /  ±0.0        ==  ±Infinity
  ±x  /  ±Infinity   ==    ±0.0
±0.0  *  ±Infinity   ==     NaN
```

- Über-/Unterlauf bei ganzen Zahlen: Die größtmögliche Zahl plus 1 entspricht der kleinstmöglichen und umgekehrt, z. B. `Integer.MIN_VALUE - 1 == Integer.MAX_VALUE`.
- Zwischen je zwei Ziffersymbolen einer Zahl dürfen beliebig viele Unterstriche (_) stehen.
- Wird ein `String`-Objekt durch den Konkatenationsoperator (+) mit einem anderen Objekt oder Wert verknüpft, erfolgt eine automatische Typumwandlung des Objekts/Werts in einen `String` (bei Objekten über den Aufruf der `toString()`-Methode).
- Arithmetische Operatoren sind nur für `int`, `long`, `float` und `double` definiert, d. h. für `char`, `short` und `byte` erfolgt dann immer ein impliziter Cast zu `int`.
- Wrapper-Klassen packen primitive Werte in Objekte (Referenzdatentypen).
  » `int a = 5;` speichert in `a` den Wert `5`.
  » `Integer a = 5;` speichert in `a` ein Objekt (der Klasse `Integer`) mit dem Inhalt `5`.
- Die Wrapper-Klassen erben nicht voneinander, d. h. Autoboxing von `5` führt bspw. zu einem `Integer(5)`. Nicht möglich ist bspw. `Long x = new Integer(1);` oder `Double x = 5;`

① ☆☆☆	**Primitive Datentypen**: Kompilieren folgende Anweisungen?	Ja	Nein
a)	`int ä = 'a';`	☐	☐
b)	`byte b = '\u0000';`	☐	☐
c)	`int c = 019;`	☐	☐
d)	`int d = 17/0;`	☐	☐
e)	`long éâ = Integer.MAX_VALUE;`	☐	☐
f)	`int 2f = 2;`	☐	☐
g)	`short _g_ = 100_00;`	☐	☐
h)	`float h = -42;`	☐	☐
i)	`double i = 314e-2;`	☐	☐
j)	`short j = 32768;`	☐	☐
k)	`char k1 = 32768;`	☐	☐
l)	`long l = 2_147_483_647 + 1;`	☐	☐
m)	`int m2 = 1L;`	☐	☐
n)	`float n = 1.234;`	☐	☐
o)	`double o = 0x0e0;`	☐	☐
p)	`double p = 0.5d;`	☐	☐
q)	`int int_ = 0017;`	☐	☐
r)	`double r = 0b10110;`	☐	☐
s)	`float s = 028F + 0.1;`	☐	☐
t)	`byte t = '\uFFFF';`	☐	☐
u)	`int u = 0x0BCe2;`	☐	☐
v)	`double super = .1;`	☐	☐
w)	`double cool = 5.5e;`	☐	☐
x)	`char x = -1;`	☐	☐
y)	`long y = 0b1_0_1_0_L;`	☐	☐
z)	`double €_$5 = 0XABCe5;`	☐	☐
A)	`double A = 0.e5;`	☐	☐
B)	`byte B = 250_;`	☐	☐
C)	`double C = 0xABC.9e51;`	☐	☐
D)	`float D¥ = 018.f;`	☐	☐
E)	`char E = "A" + "B";`	☐	☐
F)	`double F = 0x.AP1d;`	☐	☐
G)	`char G! = '';`	☐	☐
H)	`float H = 0.1f * 2;`	☐	☐
I)	`float I = 1.0f + 0.1;`	☐	☐
J)	`char J = 'B' - 'A' + 2;`	☐	☐
K)	`double K = 0xApA;`	☐	☐
L)	`long L = 0_7L;`	☐	☐
M)	`double M-M = -'@';`	☐	☐
N)	`short N = Integer.MIN_VALUE;`	☐	☐
O)	`float Oß = 0xa2.382ap3f;`	☐	☐
P)	`char P#P = 'H';`	☐	☐

② ★☆☆ **Primitive Datentypen**: Sind folgende Aussagen korrekt?

		Wahr	Falsch
a)	Sei `a` ein beliebiger `double`, dann sind die folgenden Programmstücke semantisch (= Bedeutung) äquivalent: `double b = a/a;` und `double b = 1.0;`	☐	☐
b)	Der Ausdruck `int b = a/a;` kompiliert *immer*, wenn `a` den Typ `int`, `short`, `char` oder `byte` hat.	☐	☐
c)	Mit dem Datentyp `float` können größere Zahlen gespeichert werden als mit dem Datentyp `long`.	☐	☐
d)	Bei einem `Integer`-Overflow wird *weder* eine `Exception` geworfen, *noch* entsteht ein „compile time error“.	☐	☐

e)	Am Ende des folgenden Codeauszugs ist die Zusicherung „`a == b`“ für alle (gültigen) Werte für `a` gültig:
	`int a = _____;`
	`float tmp = a;`
	`int b = (int) tmp;`
		☐	☐

f)	Folgender Ausdruck wertet zu **`true`** aus: `Integer.MAX_VALUE + 1 == Integer.MIN_VALUE`	☐	☐
g)	Sei `x` vom Typ `int`, dann kann die bedingte Anweisung `if (x + 1 > 1)` ohne semantische Änderung zu `if (x > 0)` gekürzt werden.	☐	☐

h)	Angenommen `left` und `right` haben den Typ `long`, dann sind folgende Berechnungen äquivalent:
	`long middle = (left+right) / 2;`
	und
	`long middle = left + (right-left)/2;`
		☐	☐

i)	Folgender Ausdruck wertet zu **`true`** aus: `'\uFFFF' + 1 == 0`	☐	☐
j)	Sei `d` eine als `double` deklarierte Variable, dann führt der Ausdruck `d / 0` *nie* zu einem Fehler (`Exception`).	☐	☐
k)	Jeder `float` kann als `double` dargestellt werden.	☐	☐

l)	Das Konstrukt
	`for (double d = 0.1; d < 1; d += 0.1)`
	`  System.out.println(d);`
	gibt die Zahlen `0.1` bis `0.9` auf der Konsole aus.
		☐	☐

③ ☆☆☆ **Typecasting**: Kompilieren die folgenden Codeausschnitte?

		Ja	Nein
a)	`int a1 = 1; short a2 = a1;`	☐	☐
b)	`String b = 5.2 + "";`	☐	☐
c)	`int c = (long) 17;`	☐	☐
d)	`String d = 25;`	☐	☐
e)	`long e = 'e' + (char) .5;`	☐	☐
f)	`int f = (short) (byte) 0;`	☐	☐
g)	`String g = (String) '6';`	☐	☐
h)	`float h = (float) 5*5.0;`	☐	☐
i)	`String i = 7 + "" * 8;`	☐	☐
j)	`int _j, j_;  _j = j_ = 5;`	☐	☐
k)	`String k = "Hello" + 'World';`	☐	☐
l)	`byte l = (short) 12;`	☐	☐
m)	`float m = (float) (2f + 2d);`	☐	☐
n)	`int nn = 1;` `float n = (char) Integer.MAX_VALUE+nn;`	☐	☐
o)	`float o = 4;  int ole = (int) 2 * o;`	☐	☐
p)	`long _p_ = 6;` `int p = (int) _p_ * (int) _p_;`	☐	☐
q)	`float q = (float) 018 + 1.0f;`	☐	☐
r)	`boolean r = (boolean) 0;`	☐	☐
s)	`String s = "";  s += 5;`	☐	☐
t)	`float t = (float) 2.4 * 13;`	☐	☐
u)	`long u = (double) 2f;`	☐	☐
v)	`float v = 2f + 2 * (int) 2.0;`	☐	☐
w)	`String w = 'w' * 6 + "w" + 3 * 2;`	☐	☐
x)	`boolean x = .5 > 2;`	☐	☐
y)	`int y = 9 + (char)1.23e4 - (int)0xABCp2;`	☐	☐
z)	`double z = (float) 2.0_0e+0___2;`	☐	☐
A)	`String A; Object o;` `A = 5 - 2 + o.toString() + (2 - -1);`	☐	☐
B)	`double B = .1 * 'B' + 5f;`	☐	☐
C)	`int C = 1;` `byte C0 = (byte) 100 + (byte) C;`	☐	☐
D)	`String D = "DeDe";  D++;`	☐	☐
E)	`short E = 1;  short EE = E + 'H';`	☐	☐
F)	`int F = (int) "5";`	☐	☐
G)	`boolean G = (boolean) (15 - 3 > 1 || false);`	☐	☐
H)	`long H = 2 + (int) 2L - ' ';`	☐	☐
I)	`String I = "I" + ((42 > 5 * 8) ? 2 : 4);`	☐	☐
J)	`char J = 2*'J' + (char)((long) 0.2e4 + 5);`	☐	☐
K)	`short K = 1;  char KK = K;`	☐	☐
L)	`char L = 3 * ('L' + (int) 0x2L);`	☐	☐

④ **Wrapperklassen** / Boxing: Stimmen folgende Aussagen?

★☆☆ **Hinweis:** Klassen und Objekte behandeln wir erst in Kapitel 12, Generics in Kapitel 13!

		Ja	Nein
a)	Einer Variable eines primitiven Datentyps können aufgrund von automatischem *Unboxing* Objekte der zugehörigen Wrapperklasse oder eines erlaubten „untergeordneten" Typs zugewiesen werden, wie z. B. `double a = new Integer(8);`	☐	☐

b)	Einer Variable vom Typ `Object` können alle primitiven Werte sowie Objekte zugehöriger Wrapperklassen zugewiesen werden, also z. B.
	`Object b1 = 0x4e2;`
	`Object b2 = new Integer(4);`
		☐	☐

c)	Einer Variable eines „Wrapperklassen"-Typs können aufgrund von *Autoboxing* primitive Werte des entsprechenden primitiven Typs zugewiesen werden, bspw. `Integer c = 123;`	☐	☐
d)	Einer Variable eines „Wrapperklassen"-Typs können außerdem Werte „untergeordneten" Typs zugewiesen werden, wie `Integer d = 'd';`	☐	☐
e)	Primitive Typen können als *„Generics"* (Typparameter beim Erzeugen) verwendet werden, bspw. `LinkedList<int> list = new LinkedList<>();`	☐	☐
f)	Wrapperklassen können als Typparameter verwendet werden werden, bspw. `List<Integer> list;`	☐	☐

g)	In folgendem Codeauszug gilt die Zusicherung ...
	`Integer g1 = new Integer(1);`
	`Integer g2 = 1;`
	`int g3 = g1;`		

	... „`g1 == g2`"	☐	☐
	... „`g2 == g3`"	☐	☐
	... „`g3 == g1`"	☐	☐

h)	Einer Variable vom Typ `Object` können alle Objekte zugewiesen werden, schließlich ist `Object` die „oberste" Oberklasse. D. h. z. B. Folgendes kompiliert:
	`Object o = new int[]{1, 2, 3};`
	`o = 17.2;`
	`o = new LinkedList<Integer>();`
		☐	☐

i)	Folgende Statements kompilieren:		
	`Double i1 = 1;`	☐	☐
	`double i2 = new Float('?');`	☐	☐
	`Long i3 = new Integer(17);`	☐	☐
	`Float i4 = (float) (new Integer(1));`	☐	☐
	`byte i5 = new Byte(0);`	☐	☐
	`boolean x = new Integer(0) >= new Integer(0);`	☐	☐

3. ZAHLENSYSTEME

Hinweis: *Voraussichtlich nicht klausurrelevant (WS 19/20)*

3.1. Rechnen mit verschiedenen Basen

Zahlen zur selben Basis können genauso schriftlich addiert, subtrahiert, multipliziert oder dividiert werden, wie aus der Schule bereits bekannt sein sollte. Anders als im Zehnersystem existieren für Zahlen zu einer anderen Basis aber entsprechend weniger oder mehr Ziffern, also z. B. nur die Ziffern 0 und 1 im Binärsystem (Basis: 2) oder 0 bis 9 und A bis F im Hexadezimalsystem (Basis: 16). Im 21er-System wäre die größte Ziffer bspw. K (Groß-/Kleinschreibung nicht relevant). Während wir im Zehnersystem also $1, 2, 3, 4, 5, 6, 7, \ldots, 10, 11, \ldots$ zählen, heißt es im Dreiersystem (Basis 3) analog $1, 2, 10, 11, 12, 20, 21, \ldots, 101, 102, \ldots$. Die Basis wird i. d. R. tiefgestellt angehängt (z. B. $7_{10} = 21_3$). Hexadezimalzahlen wird stattdessen häufig „0x“ vorangestellt.

⑤ ☆☆☆ Führen Sie die folgenden Berechnungen schriftlich durch, ohne die Zahlen in ein anderes Zahlensystem umzuwandeln. Tragen Sie auch die Überträge zur selben Basis ein.

a)

		6	3	5	2	0	1	6	7
			4	3	6	5	6	0	7
	+								
									7

b)

			A	F	E	0	5	D	C	9	16
		3	B	4	C	D	F	0	3	1	16
	+										
											16

c)

	5	A	B	1	C	3	A	4	13
		7	B	A	B	4	9	2	13
-									
									13

d)

	1	1	0	1	1	0	0	0	1	1	2
		1	1	0	0	1	0	1	1	0	2
-											
											2

e)

	2	3	3	4	3	0	1	5
		4	1	4	2	2	3	5
	1	4	2	1	2	4	0	5
+								
								5

f)

	1	0	1	1	1	0	1	1	0	2
		1	1	1	1	1	0	0	1	2
			1	0	1	1	0	1	0	2
+										
										2

g)

	1	1	0	0	2	1	1	2	3
			2	2	1	2	1	0	3
		1	2	1	2	1	0	1	3
-									
									3

h)

	8	A	9	C	1	6	B	3	A	7	15
	7	3	E	4	A	D	0	6	5	2	15
			E	5	E	5	A	2	4	3	15
-											
											15

i)

		A	3	2	A	5	9	0	11
		4	6	7	9	3	4	2	11
		9	0	6	A	8	1	A	11
				A	8	6	6	7	11
+									
									11

j)

	1	1	1	0	0	1	0	1	0	1	2
				1	1	0	0	1	1	0	2
		1	1	1	1	0	1	0	0	1	2
				1	1	0	1	1	1	0	1	2
-											
											2

⑥ ☆☆☆ Führen Sie die folgenden Multiplikationen schriftlich ohne Umwandlung in ein anderes Zahlensystem durch. Das „Kleine Einmaleins“ für Hexadezimalzahlen ist vorgegeben:

·	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**A**	**B**	**C**	**D**	**E**	**F**
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
2	0	2	4	6	8	A	C	E	10	12	14	16	18	1A	1C	1E
3	0	3	6	9	C	F	12	14	18	1B	1E	21	24	27	2A	2D
4	0	4	8	C	10	14	18	1C	20	24	28	2C	30	34	38	3C
5	0	5	A	F	14	19	1E	23	28	2D	32	37	3C	41	46	4B
6	0	6	C	12	18	1E	24	2A	30	36	3C	42	48	4E	54	5A
7	0	7	E	15	1C	23	2A	31	38	3F	46	4D	54	5B	62	69
8	0	8	10	18	20	28	30	38	40	48	50	58	60	68	70	78
9	0	9	12	1B	24	2D	36	3F	48	51	5A	63	6C	75	7E	87
A	0	A	14	1E	28	32	3C	46	50	5A	64	6E	78	82	8C	96
B	0	B	16	21	2C	37	42	4D	58	63	6E	79	84	8F	9A	A5
C	0	C	18	24	30	3C	48	54	60	6C	78	84	90	9C	A8	B4
D	0	D	1A	27	34	41	4E	5B	68	75	82	8F	9C	A9	B6	C3
E	0	E	1C	2A	38	46	54	62	70	7E	8C	9A	A8	B6	C4	D2
F	0	F	1E	2D	3C	4B	5A	69	78	87	96	A5	B4	C3	D2	E1

a) $E5A7_{16} \cdot B_{16}$

+

$_{16}$

b) $F01F6E_{16} \cdot C_{16}$

+

$_{16}$

c) $123_{16} \cdot 90_{16}$

d) $5B73 \cdot 1020_{16}$

e) $10111010_{2} \cdot 100010_{2}$

+

$_{2}$

f) $5F02C_{16} \cdot 71D_{16}$

g) $feed_{16} \cdot c0f4_{16}$

h) $1110010100110_2 \cdot 1001101112_2$

i) $CBF68FE_{16} \cdot E2D593C_{16}$

3.2. Basisumwandlung

(1) **Basis X → Basis Y** (insb. X → 10):

Die Umwandlung von einer Zahl zur Basis X in eine Zahl zur Basis Y machen wir intuitiv, indem wir die Zahl zunächst ins Zehnersystem und anschließend in das Zielsystem konvertieren. Dieser Schritt ist jedoch nicht nötig und es kann in einer entsprechenden Aufgabe verlangt sein, eine Konvertierung ohne Umwandlung ins Zehnersystem durchzuführen.

Wir wollen Zahlen nun möglichst weit (d. h. auf die einzelnen Ziffern) „runterbrechen". Die Zerlegung von 2739 im Zehnersystem wäre:

$$2739 = 2000 + 700 + 30 + 6 = 2 * 10^3 + 7 * 10^2 + 3 * 10^1 + 9 * 10^0$$

Nichts anderes tun wir nun in anderen Zahlensystemen, d. h. bspw. im Hexadezimalsystem:

$$2739_{16} = 2_{16} * 10_{16}^3 + 7_{16} * 10_{16}^2 + 3_{16} * 10_{16}^1 + 9_{16} * 10_{16}^0 = \ ...$$

Wollen wir 2739_{16} nun bspw. im Oktalsystem (Basis: 8) darstellen, müssen wir die einzelnen Teile zur Basis 8 schreiben. Im Kopf können wir hier über das Zehnersystem gehen. Zu beachten ist dabei, dass 10 zur Basis 16 dezimal nicht 10 ist, denn $10_{16} = 16_{10}$! Allgemein: $10_X = X_{10}$. 16_{10} entspricht wiederum 20_8, denn $16 = 2 * 8^1 + 0 * 8^0$. Weiters ist z. B. $9_{16} = 9_{10} = 1 * 8^1 + 1 * 8^0 = 11_8$, also:

$$... = 2_8 * 20_8^3 + 7_8 * 20_8^2 + 3_8 * 20_8^1 + 11_8 * 20_8^0 = \ ...$$

Nun müssen wir nur noch die ganzen Multiplikationen und Additionen im Oktalsystem durchführen (schriftlich), d. h. $20_8^2 = 20_8 * 20_8 = 400_8$, $20_8^3 = 20_8^2 * 20_8 = 400_8 * 20_8 = 10000_8$, also:

$$... = 2_8 * 10000_8 + 7_8 * 400_8 + 3_8 * 20_8 + 11_8 = \ ...$$

Man multipliziert wieder schriftlich und addiert dann schriftlich, sodass man Folgendes erhält:

$$... = 20000_8 + 3400_8 + 60_8 + 11_8 = 23471_8$$

(2) **Basis 2 ↔ Basis 16**:

Immer vier Stellen der Binärzahl entsprechen einer Stelle der Hexadezimalzahl. Man ergänzt führende Nullen am besten.

Beispiel: $0001\ 0010\ 1010\ 0110\ 1000\ 0101_2 = 1\ 2\ A\ 6\ 8\ 5_{16}$

(3) **Basis 10 → Basis Y**:

Dividiere die Zahl mit Rest durch Y. Beispiel: 2739_{10} ins 5er-System:

2739 : 5	=	547	Rest	4
547 : 5	=	109	Rest	2
109 : 5	=	21	Rest	4
21 : 5	=	4	Rest	1
4 : 5	=	0	Rest	4

→ $2739_{10} = 41424_5$

Weitere Erläuterungen: *www.arndt-bruenner.de/mathe/scripts/Zahlensysteme.htm*

⑦ ☆☆☆ Stellen Sie jeweils die zuerst gegebene Zahl zur genannten Zielbasis dar.

a) $1100\ 0111\ 1001\ 0100\ 1111\ 0011_2$ → Basis 16

b) $10101001101110100001011_2$ → Basis 16

c) $8C51F_{16}$ → Basis 2

d) 1010010111_2 → Basis 10

e) 1425_6 → Basis 10

f) 2573_{10} → Basis 16

g) 18293_{10} → Basis 5

h) 2120112_3 → Basis 7

i) 4521220_6 → Basis 3

j) 40513_7 → Basis 12

4. Operatoren, Ausdrücke & Auswertungen

Die folgende Tabelle listet alle in Java definierten Operatoren absteigend nach ihrer Bindungsstärke (Präzedenz) auf. Je weiter oben ein Operator in der Tabelle aufgeführt ist, desto stärker bindet er, d. h. die Operation wird umso früher durchgeführt. Operatoren mit demselben Rang (bspw. `*` und `/` mit Rang 4) haben auch dieselbe Bindungstärke – die Auswertung erfolgt dann von links nach rechts (Linksassoziativität). Eine Ausnahme stellen die Operatoren der Stufen 3, 14 und 15 dar, welche von rechts nach links (rechtsassoziativ) ausgewertet werden.

Beispiel: Der Ausdruck `1 + 5 / 8 * 8` wertet zu `1` aus, weil Division (`/`) und Multiplikation (`*`) gleich stark und die Addition (`+`) schwächer bindet: `1 + 5 / 8 * 8 = 1 + 0 * 8 = 1 + 0 = 1`. Zuerst wird also `5 / 8` ausgewertet, was `0` ergibt, weil Operatoren in Java nur zwischen gleichen Typen definiert sind und das Ergebnis ebenfalls denselben Typ besitzt – anders ausgedrückt: `int / int = int`. Der mathematisch korrekte Nachkommateil (5/8 = 0,625) wird verworfen.

Rang	Beschreibung	Operator / Symbol
0	Klammerung	`(...)`

1	Funktionsaufruf
	Array-Indizierung
	Objekterzeugung
	Memberzugriff (z. B. Attribute)
		`someFunction(...)`
		`someArray[someIndex]`
		`new ...`
		`someObject.someMember`

2	Postinkrement/-dekrement	`x++, x--`

3	Präinkrement/-dekrement
	unäre Operatoren
	Typecast
		`++x, --x`
		`+x, -x, !x, ~x`
		`(someType) x`

4	Multiplikation, Division Modulo (Teilerrest)	`a * b, a / b` `a % b`
5	Addition, Subtraktion Stringkonkatenation (Verknüpfung)	`a + b, a - b` `s + t`
6	Bitverschiebung (Shifts)	`d << k, d >> k, d >>> k`
7	Vergleich	`a < b, a > b, a <= b,` `a >= b, obj instanceof Cls`
8	Gleich / Ungleich	`a == b, a != b`
9	bitweises „Und“	`b & c`
10	bitweises „exklusives Oder“	`b ^ c`
11	bitweises „Oder“	`b | c`
12	logisches/boolesches „Und“	`B && C`
13	logisches/boolesches „Oder“	`B || C`
14	Bedingung (ternärer Operator)	`B ? y : n`

15	Zuweisung
	bzw.
	zusammengesetzte Zuweisungsoperatoren
	`a Δ= b` entspricht `a = (Typ von a) (a Δ b)`
		`a = b`
		`a += b, a -= b, a *= b,`
		`a /= b, a %= b, b &= c,`
		`b ^= c, b |= c, d <<= k,`
		`d >>= k, d >>>= k`

Kurzschluss-auswertung

`x++` erhöht `x` um 1 und wertet zum *alten* Wert von `x` aus.

```
y = x++;
```
entspricht
```
y = x;
x = x + 1;
```

```
y = ++x;
```
entspricht
```
x = x + 1;
y = x;
```

`++x` erhöht `x` um 1 und wertet zum *neuen* Wert von `x` aus.

Man unterscheidet in Java grundsätzlich zwischen **Ausdrücken** (*expressions*) und **Anweisungen** (*statements*). Jeder Ausdruck hat dabei einen festen **Typ** und einen **Wert**. Der Typ steht immer schon vor der Programmausführung (also *bei der Kompilierung*) fest, während der Wert erst *zur Laufzeit* des Programms berechnet wird. Beispiele für Ausdrücke sind:

- **Variablen** (z. B. `x`): Der Typ ist durch die Deklaration (z. B. `float x;`) bekannt. Eine Variable wertet bei Benutzung einfach zu ihrem derzeit gespeicherten Wert (z. B. `1.0`) aus.
- **Konstanten** (z. B. `1.2` oder `false`): Typ und Wert sind sofort erkennbar.
- **Unäre Operationen** (z. B. `-x` oder `!c`): Der Typ steht bereits vor der Ausführung fest. Der Typ von `!c` wäre bspw. sicher `boolean`, d. h. auch `c` muss den Typ `boolean` besitzen. Der Typ von `-x` hängt hingegen vom Typ von `x` ab. Wurde `x` zuvor als `float` deklariert, so hat auch `-x` den Typ `float`. Wurde `x` hingegen als `String` deklariert, würde der Ausdruck `-x` nicht kompilieren, da das unäre Minus nur auf Zahlentypen definiert ist. Der Wert hängt natürlich vom Wert des Operanden ab, z. B. ist der Wert von `!c` genau dann `true`, wenn `c` den Wert `false` hat, und der Wert von `-x` genau dann `2`, wenn `x` den Wert `-2` hat.
- **Binäre Operationen** (z. B. `5.2f + 2`, `b && c`, `d / 2` oder `f == g`): Wert und Typ gehen aus den Werten und Typen der Operanden hervor, wobei auch hier der Typ bereits vor der Ausführung feststeht. So wertet `f == g` sicher zu einem `boolean` aus, wobei `f` und `g` unterschiedliche Typen besitzen können (z. B. `int` und `float`, oder `boolean` und `boolean`). Der Typ von `d / 2` hängt hingegen vom Typ von `d` ab, welcher hier z. B. `int` oder `double`, nicht aber `boolean` oder `Object` sein könnte. Wurde `d` bspw. mit `d = 7;` initialisiert aber als `double` deklariert, so hätte bei der Zuweisung ein impliziter Cast von `7` (`int`) zu `7.0` (`double`) stattgefunden und die Operation würde zu `7.0 / 2 = 7.0 / 2.0 = 3.5` auswerten (2 wird also implizit zu `2.0` gecastet). Wurde `d` hingegen als `int` deklariert, so würde der Ausdruck zu `7 / 2 = 3` auswerten, denn `int` / `int` ergibt wieder `int`. Im Beispiel `5.2f + 2` (also `float + int`) fände ebenfalls ein impliziter Cast von `2` zu `2f` statt und das Ergebnis wäre `7.2f`.
- **Ternärer Operator** (z. B. `b ? "x" : 2.1`): `b` muss ein boolescher Ausdruck sein, während die Typen der Ausdrücke im *yes*- und *no*-Teil (hier `"x"` bzw. `2.1`) beliebig sein können. Der Ausdruck „`b ? "x" : 2.1`“ wertet also zu dem `String` `"x"` aus, falls `b` den Wert `true` annimmt, anderenfalls zu dem `double` `2.1`. Auch hier steht der Typ des gesamten Ausdrucks allerdings bereits zur *compile time* fest. *Wie kann das sein?* Hier wird immer der „kleinste gemeinsame Typ“ benutzt. In diesem Beispiel wäre das die Klasse `Object` (oberste Oberklasse), d. h. der (statische) Typ von „`b ? "x" : 2.1`“ ist `Object`! Normalerweise besitzen *yes*- und *no*-Teil aber denselben Typ. Beispiel: `c >= 5 ? 2.0 : 5.2` hat den Typ `double` (da sowohl `2.0` als auch `5.2` vom Typ `double` ist) und würde zur Laufzeit immer dann zu `5.2` auswerten, wenn `c` einen Wert kleiner als `5.0` annimmt, sonst zu `2.0`.
- **Funktionsaufrufe** (z. B. `sum(2,4)` oder `Terminal.readInt()`): Der Typ eines Funktionsaufrufs entspricht dem Rückgabetyp der Funktion/Methode*. Der Wert ist der Rückgabewert der Methode, welcher von Parametern, Eingaben, Attributwerten usw. abhängen kann. Z. B. könnte der Aufruf `sum(2,4)` zur Methodensignatur `sum(double,float)` führen und zu `6.0` auswerten. Gibt eine Methode nichts zurück (`void`), so hat sie auch keinen Typ bzw. Wert.

* Eine Funktion ist i. d. R. statisch (`static`) und kapselt Code, der an mehreren Stellen benötigt wird, um Redundanz zu vermeiden. Eine Methode ist hingegen immer an ein Objekt gebunden. Kurz: Statische Methoden nennt man Funktionen.

- **Zuweisungen** (z. B. `x = 5`): Eine Zuweisung ist sowohl eine Anweisung als auch ein Ausdruck. Neben der Durchführung der Wertzuweisung (Seiteneffekt des Operators) wertet der Zuweisungsoperator nämlich immer zum zugewiesenen Wert aus. Ist die Variable `x` als `double` deklariert, so schreibt die Zuweisung `x = 5` einerseits den Wert `5.0` in die Variable `x`, wertet aber außerdem zu `5.0` aus. Aus diesem Grund sind auch ineinander verschachtelte Zuweisungen der Form `y = x = a` möglich. Hier zunächst der Wert von `a` (bspw. `5`) an die Variable `x` zugewiesen (wie zuvor) und anschließend der zugewiesene Wert (i. d. R. der Wert von `a` selbst, ggf. aber implizit gecastet) an `y` zugewiesen. Der Operator ist folglich rechtsassoziativ (→ `y = (x = a)`). **Anfängerfehler:** Die Zuweisung „=“ ist kein Vergleichsoperator („==“):

```
boolean myBool = false;
if (myBool = true) {   // „=“ ist Zuweisungsoperator: true wird an myBool zugewiesen
    System.out.println("Ja, dieser Text wird ausgegeben!");
}
```

- **`this`** und **`super`** (Vorgriff auf Kap. 12): Das Schlüsselwort `this` kann nur in einem objektorientierten Kontext (also nicht aus einem statischen Kontext, vgl. Kap. 6.2) verwendet werden. Es wertet einfach zum aktuellen Objekt aus, also zu dem Objekt, auf dem wir gerade operieren. `super` kann verwendet werden, um explizit auf eine Membervariable, Methode oder einen Kontruktor der Oberklasse zuzugreifen (auch statische Member sind möglich).
  Die Verwendung dieser beiden Schlüsselwörter wird später klarer.

⑧ ★☆☆ **Zuweisungen**: Bestimmen Sie die Werte aller Variablen nach Ausführung des Codes:

```
boolean b1, b5, b10, b12;
b1 = b5 = true;
b10 = b12 = false;
boolean b2 = false;
boolean b7 = false;
boolean b3 = !(!b1 && b2);
boolean b4 = b2 || (b1 = false);
boolean b6 = b2 && (b2 = true);
boolean b8 = (b7 = false) || true;
if (b5 = false) {
    b7 = b5;
}
boolean b9 = true;
if (b10 = true) {
    b9 = false;
}
boolean b11 = b12 ? (b12 = false) : true;
```

b1 =	b2 =	b3 =	b4 =
b5 =	b6 =	b7 =	b8 =
b9 =	b10 =	b11 =	b12 =

⑨ ★☆☆ Gegeben sei der folgende (korrekte) Java-Code. Bestimmen Sie den Wert jeder Variable am Ende der *main*-Funktion, also nachdem *main()* ausgeführt wurde. Schreiben Sie Werte vom Typ `char` in Hochkommas (`'a'`), `String`s mit Anführungszeichen (`"a"`) und `double`s in Punktnotation (`1.0`).

Hinweis: Das Zeichen `'A'` hat den ASCII-Dezimalwert 65.

```
public class Main {
    static int d, k;
    private static boolean e, t;
    protected static String u, f;

    public static void main(String[] args) {
        int a, b;
        char c = 'A';
        int l = 2;
        int r = l;
        int O = 7;
        long w = 0;
        char A = 'c';
        double x, y, z = 3F;
        boolean g = false;
        String s = c + "";
        x = a = d = 3 / 2;
        int p = O + l;
        A = c;
        t = r == 2;
        int h = c++;
        double i = O / l; i++;
        long j = O / l;
        int q = 2 + A;
        double m = l / 2 + 2 / 1;
        String v = s + u;
        long n = 5 % ++r;
        f = 1 + 2 + "" + 4 + 5;
        s += g; g = !g;
        t = t || (++i == 0);
    }
}
```

Variablen:

```
a =
b =
c =
d =
e =
f =
g =
h =
i =
j =
k =
l =
m =
n =
O =
p =
q =
r =
s =
t =
u =
v =
w =
x =
y =
z =
A =
```

⑩ ★☆☆ Vorgriff auf Kapitel 12: Begründen Sie stichpunktartig, warum der Ausdruck (1) zu `false` auswertet während Ausdruck (2) `true` ergibt. Verbessern Sie den Vergleich in Ausdruck (1) sinnvoll.

(1) `new Integer(6) == new Integer(6)`

(2) `6 == 6`

⑪ ★☆☆ Geben Sie jeweils an, wozu die folgenden Ausdrücke auswerten („Ergebnis“) und welche Werte die Variablen `a` und `b` nach der Auswertung haben. Schreiben Sie Strings in Anführungszeichen (bspw. `"1234"`). Welche Ausdrücke kompilieren überhaupt nicht?

Vor *jeder* Teilaufgabe seien die Startwerte `int a = 1, b = 3;`

		Ergebnis	a	b
a)	`a++ - --b`			
b)	`2 + -b + ++a`			
c)	`a++ + --a + (b-1)`			
d)	`"" + 3 * b`			
e)	`b++ + ++a - b--`			
f)	`++a + 2 * (--b - --b)`			
g)	`--b + (++a + b) * a`			
h)	`a - b + "" + ++a`			
i)	`++a++`			
j)	`++a % --b`			
k)	`5.5 + b++ - (++a + 2)`			
l)	`a = ++a + ++a - a++`			
m)	`b = a = b-- + 1 - b / 2`			
n)	`a++ > --b || b-1 > ++a`			
o)	`a = (++b < --b)`			
p)	`b = --a + a++ - b - b--`			
q)	`b += ++a`			
r)	`++a >= --b ? a++ : b--`			
s)	`(a++ != --b) && (a >= 2)`			
t)	`(b-1) < --b || (a += 1) > 2`			
u)	`++a + 1 + "1" + (b > a ? a * b : "7")`			
v)	`a *= --b / ++a+1`			
w)	`++b > b || a++ > a`			
x)	`(b <= b--) && (++a == a+1)`			
y)	`a = (++a > b-1) ? b-- : a++`			
z)	`b == a * ++a + --b-1`			
A)	`(a++ == --b) ? (b *= -1) : ++a`			
B)	`(b = 1) > --a && ++b == a + 1`			
C)	`a-- * ((a += ++a) < 2 * b ? a*a : b-2)`			
D)	`++b % --b > 1 ? (b = 14) : (a = 21)`			
E)	`(++a + 7) % ((a = 5) + --b) + --a`			
F)	`++a > 1 || ++b > 4  ==  false`			

Hinweis: *Auswertungsbäume sind voraussichtlich nicht klausurrelevant (WS 19/20)*

⑫ ☆☆☆ Geben Sie zu jedem der folgenden Ausdrücke den **Auswertungsbaum** an. Es muss eindeutig erkennbar sein, welchen Typ jeder Wert besitzt. Kennzeichnen Sie also wie üblich Werte vom Typ `char` mit Hochkommas (`'a'`), Werte vom Typ `String` mit Anführungszeichen (`"a"`), `double`s durch Punktnotation (`1.0`) usw.

Hinweis: `(int)'A' == 65` und `(int)'a' == 97`

a) `2 * ( 1 / 2 )`

b) `1 + 2 + "3" + 4 + 5`

c) `1 < 3 / 2`

d) `(char)('B'-'A'+'a')`

e) `( 16 % 5 >= 2 ) + "" + (double) 0 + 6`

f) `1.75 - 7 / 4 + ( - (double) 1 )`

g) `(double) 15/10 * 5%5 + (int) 0.8`

h) `! (-7 == (double) Integer.parseInt("-7")) != ! ! false`

i) `5d == (int) 5.5 && 0x10 > 6 * 2`

j) `(double) (int) 2.5 > 2F && (7 < 15 / 2 || false)`

⑬ ☆☆☆ Für diese Aufgabe gelte eine andere Präzedenzordnung der bekannten Operatoren, wobei die Priorität eines Operators höher ist, je weiter oben er in der Tabelle steht. Operatoren derselben Prioritätsstufe werden von links nach rechts (d. h. linksassoziativ) ausgewertet.

Priorität	**Operator**(en)
1	`%`
2	`-`
3	`*`
4	`/ +`

Der `%`-Operator (Modulo bzw. Teilerrest) hat somit die höchste Bindungsstärke und wird daher als erstes ausgewertet.

Beispiel: `7 - 5 * 3` → `( ( 7 - 5 ) * 3 )`

Geben Sie für die folgenden Ausdrücke durch *vollständige* Klammerung an, wie diese ausgewertet werden (vgl. obiges Beispiel). Zeichnen Sie die Klammern entsprechend ein. Geben Sie außerdem an, wozu die Ausdrücke unter Annahme obiger Präzedenzordnung auswerten würden. Nennen Sie ggf. auftretende *Compiler-Fehler*.

	Ausdruck (Hier Klammern setzen!)	**Ergebnis** (oder Fehler?)
a)	`4 - 2 + 3`	
b)	`5 + 5 - 5 * 2`	
c)	`3 * 5 - 1 + 6`	
d)	`2 * 7 % 5 / 2`	
e)	`4 + "" + 3 * 2`	
f)	`1 + "2" + 6 / 2`	
g)	`4 - 3 / 2 + 1`	
h)	`"6" + 3 - 2 * 3`	
i)	`2 * 3 / 5 % 3`	
j)	`3 / 3 - 1 + 1`	
k)	`6d + 0x1 % 4 / 2`	
l)	`3d / 2 + ""`	
m)	`"" + .5 * 3 - 1`	
n)	`1e2 * 10 - 2 + ""`	
o)	`"" + 4 - 2 / 1`	
p)	`0x0A - 0b10 % 3`	

5. Anweisungen

Ausdrücke drücken etwas aus (Wert), Anweisungen steuern den Kontrollfluss. Die *Zuweisung* haben wir bereits im letzten Kapitel angesprochen, da sie auch ein Ausdruck ist (vgl. Kap. 4), wobei sie tatsächlich mehr Anweisung als Ausdruck ist, schließlich ist ihre Kernaufgabe die Wertzuweisung an eine Variable. Neben der Zuweisung gibt es u. a. folgende Anweisungen:

5.1. **Selektion** (bedingte Anweisung / *if*-Abfrage / Verzweigung)

```
if ( <boolescher Ausdruck> ) {
    <Statements 1. Fall>
} else if ( <boolescher Ausdruck> ) {
    <Statements 2. Fall>
} else if (<...>) {
    <...>
} else {
    <Statements sonst>
}
```

Auf den notwendigen `if`-Block *können* beliebig viele „`else if`“-Blöcke folgen, müssen aber nicht. Auch der `else`-Block ist *optional*. Die geschweiften Klammern können im Allgemeinen (d. h. auch bei Schleifen) immer dann weggelassen werden, wenn der entsprechende Block aus nur *exakt einem* Statement besteht (darf keine Deklaration sein). Einrückungen spielen keine Rolle.

Beim **ternären Operator** existieren nur `if` und `else`-Teil:

```
<boolescher Ausdruck> ? <Statements yes/if> : <Statements no/else>;
```

Wie wir bereits wissen ist der ternäre Operator jedoch keine Anweisung, sondern ein Ausdruck (vgl. Kap. 4), d. h. man kann nicht jedes *if-else* mit ternärem Operator schreiben. Man benutzt ihn häufig bei Rückgaben und Zuweisungen. Beispiele:

Kurzschreibweise mit tern. Op.	**Äquivalenter Code** mit *if*s

`int d = (a == 0 ? b : c);`	`int d;`
	`if (a == 0) {`
	`  d = b;`
	`} else {`
	`  d = c;`
	`}`

`System.out.println(z ? "x" : a);`	`if (z == true) {  // „== true“ ist unnötig`
	`  System.out.println("x");`
	`} else {`
	`  System.out.println(a);`
	`}`

`return n == 0 ? a`
`  : n == 1 ? b : c;`
	`if (n == 0) {`
	`  return a;`
	`} else if (n == 1) {`
	`  return b;`
	`} else {`
	`  return c;`
	`}`

⑭ ☆☆☆ Ergänzen Sie alle geschweiften Klammern (= Blöcke) in folgendem Java-Konstrukt:

```
1   if (x > 0)
2       if (myBool)
3           if (z > 0)
4               return x;
5       else
6           return y;
7   else if (y < 0)
8       return y;
9   return z;
```

5.2. Mehrfachverzweigung (*switch*)

```
int result;
if (x == 0)
    result = 2;
else if (x == 1)
    return;
else if (x == 2 || x == 3)
    result = 5;
else
    result = 0;
```

```
switch (x) {
    case 0:
        result = 2; break;
    case 1:
        return;
    case 2:
    case 3:
        result = 5; break;
    default:
        result = 0;
}
```

Ganzzahl-/Stringvariable

Konstante

`switch`-Anweisung verlassen
(nach `return` kommt kein `break`!)

5.3. *while*-Iteration (Schleife mit Vorabprüfung)

```
while ( <boolescher Ausdruck> ) {
    <Statements>
}
```

5.4. *do-while*-Iteration (Schleife mit Nachprüfung)

```
do {
    <Statements>
} while ( <boolescher Ausdruck> );
```

Diese Schleife kann hilfreich sein (anstelle der „normalen“ *while*-Schleife), wenn man etwas mindestens einmal ausführen möchte bzw. erst nach der ersten Ausführung weiß, ob man die Schleife noch einmal ausführen muss. Jede *do-while*-Schleife kann problemlos als *while*-Schleife geschrieben werden, sie ist nur manchmal (für uns: selten) schöner.

5.5. *for*-Statement (*for*-Schleife)

```
for ( <Initialisierung> ; <Bedingung> ; <Modifikation> ) {
    <Statements>
}
```

Wie die *do-while*-Schleife kann auch jede *for*-Schleife als *while*-Schleife geschrieben werden. Die *for*-Schleife ist nur schöner, wenn z. B. über etwas (häufig: Array oder String) iteriert wird oder kontinuierlich gezählt (z. B. von `0` bis `n`) wird.

Die *for*-Schleife entspricht folgender *while*-Schleife:

```
<Initialisierung>;
while ( <Bedingung> ) {
    <Statements>
    <Modifikation>;
}
```

Die „**for-each**“**-Schleife** kann bei „iterierbaren“ Objekten verwendet werden und zählt „im Hintergrund“, so dass wir direkt das nächste Element der Kollektion erhalten. Derartige Objekte sind bspw. Arrays, Listen und Sets, insb. also *Collections*, da diese das Interface `Iterable` implementieren und somit einen *Iterator* besitzen. Mehr zu Iteratoren gibt es in Kapitel 16, aber im Grunde werden durch eine *for-each*-Schleife einfach automatisch im Wechsel die Iterator-Methoden `hasNext()` und `next()` aufgerufen.

```
for ( <Variablendeklaration> : <Kollektion> ) {
    <Statements>
}
```

Beispiel:

```
String[] arr = {"a", "B", "3"};
for (String s : arr)   // s ist erst "a", dann "B" und dann "3"
    System.out.println(s);
```

5.6. `break`

Mit dem Schlüsselwort `break` wird eine Schleife (*while*/*for*/*do*) oder eine Mehrfachverzweigung (*switch*) verlassen. Bei verschachtelten Schleifen wird durch ein „`break;`“-Statement jedoch nur die innerste Schleife verlassen! Um äußere Schleifen mittels `break` zu verlassen, kann man ***Labels*** nutzen und `break labelName;` schreiben. Beispiel:

```
fridolin:
while (x < 10) {
    for (; y < 100; y++) {
        y *= x;
        if (y % 3 == 0)
            break fridolin;
        x *= 2;
    }
    x++;
}
```

Diese Schleife heißt *fridolin*.

Wenn `y` durch 3 teilbar ist wird die Schleife *fridolin* verlassen (egal ob `x < 10` gilt oder nicht). `x *= 2` und `x++;` werden dann nicht mehr ausgeführt.

5.7. `continue`

Mit dem Schlüsselwort `continue` wird der aktuelle Schleifendurchlauf wie bei `break` abgebrochen, die Schleife dann allerdings bei der Bedingung bzw. mit dem nächsten Element (bei *for-each*) fortgesetzt. `continue` kann nur in Schleifen und somit nicht in einem *switch* verwendet werden.

5.8. **Rückgabeanweisung** (`return`)

Mit der *return*-Anweisung verlassen wir eine Funktion/Methode.

- Eine `void`-Methode kann über leere „`return;`“-Statements verfügen, muss aber nicht. Anderenfalls wird sie verlassen, wenn man am Ende angekommen ist.
- Alle anderen Methoden müssen *in jedem Fall* in einem `return ...;` enden.
- Eine Methode kann mehrere *return*-Statements enthalten, allerdings wird stets nur das zuerst erreichte ausgeführt. Dies impliziert, dass nach einer *return*-Anweisung (bspw. in der nächsten Zeile) kein anderes Statement mehr stehen darf.
- Die Rückgabeanweisung ist stets das letzte, was eine Methode ausführt. Steht `return` in einem `try`- bzw. `catch`-Block, so wird zuerst noch der `finally`-Block ausgeführt.

5.9. ***try-catch-*** und ***throw-***Statement benutzen wir für Exceptions (→ Kapitel 14)

5.10. **`synchronized`** ist zugleich Statement und Modifier (→ Kapitel 6.6)

Übungsaufgaben zu Anweisungen: Kontrollflussdiagramme (Kapitel 10)

⑮ ★☆☆ **Programmverständnis**:

Geben Sie jeweils an, zu welchen Werten die Aufrufe *f(2)* und *f(5)* auswerten. Geben Sie insb. an, wenn ein Aufruf in einer Endlosschleife (→ terminiert nicht) oder Endlosrekursion („Stack Overflow“) endet. Schreiben Sie Strings in Anführungszeichen.

		f(2)	f(5)
a)	`static int f(int x) { int a = 0; while (x > 0) { a += x * x; x /= 2; } return a; }`		
b)	`public static int f(int x) { if ((x -= 4) <= 0) return x; return f(x--); }`		
c)	`static int f(int x) { int c = x; for (int i = 1; i < x; i *= 2) --c; return c; }`		
d)	`static String f(final int x) { if (x > 1) return "x" + f(x-1); return "x"; }`		
e)	`static int f(int x) { int e = x--; for (; x > 1; --x) e %= x/2; return (e == 0) ? f(--x) : e; }`		
f)	`public static int f(int x) { int f = x; while (-f <= 0) f -= x++/2; return f; }`		
g)	`static String f(double f) { do { f /= 2; } while(f > 2); return 1 + f + ""; }`		

h)
```
static boolean f(int x) {
    boolean h = false;
    switch (x) {
        case '5': break;
        case 2: h = f((short)(x+1));
        default: h = !h;
    }
    return h;
}
```

i)
```
static String f(int x) {
    int i = 0;
    String i_ = "";
    while (x > -2) {
        i_ += x;
        x -= 2 * i++;
    }
    return i_;
}
```

j)
```
private static int f(int x) {
    class F {
        int f(int x, int g) {
            if (g == 0)
                return x;
            return f(x, x % g);
        }
    }
    return new F().f(++x, x+3);
}
```

k)
```
static int f(int x) {
    return (x<-2 || x++>2) ? f(-x) : x;
}
```

l)
```
static int f(int x) {
    for (int y = 1; y < x; y++) {
        x = x-y;
        if (x >= 2)
            continue;
        x -= 2;
    }
    return x;
}
```

m)
```
public static int f(int x) {
    outer: while (true)
        inner: while (x < 7)
            if (++x == 5)
                break outer;
    return x;
}
```

6. MODIFIER

Dieses Kapitel stellt alle für uns relevanten Modifikatoren vor (`volatile`, `transient`, `native` und `strictfp` werden nicht behandelt), wenngleich wir diese erst später benötigen und vollständig verstehen werden. Erinnere dich dann an dieses Kapitel!

6.1. Zugriffsmodifikatoren: Jede Klasse, Membervariable (nicht bei lokalen Variablen!) und Methode besitzt eine bestimmte Sichtbarkeit. Wird kein Sichtbarkeitsmodifier benutzt, nennt man die Sichtbarkeit *package-private* oder *default* (das sind keine Modifier).

- **`public`** : Auf das Element kann von überall aus zugegriffen werden. Pro Datei darf maximal eine *öffentliche* Klasse existieren (Name = Dateiname).
- **`protected`** : Auf das Element kann von allen Klassen zugegriffen werden, die sich im selben Package befinden oder von der Klasse, in der dieses Element definiert wurde, erben. Klassen können nur `protected` markiert werden, wenn sie in einer anderen Klasse definiert werden.
- *package-private*: Auf das Element kann von allen Klassen zugegriffen werden, die sich im selben Package befinden. In einer Datei können beliebig viele derartige *top-level* Klassen existieren.
- **`private`** : Auf das Element kann nur von innerhalb der Klasse, in der es definiert wurde, zugegriffen werden. Private Klassen müssen innere Klassen sein (in einer anderen Klasse definiert).

Zusammenfassung (Zugriffsübersicht):

	Überall	Subklasse	Package	Klasse
`public`	☑	☑	☑	☑
`protected`	☐	☑	☑	☑
default	☐	☐	☑	☑
`private`	☐	☐	☐	☑

6.2. Speichermodifikator (**`static`**): Alle in einer Klasse definierten Membervariable und Methoden sind grundsätzlich an ein Objekt der umliegenden Klasse gebunden. Mit **`static`** markierte Member sind frei von dieser Bindung, existieren also unabhängig von Instanzen der Klasse. Auch innere Klassen können statisch sein. Sie sind dann ebenfalls nicht mehr an ein Objekt der umliegenden Klasse gebunden.

Eine nicht-statische Membervariable heißt *Objektvariable/Attribut*. Eine statische Variable bzw. Methode nennt man auch *Klassenvariable* bzw. *Klassenmethode/Funktion*. Eine Klassenvariable existiert also nur ein einziges Mal; für alle Objekte gemeinsam (*shared variable*).

Klassenvariablen/-methoden gehören somit zur Klasse, nicht zu einem Objekt. Daher kann man aus einem statischen Kontext (z. B. einer Funktion) bspw. nicht auf Attribute zugreifen, schließlich ist ein Attribut an ein Objekt gebunden. Aus einem objektbezogenen Kontext (bspw. einer nicht-statischen Methode) kann man hingegen bspw. sowohl auf Attribute als auch auf statische Member zugreifen.

6.3. Veränderbarkeitsmodifikator (`final`): Finale Variablen sind nach erfolgter Initialisierung (erstmalige Wertzuweisung) unveränderlich. Globale Konstanten werden häufig mit `static final` markiert. Auch lokale Variablen oder Parameter können `final` sein. Natürlich muss eine Variable nicht unbedingt `final` sein um nicht verändert zu werden, schließlich kann man auch einfach sicherstellen, dass eine Variable an keiner Stelle verändert wird. Das Schlüsselwort `final` hilft uns aber dabei, dass wir eine Variable, die wir eigentlich nicht verändern wollten, nicht doch irgendwo verändern. Mit `final` können wir uns also bewusst selbst einschränken.

Mehr dazu: Besonders sinnvoll ist die Verwendung dieses Schlüsselsworts bspw. wenn wir innerhalb einer anonyme Klasse (→ Kap. 19.1) auf Variablen der umgebenden Methode oder Klasse zugreifen wollen, da diese Variablen innerhalb nicht verändert werden dürfen.

Beachte: `final` bezieht sich auf die Veränderbarkeit der Variable, nicht des Objekts. Das Objekt, das eine finale Variable speichert, kann nach wie vor verändert werden (→ Setter).

Mit `final` markierte Methoden können in Unterklassen nicht überschrieben werden.
Von finalen Klassen kann man nicht erben (all ihre Methoden sind automatisch `final`).

6.4. Abstrakte Klassen und Methoden (`abstract`):

Eine *abstrakte Klasse* (`public abstract class MyClass`) kann nicht instanziiert werden, d. h. es gibt keine Objekte einer solchen Klasse. Außerdem dürfen abstrakte Klassen zusätzlich zu „normalen" Methoden auch abstrakte Methoden definieren.

Eine *abstrakte Methode* verfügt lediglich über einen Methodenkopf, ihr fehlt die Implementierung (Rumpf). Eine solche Methode ist also definiert aber nicht implementiert und kann deshalb auch nicht ausgeführt werden. Alle nicht-abstrakten Unterklassen einer abstrakten Klasse müssen dann alle abstrakten Methoden implementieren (d. h. nicht `abstract`).

*Ein **Interface*** ist keine Klasse, kann aber wie eine „ausschließlich" abstrakte Klasse beschrieben werden. Das heißt: Interfaces können ebenfalls abstrakte Methoden definieren, können im Gegensatz zu abstrakten Klassen jedoch keine normalen Methoden, Attribute oder Konstruktoren enthalten. Alle nicht-statischen Methoden eines Interfaces sind deshalb implizit `public abstract`. Interfaces können dennoch über statische Methoden und Konstanten verfügen (alle Variablen sind automatisch `static final`)

Jede Klasse erbt (`extends`) von genau einer (eventuell abstrakten) Klasse (→ Kap. 12), kann jedoch keines oder beliebig viele Interfaces implementieren (`implements`). Implementiert eine Klasse ein Interface, so wird versichert, dass die Klasse die vom Interface „geforderten" Methoden bereitstellt, also bestimmte Funktionalitäten anbietet. Interfaces werden daher oft mit Adjektiven benannt (bspw. `Iterable`, `Runnable`, `Comparable`).

6.5. `default`-Methoden:

Tatsächlich können seit Java-Version 8 auch Interfaces „normale" Methoden implementieren. Derartige Methoden müssen mit dem Schlüsselwort `default` (statt `abstract`) versehen werden und dienen dann als Standardimplementierung für Klassen, die das jeweilige Interface implementieren (mit `implements`). Derartige Methoden werden für uns nicht von großer Bedeutung sein.

6.6. **`synchronized`**: ***Synchronisierung*** ist nötig, wenn man das Problem des *kritischen Abschnitts* lösen möchte. Unter einem ***kritischen Abschnitt*** versteht man Programmabschnitte, die nicht von mehreren Threads (→ Kap. 21) gleichzeitig ausgeführt werden dürfen, um Probleme wie *Race Conditions* zu vermeiden. Eine ***Race Condition*** (auch *Hazard*) bezeichnet eine Situation, in der das Ergebnis der Ausführung eines Programmabschnitts vom zeitlichen Ablauf einzelner Operationen abhängt. Beispiel:

Angenommen zwei Threads *T1* und *T2* wollen nacheinander dieselbe Variable ($x = 1$) verändern – *T1* möchte 3 addieren, *T2* möchte 2 subtrahieren. Dann müssen diese jeweils zuerst den Wert von x lesen, um anschließend das berechnete Ergebnis in x zurückschreiben. Neben den beiden intuitiven Konstellationen *„T1 liest* $x = 1$*“, „T1 schreibt* $x = 4$*“, „T2 liest* $x = 4$*“, „T2 schreibt* $x = 2$*“* (also $1 + 3 - 2 = 2$) und *„T2 liest* $x = 1$*“, „T2 schreibt* $x = -1$*“ „T1 liest* $x = -1$*“, „T1 schreibt* $x = 2$*“* (also $1 - 2 + 3 = 2$), welche das gleiche Ergebnis liefern, sind ohne Synchronisation wegen des zufälligen* Schedulings auch andere Reihenfolgen wie *„T1 liest* $x = 1$*“, „T2 liest* $x = 1$*“, „T1 schreibt* $x = 4$*“, „T2 schreibt* $x = -1$*“* (also Ergebnis $x = -1$) möglich 😕 Wir sprechen von einer *Race Condition*. Der *kritische Abschnitt* ist hier also die Kombination aus „x lesen“ und „x überschreiben“. Um zu vermeiden, dass ein Thread die Variable x liest, bevor ein anderer Thread sie zurückgeschrieben hat, müssen wir sicherstellen, dass beide Operationen ununterbrechbar nacheinander ausgeführt werden. Dazu nutzen wir *Locks* (oder *Semaphore*).

Ein ***Lock*** (auch *Mutex*) ist ein simples Objekt, welches mit den Zuständen „frei“ oder „belegt“ markiert sein kann. Entsprechend kann man ein `Lock`-Objekt belegen (`myLock.acquire()`) und freigeben (`myLock.release()`). Das Interface `Lock` definiert diese Methoden – implementiert werden sie in konkreten Klassen wie z. B. `ReentrantLock`. In unserem Beispiel wäre der Ablauf also „Lock belegen“, „x lesen“, „x schreiben“, „Lock freigeben“. Beim Aufruf von `acquire()` wartet ein Thread solange, bis das Lock frei ist (alternativ `myLock.tryAcquire()` mit Rückgabetyp `boolean`). Ein **`Semaphor`**(n) funktioniert wie ein Lock, erlaubt aber nicht nur einen sondern bis zu n gleichzeitige Zugriffe (in unserem Beispiel wäre das nicht sinnvoll). Praxisbeispiele: Lock = WC, Semphor = Parkhaus

Da Locks relativ häufig genutzt werden, wurde das `synchronized(...){...}`-Statement eingeführt. Vor dem Betreten des `synchronized`-Blocks (`{...}`) muss das in Klammern angegebene Lock belegt werden (ggf. wird darauf gewartet); beim Verlassen wird es wieder freigegeben. Tatsächlich kann in Klammern ein beliebiges Objekt angegeben werden, weil jedes Objekt automatisch über ein sog. *intrinsisches Lock* verfügt (das ist eine Art „Built-in“-Lock). Wir müssen nur sicherstellen, dass alle Threads über dasselbe Objekt synchronisieren. Aus diesem Grund nutzt man häufig „`static final Object`“-Variablen.

Steht `synchronized` wiederum im Methodenkopf (wie im Beispiel links), so ist das äquivalent zu einem `synchronized(this)`-Block (Sync. über dem aktuellen Objekt, Beispiel rechts) bzw. `synchronized(CurrentClassName.class)` bei statischen Methoden:

```
synchronized void method() {
    <Statements>
}
```

```
void method() {
    synchronized(this) {
        <Statements>
    }
}
```

* Zufall gibt es nicht. Der Scheduler arbeitet nach einer definierten Strategie. Für uns scheint es manchmal wie Zufall.

7. Referenzdatentypen

7.1. Arrays

(16) ★★☆ Sind folgende Aussagen korrekt?

		Wahr	Falsch
a)	Die Deklarationen `type[] name;` *oder* `type name[];` sind beide möglich und identisch.	☐	☐
b)	Folgendes Statement initialisiert ein Array der Größe 10: `int[] b = new int[9];`	☐	☐
c)	Egal an welcher Stelle ein Array deklariert wird (Membervariable, lokale oder statische Variable), es gilt immer: Alle Elemente werden immer *implizit* mit dem Wert `null` oder einem `null`-Äquivalent (`0`, `0.0`, `false`, ...) initialisiert.	☐	☐
d)	Wird ein Array als lokale Variable deklariert aber nicht initialisiert, so ist ihr Wert automatisch `null`.	☐	☐
e)	Die Länge eines Arrays kann über die *`length()`*-Methode ermittelt werden, d. h. bspw. `arr.length()`.	☐	☐
f)	Folgende Initialisierungen sind zulässig:		

```
int[] a = new int[5];                                          ☐ ☐
int[] b = new short[5];                                        ☐ ☐
char[] c = new int[5];                                         ☐ ☐
int[] d = new int[]{1, 2, 3, 4, 5};                            ☐ ☐
char[] e = {1, 2, 3};                                          ☐ ☐
boolean[] f = new boolean[]{true, false};                      ☐ ☐
int[] g = new int[2]{1, 2};                                    ☐ ☐
String[] h = {};                                               ☐ ☐
int[][] i = {{'a', 'b'}, 'c', 'd'};                            ☐ ☐
int[][] j = {{1, 2}, new int[2], {} };                         ☐ ☐
double[] k = new {1.0, 0.2};                                   ☐ ☐
double[] l = new double[]{'A'};                                ☐ ☐
double[] m = new double[]{1; 2; 3};                            ☐ ☐

int[] o = new int[3];                                          ☐ ☐
o = {1, 2, 3};

int[] p, q;                                                    ☐ ☐
p = q = new int[]{1, 'B', 3};

int r = {1, 2, '3'};                                           ☐ ☐
int[] s = new Integer[5];                                      ☐ ☐
Integer[] t = {1, 2, 3};                                       ☐ ☐
Integer[] u = new int[1];                                      ☐ ☐
Number[] v = new Integer[5];                                   ☐ ☐
Double[] x = {0.1, new Double(-1)};                            ☐ ☐
Double[] y = {1};                                              ☐ ☐
```

g)	Wird ein Array als Membervariable deklariert aber nicht initialisiert, so erfolgt in jedem Fall eine implizite Initialisierung mit dem Wert `null`.	☐	☐

h)	Sei ein Array `arr` vom Typ `char[]` gegeben, dann kann mit folgendem Schleifenkonstrukt jedes Element auf der Konsole ausgegeben werden:
	`for (int elem : arr)`
	`  System.out.println(elem);`
		☐	☐

i)	Sei ein Array `arr` vom Typ `int[]` gegeben, dann kann mit folgendem Schleifenkonstrukt jedes Element auf der Konsole ausgegeben werden:
	`for (char elem : arr)`
	`  System.out.println(elem);`
		☐	☐

j)	Sei `arr` ein Array vom Typ `boolean[]`, dann weist folgendes Konstrukt allen Elementen (Positionen) des Arrays den Wert **`true`** zu:
	`for (boolean e : arr)`
	`  e = true;`
		☐	☐

k)	Es gibt eine Möglichkeit, ein Array mit folgendem Inhalt zu erzeugen: `{"Hello", 1, false}`	☐	☐

l)	Sei `arr` ein Array vom Typ `Integer[]`, dann weist folgendes Konstrukt allen Elementen (Positionen) des Arrays ein Integer-Objekt mit dem Wert 1 zu:
	`for (Integer elem : arr)`
	`  elem = new Integer(1);`
		☐	☐

(17) ★☆☆ Bestimmen Sie jeweils das Element an der genannten Position (Zugriff) in dem gegebenen Array. Existiert das Element nicht, so kennzeichnen Sie dies entsprechend (Querstrich).

		Zugriff	Inhalt
	`x = {1, 'A', 3}`	`x[1]`	`'A'`
a)	`a = {{}, null, {1, 2}}`	`a[2][1]`	
b)	`b = {{1, 2, 3}, {4, 5, 6}, {7, 8}}`	`b[1][1]`	
c)	`c = {{1, 2, 0}, {4}, {7, 8}}`	`c[1]`	
d)	`d = {{'A', 'C', 'D', 5}, {}, {1}}`	`d[0]`	
e)	`e = {{{{true, false}}}, {}}`	`e[0][0]`	
f)	`f = {{1, 2}, {3, 4, 5}, {6}}`	`f[2][1]`	
g)	`g = {{3, 2}, {0, 1}, {-1, -2}}`	`g[g[1][0]]`	

⑱
★★☆

Ein Temperatursensor misst die Temperatur zweimal pro Tag (einmal morgens und einmal mittags) und speichert sie in einem `double`-Array `t`. Die erste Messung erfolgt morgens und das Array enthält mindestens zwei Messwerte (keine Fehlerbehandlung nötig). Das Array könnte somit bspw. wie folgt gegeben sein (Das ist nur ein Beispiel):

Tag:	1		2		3		4
Index in `t`:	`[0]`	`[1]`	`[2]`	`[3]`	`[4]`	`[5]`	`[6]`
Array `t`:	`1.2`	`7.7`	`3.1`	`10.3`	`-2.4`	`8.1`	`0.9`

a) Implementieren Sie eine Methode zur Ermittlung der Durchschnittstemperatur im gesamten Messzeitraum (im obigen Beispiel wäre das ca. `4.129`):

```
public static double avgTemp(double[] t) {




}
```

b) Implementieren Sie eine Methode zur Ermittlung der niedrigsten, mittags gemessenen Temperatur (im obigen Beispiel wäre das also `7.7`):

```
public static double minTempNoon(double[] t) {




}
```

c) Mittels `insert` soll ein neuer Messwert `value` hinten in das Array `t` eingefügt werden können. Da die Größe eines Arrays bekanntlich unveränderlich ist, soll dafür ein neues Array erzeugt und korrekt befüllt zurückgegeben werden.

```
public static double[] insert(double[] t, double value) {




}
```

d) Schreiben Sie eine Methode, um den Tag mit der größten Temperaturschwankung zu bestimmen. Mit obigem Beispiel müsste die Methode somit den Wert 3 zurückgeben, da der Temperaturunterschied zwischen dem Messwert morgens und dem Messwert mittags am dritten Tag `10.5` beträgt, wohingegen er an Tag 1 bei `6.5` und an Tag 2 bei `7.2` liegt.

Sollte der Messwert des letzten Tages fehlen (wie oben), so soll dieser Tag ignoriert werden. Gibt es mehrere Tage mit der maximalen Temperaturschwankung, so soll der letzte Tag zurückgegeben werden, an dem dieses Maximum erreicht wurde.

Beachten Sie, dass die Temperatur nachts höher sein könnte als mittags.

```
public static int dayWithHighestVariation(double[] t) {




}
```

⑲ ★★☆ **Call by Value**: Gegeben seien die folgenden Methoden (links). Welche Konsolenausgabe wird durch die sequentielle Ausführung der Statements (rechts) generiert?

```
static int[] set1(int[] a) {
    int[] b = {1};
    a = b;
    set3(a);
    return b;
}
static void set5(int[] a) {
    a = new int[] {5};
}
static void set3(int[] a) {
    a[0] = 3;
}
static void print(int[] a) {
    System.out.println(a[0]);
}
```

```
int[] a = new int[] {2};
print(a);     // Ausgabe:

set3(a);
print(a);     // Ausgabe:

set5(a);
print(a);     // Ausgabe:

a[0] = 4;
print(a);     // Ausgabe:

set1(a);
print(a);     // Ausgabe:

a = new int[2];
print(a);     // Ausgabe:

a = set1(a);
print(a);     // Ausgabe:
```

⑳ ★★★

Implementieren Sie die Methode `int[] removeDuplicates(int[] arr)`, welche ein neues Array zurückgibt, das alle Elemente aus `arr` in gleicher Reihenfolge jedoch ohne Duplikate enthält. Mehrfach in `arr` enthaltene Werte sollen im Ergebnisarray also nur noch einfach enthalten sein, wobei für die Reihenfolge das jeweils erste Vorkommen einer Zahl ausschlaggebend ist. `arr` darf dabei nicht verändert werden.

Beispiel: $removeDuplicates([4,\ 2,\ 1,\ 2,\ 2,\ 3,\ 1,\ 5,\ 4]) = [4,\ 2,\ 1,\ 3,\ 5]$

```
public static int[] removeDuplicates(int[] arr) {




}
```

㉑ ★★☆

Schreiben Sie eine Methode `int[] extractMultiplesOf(int[] array, int num)`, welche ein Array zurückgibt, das alle Elemente aus `array` enthält, die Vielfache von `num` sind (Tipp: Modulo-Operator). Das Ergebnisarray soll außerdem frei von Duplikaten sein. Der Parameter `array` darf nicht verändert werden.

Beispiel: $extractMultiplesOf([2,\ 6,\ 12,\ 8,\ 0,\ 1,\ 6,\ 6,\ 5,\ -9,\ 1],\ 3) = [6,\ 12,\ 0,\ -9]$

```
public static int[] extractMultiplesOf(int[] array, int num) {




}
```

(22) ★★★

Implementieren Sie die Methode `void reverse(Object[] a)`, welche das Array `a` *in-place* (also ohne ein neues Array zu erzeugen!) invertiert, sodass das Array `a` nach dem Aufruf `reverse(a)` die Elemente in umgekehrter Reihenfolge enthält.

Beispiel: $a = [5,\ -7,\ 0,\ 3] \xrightarrow{reverse(a)} a = [3,\ 0,\ -7,\ 5]$

```
static void reverse(Object[] a) {




}
```

(23) ★★☆

Schreiben Sie eine Methode `long[] powerOfX(int x, byte n)`, welche alle Potenzen x^i für $i \in [0, n]$ (0 und n inklusive) berechnet und in einem Array speichert, sodass für jedes Element (jeden Index `i`) des Rückgabearrays gilt:

$$result[i] = x^i$$

```
public static long[] powerOfX(int x, byte n) {




}
```

(24) ★★☆

Vervollständigen Sie die folgende Klasse, so dass die Methoden wie folgt arbeiten:

- *count*`(int[] numbers, int element)` gibt zurück, wie oft das Element `element` im Array `numbers` vorkommt.
- *indexOf*`(Object[] arr, Object elem)` gibt den Index des Elements `elem` im Array `arr` zurück. Kommt das Element mehrfach im Array vor, so wird der Index des letzten Vorkommens zurückgegeben; kommt es nicht vor, so wird -1 zurückgegeben.
- *remove*`(int[] arr, int elem)` gibt ein Array zurück, welches alle Elemente des Arrays `arr` außer `elem` enthält. Dazu werden alle Vorkommen von `elem` entfernt.

Warum konnte man die geschweiften Klammern der Kontrollstrukturen hier weglassen?

```
public class Arrays {
    static int count(int[] numbers, int element) {
        [                              ];
        for (int i = 0; i < [                              ]; ++i)
            if ([                              ])
                counter++;
        return counter;
    }

    static int indexOf(Object[] arr, Object elem) {
        int i = [                    ];
        while (i-- > 0)
            if (arr[i] == elem)
                [                    ];
        [                          ];
    }

    static int[] remove(int[] arr, int elem) {
        // Anzahl der Vorkommen des Elements im Array
        int count = count([                              ]);
        int[] result = new int[[                              ]];
        int b = result.length; // Zähler für das Ergebnisarray

        // Alle Elemente (außer elem) in Ergebnisarray kopieren:
        for (int a = [                    ]; a >= 0; a--)
            if (arr[a] != elem)
                [                              ];
        return result;
    }
}
```

㉕ ★★★ Schreiben Sie eine Methode, die *beliebig viele verschiedene* Zahlen vom Benutzer einliest und anschließend die Summe dieser Zahlen ausgibt. Die Summe der bisher eingegebenen Zahlen soll mithilfe von `System.out.println(...)` ausgegeben werden, sobald die Benutzerin zum ersten Mal eine Null (0) eingibt. Das Programm wird anschließend beendet.

Achten Sie darauf, bereits eingegebene Zahlen nicht mehrfach zu addieren. Sie können ein Array benutzen, um Zahlen zwischenzuspeichern.

Beispiel: Bei der Eingabefolge 3, 1, 3, 2, 0 wäre die Ausgabe 6.

Hinweis: Für das Einlesen einer Zahl vom Benutzer stehe Ihnen die Methode `public static int read()` zur Verfügung.

Hinweis: Durch die Speicherung mittels Array kann es passieren, dass der Nutzer mehr Zahlen eingibt, als gespeichert werden können (Speicher voll). Diesen Fall müssen Sie *nicht* behandeln!

```
public static void userSum() {









}
```

7.2. Strings

Wichtige Objektmethoden für String-Objekte:

- `char charAt(int index)`: Gibt das Zeichen an der Stelle `index` zurück.
- `int length()`: Gibt die Länge des Strings zurück.
- `boolean equals(Object anObject)`: Gibt `true` zurück, falls `anObject` ein String ist, der *zeichenweise* gleich zu diesem String ist.
- `String substring(int beginIndex)`: *bzw.* `String substring(int beginIndex, int endIndex)` — Gibt den Teilstring beginnend bei `beginIndex` (bis Ende) zurück. Wird zusätzlich ein zweiter Parameter übergeben, so ist der Teilstring bis zum Index `endIndex-1` beschränkt.
- `boolean startsWith(String prefix)`: Gibt zurück, ob der String mit `prefix` beginnt.
- `boolean endsWith(String suffix)`: Gibt zurück, ob der String mit `suffix` endet.
- `String replace(char oldChar, char newChar)`: Ersetzt jedes Zeichen `oldChar` durch `newChar` und gibt das Ergebnis zurück.
  geht auch mit zwei `Strings` als Parameter, z. B.: `"aba".replace("a", "xx")` gibt `"xxbxx"` zurück.
- `String[] split(String regex)`: Trennt den String an den Stellen `regex` in ein Array, z. B. `"axbcxd".split("x")` = `{"a","bc","d"}`
- `int indexOf(int ch)`: *bzw.* `int indexOf(String str)` — Gibt den Index des ersten Vorkommens des Zeichens `ch` (bzw. Strings `str`) im String zurück (`-1`, falls nicht enthalten). Es kann ein zweiter Parameter `int fromIndex` übergeben werden, um eine Start-Suchposition zu spezifizieren.
  analog dazu gibt es `lastIndexOf(...)`
- `char[] toCharArray()`: Gibt ein Array zurück, das alle Zeichen des Strings unverändert enthält.
- `String toUpperCase()`: *bzw.* `toLowerCase()`: Gibt einen String zurück, der dem String in Groß- bzw. Kleinschreibung entspricht.

Es gibt noch weitere nützliche Methoden (`strip`, `matches`, `join`, ...), im Hinblick auf Streams (→ Kap. 19) insb. `chars()` und `lines()`. Beachte stets, dass Strings im Gegensatz zu Arrays *unveränderlich* sind, d. h. einzelne Zeichen kann man nachträglich nicht ändern. Man überschreibt eine Variable stattdessen immer mit einem *neuen* String.

Beispiel:

```
String s = "rot";       // Variable s speichert einen neuen String "rot"
s += "blau";            // Variable s speichert neuen String "rotblau"
s.toUpperCase();        // Variable s bleibt unverändert.
s = s.toUpperCase();    // Variable s speichert neuen String "ROTBLAU"
```

Beachte außerdem, dass wir Strings (genau wie alle anderen Objekte außer Arrays) i. d. R. mit der `equals`-Methode vergleichen wollen, denn `==` prüft auf Referenzgleichheit (vgl. Seite 86).

(26) ★☆☆ Kompilieren folgende Statements? Begründen Sie *kurz*, falls nicht.

- ☐ `String a = "Hello World";`
- ☐ `String b = 5 + "5";`
- ☐ `String c = '5' + 5;`
- ☐ `String d = null;`
- ☐ `String e = "" + -1 * 5;`
- ☐ `String f = null + " World";`
- ☐ `String g = new String("unknown");`
- ☐ `String h = 'null';`
- ☐ `String i = (String) 'Test';`
- ☐ `String j = '5' + 5.0 + "5" + 5;`
- ☐ `String k = 'J';`
- ☐ `String l = (String) 'K';`
- ☐ `String m = new String(5);`
- ☐ `String n = new String();`
- ☐ `String o = new String("Hello") + new String("World");`
- ☐ `String p = "Beispiel" * 3;`
- ☐ `String[] q = new String[2];`
- ☐ `String r[] = null;`
- ☐ `String[] s = new String[]{1, 2, 3};`
- ☐ `String t = new Integer(5).toString();`
- ☐ `class U extends String { }`
- ☐ `String v = (String) "V";`
- ☐ `Object[] w = new Object[]{"A", "B", "A"};`
- ☐ `Object x = "XxX";`
- ☐ `String y = "Y" + new Object();`
- ☐ `String z = new Object();`
- ☐ `Object[] A = {"A".toString(), "B"};`
- ☐ `boolean B = 'A' == "B";`
- ☐ `String C = (String) new Object();`
- ☐ `String[] D = (String[]) new Object[17];`
- ☐ `boolean E = new String("x") == new String("x");`
- ☐ `boolean F = "x" == "x";`
- ☐ `String G = String.valueOf("rot".toUpperCase() == "ROT");`

(27) ★☆☆ Implementieren Sie eine Methode, die die Anzahl der Vokalbuchstaben (a, e, i, o, u, y) in einem Text zurückgibt. Umlaute (ä, ö, ü) sollen nicht beachtet werden, allerdings können die Vokale sowohl groß- als auch kleingeschrieben sein.

Beispiel: Der Text `"Annabell spielt Xylophon."` enthält 8 Vokalbuchstaben.

```
public static int countVowels(String text) {




}
```

(28) ★★☆ Schreiben Sie eine Methode `String invert(String s)`, welche den als Parameter übergebenen String `s` umgekehrt zurückgibt. Der Aufruf `invert("Fridolin")` sollte also bspw. zu `"nilodirF"` auswerten.

```
static String invert(String s) {




}
```

(29) ★★☆ Vervollständigen Sie folgenden Code, so dass die Methode *filterAlphaToUpper* den als Parameter übergebenen String `s` mit folgenden Veränderungen zurückgibt:

- Großbuchstaben (A-Z) bleiben erhalten
- Kleinbuchstaben (a-z) werden zu Großbuchstaben umgewandelt
- alle anderen Zeichen (z. B. Ziffern) werden jeweils durch einen Unterstrich ersetzt

Der resultierende String enthält somit ausschließlich Großbuchstaben und Unterstriche.

Beispiel: *filterAlphaToUpper*`("He1lo w0rLD!")` liefert `"HE_LO_W_RLD_"`.

Hinweis: Sie dürfen die Objektmethoden `charAt(...)` und `length()` aus der Klasse `String` verwenden.

```
private static String filterAlphaToUpper(String s) {
    String result = [          ]; // Ergebnisstring anlegen

    for (int i = 0; i < [          ]; i++) {
        // Aktuelles Zeichen zwischenspeichern
        char c = [          ];

        if (c >= 'a' && c <= 'z') {
            // Umwandlung von Klein- zu Großbuchstaben
            c -= 'a';
            c += [          ];
        } else if ([          ])
            c = '_';

        // Aktuelles (ggf. verändertes) Zeichen anhängen
        result = [          ];
    }

    return result;
}
```

Die `main`-Methode soll nun alle übergebenen Strings wie beschrieben verändern und ausgeben. Für die Ausgabe können Sie `System.out.println` verwenden.

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









}
```

(30) ★★★ Implementieren Sie die Klassenmethode `contains(String s, String sub)`, welche genau dann `true` zurückgibt, wenn der als Parameter übergebenen Text `s` den Text `sub` enthält. Anderenfalls soll die Funktion `false` zurückgeben.

Beispiele:

- `contains("abcde","cd")` gibt `true` zurück
- `contains("e","ef")` gibt `false` zurück

Hinweis: Sie dürfen die Methoden `charAt(…)` und `length()` der Klasse `String` verwenden. Abgesehen davon dürfen Sie keine Methoden verwenden.

```
public static boolean contains(String s, String sub) {




}
```

Erweitern Sie die `contains`-Methode nun so, dass der Punkt in `sub` eine besondere Funktion hat: Jeder in `sub` vorkommende Punkt stehe nun für beliebiges Zeichen.

Beispiele:

- `contains("abcd","a.c") == contains("abcd","b..") == true`
- `contains("abcd","a.d") == contains("abc","b..") == false`

Zusatzaufgabe (ohne Lösung): Erweitern Sie die `contains`-Methode anschließend so, dass auch das Fragezeichen in `sub` eine besondere Funktion bekommt: Jedes in `sub` vorkommende Fragezeichen bedeutet, dass das jeweils vorhergehende Zeichen optional ist, also in `s` vorkommen kann oder nicht. Sie dürfen davon ausgehen, dass ein Fragezeichen nicht auf einen Punkt oder ein anderes Fragezeichen folgt.

㉛ ★★☆ Befüllen Sie die Lücken in folgendem Code entsprechend der Kommentare sinnvoll:

```
/* Wandelt die als Parameter übergebene Hexadezimalzahl in ihren korrespondierenden
 * Dezimalwert um und gibt diesen zurück. In Fehlerfällen wird -1 zurückgegeben (z. B.
 * wenn der String hex, der die Hexadezimalzahl repräsentiert, nicht über das Präfix
 * 0x bzw. 0X verfügt oder ungültige Zeichen enthält). hex sei nicht null.
 */
public static long hexToDez(String hex) {

    hex = [                              ]; // kleinschreiben damit z. B. 0X3A == 0x3a

    // Fehlerprüfung: Hexadezimalzahl muss mit 0x beginnen, sonst ungültig:
    if ([                                        ])
        return -1;

    hex = hex.substring(2); // Hex-Präfix 0x entfernen

    // Führende Nullen zählen:
    int zerosIndex = 0;   // Anzahl führender Nullen
    while (zerosIndex < hex.length() && [                              ])
        zerosIndex++;

    // Wenn Hex-Zahl keine Ziffern ungleich 0 enthält, dann entspricht sie dezimal 0:
    if (zerosIndex == [                        ])
        return 0;

    // Führende Nullen entfernen:
    hex = [                                  ];

    // Fehlerprüfung: Mehr als 16 Hex-Ziffern sind nicht als long darstellbar.
    // Bei exakt 16 Ziffern muss die erste kleiner 8 sein, sonst nicht darstellbar.
    if (hex.length() > 16 || [                                  ])
        return -1;

    long multiplier = 1, result = 0; // Positionsfaktor, Ergebnis

    // Iteriere über alle Ziffern der Hex-Zahl, beginnend bei der niederwertigsten:
    for (int i = [                                  ]; i--) {
        int digit = hex.charAt(i); // nächste Hex-Ziffer

        // Konvertiere die Hex-Ziffer in ihren äquivalenten Dezimalwert:
        if (digit >= '0' && digit <= '9')
            digit -= '0';
        else if (digit >= 'a' && digit <= 'f')
            digit = [                          ];
        else
            return [                    ];

        // Modifiziere Zwischenergebnis; bereite Multiplikator für nächste Pos. vor:
        result += [                          ];
        multiplier *= 16;
    }

    return result;
}
```

(32) ★★☆ Implementieren Sie die Klassenmethode mit der Signatur `cutOut(String[],String)`. Die Methode bekommt ein `String`-Array und einen Suchstring als Parameter übergeben und soll in jedem Array-Element alle Vorkommen des Suchstrings entfernen. Der Parameter soll direkt modifiziert werden. Schlussendlich soll zurückgeben werden, wie viele Elemente des Arrays verändert wurden.

Der Aufruf *cutOut*(…,"*a*") würde bspw. den Buchstaben *a* aus allen im Array enthaltenen Strings entfernen und zurückgeben, wie viele Strings ursprünglich mind. ein *a* enthielten.

Hinweis: Aus der Java-Standardbibliothek dürfen Sie lediglich die Methoden `equals`, `length` und `charAt` der Klasse `String` verwenden.

Hinweis: Mögliche Duplikate und Uneindeutigkeiten müssen Sie nicht beachten.

Beispiel: *cutOut*(["*ab*", "*aabab*", "*acba*", "*abababb*", "*a*", "*baba*"], "*ab*") gibt 4 zurück und verändert das Array zu ["", "*a*", "*acba*", "", "*a*", "*ba*"].

7.3. Weitere Referenzdatentypen

Neben Arrays und `String`s existieren natürlich unendlich viele weitere Referenzdatentypen. Jede Klasse und jedes Interface, das wir entweder selbst schreiben (z. B. Klasse `Rechteck` aus Kapitel 1) oder schon in der Java-Biliothek definiert sind (z. B. `Object`, `Comparable`, Wrapper-Klassen wie `Integer`), definiert einen Referenzdatentyp. Arrays stellen gewissermaßen eine Art Sonderfall dar, da sie nicht in einer bestimmten Klasse implementiert sind.

(33) ★☆☆ In dieser Aufgabe sollen Sie einen **Zahlenbasisumwandler** und -rechner implementieren, der eine vorzeichenlose Zahl von der einen in eine andere Basis konvertiert, wie Sie es in Kapitel 3 bereits schriftlich getan haben.* Der Einfachheit wegen wollen wir die Umwandlung hier über einen Zwischenschritt durchführen, sodass die Zahl zunächst ins Zehnersystem und dann ins Stellenwertsystem der Zielbasis umgewandelt wird.

Das Alphabet gültiger Ziffern besteht aus den Zeichen `'0'` bis `'9'` und `'A'` bis `'Z'` (bzw. `'a'` bis `'z'`). Wird eine ungültige Start-/Zielbasis übergeben (z. B. 1 oder 67) oder enthält die Eingabe eine ungültige Ziffer (z. B. `'A'` für die Basis 10, `'7'` für die Basis 7 oder `'?'` für jede Basis), so soll eine informative `IllegalArgumentException` geworfen werden.

Implementieren Sie die folgenden Methoden:

(a) `private static char intToDigit(int d)`: Wandelt eine im Dezimalsystem dargestellte Ziffer (ausgedrückt als Zahl `d`) in das entsprechende Zeichen um.
Beispiele: 6 → `'6'`; 10 → `'A'`; 16 → `'G'`

(b) `private static String convertDecimalToBase(long number, int base)`: Stellt die Dezimalzahl `number` zur Basis `base` dar und gibt das Ergebnis zurück. Das Verfahren dazu ist, die Zahl solange mit Rest durch die Basis zu teilen, bis das ganzzahlige Ergebnis 0 ist. Der Divisionsrest ist jeweils die nächste Ziffer (beginnend bei der letzten). Beispieldarstellung von 363 zur Basis 16:

$$363 : 16 = 22, \text{ Rest: } 11 \rightarrow \text{Ziffer } \texttt{'B'}$$
$$22 : 16 = 1, \text{ Rest: } 6 \rightarrow \text{Ziffer } \texttt{'6'}$$
$$1 : 16 = 0, \text{ Rest: } 1 \rightarrow \text{Ziffer } \texttt{'1'}$$
$$\Longrightarrow 363 = 16B_{16} \rightarrow \text{Ergebnis } \texttt{"16B"}$$

(c) `private static int digitToInt(char c)`: Wandelt (gegensätzlich zu (a)) das Zeichen `c` in eine Zahl um, die das entsprechende Zeichen im Zehnersystem darstellt.
Beispiele: `'6'` → 6; `'a'` → 10; `'G'` → 16

(d) `private static long convertBaseToDecimal(String number, int base)`: Wandelt die Zahl `number`, welche zur Basis `base` dargestellt ist, in einen Integer (also ins Zehnersystem) um.
Beispielumrechnungen sehen so aus (Ergebnis rechts):

- $10101_2 = 1 * 2^4 + 0 * 2^3 + 1 * 2^2 + 0 * 2^1 + 1 * 2^0 = 21$
- $A6C_{16} = 10 * 16^2 + 6 * 16^1 + 12 * 16^0 = 2748$
- $t3sT_{30} = 29 * 30^3 + 3 * 30^2 + 28 * 30^1 + 29 * 30^0 = 786569$

Für mehr Effizienz wollen wir die Potenz nicht für jeden Summand berechnen, sondern das Horner-Schema verwenden (Potenzberechnungen ausklammern). Beispiel:

$$3A6C_{16} = 3 * 16^3 + 10 * 16^2 + 6 * 16 + 12 =$$
$$= (3 * 16^2 + 10 * 16^1 + 6) * 16 + 12 =$$
$$= \big((3 * 16 + 10) * 16 + 6\big) * 16 + 12$$

(e) `public static String convert(String number, int baseA, int baseB)`: Stellt die zur Basis `baseA` angegebene Zahl `number` zur Basis `baseB` dar und gibt das Ergebnis zurück. Nutzen Sie dazu die Methoden aus (b) und (d).

(f) `public static String add(String[] numbers, int base)`: Addiert alle im Array enthaltenen Zahlen (angegeben zur Basis `base`) und gibt das Ergebnis dargestellt zur selben Basis zurück (analog für `sub`, `mul`, `div`). Nutzen Sie (b) und (d).

* Mathematische Grundlagen: *http://www.arndt-bruenner.de/mathe/scripts/Zahlensysteme.htm*

8. KONTEXTFREIE GRAMMATIKEN & REGULÄRE AUSDRÜCKE

Reguläre Ausdrücke (engl. *regular expression*, kurz: „regex“) werden durch reguläre Grammatiken – das sind eingeschränkte *kontextfreie Grammatiken* – erzeugt. Die Einschränkung ist im Speziellen, dass reguläre Ausdrücke eigentlich tatsächlich Ausdrücke sind und somit in einer Zeile, ohne Definition bzw. Verwendung jeglicher Nichtterminale, geschrieben werden (oder zumindest geschrieben werden können).

Beispiel: Die in erweiterter Backus-Naur-Form (EBNF) angegebene kontextfreie Grammatik rechts lässt sich als regulärer Ausdruck wie folgt schreiben:

```
pdigit ::= 1 | ... | 9
digit ::= 0 | pdigit
number ::= -? pdigit digit* | 0
```

```
-? (1 | ... | 9) (0 | 1 | ... | 9)* | 0
```

Gegenbeispiel: `word ::= (a word b)*` lässt sich nicht einfach ohne Nichtterminale (`word`) darstellen und ist daher zwar eine kontextfreie Grammatik, aber kein regulärer Ausdruck.

Relevante Operatoren: **Bindungsstärke:**

a	Literal/Terminalzeichen (*a*)	
`x?`	**Option** (keinmal oder einmal)	stark
`x*`	**Iteration** (keinmal, einmal oder beliebig oft)	
`x+`	Iteration (mindestens einmal)	
`x y`	**Konkatenation** (Verknüpfung)	mittel
`x | y`	**Alternative** (entweder x oder y)	schwach
a `|` ... `|` *z*	Alternativen (eines der Zeichen von *a* bis *z*)	

(34) ☆☆☆ Vermeiden Sie im Folgenden Redundanz. Geben Sie eine Grammatik an für ...

a) ... bestimmte Benutzernamen. Diese setzen sich aus zwei durch einen Unterstrich getrennten Teilen zusammen. Beide Teile bestehen aus einer beliebigen Anzahl an Buchstaben (mind. ein Zeichen), wobei auch Punkte vorkommen dürfen, sofern vor und nach jedem Punkt mindestens ein Buchstabe folgt. Der Benutzername muss mit einem Großbuchstaben beginnen.

b) ... die Angabe von Geldbeträgen in Euro. Eine solche Preisangabe besteht aus einem Teil vor dem Komma (eine mindestens einstellige Zahl ohne führende Nullen) und einem Nachkommateil (eine genau zweistellige Zahl). Nach dieser Kombination (bspw. *1,99*) folgt entweder ein €-Zeichen oder „*Euro*“. Alternativ kann dem Geldbetrag auch „*EUR*“ vorangestellt sein.

   Beispiele: *0,99 Euro* oder *100,01 €* oder *EUR 1024,20*

c) ... Datumsangaben, wobei ein Datum aus einer zweistelligen Tag- (*01-31*) und einer zweistelligen Monatsangabe (*01-12*) besteht, jeweils gefolgt von einem Punkt (.) (zum Beispiel *30.03.*). Im Anschluss kann noch eine zwei- oder vierstellige Jahresangabe folgen (z. B. *01.01.00* oder *01.12.2017*).

d) ... Uhrzeiten im 24-Stunden-Format, wobei eine Uhrzeit aus einer ein- oder zweistelligen Stundenzahl gefolgt von einem Trennzeichen (Punkt oder Doppelpunkt) und einer zweistelligen Minutenzahl besteht (z. B. *16:00*, *04:59*, *0:00* oder *5.43*). Optional kann „*Uhr*“ angehängt werden (z. B. *16:00 Uhr*). Alternativ können Uhrzeiten nur mit der Stundenzahl (ohne führende Null) gefolgt von „*Uhr*“ angegeben werden (z. B. *7 Uhr*).

㉟ ★☆☆ Verwenden Sie für folgende Teilaufgaben abgesehen von `letter ::=` *a*|...|*z* keine Definition. Nennen Sie einen regulären Ausdruck für Textmuster, ...

a) die weder *a* noch *z* enthalten:

b) die eine gerade Anzahl (möglicherweise auch 0) an Buchstaben enthalten:

c) die nicht mit *a* beginnen und entweder mit *b* oder mit *a* enden:

d) die mit *a* enden, sofern sie auch mit *a* beginnen:

e) die genau zwei oder genau vier Zeichen lang sind:

f) die genau zwei *a*'s enthalten, wobei diese aufeinander folgen:

g) die mit beliebig vielen *a*'s beginnen dürfen, sonst aber nicht zwei oder mehr aufeinanderfolgende *a*'s enthalten:

9. SYNTAXBÄUME

Mit *Syntaxbäumen* wird Code syntaktisch in Einheiten gegliedert und so baumartig dargestellt. Die Grammatik ist vorgegeben und kann beispielsweise der nachfolgenden entsprechen, welche (weit) nicht alle Möglichkeiten abdeckt, aber für „MiniJava“ genügt. Deklarationen können laut dieser Grammatik bspw. nur ganz am Anfang eines jeden Codes stehen. Außerdem werden keine Auswertungsreihenfolgen (d. h. Bindungsstärken) definiert, was manchmal mehrere Darstellungsmöglichkeiten zulässt. Die Grammatik wäre in der Klausur ziemlich sicher gegeben und kann von der nachfolgenden Grammatik abweichen.

Vorgehen (grobe Erklärung): Das oberste Statement ist immer program – hier starten wir. Wir betrachten nun den gesamten Quelltext und gliedern ihn in Deklarationen und Statements, um den program-Knoten entsprechend der Grammatik in Deklarationen (decl) und Statements (stmt) zu unterteilen. Jeder <Nichtterminal>-Knoten wird nun einzeln betrachtet und weiter unterteilt, bis alle Terminalsymbole (also Code-Stücke, z. B. write) zugeordnet wurden. Man sollte durch Übung ein Gefühl dafür entwickeln, was z. B. ein stmt ist.

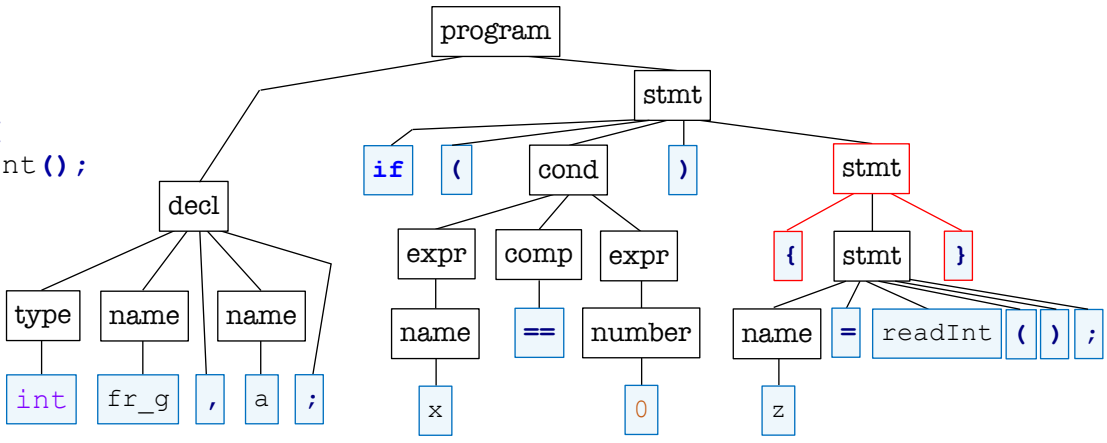
In der Grammatik (vgl. nächste Seite) werden Nichtterminale durch spitze Klammern gekennzeichnet (z. B. <expr>). Ein „|“ steht für „Oder“, d. h. wir können expr bspw. zu number oder zu name auswerten. Ein Stern (<something>*) bedeutet, dass das vorhergehende Nichtterminal beliebig oft (keinmal, einmal, zweimal, ...) vorkommen kann. Ein Fragezeichen (<something>?) kennzeichnet Optionalität (something kann keinmal oder genau einmal vorkommen).

Häufige Fragen/Antworten:

- Ebenen (vertikale Anordnung) spielen keine Rolle.
- Die Reihenfolge (links → rechts) ist hingegen relevant (Kanten dürfen nicht überkreuzen).
- Terminalsymbole können zusammengefasst werden (z. B. „write“ und „ (“ zu „write(“).
- Es müssen keine Kästchen um die Knoten gezeichnet werden wie in den Lösungen, allerdings sind die Verbindungskanten wichtig (z. B. Linie zwischen stmt und expr). Vergessene Kanten sind ein sehr häufiger Leichtsinnsfehler und führen zu Punktabzug!
- Der fertige Syntaxbaum muss den gesamten Code in richtiger Reihenfolge enthalten, d. h. in der untersten Baumebene (Knoten ohne Nachfolger) stehen nur noch Terminalsymbole, welche von links nach rechts gelesen den kompletten Code (jedes einzelne Zeichen!) ergeben.
- **Häufigster Fehler**: Geschweifte Klammern dürfen nicht direkt zu einem if- oder while-stmt zugeordnet werden. Es ist immer ein separates <stmt> = { <stmt>* } nötig (siehe Grammatik)!

Beispiel:

```
int fr_g, a;
if (x == 0) {
    z = readInt();
}
```



(36) ★★★

Zeichnen Sie die Syntaxbäume zu folgenden Codeauszügen.

Die MiniJava-Grammatik sei wie rechts dargestellt gegeben, wobei hier Terminalsymbole blau und Nichtterminale grün dargestellt wurden. ***Hinweis:*** *Lerne diese Grammatik nicht auswendig! Sie wäre im Klausur-Anhang gegeben (schwarz-weiß) und kann von der hier abgedruckten bzw. der in der Vorlesung vorgestellten Grammatik abweichen!*

Aufgabe a) *[Schwierigkeit: einfach]*

```
int a;
a = 0;
while (a < 10)
    a = readInt();
if (a > 100) {
    write(10);
}
```

Aufgabe b) *[Schwierigkeit: einfach]*

```
int x, y;
y = 1;
while (y > 0) {
    x = readInt();
    y = y - x;
}
write(y);
```

Aufgabe c) *[Schwierigkeit: mittel]*

```
if (x % 10 == x / 10 &&
    !(false || x != -1))
    write(-x);
```

Aufgabe d) *[Schwierigkeit: schwer]*

```
int a, b;
a = 1;
b = -2;;
if ((a%b) <= -a || b == 2) {
    a = 0;
} else if (true)
    write(b+1);
```

```
<program> ::= <decl>* <stmt>*

<decl>    ::= <type> <name> (, <name>)* ;

<stmt>    ::= ;
            | { <stmt>* }
            | <name> = <expr> ;
            | <name> = readInt();
            | write( <expr> );
            | if ( <cond> ) <stmt>
            | if ( <cond> ) <stmt> else <stmt>
            | while ( <cond> ) <stmt>

<expr>    ::= <number>
            | <name>
            | ( <expr> )
            | <unop> <expr>
            | <expr> <binop> <expr>

<cond>    ::= true
            | false
            | ( <cond> )
            | <expr> <comp> <expr>
            | <bunop> <cond>
            | <cond> <bbinop> <cond>

<unop>    ::= -
<binop>   ::= - | + | * | / | %
<bunop>   ::= !
<bbinop>  ::= && | ||
<comp>    ::= == | != | <= | < | >= | >

<type>    ::= int
<name>    ::= [A-Za-z_] [A-Za-z_]*
<number>  ::= 0 | ([1-9] [0-9]*)
```

Hinweis: *Diese Teilaufgabe wäre deutlich zu umfangreich für die Klausur. Wenn in der Klausur ein Syntaxbaum abgeprüft wird, dann vermutlich vom Schwierigkeitsgrad „einfach“ oder ein sehr kleiner schwieriger (z. B. mit else if). Klausuraufgaben dazu waren bisher aus Zeitgründen aber eher einfach.*

Aufgabe e) *[Schwierigkeit: schwer]*

Für diese Teilaufgabe sei die MiniJava-Grammatik wie im WS 2018/19 gegeben:

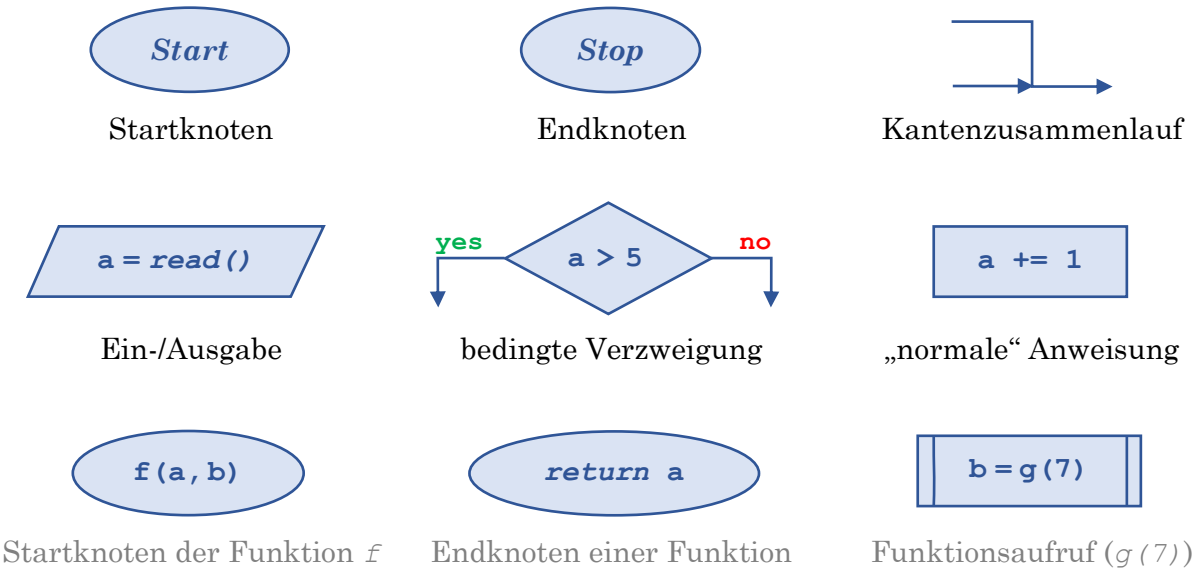
```
<program>     ::=   <decl>* <stmt>*
<decl>        ::=   <type> <name> ( , <name>)* ;
<stmt>        ::=   <expr>? ;
                |   { <stmt>* }
                |   <name> = <expr> ;
                |   <expr> [ <expr> ] = <expr> ;
                |   if ( <cond> ) <stmt> (else <stmt>)?
                |   while ( <cond> ) <stmt>
                |   do <stmt> while ( <cond> );
                |   return <expr> ;
<expr>        ::=   <number>
                |   <name>
                |   new int [ <expr> ]
                |   <expr> [ <expr> ]
                |   length( <expr> )
                |   read ( )
                |   write ( <expr> )
                |   <name> ( (ϵ | <expr> (, <expr>)*) )
                |   ( <expr> )
                |   <unop> <expr>
                |   <expr> <binop> <expr>
<cond>        ::=   true
                |   false
                |   ( <cond> )
                |   <expr> <comp> <expr>
                |   <bunop> ( <cond> )
                |   <cond> <bbinop> <cond>
<unop>        ::=   -
<binop>       ::=   - | + | * | / | %
<bunop>       ::=   !
<bbinop>      ::=   && | ||
<comp>        ::=   == | != | <= | < | >= | >
<type>        ::=   int | int[]
<name>        ::=   [A-Za-z_] [A-Za-z_]*
<number>      ::=   0 | ([1-9] [0-9]*)
```

```
int[] a;
int vA, vB;
{
    a = new int[4];
    vB = length(a);
}
do {
    vA = a[vB-1];
    vB = f(vB, vA+a[0]);
    a[vB] = (read());
} while (!(vB <= 0 || (vB > 4)));
return g();
```

Hinweis: Diese Teilaufgabe wäre zu umfangreich für die Klausur.

10. KONTROLLFLUSSDIAGRAMME

Mit *Kontrollflussdiagrammen* (auch: *Kontrollflussgraphen*) wird veranschaulicht, wie bzw. in welcher Reihenfolge einzelne Programmstücke ausgeführt werden. Es kann bspw. nachvollzogen werden, ob und unter welchen Umständen ein Programm terminiert. In einem solchen Diagramm werden im Grunde alle Statements in Knoten (Ellipse, Rechteck, Raute, Parallelogramm) gezeichnet und mit Kanten (Pfeilen) verbunden. Deklarationen von Variablen (z. B. „`int i;`“) werden dabei nicht eingezeichnet! Es existieren folgende Knotenarten:



Startknoten | Endknoten | Kantenzusammenlauf

Ein-/Ausgabe | bedingte Verzweigung | „normale“ Anweisung

Startknoten der Funktion `f` | Endknoten einer Funktion | Funktionsaufruf (`g(7)`)

Ein Kontrollflussdiagramm verfügt über *genau einen Startknoten* und *beliebig viele Endknoten*. Soll eine ganze Methode/Funktion dargestellt werden, so wird anstelle des normalen Startknotens der *Startknoten für Funktionen* und als Endknoten ein (ggf. leeres) *return*-Statement verwendet. Besteht der Code-Auszug aus mehreren Funktionen, dann existiert für jede Funktion ein separater Startknoten. Der Start-/Endknoten steht in einer Ellipse (wie hier), einem Kreis, oder einem Rechteck mit abgerundeten Ecken.

Grundsätzlich werden alle Anweisungen in Rechtecke geschrieben. Ausnahmen bilden Eingabe- und Ausgabe-Statements wie `read`, `write` oder `System.out.println` (→ Parallelogramm) und bedingte Verzweigungen wie bei `if`, `while` oder `switch-case` (→ Raute).

Sonstige Hinweise:

- Das Semikolon (`;`) kann eingezeichnet werden, muss aber nicht. Farben spielen keine Rolle.
- Mehrere aufeinanderfolgende „normale“ Anweisungen (bspw. `a += 1` und `--b`) können in *einem* Knoten zusammengefasst werden (bestenfalls getrennt durch eine horizontale Linie).
- Ob der **`yes`**- bzw. **`no`**-Teil einer bedingten Verzweigung links oder rechts steht, ist egal. Man kann innerhalb eines Kontrollflussdiagramms auch variieren.
- Statt **`yes`** und **`no`** kann auch **`true`** und **`false`**, **`ja`** und **`nein`** oder Ähnliches geschrieben werden.
- Wo die Kanten einen Verzweigungsknoten verlassen oder erreichen spielt keine Rolle.
- **Häufigster Fehler:** Pfeilspitze oder **`yes`**-/**`no`**-Beschriftung vergessen (jeweils -½ Punkt).

(37) ★★★ Geben Sie die Kontrollflussdiagramme zu folgenden Codeauszügen an.

a)
```
int x = 1;
while (x < 5) {
    int y = x - 1;
    if (y > 3) {
        write("x");
    } else
        break;
    x = x * 2;
}
```

b)
```
int x;
x = read();
while (x < 0 || x > 100) {
    write("Wrong Input");
    x = read();
}
```

c)
```
for (int i = 1; i < 10; i *= 2) {
    if (i % 3 == 0)
        write(i);
}
```

d)
```
if (b) {
    b = a;
    if (a)
        write("b");
} else if (a) {
    a = read();
} else
    write(b == a);
a = b;
```

e)
```
write("Hello User");
for (int x = 1000; x > 0; x--) {
    int y = x & 14;
    while (y != 0) {
        write(y);
        y /= 2;
    }
    write(0);
}
write("Bye bye");
```

f)
```
public int f() {
    int a = read();
    if (a > 5)
        a = g(5);
    return a;
}

int g(int x) {
    if (x <= 5)
        return x-1;
    return 5;
}
```

g)
```
while (var != null) {
    var = read();
    if (var.equals("quit")) {
        break;
    }
    if (var.equals("clear")) {
        continue;
    }
    evaluate(var);
}
```

h)
```
while(true)
    write("1");
    write("2");
```

i)
```
public int fun(int x) {
    switch(x) {
        case 0:
            x = 5; break;
        case 1: case 3:
            x = 2;
        case 2:
            return 2;
        default:
            write(x);
    }
    return 1;
}
```

j)
```
int x = 1, y;
do {
    y = read();
    if (y % 2 == 0)
        x = read();
    write(x * y);
} while (x != y);
```

k)
```
int a, b;
loop1: while (true) {
    loop2: while (true) {
        a = read();
        int c = b = read();
        if (b < 0)
            break;
        loop3: do {
            b *= 2;
        } while (b < a);
        if (c == 0)
            break loop1;
        else if (c == 1)
            continue loop2;
    }
    System.out.println("fail");
}
```

l)
```
int a, b, c;
final int d = read();
c = read();
outer: for (b = d; b < c; ++b) {
    if (b % 2 != 0) {
        a = b % 10;
        switch (a) {
            case 1:
                ++c;
            case 3:
                write(c);
                continue;
            case 5:
                return;
            case 7: {
                b %= 2;
            }
            case 9:
                break outer;
        }
        write(a);
    }
    write(b);
}
```

ZUSTANDSTABELLEN

Der Zustand eines Programms zu einem bestimmten Zeitpunkt während der Ausführung wird über die Werte der vorhandenen Variablen beschrieben. Zum Festhalten der Zeitpunkte dienen Markierungen im Code – das sind hier einfach die Zeilennummern. Üblicherweise werden nur Zustände angegeben, bei denen Änderungen an den Variablen stattfinden. In unserem Fall tragen wir deshalb ausschließlich die Zustände nach *Zuweisungen* ein (könnte in der Klausur anders sein, vgl. Aufgabenstellung). Die Zustände der Variablen werden in unserem Fall immer *nach* Ausführung der jeweiligen Zuweisung angegeben (auch das steht in der Aufgabenstellung). Nicht initialisierte oder noch nicht deklarierte Variablen werden dabei mit „–“ gekennzeichnet.

Beispiel: Zustandstabelle zu 37 a)

Zeilennr.	x	y
1	1	–
3	1	4
8	2	4
3	2	3

In Codezeile 1 wird die Variable `x` mit dem Wert `1` initialisiert, während `y` noch nicht existiert. In Zeile 2 finden keine Zuweisung statt, weshalb sie in der Zustandstabelle nicht erscheint. In Zeile 3 wird die Variable `y` deklariert und mit dem Wert `5-1 = 4` initialisiert, `x` bleibt unverändert. Der `if`-Block wird ausgeführt, enthält aber keine für die Zustandstabelle relevanten Änderungen. Anschließend führt Zeile 8 dazu, dass der Wert von `x` auf `2` verdoppelt wird. Da die `while`-Bedingung „`x < 5`“ immer noch wahr ist, wird erneut Zeile 3 ausgeführt, wodurch die (eigentlich neue) Variable `y` den Wert `5-2 = 3` erhält. Nachdem die `if`-Bedingung „`y > 3`“ jetzt nicht mehr wahr ist, wird der `else`-Teile ausgeführt und die `while`-Schleife daher mittels `break` verlassen.

(38) ★☆☆ Erstellen Sie die **Zustandstabelle** für den Code aus Aufgabe 37 l). Die Tabelle soll die Werte der Variablen `a`, `b`, `c` und `d` *nach Zuweisungen* angeben. Als Zuweisung wird hier jede Anweisung verstanden, die eine oder mehrere Variablen modifiziert (d. h. bspw. `x = 0` oder `x++`). Geben Sie außerdem jeweils die Zeilennummer der Zuweisung an, durch deren Ausführung der jeweilige Zustand produziert wurde. Noch nicht initialisierte oder deklarierte Variablen sollen mit „–“ (Minus) gekennzeichnet werden.

Hinweis: Die *for*-Schleife in Zeile 4 enthält zwei Zuweisungen. Unterteilen Sie diese Zeile daher in die Zeilennummern *4i* (Initialisierung: `b = d`) und *4m* (Modifikation: `++b`).

Der Code enthält zwei `read()`-Aufrufe. Gehen Sie davon aus, dass die `read()`-Methode folgenden Zahlen (in ebendieser Reihenfolge) zurückgibt:

a) `6` und `8`:

Zeilennr.	a	b	c	d

b) `-1` und `2`:

Zeilennr.	a	b	c	d

c) `0` und `5`:

Zeilennr.	a	b	c	d

11. Rekursion

Iteration vs. Rekursion in prozeduraler Programmierung

Ruft eine Funktion bzw. Methode sich selbst auf, so nennt man sie *rekursiv*. Eine rekursive Methode ruft also an mind. einer Stelle im Code (ggf. aber auch an mehreren) sich selbst auf (*rekursiver Aufruf*). Damit das nicht endlos lange so weiter geht (z. B. `f(2)` ruft `f(1)` auf, `f(1)` ruft `f(0)` auf, `f(0)` ruft `f(-1)` auf usw.), benötigt jede rekursive Methode mind. eine Abbruchbedingung (*Rekursionsanker*), d. h. einen oder mehrere Fälle, in denen kein rekursiver Aufruf mehr stattfindet (z. B.: `if(n <= 0) { return 0; }`). Außerdem muss sichergestellt sein, dass in jedem Fall irgendwann ein Rekursionsanker erreicht wird.

Rekursionsarten:

- ***Lineare Rekursion*** (häufigste Form): In jedem (rekursiven) Aufruf der Methode findet maximal ein rekursiver Aufruf statt, d. h. es wird entweder ein Rekursionsanker erreicht oder es findet exakt ein rekursiver Aufruf statt. Beispiele: Fakultät, GGT
- ***Endrekursion***: Nach Durchführung des rekursiven Aufrufs kann die Methode direkt verlassen werden, d. h. es sind keine weiteren Berechnungen oder Methodenaufrufe mehr nötig. Bei `void`-Methoden bedeutet das, dass der rekursive Aufruf das letzte sein muss, was vor dem Verlassen der Methode passiert. Bei allen anderen Methoden muss das Ergebnis des rekursiven Aufrufs unverändert zurückgegeben werden (z. B. `return f(n+1);`). Bei mehreren rekursiven Aufrufen innerhalb einer Methode muss diese Bedingung entsprechend für jeden rekursiven Aufruf erfüllt sein. Beispiele findest du im Unterkapitel „Endrekursion“.
- ***Verschachtelte Rekursion***: Rekursive Aufrufe werden als Parameter rekursiver Aufrufe verwendet (z. B. `f(f(n+1));`). Achtung: So etwas ist niemals endrekursiv.
- ***Kaskadenförmige Rekursion***: In mind. einem Fall finden mehrere rekursive Aufrufe statt (und werden ggf. miteinander verknüpft). Beispiele: Fibonacci, Binomialkoeffizient
- ***Verschränkte Rekursion***: Zwei verschiedene Methoden rufen sich wechselseitig auf. Beispiel: `f()` ruft `g()` auf und `g()` ruft `f()` auf.

Die Informationsblöcke der rekursiven Aufrufe, welche bspw. die Rücksprungadresse (zuletzt aktive Zeile im Code) und Werte der lokalen Variablen und Parameter enthalten, werden dabei auf dem *Stack* gespeichert, während Objekte auf dem *Heap* abgelegt werden. Wichtig: Jeder rekursive Funktionsaufruf besitzt somit seine eigenen lokalen Variablen. Diese müssen von der JVM* jeweils gespeichert werden, damit bei der Rückkehr zu einem vorherigen rekursiven Aufruf dessen lokale Variablen wiederhergestellt werden können. Ist der Stack voll (also der gesamte Speicher belegt), so sind keine weiteren Methodenaufrufe mehr möglich und es kommt zu einem *StackOverflowError*. Dieser Fehler ist das Ende einer „Endlosrekursion“ (anders als bei einer Endlosschleife, welche tatsächlich unendlich lange ausgeführt wird) und meist ein Indiz für fehlende oder unvollständige Abbruchbedingungen.

Ist eine rekursive Implementierung gefordert, dürfen meist keine Schleifen verwendet werden, da die Implementierung dann wahrscheinlich (nicht zwangsweise!) rein iterativ geschieht. Eine Methode wird auch dann als rekursiv angesehen, wenn sie selbst nicht rekursiv ist, aber lediglich eine rekursive Hilfsmethode aufruft. Das kann Sinn machen, bspw. um eine Art „Zählervariable“ oder einen Akkumulator (vgl. Aufgaben zu Endrekursion) als zusätzlichen Parameter einzuführen, den man in der ursprünglichen Methode mit einem bestimmten Startwert initialisiert.

* Die JVM (Java Virtual Machine) führt deinen Code aus und kümmert sich um diesen Speicher- und Wiederherstellungsvorgang.

(39) ★★☆ Überführen Sie die folgende Methode zur Berechnung der ***Fakultät*** in eine rekursive Variante (ohne Schleifen), indem Sie die Methode `int facRec(int n)` implementieren, welche für *alle* möglichen Werte für `n` gleich auswertet wie `facIter(n)`.

```
int facIter(int n) {
    int result = 1;

    while (n > 1) {
        result *= n;
        n--;
    }

    return result;
}
```

```
int facRec(int n) {




}
```

(40) ★★☆ Gegeben sei folgende rekursiv definierte mathematische Funktion zur Berechnung der n-ten ***Fibonacci***-Zahl:

$$fib(n) := \begin{cases} 0 & \text{falls } n = 0 \\ 1 & \text{falls } n = 1 \\ fib(n-1) + fib(n-2) & \text{sonst } (n > 1) \end{cases}$$

Implementieren Sie die Funktion *rekursiv* in einer Methode `fibRec`. Für ungültige Parameter soll `-1` zurückgegeben werden.

```
int fibRec(int n) {









}
```

Mathematische Zusatzaufgabe (anspruchsvoll!):
Implementieren Sie eine *iterative* Variante.

```
int fibIter(int n) {









}
```

(41) ★★☆

Gesucht ist eine Methode, die ein **Sternendreieck** der Höhe und Breite `size`, welche als (einziger) Parameter übergeben wird, auf der Konsole ausgibt. Es sollen in der gesamten Implementierung keine Schleifen verwendet werden.

Eine (private) Hilfsmethode kann dabei helfen, um eine bestimmte Anzahl an Sternen miteinander zu verknüpfen und so die einzelnen Zeilen zu bilden.

Beispielausgabe für den Parameterwert `size = 5`:

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

Geschafft? Dann versuch' es doch mal rechtsbündig!

(42) ★☆☆

Mit dem *euklidischen Algorithmus* kann der **größte gemeinsame Teiler** (ggT, eng. *gcd*) zweier ganzer, positiver Zahlen berechnet werden:

$$ggt(a,b) \coloneqq \begin{cases} a & \text{falls } a = b \\ ggt(a-b,b) & \text{falls } a > b \\ ggt(a,b-a) & \text{falls } b > a \end{cases}$$

Implementieren Sie den Algorithmus in einer *rekursiven* Methode. Ungültige Parameterwerte müssen *nicht* behandelt werden.

```
static int ggt(int a, int b) {




}
```

(43) Grundlegende Verständnisfragen zu Rekursion

★★★

1. Betrachten Sie die Methode rechts:

   a) Ist die Methode rekursiv?

   b) Kompiliert der Code?

   c) Terminiert der Methodenaufruf `kleiner(x)` immer, also für alle `x`?

```
public int kleiner(int x) {
    while (x > 0)
        kleiner(x--);
    return x + 1;
}
```

2. Welche Ausgabe produziert folgender Code? Ist *happy()* endrekursiv?

```
public static void happy(int x) {
    if (x < 0) return;
    x *= -3;
    happy(x);
}

public static void main(String[] args) {
    int x = 2;
    happy(x);
    System.out.println(x);
}
```

3. Wozu werten die Aufrufe `f(2)`, `f(-1)` und `f(0)` aus?

```
public int f(int f) {
    if(f > 5) return f;
    f = -2 * f;
    f(f);
    return f;
}
```

4. Welche Ausgabe produziert folgender Code?

```
private static void a(int[] a) {
    System.out.println(a[0]);
    int[] b = {a[0] * 2};
    a[0] = b[0];
    a = b;
    if (a[0] < 15) a(a);
}

public static void main(String[] args) {
    int[] b = {3};
    a(b);
    System.out.println(b[0]);
}
```

5. Existiert zu jedem rekursiv definierten Algorithmus ein äquivalenter iterativer Algorithmus?

   ☐ Ja ☐ Nein

   ☐ Wer weiß das schon

(44) ★★☆

Implementieren Sie zur Berechnung der ***Potenz*** a^b (b sei nie negativ) eine

a) *iterative* Methode:

```
int powerIter(int a, int b) {



}
```

b) *rekursive* Methode, welche keine Schleifen verwendet:

```
int powerRec(int a, int b) {



}
```

(45) ★☆☆

Schreiben Sie eine *rekursive* Methode `reverse(int)`, die für einen *positiven* Zahlenwert `x` die Ziffern in umgekehrter Reihenfolge auf der Konsole ausgibt.

Beispiel: Der Aufruf von *`reverse(14876)`* gibt *`67841`* aus.

```
public void reverse(int x) {



}
```

(46) ★★☆ Im Online-Shop eines Großhändlers können unbedruckte T-Shirts in Einheitsgrößen bestellt werden, wobei ab bestimmten Bestellmengen Mengenrabatte angeboten werden.

Beispiel: 121 T-Shirts kosten 250 + 28 + 28 + 3 = 309 €.

Menge	Preis
1	3 €
10	28 €
50	130 €
100	250 €

Implementieren Sie die Methode `int price(int n)`, welche die Gesamtkosten der Bestellung in Abhängigkeit von der Anzahl an T-Shirts `n` *rekursiv* berechnet:

```
int price(int n) {




}
```

(47) ★★☆ Implementieren Sie die Berechnung des ***Binomialkoeffizienten*** („*n über k*“)

a) auf Basis des „pascalschen Dreiecks“ nach folgender Definition

$$\binom{n}{k} \coloneqq \binom{n-1}{k-1} + \binom{n-1}{k}$$
$$\binom{n}{0} \coloneqq \binom{n}{n} \coloneqq 1$$

wobei $0 \leq k \leq n$, d. h. insb. $n \geq 0$, in einer *rekursiven* Methode.
Bei ungültigen Parameterwerten soll die Methode zu `-1` auswerten.

```
int binomialRec(int n, int k) {




}
```

b) nach der nachfolgend genannten Formel, indem Sie auf Ihre Implementierung der Fakultätsfunktion `facRec(int n)` aus Aufgabe 39 zurückgreifen:

$$\binom{n}{k} \coloneqq \frac{n!}{k! * (n-k)!}$$

```
int binomialDef(int n, int k) {




}
```

Ist nun (a) der rekursive (b) oder der iterative Ansatz besser? Grundsätzlich ist die iterative Lösung einer Aufgabe in Java so gut wie immer besser, weil die rekursiven Aufrufe mehr Zeit und Speicherplatz benötigen. Bei einer iterativen Lösung kann es auch für große Zahlen *nicht* zu einem Stack-Overflow kommen. Ansatz b) wäre daher effizienter, wenn man auch die Fakultätsberechnung iterativ durchführen würde (`facIter` statt `facRec`).

Warum üben wir Rekursion dann überhaupt? Tatsächlich existiert zu jedem rekursiven Ansatz auch eine äquivalente und meist effzientere iterative Lösung. Viele Probleme lassen sich jedoch nicht so einfach iterativ lösen – nämlich gerade wenn es sich um ein rekursiv formuliertes Problem handelt. Hast du es z. B. geschafft, die Fibonacci-Zahlen iterativ zu berechnen (Zusatzaufgabe 40, S. 67)? Du kannst ja mal versuchen, den Quicksort-Algorithmus (Aufgabe 105, S. 153) iterativ zu implementieren... Möglich? Ja. Einfach? Nein. Deshalb üben wir rekursives Denken.

(48) ★☆☆ Gesucht ist eine *rekursive* Methode, die alle durch 3 teilbaren Zahlen kommagetrennt beginnend bei 0 bis maximal *n* ausgibt, wobei *n* als Parameter übergeben wird. Ergänzen Sie die Hilfsmethode und initialisieren Sie den Parameter *i* beim Aufruf sinnvoll.

Hinweis: Für die Ausgabe können Sie die Methode `System.out.print()` verwenden.

```
public static void multiplesOfThree(int n) {
    multiplesOfThree(n, [    ]);
}

private static void multiplesOfThree(final int n, int i) {




}
```

(49) ★★☆ Schreiben Sie eine Methode, die den als Parameter übergebenen `String` durchläuft und alle darin enthaltenen Vokalbuchstaben (a, e, i, o, u, y) rekursiv durch *x* bzw. *X* ersetzt, wobei die Klein- bzw. Großschreibung erhalten bleiben soll.

Hinweis: Da `String`s unveränderlich sind, muss entsprechend ein *neuer* `String` aufgebaut werden. Implementieren Sie daher eine Hilfsmethode (wie in Aufgabe 48). Sie dürfen keine Schleifen verwenden!

```
public static String replaceVowels(String s) {

}

private static String replaceVowels(final String s, int i) {




}
```

(50) ☆☆☆

Implementieren Sie den „+“-Operator (***Addition***) in einer *rekursiven* Methode `int sum(int a, int b)`, wobei erlaubte arithmetische Operationen lediglich die Inkrementierung und Dekrementierung (Addition bzw. Subtraktion von 1, d. h. $x \pm 1$) sind. Beachten Sie, dass a und b negativ sein können.

```
static int sum(int a, int b) {




}
```

(51) ★★☆

Schreiben Sie ohne Verwendung von Schleifen eine *rekursive* Methode

`replaceDigit(long number, byte digit, byte newDigit)`,

die alle Vorkommen der Ziffer `digit` in der Zahl `number` durch die Ziffer `newDigit` ersetzt und die neue Zahl zurückgibt. Gehen Sie davon aus, dass für `digit` und `newDigit` nur gültige Werte, also Ziffern (einstellig), übergeben werden. Wandeln Sie die Zahl an keiner Stelle in einen `String` um!

Beispiel: `replaceDigit(12123221, 2, 4)` gibt `14143441` zurück.

(52) ★★☆

Ein ***Palindrom*** ist ein Wort, welches vorwärts und rückwärts gelesen identisch ist, also bspw. „wow“, „anna“ oder „legovogel“. Zur Erkennung von Palindromen sollen Sie eine Methode `boolean isPalindrome(char[] c)` schreiben, die als Parameter ein `char`-Array erwartet und `true` zurückgibt, wenn es sich bei dem Feld um ein Palindrom handelt. Groß- und Kleinschreibung soll unterschieden werden und die Berechnung *rekursiv* durchgeführt werden. Das Array `c` darf nicht verändert werden.

Beispiele:

- `isPalindrome({'w', 'o', 'w'}) == true`
- `isPalindrome({'A', 'n', 'n', 'a'} == false`

Sie dürfen sich (falls nötig) private Hilfsmethoden definieren.

Zusatzaufgaben:

- Implementieren Sie außerdem eine *iterative* Fassung.
- Ignorieren Sie die Groß-/Kleinschreibung, sodass bspw. auch „Anna“ als Palindrom erkannt wird.

(53) ★☆☆

Eine ***Primzahl*** ist eine natürliche Ganzzahl, die größer ist als `1` und nur durch `1` und sich selbst ganzzahlig (d. h. ohne Rest) teilbar ist. Eine Methode `isPrime(int)` sei wie unten angegeben implementiert. Ihre Aufgabe ist es nun, die darin verwendete private Hilfsmethode so zu implementieren, dass `isPrime` genau dann zu `true` auswertet, wenn es sich bei der als Parameter übergebenen Zahl um eine Primzahl handelt. Die Implementierung muss *rekursiv* sein.

Tipp: Die Hilfsmethode erhält als zweiten Parameter einen Teiler, welcher in jedem rekursiven Schritt heruntergezählt wird, ähnlich wie es in einer primitiven iterativen Fassung gemacht werden würde, um alle Teiler von `n-1` bis `1` auf Teilbarkeit zu prüfen.

```
public boolean isPrime(int n) {
    if (n <= 1)
        return false;
    return isPrime(n, n-1);
}
```

(54) ★☆☆

Überführen Sie die folgende *iterative* Methode in eine *rekursive* Variante (ohne Schleifen), indem Sie die Methode `int ultraRec(short x)` implementieren, welche für alle möglichen Werte für `x` gleich auswertet wie `ultraIt(x)`:

```
int ultraIt(short x) {
    int ret = 0;
    for(int i=x; i>0; i-=2)
        ret += i * (i-1);
    return ret;
}
```

```
int ultraRec(short x) {




}
```

(55) ★★☆ **Reverse *rekursiv***: Die *reverse*-Methode aus Aufgabe 22 (Seite 43) invertiert ein Array *in-place*, d. h. der Parameter wird direkt verändert. Nun soll eine Methode

```
static Object[] reverse(Object[] a)
```

implementiert werden, die das als Parameter übergebene Array `a` umkehrt, `a` dabei jedoch *nicht* verändert. Stattdessen soll ein *neues* Array zurückgegeben werden, welches alle Elemente des Parameters `a` in umgekehrter Reihenfolge enthält. Wird der Methode bspw. das Array `{"A","B","C"}` übergeben, soll sie das Array `{"C","B","A"}` zurückgegeben werden. Wird `null` übergeben, soll ebenfalls `null` zurückgegeben werden.

Zur Umkehrung des Arrays wird auf die *rekursive* Hilfsmethode `reverse(Object[] a, int from, int to)` zurückgegriffen, welche das Array im Bereich von `from` bis `to` (jeweils *inklusive*) umkehrt. Sie teilt das Array dazu in zwei disjunkte Bereiche (`left` und `right`), welche dann (rekursiv) invertiert werden und die beiden Ergebnis-Arrays zusammenfügt (*merge*). Wird also bspw. das Array `{1, 2, 3, 4}` und die Grenzen `0` (`from`) und `1` (`to`) übergeben, so soll das Array `{2, 1}` zurückgegeben werden.

Füllen Sie die Lücken:

```
public static Object[] reverse(Object[] a) {
   return reverse(a, 0, a.length-1);
}

private static Object[] reverse(Object[] a, int from, int to)
{
   if (                          ) return null;
   if (to < from)  return new Object[0];  // kein Element
   if (to == from) return                          ;

   int middle = (from+to)/2;
   Object[] left = reverse(a, from, middle);
   Object[] right = reverse(a,                 , to);
                  result = new Object[                    ];

   for (int i = 0; i <                       ; ++i)
       result[i] =                       ;
   for (int i = 0; i < left.length; ++i)
       result[                       ] = left[                ];

   return result;
}
```

(56) **Zinseszins**-Rechner:

★☆☆

Alfi (15) vergleicht verschiedene Anlagemöglichkeiten und stößt auf ein attraktives Angebot der „Fairness Bank“, welche ihm vorschlägt, sein Erspartes (stolze 1.000 Euro) zu einem festen Zinssatz von 10 Prozent (jährlich) anzulegen. Der Haken ist, dass Alfi nicht an das Geld kommt, bis der Kontostand 1.000.000 Euro beträgt. Henni (13) hat nur die Hälfte des Kapitals von Alfi und soll daher jährlich nur 5 Prozent Zinsen bekommen, allerdings hat sie bereits bei einem Kontostand von 100.000 Euro vollen Zugriff auf das Sparvermögen. Wer von beiden ist mit dem Angebot wohl besser beraten?

a) Schreiben Sie eine *rekursive* Methode, die in Abhängigkeit von Startkapital, Zielkapital (jeweils in Euro) und Zinssatz (in Prozent) die Anzahl an Jahren berechnet, nach denen das Ziel erreicht ist:

```
public static int anlagedauerBerechnen(double kapital,
                                        double zielkapital,
                                        double zinssatz) {




}
```

Alternativ kann Alfi auf einen monatlichen Sparplan der „Money AG“ setzen, wobei monatlich ein bestimmter Betrag eingezahlt wird und am Ende des Jahres die Zinsen überwiesen werden (dabei werden alle 12 Einzahlungen miteingerechnet). Der Monatsbetrag erhöht sich jedoch jährlich um einen bestimmten Prozentsatz (Dynamisierungsrate). Auch hier kann Alfi mit einem bestimmten Kapital starten (wird auch verzinst). Hat Alfi nun bessere Chancen, wenn er monatlich 20 Euro mit einer jährlichen Dynamisierungsrate von 5% zu einem Zinssatz von 7 Prozent über den gleichen Zeitraum wie im obigen Beispiel anlegt, jedoch ohne Startkapital startet?

b) Schreiben Sie eine *rekursive* Methode, die das Sparguthaben in Abhängigkeit von der Anlagedauer (in Jahren), dem Startkapital (in Euro), dem Monatsbeitrag, dem Zinssatz (Rendite in Prozent) und der Dynamisierungsrate (in Prozent) berechnet.

```
public static double vermoegenBerechnen(double kapital,
                                         double monatsbeitrag,
                                         int jahre,
                                         double zinssatz,
                                         double dynamisierung) {




}
```

ENDREKURSION

(57) ★☆☆ Ist Ihre rekursive Implementierung zu den Aufgaben 39, 42, 44, 45, 46, 48, 49, 52, 53 und 54 *endrekursiv*? Falls nicht: Implementieren Sie die Methoden nun *endrekursiv*.

Hinweis: Die Implementierungen aus den Aufgaben 40, 47 und 55 sind *nicht* endrekursiv, wären aber nur mit sehr viel Aufwand endrekursiv implementierbar, daher verzichten wir an dieser Stelle darauf.

Sie dürfen weitere (private) Methoden implementieren, wobei diese dann ebenfalls *endrekursiv* sein müssen und keine Schleifen enthalten dürfen. Zur Erinnerung: Für uns gilt eine Methode auch dann als endrekursiv, wenn sie eigentlich gar nicht rekursiv ist, aber eine endrekursive Hilfsmethode aufruft.*

* Das ist streng genommen nicht korrekt, für uns aber ok. Die nicht-rekursive Methode fungiert als eine Art Wrapper.

(58) ★★☆ Schreiben Sie eine Methode zur Berechnung der ***Summe*** *aller ganzen Zahlen* von 1 bis n, wobei n als Parameter übergeben wird und den „größtmöglichen“ Typ zur Speicherung ganzer Zahlen haben soll. Die zu implementierende Methode soll also (in Abhängigkeit von n) folgende Berechnung durchführen:

$$\sum_{i=1}^{n} i$$

Implementieren Sie den Algorithmus ...

- ... iterativ in einer öffentlichen Methode mit dem Namen `sumIt`,
- ... rekursiv in einer öffentlichen Methode mit dem Namen `sumRec` und
- ... endrekursiv in einer öffentlichen Methode mit dem Namen `sumTailRec`.

Verwenden Sie *keine* Membervariablen. Sie dürfen weitere private Methoden schreiben (der Nutzer muss kein Wissen über diese Methoden haben). Alle öffentlichen Methoden müssen als Parameter das oben beschriebene n übernehmen. Wählen Sie einen sinnvollen Ergebnistyp. Für alle $n \leq 0$ ist das Ergebnis 0.

(59) ★☆☆ Gegeben sei die *endrekursive* Implementierung eines sinnfreien Algorithmus in der Methode `nonsenseTail(int)`. Überführen Sie diese Implementierung in ...

a) eine *rekursive* Implementierung in der Methode `int nonsenseRec(int)`.

b) eine *iterative* Implementierung in der Methode `int nonsenseIter(int)`.

```
public int nonsenseTail(int x) {
    return nonsenseTailHelper(x, 0);
}

private int nonsenseTailHelper(int x, int acc) {
    if (x >= 100)
        return acc;
    return nonsenseTailHelper(x+3, acc+x);
}


public int nonsenseRec(int x) {









}


public int nonsenseIter(int x) {









}
```

(60) ★★★

a) Implementieren Sie die Methode `quersumme(int)` zur *rekursiven* Berechnung der **Quersumme** der als Parameter übergebenen ganzen Zahl. Die Quersumme ist die Summe aller Ziffern einer natürlichen Zahl, wobei Ihre Implementierung auch für negative Zahlen funktionieren soll. Sei q die Quersumme einer positiven Zahl a, dann definieren wir $-q$ als die Quersumme von $-a$.
Es dürfen keine Schleifen vorkommen.

Tipp: Modulo-Operator („Teilerrest“) wie in der `reverse(int)`-Aufgabe.

```
public int quersumme(int a) {




}
```

b) Geht das auch *iterativ*? Hier dürfen Sie selbstverständlich Schleifen verwenden.

```
public int quersumme(int a) {




}
```

c) Die Lösung aus Aufgabe a) ist nicht *endrekursiv*? Let's go!

```
private int quersumme_h(int a, int acc) {




}

public int quersumme(int a) {
    return quersumme_h(a, );
}
```

(61) ★★☆

Verlangt ist eine Methode `invert`, die einen `String` als Parameter erwartet, welchen sie in umgekehrter Reihenfolge zurückgibt (schließlich sind `String`-Objekte unveränderlich). Der Aufruf `invert("Klausur")` soll also bspw. zu `"rusualK"` auswerten.

Füllen Sie die Lücken in der unterhalb angegebenen endrekursiven Implementierung, sodass der Aufruf der Methode `invert(str)` für einen beliebigen `String str` wie oben beschrieben auswertet, also den invertierten `String` zurückgibt. Wird **`null`** übergeben, so soll eine `IllegalArgumentException` geworfen werden (Vorgriff auf Kap. 14)

Die Idee hinter der privaten Hilfsmethode ist, dass beginnend bei dem letzten Index pro rekursivem Aufruf das am Index `index` stehende Zeichen an das im Parameter `result` gespeicherte Zwischenergebnis angehängt wird.

Außer den unterhalb zu vervollständigenden `invert`-Methoden und den Methoden `charAt(int)` und `length()` der Klasse `String` dürfen keine weiteren Methoden benutzt oder implementiert werden.

```
public String invert(String str)
{
    if (str == null)
        [                                                    ]
    return invert(str, [                    ], [                    ]);
}

private String invert(String str, int index, String result)
{
    if (index < 0)
        return [                    ];
    result += [                    ];
    return invert(str, [                    ], result);
}
```

(62) ★☆☆

Überführen Sie folgende endrekursive Methode `x` in eine *iterative* Fassung `y`, die für alle möglichen Parameterwerte gleich auswertet wie `x`. Hinweis: Der ternäre Operator wertet bei erfolgreicher Prüfung zum Wert nach dem „**`?`**“, sonst zum Wert nach dem „**`:`**“ aus.

```
static int x(int x) {
   return x < 100 ? x(x*x) : x;
}
```

```
static int y(int x) {




}
```

12. Objektorientierte Programmierung

Ein *Objekt* ist eine Instanz einer Klasse (wobei es beliebig viele Instanzen einer Klasse geben kann). Eine *Klasse* ist eine Art (nicht ausgefüllte) Vorlage für die Instanziierung der Objekte und existiert nur einmal. Unter *Instanziierung* versteht man die Erzeugung (das Anlegen) eines Objekts mithilfe des `new`-Operators. Dieser Instanziierungsoperator wird genutzt, um Konstruktoren aufzurufen (bspw. `new X()`) oder Arrays zu erzeugen (bspw. `new int[5]`).

Variablen können Objekte speichern (bspw. `String s = new String()`). Streng genommen werden Objekte aber auf dem *Heap* (das Gegenstück zum *Stack*) gespeichert und eine Variable speichert nie das Objekt selbst, sondern nur eine Referenz auf das Objekt. Anders als die Erzeugung muss die Vernichtung eines Objekts nicht explizit durchgeführt werden. Die sog. *Garbage Collection* (das ist ein Thread, der immer nebenher läuft) kümmert sich darum, dass nicht mehr benötigte Objekte – also Objekte, die nicht mehr referenziert werden – gelöscht werden.

Java verfolgt das ***„Call by Value"***-Konzept:

- Erwartet eine Methode einen Parameter primitiven Typs (z. B. `int v`) und wird beim Aufruf der Methode eine Variable `x` übergeben, so wird der Wert von `x` in die Parametervariable `v` kopiert. Durch die Veränderung von `v` (innerhalb der Methode) wird `x` nicht verändert.
- Erwartet eine Methode einen Referenzdatentyp (z. B. `Fahrzeug v`) und wird beim Aufruf der Methode eine Variable `x` übergeben, so wird ebenfalls der Wert von `x` in `v` kopiert. Aber Vorsicht: Der Wert von `x` ist kein Fahrzeug, sondern eine *Referenz* auf ein Fahrzeug. Eine Zuweisung der Form `v = ...;` wird den Wert von `v` ändern (d. h. `v` zeigt nun bspw. auf ein anderes Fahrzeug), nicht aber den von `x`, genau wie bei primitiven Werten. Führen wir hingegen eine Operation auf dem von `v` referenzierten Objekt durch (z. B. `v.tanken()`), so wird damit „auch" `x` verändert, schließlich referenzieren die beiden Variablen ein- und dasselbe Objekt.

Vererbung

Jede Klasse *erbt von* (auch: *erweitert*) genau einer anderen Klasse (`extends`). Die einzige Ausnahme ist `Object` (die oberste Oberklasse), welche von keiner Klasse erbt. Wird keine Oberklasse spezifiziert, so erbt eine Klasse implizit von `Object`. Eine Klasse erbt alle zugreifbaren Member (Methoden und Attribute) ihrer Oberklasse, d. h. private Member werden nicht vererbt (allerdings existieren für Objekte einer Unterklasse im Hintergrund auch die privaten Attribute der Oberklasse). *Erben* bedeutet also, dass eine Unterklasse über alle (nicht-privaten) Methoden und Variablen der Oberklasse verfügt und diese benutzen kann; die Oberklasse wird erweitert.

Geerbte Methoden können aber auch **überschrieben** (*overriding*) werden, indem man sie in der Unterklasse neu implementiert (mit gleicher Signatur und mindestens gleich großer Sichtbarkeit). **Überladen** von Methoden (*overloading*) ist davon unabhängig und bezeichnet die Implementierung mehrere Methoden mit dem gleichen Namen (aber unterschiedlicher Signatur).

Attribute können durch Parameter und lokale Variablen **verschattet** (*hiding*) werden. Das ist bspw. der Fall, wenn eine lokale Variable mit dem gleichen Namen wie ein Attribut existiert. Auf ein durch `var` verschattetes Attribut kann mit „`this.var`" zugegriffen werden. Analog können geerbte Attribute durch (neue) Attribute verschattet werden. Auf ein durch `var` verschattetes geerbtes Attribut kann mit „`super.var`" zugegriffen werden.

Auch statische Member werden vererbt, allerdings können statische Methoden in der Unterklasse nicht überschrieben, sondern nur verschattet werden.

Abstrakte Klassen und **Interfaces** (siehe Kapitel 6.4, Seite 36):

Interfaces sind keine Klassen und verfolgen einen anderen semantischen Sinn. Sie werden benutzt, um Klassen mit bestimmten Eigenschaften zu identifizieren. Implementiert eine Klasse ein Interface, so sichert sie uns zu, dass sie alle in dem jeweiligen Interface definierten Methoden implementiert. Klassen können beliebig viele Interfaces implementieren, z. B.:

```
public class Auto extends Fahrzeug implements Fahrbar, Lackierbar
```

Eine nicht-abstrakte Klasse, die ein Interface implementiert muss alle im Interface definierten Methoden implementieren. Da abstrakte Klassen auch abstrakte Methoden enthalten dürfen, müssen abstrakte Klassen, die Interfaces implementieren, nicht alle dort definierten Methoden implementieren, können aber.

Interfaces können von beliebig vielen anderen Interfaces erben (kommagetrennt, in der Praxis selten). Sie übernehmen damit die abstrakten Methoden des „Oberinterfaces“, z. B.:

```
public interface Lackierbar extends Faerbbar
```

Die häufigste Form des Interfaces ist das *funktionale Interface*. Ein solches Interface verfügt nur über eine einzige abstrakte Methode. Bekannte Beispiele aus der Java-Standardbibliothek sind `Runnable` (→ Kap. 21) und `Comparable<T>` (→ Kap. 18).

Folgendes Beispiel zeigt ein funktionales Interface. Zur Erinnerung: Alle nicht-statischen Methoden eines Interfaces sind implizit `public abstract`.

```
public interface Faerbbar {
    void einfaerben(String farbe);
}
```

Die Klasse `Auto` bietet also auf jeden Fall eine Methode `einfaerben(String)` an.

`Object`-Methoden: Jede Klasse erbt von `Object` (ggf. indirekt) und daher u. a. die Methoden:

- `String toString()` gibt eine `String`-Repräsentation des Objekts zurück. Die `toString`-Methode wird bspw. automatisch aufgerufen, wenn man versucht, ein Objekt mit einem String zu konkatenieren (`"Obj: " + meinObj` entspricht `"Obj: " + meinObj.toString()`) oder das Objekt mittels `System.out.println()` auszugeben. Man sollte die `toString`-Methode immer überschreiben, da die Standardimplementierung (geerbt von `Object`) lediglich einen `String` der Form `<Klassenname>@<Hash-Code des Objekts>` zurückgibt (bspw. `AuessereKlasse$InnereKlasse@6d06d69c`).
- `boolean equals(Object)` werden wir später in diesem Kapitel betrachten
- `void wait()`, `void notify()` und `void notifyAll()` spielen eine wesentliche Rolle bei der Thread-Synchronisierung, insbesonde im typischen Erzeuger-Verbraucher-Problem (→ Kap. 21).

Referenzgleichheit (==), **Objektgleichheit** (`equals()`) und `instanceof`

Primitive Werte (z. B. `int`, vgl. Kapitel 2) haben wir immer mit dem Vergleichsoperator (==) auf Gleichheit geprüft, d. h. bspw. `(5 == 5 ? 1 : 0)` wertet zu `1` aus. Vergleichen wir nun jedoch zwei Objekte mittels dieses Operators, verhält sich dieser anders und prüft nun auf Referenzgleichheit, also ob die beiden Objekte *identisch* sind (Handelt es sich um *dasselbe* Objekt?).

Daher wertet z. B. der Ausdruck `(new String("Test") == new String("Test"))` zu `false` aus, d. h. auch `String`s sollten niemals mit == verglichen werden.

Um die Inhalte zweier Objekte auf Äquivalenz zu prüfen, nutzen wir stattdessen die `equals`-Methode. Diese sollte genau dann `true` zurückgeben, wenn die beiden Objekte als gleich angesehen werden können (Handelt es sich um *das gleiche* Objekt?).

Beispiel: Objekte der von mir definierten Klasse `TShirt` haben eine Farbe (String) und eine Produktionsnummer (`int`). In der `equals`-Methode kann ich nun selbst festlegen, wann zwei Objekte als „gleich“ angesehen werden sollen. Ich könnte zwei T-Shirts nun bspw. als „gleich“ bezeichnen, wenn die Farbe gleich ist (mir die Produktionsnummer hingegen egal ist):

```
1  public class TShirt {
2     public String farbe;
3     private int produktionsnummer;
4     @Override
5     public boolean equals(Object other) {
6        if (other == null)
7            return false;
8        if (!(other instanceof TShirt))
9            return false;
10       TShirt t = (TShirt) other;
11       return (farbe != null ? farbe.equals(t.farbe) : t.farbe == null);
12    }
13 }
```

Im Regelfall müssen wir in der `equals`-Methode einen Cast durchführen, da die Methode (sofern man die von `Object` geerbte Methode überschreiben will) ein Objekt vom Typ `Object` übergeben bekommt, wobei ja nicht sichergestellt ist, dass dieses Objekt dann ein Attribut `farbe` hat (Es könnte ja auch ein Integer übergeben werden...).

Mit dem `instanceof`-Operator kann man überprüfen, ob das links davon aufgeführte Objekt (hier: `other`) mindestens so speziell ist (d. h. auch Unterklassen sind möglich; ohne Unterklassen: `other.getClass() == this.getClass()`) wie die rechts vom Operator genannte Klasse (hier: `TShirt`). Ist das nicht der Fall (`== false`, hier mit `!`), können die Objekte nicht gleich sein. Die `equals`-Methode sollte gewisse Eigenschaften (Reflexivität, Symmetrie, Transitivität, `false` wenn `other == null`, ...) besitzen, siehe *Javadoc**.

In der von `Object` geerbten Standardimplementierung tut die `equals`-Methode übrigens das gleiche wie der ==-Operator, prüft also auf Referenzgleichheit:

```
public boolean equals(Object other) {  return (this == other);  }
```

* *https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html#equals-java.lang.Object-*

(63) ★★☆ **`equals()`, Gleichheitsoperator und String-Pool** (Sonderfall: Strings)

1. Kreuzen Sie jeweils an, wozu die Ausdrücke auswerten bzw. kreuzen Sie nichts an, falls Sie der Meinung sind, dass der Ausdruck fehlerhaft ist bzw. kein Ergebnis hat. Die Klasse `TShirt` sei wie auf der vorherigen Seite implementiert.

```
TShirt t1, t2, t3, t4;
t1 = new TShirt();       t1.farbe = "grün";
t2 = t4 = new TShirt(); t2.farbe = "grün";
t3 = new TShirt();       t3.farbe = "blau";
```

	Ausdruck	true	false
a)	`t1 == t2`	☐	☐
b)	`t1 == t3`	☐	☐
c)	`t4 == t2`	☐	☐
d)	`t1.equals(t2)`	☐	☐
e)	`t2.equals(t1)`	☐	☐
f)	`t3.equals(t2)`	☐	☐
g)	`t1.equals(new TShirt())`	☐	☐
h)	`t2.equals(t4)`	☐	☐
i)	`t2.equals(t4.farbe)`	☐	☐
j)	`t2.farbe.equals(t4.farbe)`	☐	☐
k)	`t1.farbe.equals(t2.farbe)`	☐	☐
l)	`t2.farbe.equals(t3.farbe)`	☐	☐
m)	`t2.farbe == t4.farbe`	☐	☐
n)	`t1.farbe == t4.farbe`	☐	☐
o)	`t2.farbe == t3.farbe`	☐	☐
p)	`t1.farbe + "" == t2.farbe`	☐	☐
q)	`t2.farbe + "" == t4.farbe + ""`	☐	☐
r)	`t1.produktionsnummer == t2.produktionsnummer`	☐	☐
s)	`t1.produktionsnummer.equals(t2.produktionsnummer)`	☐	☐

2. Welche Ausgabe produziert folgendes Programm?

```
char x = 'x';
boolean y = false;
String a = "x";
String b = "" + a;
String c = "" + "x";
if (a == b) System.out.print(1);
if (a == c) System.out.print(2);
if (b == c) System.out.print(3);
if (c == "x") System.out.print(4);
if (x + 0 == 'x') System.out.print(5);
if (c.equals(b)) System.out.print(6);
y = (c == "y");
if (y = true) System.out.print(7);
```

(64) ★★★

Modellierung: Implementieren Sie eine Klasse **`Book`** zur Modellierung von Büchern.

Dem Konstruktor müssen in jedem Fall Titel und Autor, optional zusätzlich noch der Preis in Cent (`int`) übergeben werden. Nach der Erzeugung eines Buches sollen Titel und Autor nicht mehr geändert werden können, während der Preis veränderlich ist. Nutzen Sie entsprechend passende Schlüsselwörter. Standardmäßig soll ein Buch kostenlos sein; der Preis darf nicht negativ sein. Initialisieren Sie alle Membervariablen explizit.

Bei der Erzeugung eines `Book`-Objektes soll automatisch eine *eindeutige* Identifizierungsnummer vom Typ `long` zugewiesen werden, welche weder eigenständig vom Nutzer bestimmt, noch nachträglich verändert werden kann und selbst nach dem Löschen des Buches nicht erneut vergeben wird.

Bieten Sie entsprechende *Getter*- und *Setter*-Methoden an, wobei die neuen Werte in den *Settern* vor der Übernahme/Speicherung überprüft werden sollen. Werden einem Konstruktor oder einer solchen Methode ungültige Parameter übergeben, so soll eine `IllegalArgumentException` geworfen werden (Vorgriff auf Kap. 14). Verwenden Sie Modifikatoren an geeigneten Stellen und halten Sie sich an das Konzept des *Information Hidings*, indem Objektvariablen nicht direkt von außen zugreifbar sind.

Überschreiben Sie die von `Object` geerbten Methoden ...

a) `public String toString()`: gibt eine *String*-Repräsentation des Buches in der Form "*<Autor>*`: `*<Titel>* `(EUR `*<Preis>*`)`" zurück. Der Preis soll dabei immer mit genau zwei Nachkommastellen angegeben werden.
   Beispiel: `"Max Muster: Java 4 life (EUR 15,20)"`.

b) `public boolean equals(Object o)`: gibt **`true`** zurück, wenn die Identifikationsnummern der beiden Bücher übereinstimmen (sofern es sich denn um zwei Bücher handelt), sonst **`false`**.

Implementieren Sie außerdem die Methoden ...

c) `public void increasePrice(float euro)`: erhöht den Preis um den als Parameter übergebenen Euro-Betrag (bspw. `1.25f`). Angaben mit mehr als zwei Nachkommastellen können beliebig gerundet werden.

d) `public boolean hasSameAuthor(Book b)`: gibt **`true`** zurück, wenn das als Parameter übergebene Buch denselben Autor hat, sonst **`false`**.

e) `public int authorLetters()`: gibt die Anzahl der Buchstaben des Namens des Autors zurück. Leerzeichen werden *nicht* gezählt. Sie können hierfür bspw. die Methoden `charAt(int)` und `length()` aus der Klasse `String` benutzen.

Des Weiteren soll die öffentliche Klassenmethode `equals(Book, Book)` existieren, welche wie die Objektmethode `equals` auswertet. Entsprechend können und sollen Sie die Auswertung des Wahrheitswertes einfach an die Objektmethode delegieren.

Zusatzfrage: Warum ist die `equals`-Methode in diesem Fall eigentlich überflüssig?

(65) ★★☆

Implementieren Sie basierend auf der Klasse `Book` aus Aufgabe 64 eine Klasse **`Library`** zur Modellierung einer Bibliothek (für Bücher). Zur internen Speicherung der Bücher soll ein Attribut vom Typ `LinkedList<Book>` (→ `import java.util.LinkedList`) angelegt werden, welches im Konstruktor als neue leere Liste initialisiert wird (siehe Hinweis unterhalb). Die Klasse `LinkedList<Book>` stellt die Methoden `boolean add(Book e)`, `boolean remove(Object o)` und `boolean contains(Object o)` bereit, welche Sie für die folgenden Aufgaben nutzen können, um ein Buch in die Liste einzufügen, zu entfernen oder zu prüfen, ob ein Buch enthalten ist. `contains` wertet zu **`true`** aus, wenn das Objekt in der Liste enthalten ist, sonst **`false`**.

Hinweis: Generische Klassen und Collections (→ `LinkedList<Book>`) werden eigentlich erst in den Kapiteln 13 und 16 behandelt. Wirf vor der Bearbeitung dieser Aufgabe ggf. einen Blick auf Seite 115 f.

Um über alle Elemente der Liste zu iterieren, kann man die bereits bekannte *for-each*-Schleife nutzen (→ Kap. 5.5):

```
for (Book b : meineBuecherListe) { /* tue etwas mit b */ }
```

Implementieren Sie die Methoden ...

a) `public void add(Book b)`: nimmt das als Parameter übergebene Buch in die Bibliothek auf (wenn es noch nicht in der Bibliothek enthalten ist).

b) `public Book get(long id)`: sucht in der Bibliothek nach dem Buch mit der Identifikationsnummer `id` und gibt dieses zurück. Existiert ein solches Buch nicht, so soll **`null`** zurückgegeben werden.

c) `public boolean delete(Book b)`: entfernt das Buch `b` aus dem Bibliotheksinventar und gibt **`true`** zurück, wenn die Operation erfolgreich durchgeführt wurde, sonst **`false`**.

d) `public LinkedList<Book> getBooksWrittenBy(String author)`: gibt eine Liste aller Bücher zurück, die von dem gegebenen Autor `author` verfasst wurden.

(66) ★★☆

Die Bibliothek aus Aufgabe 65 soll nun zusätzlich zu Büchern (`Book`) auch DVDs aufnehmen können. Implementieren Sie dazu eine abstrakte Klasse `Medium`, welche von den Klassen `Book` und `DVD` erweitert wird. DVDs verfügen im Gegensatz zu Büchern nicht über einen Autor, stattdessen aber über ein Array von Schauspielern (`String[]`), welches dem Konstruktor als Parameter übergeben werden soll. Alle Medien haben eine eindeutige Identifikationsnummer, d. h. ein Buch kann auch unter keinen Umständen dieselbe ID-Nr. haben wie eine DVD. Vermeiden Sie Redundanz (Code-Duplikate) zwischen den Klassen `Book`, `DVD` und `Medium`, indem Sie Code aus `Book` und `DVD` in der Oberklasse `Medium` zusammenfassen wo möglich.

Passen Sie Ihre vorherige Implementierung von `Library` an, sodass auch DVDs und sonstige Unterklassen von `Medium` aufgenommen werden können (bspw. sollte die `add`-Methode nun die Signatur `add(Medium m)` haben).

(67) ★★☆ Betrachten Sie folgenden Code vor dem Hintergrund, dass Java den *„Call by Value“*-Ansatz verwendet. Welche Ausgabe wird durch den Aufruf der `main`-Methode produziert? Warum können mehrere `set`-Methoden existieren? Wie nennt man dieses Konzept?

```
public class X {
    private int v;

    public X set(X x, int v) {
        x.v = v;
        return x;
    }

    public X set(X x) {
        x = new X();
        return set(x, 2);
    }

    public void set(int v) {
        this.v = v;
    }

    public static void main(String[] args) {
        X x1, x2;
        x1 = x2 = new X();
        X x3 = new X();

        x1.set(1);
        System.out.print(" " + x1.v + x2.v + x3.v);
        x1 = x1.set(x1);
        x3 = x1.set(x2, 3);
        System.out.print(" " + x1.v + x2.v + x3.v);
        x1.set(x2);
        x2.set(4);
        System.out.print(" " + x1.v + x2.v + x3.v);
        x3.set(5);
        x3 = x1.set(x2);
        x3.set(x1, 6);
        System.out.print(" " + x1.v + x2.v + x3.v);
    }
}
```

(68) ★★★

Modellierung: Diskussionsforum

In dieser Aufgabe soll ein Internetforum objektorientiert modelliert werden. Das Forum soll mehreren Benutzern erlauben, neue Beiträge (*Themen*) zu erstellen und *Antworten* zu bereits erstellen Themen zu verfassen. Außerdem ist es bestimmten Nutzern gestattet, ihre Beiträge nachträglich zu verändern oder zu löschen.

Nachfolgend finden Sie Erklärungen zu allen zu implementierenden Klassen. Machen Sie Gebrauch von Vererbung und halten Sie sich an Konzepte wie *Information Hiding* (d. h. Attribute stets `private` mit öffentlicher *Getter*-Methode). Die Signaturen aller erwarteten *Getter*-Methoden sind im Folgenden angegeben. Für die nicht explizit als unveränderlich beschriebenen Attribute sind zudem *Setter*-Methoden zu implementieren. Unveränderliche Variablen sollen stattdessen als `final` definiert werden. Vermeiden Sie Redundanz, indem Sie Daten – soweit möglich – in Oberklassen kapseln.

Benutzer: Wir unterscheiden zwischen *Moderatoren* (**`Moderator`**) sowie *registrierten* (**`RegisteredUser`**) und *anonymen Benutzern* (**`AnonymousUser`**). In jedem Forum existiert zudem immer genau ein *Administrator* (**`Admin`**).

Jeder Benutzer verfügt über einen Namen (`String getName()`) und eine unveränderliche E-Mail-Adresse (`String getEmail()`). Kapseln Sie diese Attribute in einer abstrakten Oberklasse **`User`** mit einem Konstruktor `User(String name, String email)`. Der Administrator speichert sich zusätzlich auch das `Forum` in einem Attribut ab, welches unveränderlich und *nicht* über einen *Getter* zugreifbar sein soll. Der Konstruktor von `Admin` bekommt daher ein `Forum`-Objekt als dritten Parameter übergeben. Die Konstruktoren der anderen Nutzer sind analog zu `User` zu implementieren.

Ergänzen Sie die Klassen, so dass die **`toString()`**-Methode für einen Administrator zu *`Name <E-Mail-Adresse>`* und für alle anderen zu *`Name`* auswertet.

Beiträge: Wir unterscheiden zwischen zwei Arten von Forumsbeiträgen – *Themen* (**`Topic`**) und *Antworten* auf bereits erstellte Themen (**`Reply`**):

Jeder Beitrag verfügt über einen Inhalt (`String getContent()`), einen Verfasser (`User getAuthor()`) und eine über alle Beiträge hinweg eindeutige Identifikationsnummer „ID“ (`int getID()`), welche beim Erzeugen eines Beitragsobjekts automatisch generiert und gespeichert wird (und auch nach dem Löschen dieses Beitrags nicht erneut vergeben wird). Beiträge können als gelöscht markiert werden (`boolean isDeleted()`), werden aber nie echt entfernt (d. h. die boolesche Markierung genügt). Verfasser und ID sollen nach der Initialisierung nicht mehr verändert werden können.

Ein *Thema* verfügt zusätzlich über einen Titel (`String getTitle()`). Eine *Antwort* speichert hingegen zusätzlich ihr zugehöriges Thema (`Topic getRelatedTopic()`), welches nach dem Setzen im Konstruktor nicht mehr verändert werden können soll.

Kapseln Sie die bei beiden Beitragsarten vorhandenen Eigenschaften in der abstrakten Oberklasse **`Post`**. Definieren Sie dort außerdem die abstrakte Methode `boolean isTopic()`, welche **`true`** zurückgibt, falls es sich um ein *Thema* handelt, **`false`** sonst.

Implementieren Sie die folgenden Konstruktoren:

- `Post(String content, User author)`
- `Topic(String title, String content, User author)`
- `Reply(String content, Topic related, User author)`

Ergänzen Sie die `User`-Klasse nun um die abstrakte Methode **`canModifiy(Post)`**, die **`true`** zurückgibt, falls der jeweilige Benutzer den übergebenen Post verändern darf, **`false`** sonst. Implementieren Sie diese Methode in den Unterklassen wie folgt:

- Ein anonymer Benutzer kann keine Änderungen durchführen.
- Ein registrierter Benutzer darf lediglich seine eigenen Beiträge verändern. Bereits gelöschte Beiträge kann er zudem nicht verändern.
- Ein Moderator kann grundsätzlich alle Beiträge verändern. Ausgenommen sind Beiträge von Administratoren sowie bereits gelöschte Beiträge.
- Ein Administrator kann alle Beiträge verändern.

Sie können sich jetzt der soeben implementierten `canModifiy`-Methode bedienen, um die Methoden **`deletePost(Post)`** und **`editPost(Post,String)`** hinzuzufügen. `deletePost` löscht den als Parameter übergebenen Beitrag (durch Setzen des `isDeleted`-Flags), falls der Nutzer diesen Beitrag verändern darf. Anderenfalls soll eine Fehlermeldung ausgegeben werden. `editPost` setzt den Beitragsinhalt analog dazu (also unter der gleichen Voraussetzung) auf den im zweiten Parameter übergebenen Wert.

Administratoren sollen weiters über die Methode **`restorePost(Post)`** einen Beitrag als *„nicht gelöscht“* markieren können (Gegenteil zu `deletePost`). Außerdem haben Administratoren mittels **`editPostTitle(Topic,String)`** zusätzlich die Möglichkeit, den Titel eines *Thema*s nachträglich zu ändern.

Forum: Ein **`Forum`** verwaltet alle Beiträge und Benutzer in einem `Post`- bzw. `User`-Array. Der Administrator des Forums wird innerhalb des Konstruktors `Forum(String adminName, String adminMail)` erzeugt und in einem Attribut gespeichert (`Admin getAdmin()`). Er kann nachträglich nicht geändert werden und wird ebenfalls in das `User`-Array aufgenommen. Implementieren Sie die Methoden:

(1) `boolean addUser(User)` fügt den als Parameter übergebenen Benutzer hinten in das `User`-Array ein. Dazu muss das Array um genau einen Platz vergrößert werden. **`null`**-Werte und anonyme Benutzer sollen nicht eingefügt werden! Existiert bereits ein Nutzer mit derselben E-Mail-Adresse, so soll der übergebene Nutzer nicht ins Array aufgenommen und stattdessen eine Fehlermeldung ausgegeben werden. Die Methode gibt zurück, ob der Nutzer eingefügt wurde oder nicht.

(2) `void addPost(Post)` fügt den als Parameter übergebenen Beitrag hinten in das `Post`-Array ein. Dazu muss das Array um einen Platz vergrößert werden.

optional

(3) `String topicToString(Topic)` wandelt das als Parameter übergebene Thema in einen String um. Der Rückgabestring muss das *Thema* und *alle Antworten*, die sich auf dieses Thema beziehen, enthalten. Die genaue Formatierung ist Ihnen überlassen. Gelöschte Antworten sollen nicht im Ergebnis enthalten sein.

(4) `void printForum()` gibt alle sichtbaren (also nicht gelöschten) Themen des Forums auf der Konsole aus. Einzelne Themen können mit (3) formatiert werden.

Definieren Sie ein `enum` **`UserRole`** mit den Werten *`ADMIN`*, *`MODERATOR`* u. *`PARTICIPANT`*. Implementieren Sie in `Admin` zuletzt die Methode `User` **`createUser(String name, String email, UserRole role)`**, welche einen Benutzer des gegebenen Typs erzeugt, in das Forum einfügt und zurückgibt. Für ungültige `role`-Werte (wie **`null`**) soll ein anonymer Nutzer erzeugt werden. Geben Sie eine Fehlermeldung aus (und **`null`** zurück), falls ein Admin hätte erzeugt werden soll (da nur ein Admin pro Forum erlaubt ist).

(69) ★☆☆	**Gemischte Aussagen** zu Klassen und Interfaces	Wahr	Falsch
a)	Objektvariablen/Attribute können statische Variablen innerhalb derselben Klasse verschatten.	☐	☐
b)	Auf verschattete Membervariablen kann über den Klassennamen zugegriffen werden.	☐	☐
c)	Auf verschattete statische Variablen kann über das Schlüsselwort `this` zugegriffen werden.	☐	☐
d)	Ein formaler Parameter kann eine mit dem Modifier `public` deklarierte statische Variable verschatten.	☐	☐
e)	Eine Klasse kann unter Umständen `private` sein.	☐	☐
f)	Eine *top-level* Klasse (keine innere Klasse) darf weder `private` noch `protected` sein.	☐	☐
g)	Konstruktoren können überladen werden.	☐	☐
h)	Eine in einer Datei als `public` deklarierte *top-level* Klasse muss immer denselben Namen haben wie die Datei.	☐	☐
i)	In einer Datei *Test.java* können beliebig viele *top-level* Klassen ohne Zugriffsmodifier definiert werden.	☐	☐
j)	Eine Klasse kann gleichzeitig beliebig viele Interfaces implementieren und von einer Klasse erben.	☐	☐
k)	In der Datei *Test.java* muss sich eine öffentliche Klasse `Test` (oder optional mit Typparameter) befinden.	☐	☐
l)	Methoden in einem Interface können überladen werden.	☐	☐
m)	Eine innere Klasse kann eine beliebige Sichtbarkeit haben (`public`, `private`, `protected` oder *package-private*).	☐	☐
n)	Ein Interface kann ebenso (wie eine innere Klasse) innerhalb einer Klasse definiert werden.	☐	☐
o)	Eine innere Klasse darf denselben Namen haben wie eine in einer beliebigen anderen Datei definierte Klasse, selbst wenn beide `public` sind.	☐	☐
p)	Mit dem Statement `new A.B()` kann (von außerhalb) eine öffentliche innere, nicht-statische Klasse `B` in der öffentlichen *top-level* Klasse `A` instanziiert werden.	☐	☐
q)	Eine abstrakte Klasse kann private innere Klassen enthalten.	☐	☐
r)	Innerhalb eines Interfaces können Klassen und Interfaces definiert werden, sofern diese nicht `private` oder `protected` sind.	☐	☐

s)	Folgender Code kompiliert:
	`public void method() {`
	`  if (true)`
	`  int x = 5;`
	`}`
		☐	☐

t)	Innere Klassen können innere Klassen besitzen.	☐	☐
u)	Finale Methoden werden vererbt.	☐	☐
v)	Methodendeklarationen ohne Verwendung des Schlüsselwortes `abstract` sind in Interfaces möglich.	☐	☐
w)	In einem Interface können Variablen angelegt werden.	☐	☐
x)	Eine Methodendeklaration der Form „`public int x();`“ ist innerhalb einer abstrakten Klasse erlaubt.	☐	☐
y)	Von einer finalen Klasse kann man nicht erben.	☐	☐
z)	Eine abstrakte Klasse kann als `final` deklariert werden.	☐	☐
A)	In einem Interface dürfen statische Methoden implementiert werden (mit Rumpf).	☐	☐
B)	Eine innere Klasse kann von einer anderen inneren Klasse erben.	☐	☐
C)	Sei `A` eine abstrakte Klasse und `C` ein Interface, dann ist folgende Klassendefinition möglich: `class B extends A implements C`	☐	☐
D)	Abstrakte Klassen können nicht-abstrakte („normale“) Klassen erweitern, also von ihnen erben.	☐	☐

E)	Folgender Code kompiliert:
	`public int method() {`
	`  {`
	`  return 5;`
	`  }`
	`}`
		☐	☐

F)	Interfaces können von abstrakten Klassen erben.	☐	☐
G)	Abstrakte Klassen können von Interfaces erben.	☐	☐

H)	Diese Art der Überladung ist möglich:
	`public void m(int x, int y) { }`
	`public void m(int x, char y) { }`
		☐	☐

I)	Interfaces können von Interfaces erben.	☐	☐
J)	Abstrakte Klassen können Interfaces implementieren.	☐	☐
K)	Innere Klassen können `final` oder `abstract` sein.	☐	☐

L)	Interfaces können Interfaces implementieren.	☐	☐
M)	Klassen können abstrakte Klassen implementieren.	☐	☐
N)	Generische* Klassen können Interfaces implementieren.	☐	☐

O)	Folgender Code kompiliert:
	`class A { final int x = 0; }`
	`class B extends A { int x = 5; }`
		☐	☐

P)	Eine finale Methode kann überladen werden.	☐	☐
Q)	Folgender Code kompiliert: `int method() { if (false) { int x = 5; } return x; }`	☐	☐
R)	Eine statische Variable kann als `final` deklariert werden.	☐	☐
S)	Interfaces können generisch* sein. * Vorgriff auf Kap. 13	☐	☐
T)	Folgender Code kompiliert: `class Foo<T, P> { void m(P x) { x.toString(); } }`	☐	☐
U)	Folgender Code kompiliert: `class Foo<R> { public static R m(R r) { return r; } }`	☐	☐
V)	Von Klassen wie `String` und `Integer` kann man erben.	☐	☐
W)	Variablen können den Typ `void` haben.	☐	☐

X)	Folgender Code kompiliert:
	`class W {}`
	`class Z<V extends W> { }`
		☐	☐

Y)	Einer Methode, die als Parameter den Typ `A` erwartet, können Objekte der Klasse `A` und Objekte einer Unterklasse von `A` übergeben werden.	☐	☐
Z)	Eine abstrakte Klasse, die (mittels des Statements **`implements`**) ein Interface implementiert, muss keine Implementierungen für die Methodendefinitionen des Interfaces anbieten.	☐	☐

α)	Klassen können in Methoden definiert werden.	☐	☐
β)	Eine Methode einer in einer Methode definierten Klasse kann auf die lokalen Variablen und Parameter der Methode, in der die Klasse definiert ist, zugreifen, sofern diese Variablen unveränderlich (`final`) sind.	☐	☐
γ)	In Interfaces können Konstruktoren definiert werden.	☐	☐
δ)	In abstrakten Klassen können Konstruktoren definiert werden.	☐	☐
ε)	Wird in einer Klasse `X` eine Klassenvariable (statische Variable) `x` ohne Zugriffsmodifikator deklariert (z. B. als `static int x;`), so hat jedes Objekt der Klasse `X` und alle Objekte aller Unterklassen von `X` Zugriff auf diese Variable.	☐	☐
ζ)	Wird in einer nicht-statischen Methode eines Objekts des Typs `X` eine in der Klasse `X` statisch deklarierte Variable (Klassenvariable) verändert, so hat das für andere Instanzen dieser Klasse keine Auswirkung (d. h. deren Klassenvariable bleibt gleich).	☐	☐
η)	Folgender Code kompiliert: `class A { public interface B { } interface C extends B { } }`	☐	☐
θ)	Jede Methode eines Interfaces ist `public`.	☐	☐
ι)	Statische Methoden in Interfaces oder abstrakten Klassen können abstrakt sein (sofern sie über keinen Rumpf verfügen). Beispiel: `static abstract int m();`	☐	☐

κ)	Betrachten Sie folgenden Codeauszug:
	`class A { public int x; public A(int x) { this.x = x; } }`
	`public static void f(A a) { a = new A(1); }`
	Folgt nach der Deklaration von `A a = new A(0);` der Aufruf `f(a)`, so hat `a.x` anschließend den Wert `1`.
		☐	☐

λ)	Eine innere Klasse `class Foo` kann statische Variablen oder Methoden definieren.	☐	☐
μ)	Folgende Klasse kompiliert: `class A { final int a; void m() { a = 1; } }`	☐	☐

ν) Klassennamen müssen großgeschrieben werden. ☐ ☐

ξ) Konstruktoren können `static` sein. ☐ ☐

ο) Konstruktoren können `final` sein. ☐ ☐

π) Abstrakte Methoden können `static` sein. ☐ ☐

ρ) Abstrakte Methoden können `final` sein. ☐ ☐

σ) Gegeben sei folgende Klasse (in einer Datei *`Private.java`*):

```
public class Private {
    private Private() {}
}
```

☐ ☐

Die Klasse kann von außen *nicht* instanziiert werden,
d. h. `new Private();` ergibt einen Compiler-Fehler.

τ) Folgende Klasse kompiliert *nicht*:

```
public class X {
    public static int m(int a, long b) {
        return m(b, a);
    }
    public static int m(long a, int b) {
        return m(b, a);
    }
}
```

☐ ☐

υ) Statische Methoden können *überschrieben* werden. ☐ ☐

φ) Statische Methoden können `final` sein. ☐ ☐

χ) Statische Methoden können *überladen* werden. ☐ ☐

ψ) Gegeben sei die Klasse `X` aus Teilaufgabe τ).

- Folgender Aufruf kompiliert *nicht*: `X.m(0, 0d);` ☐ ☐
- Folgender Aufruf kompiliert *nicht*: `X.m(0, 0);` ☐ ☐

ω) Folgender Code kompiliert:

```
public class X {
    int m() { return x; }
    int x;
}
```

☐ ☐

Δ) Folgende Methode kompiliert:

```
int m(boolean b) {
    int x;
    if (b)
        x = 1;
    return x;
}
```

☐ ☐

(70) ★★☆ **Gemischte Fragen** zu Java: Stack und Heap

		Wahr	Falsch
a)	Der Speicher ist begrenzt, daher kommt es bei der Erzeugung von zu vielen Objekten zu einem `StackOverflowError`.	☐	☐
b)	Objekte werden auf dem Heap gespeichert.	☐	☐
c)	Der Java-Stack ist nach dem FIFO-Prinzip aufgebaut.	☐	☐
d)	Lokale Variablen werden auf dem Heap gespeichert.	☐	☐

e)
```
public int a(int b) {
    if (b <= 0) return b;
    if (b == 0) return 1 / b;
    return a(b--);
}
```

Der Aufruf $a(b)$ führt für ...

... $b < 0$ *nie* zu einem Fehler.	☐	☐
... $b > 0$ *nie* zu einem Fehler.	☐	☐
... $b > 0$ *immer* zu einem Fehler.	☐	☐
... $b = 0$ *eventuell* zu einem Fehler.	☐	☐

f)
```
public static String c(int d) {
    String out = "x";
    if (d <= 0) {
        while (d < 0) {
            out = out + out;
            d++;
        }
        return out;
    }
    return out + c(d-1);
}
```

Der Aufruf $c(d)$ führt für ...

... $d < 0$ *nie* zu einem Fehler.	☐	☐
... $d < 0$ *immer* zu einem Fehler.	☐	☐
... $d > 0$ *nie* zu einem Fehler.	☐	☐
... $d > 0$ *immer* zu einem Fehler.	☐	☐
... $d = 0$ *nie* zu einem Fehler.	☐	☐

g) Folgender Code ...

... wertet beim Aufruf von `f()` zu 6 aus.	☐	☐
... ändert den Wert der Membervariable `x`.	☐	☐
... führt beim Aufruf von `f()` zu einem Laufzeitfehler.	☐	☐
... kompiliert nicht.	☐	☐

```
public class DynStat {
    int x;
    static void f() {
        this.x = 6;
    }
}
```

(71) ☆☆☆

Gegeben sei folgendes Interface:

```
interface X {
    int x();
    public abstract void y(X x);
    static void z() { /* do sth */ };
}
```

Kreuzen Sie kompilierende Code-Fragmente an.

☐
```
public class A extends X { }
```

☐
```
class B implements X { }
```

☐
```
abstract class C implements X { }
```

☐
```
interface D implements X { }
```

☐
```
interface E extends X { }
```

☐
```
class F implements X {
    public int x() { return 0; };
    public void y(X x) { };
}
```

☐
```
class G extends Object implements X {
    public int x() { return 0; };
    public void y(X x2) { return; };
    int z() { return 1; };
}
```

☐
```
abstract class H implements X {
    private int x() { return 1; }
    public void y(X x) { };
}
```

☐
```
class I implements X {
    public int x(int i) { return x(++i); }
    public int x() { return x(1); }
    public void y(X x) { }
}
```

☐
```
class J implements X {
    int x() { return 0; }
    public void y(X x) { System.out.println("J"); }
    public static void z() { }
}
```

☐
```
abstract class K implements X {
    public abstract int x();
    public abstract void y(X k);
}
```

☐
```
public class L {
    class L2 implements X {
        public int a;
        public int x() { return a; }
        public void y(X x) { }
    }

    public int a(X a) {
        return ((L2) a).a;
    }
}
```

☐
```
abstract class M implements X {
    public int x();
}
```

☐
```
abstract class N implements X {
    abstract void y(N n);
    abstract void z();
}
```

☐
```
public abstract class O implements X {
    abstract void y(X x);
    public int x() { return 1; }
}
```

☐
```
abstract class P1 implements X {
    public int x() { return 0; }
}

class P2 extends P1 {
    public void y(X x) { };
}
```

☐
```
abstract class Q1 implements X {
    public int x() { return 0; }
    public void y(Q1 x) { }
}

class Q2 extends Q1 implements X {
    public void y(X x) { y(this); }
}
```

☐
```
abstract class R implements X {
    abstract void r();
    public static void y(X x);
}
```

☐
```
public interface S extends X {
    int w();
    static void z() { };
}
```

☐
```
interface T1 extends X {
    int x();
}
class T2 implements T1 {
    public int x() { return 0; }
}
```

☐
```
class U implements X {
    public int x() { return 1; }
    public void y(X x) { y((X)this); }
}
```

☐
```
abstract class V implements X {
    public int x() { return this.x(); }
}
class V2 extends V {
    public void y(V x) { }
}
```

☐
```
public abstract class W implements X {
    class W2 extends W {
        public int x() { return 0; }
        public void y(X x) { }
        public int y(W2 x) { return x(); }
    }
}
```

☐
```
abstract class x implements X {
    public abstract char x();
    void y() {}
}
```

☐
```
abstract class Y implements X {
    protected int x = 5;
    int x() { return x; }
}
```

☐
```
interface X2 {
    public abstract int x();
}
class Z implements X, X2 {
    public int x() { return -1; }
    public void y(X x) { }
}
```

(72) ★☆☆ Größere Programmieraufgabe: **Cryptograph**

In dieser Aufgabe sollen Sie einen einfachen Verschlüsselungsalgorithmus für Strings implementieren. Wie Sie wissen sind Zeichen nichts anderes als Zahlen im Wertebereich von 0 bis 65535. Die Grundidee ist nun, jedes x-te Zeichen des Textes um einen Wert y zu erhöhen, wobei es eine beliebige Anzahl an (x, y)-Paaren gibt (je mehr desto stärker die Verschlüsselung). Der Gesamtschlüssel kann somit als eine Menge solcher Tupel angegeben werden.

Implementieren Sie die Klasse `Tuple` zur Speicherung eines (x, y)-Paars. x und y werden über den Konstruktor initialisiert und sollen danach nicht mehr verändert werden können. Die beiden Werte sollen jeweils nur 8 Bit groß sein (Größe von `byte`), allerdings vorzeichenlos (nicht negativ) sein. Nutzen Sie daher am besten den Typ `char`. Verwenden Sie passende Modifier. Schreiben Sie die folgenden Methoden:

(a) `public char combinedValue()`: x und y sollen in einem einzigen Wert kombiniert werden, x in den oberen und y in den unteren 8 Bit des Rückgabewerts.
Tipp: Werfen Sie einen Blick auf die binären Operatoren << und `|`.

(b) `public void applyTo(char[] chars, boolean encode)`: Erhöht jeden x-ten Wert im Array `chars` um y, falls für `encode` **`true`** übergeben wird. Ist `encode` **`false`**, so wird das entsprechend rückgängig gemacht (Subtraktion). Beachten Sie, dass x und/oder y 0 sein können. Behandeln Sie diesen Fall sinnvoll.

(c) `public static Tuple createTupleFromChar(char combined)`: Erzeugt (umgekehrt zu `combinedValue`) aus einem 16-Bit-Wert ein Tupel, wobei die oberen 8 Bit für x und die unteren 8 Bit für y genutzt werden sollen.

Implementieren Sie nun die Klasse `Key` zur Repräsentation eines ganzen Schlüssels. Im Konstruktor wird die Schlüssellänge übergeben und Platz für entsprechend viele (x, y)-Paare reserviert (mittels eines Arrays). Schreiben Sie die folgenden Methoden:

(d) `public String toString()`: Gibt den Schlüssel als String zurück, indem die kombinierte Darstellung aller enthaltenen Tupel aneinandergereiht werden (genau ein Zeichen pro Tupel).

(e) `public void applyTo(char[] chars, boolean encode)`: Wendet jedes (x, y)-Paar auf das Array `chars` an, wie bereits in b) beschrieben.

(f) `public static Key createKeyFromString(String str)`: Erzeugt aus einem String einen Key, indem aus jedem Zeichen jeweils ein Tupel erzeugt wird.

Implementieren Sie zu guter Letzt die Klasse `Cryptograph` mit den privaten Attributen `char[] text` und `Key key` und zugehörigen *Gettern*. Der Konstruktor `Cryptograph(String text, String key)` speichert den Text im Array und erzeugt ein Schlüssel-Objekt aus dem String. Der Konstruktor `Cryptograph(String text, int keyLength)` erzeugt hingegen einen zufälligen Schlüssel der Länge `keyLength`, während `Cryptograph(String text)` diesen mit der Schlüssellänge gleich 20% der Textlänge aufruft. Schreiben Sie dazu eine Methode `static String createRandomString(int length)` zur Generierung eines zufälligen Strings. `void encode()` verschlüsselt den Text `text` mit `key`, `void decode()` führt die Entschlüsselung durch.

13. Generische Klassen

Bei einigen Klassen macht es Sinn, sog. *Generics* zu verwenden. Dies erlaubt uns die Parametrisierung von Klassen (oder Interfaces) mit anderen Klassen. So erhält die Klasse `LinkedList` (verkettete Liste) bspw. den *Typparameter* `E`; man schreibt dann `LinkedList<E>`.

Ähnlich wie bei einem Konstruktor muss bei der Erzeugung (bzw. bereits bei der Deklaration) eines Objekts einer generischen Klasse dann ein konkreter Wert eingesetzt werden, nur sind das bei Typparametern anders als bei „normalen" Parametern dann eben keine Werte/Objekte, sondern Klassen. Die Klasse `LinkedList` ist aus dem Grund parametrisiert, damit man angeben kann, welchen Typ die Elemente der jeweiligen Liste haben sollen. So kann man bspw. eine Liste von Strings (`LinkedList<String>`), aber auch eine Liste von Büchern (`LinkedList<Book>`) erzeugen, wobei in eine Liste von Strings natürlich keine Bücher (und umgekehrt) eingefügt werden können, d. h. in einer Liste von Büchern sind nur Bücher.

Für den Typparameter `E` (man könnte ihn nennen wie man will; auch `KatzHundMaus` wäre möglich) wird also ein konkreter Wert (`String` bzw. `Book`) eingesetzt. Eingesetzt werden können alle Klassennamen und Interfaces, nicht aber primitive Datentypen. Man muss ggf. auf die zugehörige Wrapper-Klasse zurückgreifen (bspw. `LinkedList<Integer>` für `int`).

Kurz: Ein Typparameter (z. B. `T`) ist ein Platzhalter für einen Klassen-/Interfacenamen (= Typ).

Auch mehrere Typparameter sind möglich. Beispielklasse für Paare:

```
public class Pair<K, V extends Number> {
    private final K key;
    private V value;

    public Pair(K key, V value) {
        this.key = key;
        this.value = value;
    }

    public void setValue(V v) {  value = v;    }
    public V getValue()       {  return value; }
    public K getKey()         {  return key;   }
}
```

Beispiel für ein `String`-`Integer`-Paar:

```
Pair<String, Integer> p = new Pair<String, Integer>("Test", 1);
```

Der sog. *Diamond-Operator* <> erlaubt das Weglassen des Typs bzw. der Typen bei der Instanziierung (da diese nach der Deklaration eindeutig sind). Der Ausdruck oberhalb ist äquivalent zu:

```
Pair<String, Integer> p = new Pair<>("Test", 1);
```

Das „`extends Number`" ist in diesem Beispiel eine zusätzliche Angabe, die `V` auf numerische Typen (Unterklassen von `Number` wie bspw. `Long` oder `Double`) einschränkt. Genauso können Interfaces angegeben werden. Mehrere Interfaces und Oberklasse: `T extends C & I1 & I2`

Außerdem können generische Klassen als generische Typen benutzt werden. Eine Liste von Paaren könnte bspw. wie folgt definiert werden. `p` könnte man mittels `liste.add(p);` einfügen.

```
LinkedList<Pair<String, Integer>> liste = new LinkedList<>();
```

(73) ★☆☆ Kreuzen Sie die kompilierenden Code-Abschnitte an.

☐
```
class A<T> {   public A<T>() {}   }
```

☐
```
class B<T> {   public B(T t) {}   }
```

☐
```
class C<> {}
```

☐
```
class D<T> {
    public D() {  new D<Object>();  }
}
```

☐
```
class E<Example> {
    private Example e;
    public E(Example e) { e = e; }
}
```

☐
```
class F<T> {   public F() { new T(); }   }
```

☐
```
class G<T, U, M> {}
```

☐
```
class H1<T> {}
class H2<H1> {}
```

☐
```
class I<I<T>> {}
```

☐
```
class J1<Test> {}
class J2<T> extends J1<Float> {}
```

(74) ★★☆ Die Klassen `DVD`, `Book` und `Medium` seien wie in Aufgabe 64 und 66 beschrieben implementiert. Implementieren Sie nun eine Klasse **`Collection`**, die sich ähnlich verhält wie die Klasse `Library` aus Aufgabe 65, welche die gemeinsame Speicherung von DVDs und Büchern ermöglicht hat. Anders als bei `Library` sollen jetzt neben Sammlungen von Medien (sowohl DVDs als auch Bücher aufnehmbar) aber auch reine DVD- und reine Bücher-Sammlungen möglich sein, wobei dann entsprechend nur DVDs bzw. nur Bücher enthalten sein sollen. Führen Sie für `Collection` daher einen Typparameter `M` ein, für den es möglich sein soll, sowohl `Medium` als auch alle Unterklassen davon einzusetzen.

1) Implementieren Sie wieder die öffentlichen Methoden ...
   a) `void add(M m)`: nimmt das als Parameter übergebene Objekt in die Sammlung auf (wenn es noch nicht in der Sammlung enthalten ist).
   b) `M get(long id)`: sucht in der Bibliothek nach dem Objekt mit der Identifikationsnummer `id` und gibt dieses zurück. Existiert ein solches Objekt nicht, so soll **`null`** zurückgegeben werden.
   c) `boolean delete(M m)`: entfernt das Objekt `m` aus der Sammlung und gibt **`true`** zurück, wenn die Operation erfolgreich durchgeführt wurde, sonst **`false`**.

2) Statt der Methode `getBooksWrittenBy` sollen Sie nun eine öffentliche Methode mit der Signatur `getObjectsWithFirstLetter(char initial)` implementieren, welche eine Liste aller in der Sammlung enthaltenen Objekte zurückgibt, deren Titel mit dem Buchstaben `initial` beginnt.

(75) ★☆☆

Gegeben seien folgende Klassen:

```
class X<T extends A> {
    public X()    { System.out.println(T.stat()); }
    public X(T t) { System.out.println(t.type()); }
}

class A {
    public static String stat() { return "A"; }
    public String type()        { return "A"; }  }
class B extends A {
    public static String stat() { return "B"; }
    public String type()        { return "B"; }  }
class C extends A {
    public String type()        { return "C"; }  }
class D extends B {
    public String type()        { return "D"; }  }
```

Kreuzen Sie an, welche der darunter genannten Code-Auszüge kompilieren und geben Sie für diese die Ausgabe nach Ausführung an.

☐ `X<A> a = new X<A>();`

☐ `A b0 = new A();  X<A> b = new X<>(b0);`

☐ `A c0 = new A();  X<B> c = new X<B>(c0);`

☐ `X<A> d = new X<>(new D());`

☐ `X<C> e = new X<C>();`

☐ `X<T> f = new X<>();`

☐ `B g0 = new B();  X<B> g = new X<>(g0);`

☐ `X<A> h = new X<C>();`

☐ `X<D> i = new X<D>(new D());`

☐ `X<C> j = new X<>(new B());`

☐ `A k0 = new C();  X<C> k = new X<>(k0);`

☐ `new X<A>(new C());`

☐ `X<Object> m = new X<>();`

☐ `class N<T> extends X<A> {};  new N<C>();`

☐ `class O<T extends A> extends X<T> {};  new O<A>();`

☐ `class P<T extends B> extends X<T> {}`

☐ `class Q<T> extends X<T> {}`

☐ `class R<T extends B> extends X<T> {};  new R<>();`

☐ `class S<T> extends X<A> {};  X<A> s = new S<>();`

☐ `class T<T extends D> extends X<T> {}`

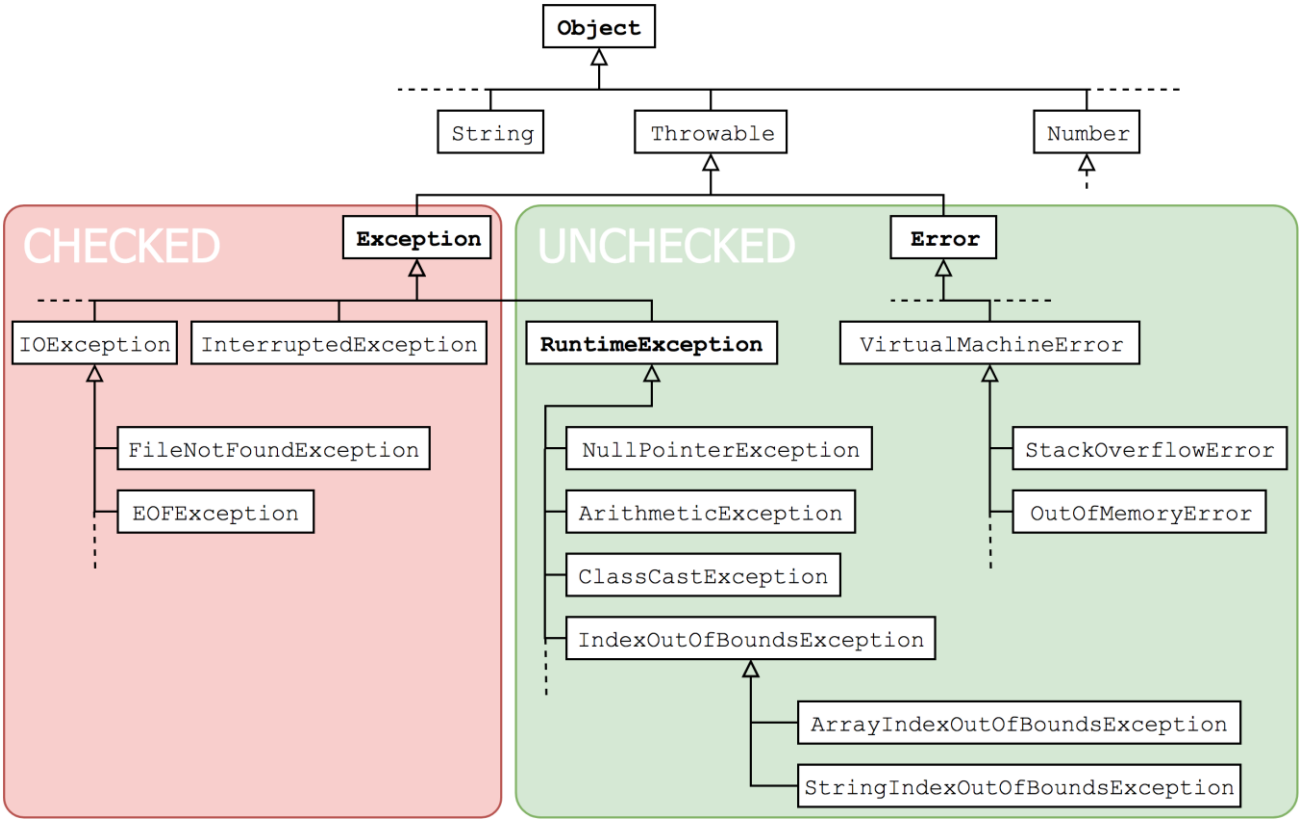
☐ `class U<K extends A> extends X<D> {};  X<B> u = new U<B>();`

☐
```
class V<T extends A> {
    public V() {
        System.out.println(new X<T>());
    }
}
new V<>();
```

14. Ausnahmen & Fehlerbehandlung

Zur Laufzeit eines Programms auftretende Fehler werden in Java durch Objekte des Typs `Throwable` repräsentiert und „Laufzeitfehler“ genannt. Wird ein solcher Fehler erzeugt, so sagt man: „Eine *Exception wird geworfen*“ (→ `throw`). Grundsätzlich unterscheiden wir zwischen zwei Fehlerkategorien, den *checked* und *unchecked* Exceptions.

Hier ein kleiner Auszug aus der Java-Standard-Bibliothek. Vielleicht findest du in diesem Schaubild auch den ein oder anderen Übeltäter, der dich schon so manchen Nerv gekostet hat.



Am häufigsten begegnen uns ***unchecked Exceptions*** (rechts im Bild):

- `RuntimeException`s treten i. d. R. durch unsaubere Programmierung auf, d. h. der Programmierer hat nicht jeden möglicherweise eintretenden Fall (z. B. jede erdenkliche Variablenkonfiguration) beachtet. Eine `ArrayIndexOutOfBoundsException` kann bspw. vermieden werden, indem der Programmierer vor jedem Array-Zugriff im Code sicherstellt, dass der Index gültig ist. Eine `ArithmeticException` kann bspw. bei einer Ganzzahldivision durch 0 auftreten, daher sichert man sich diesbezüglich zuvor ab: Wenn der Divisor 0 ist, dann ... (tue irgendetwas anderes statt der Division). `RuntimeException`s gilt es zu vermeiden, daher sollte man sie nicht über ein `try-catch` fangen (gleich mehr dazu).
- `Error`s sind schwerwiegendere Fehler, die sich häufig auf die Ausführungsumgebung oder Maschine beziehen und normalerweise ebenfalls nicht gefangen werden sollten. So tritt der `OutOfMemoryError` bspw. auf, wenn kein Heap-Speicherplatz mehr zur Verfügung steht.

„*Unchecked*“ bedeutet, dass das mögliche Auftreten eines solchen Fehlers nicht explizit angegeben bzw. diese Fehler nicht gefangen werden müssen (aber können (aber nicht sollen)).

Checked Exceptions (links im Bild) sind alle Unterklassen von `Exception` (und diese selbst) außer `RuntimeException`. Eine Methode, die einen solchen Fehler produzieren könnte, muss im Methodenheader mit einer entsprechenden `throws`-Klausel gekennzeichnet sein (wenn sie diese Exception nicht selbst mittels *try-catch* fängt), bspw.:

```
public void someMethod() throws IOException { ... }
```

Beim Aufruf von `someMethod` müssen wir nun damit rechnen, dass eventuell eine `IOException` auftritt (das schließt alle Unterklassen wie `FileNotFoundException` ein). Sicher nicht auftreten könnte hingegen bspw. eine `InterruptedException`. Es könnten aber auch mehrere Fehlerklassen angeben werden:

```
void anotherMethod() throws MyExc1, MyExc2 { ... }
```

`MyExc1` und `MyExc2` seien dabei selbst definierte Unterklassen von `Exception`, z. B.:

```
class MyExc1 extends Exception {
  // Beliebige Konstruktoren:
  public MyExc1(String message) {
    super(message);
  }
  public MyExc1() {
    super();
  }
}
```

```
class MyExc2 extends IOException {
  private int storeSomething;
  public MyExc1(String msg, int v) {
    storeSomething = v;
    super(msg);
  }
  public MyExc1(int val) {
    this("Mein Fehlercode: " + val);
  }
}
```

Jede Methode, in der `anotherMethod` aufgerufen wird, muss nun entweder ebenfalls eine entsprechende `throws`-Klausel besitzen oder die evtl. auftretenden Exceptions mit Hilfe eines `try-catch`-Statements fangen:

```
try {                          // Tue etwas, wo eine Exception auftreten könnte:
    ...
    anotherMethod();
    ...
} catch (MyExc1 ex) {          // Wenn MyExc1 auftritt wird das hier ausgeführt:
    ...
} catch (MyExc2 ex) {          // Wenn MyExc2 auftritt wird das hier ausgeführt:
    ...                        // z. B. ex.printStackTrace();
} finally {                    // Optional! Wird immer ausgeführt (auch vor return):
    ...
}
```

Möchte man für mehrere verschiedene Exceptions das gleiche tun, kann man diese in einem `catch` zusammenfassen: `catch (MyExc1 | MyExc2 ex) { ... }`

Beachte, dass die Reihenfolge der `catch`-Blöcke relevant ist, da nach dem Auftreten einer Exception immer der erste passende `catch`-Block (und nur dieser!) ausgeführt wird. Dies spielt eine Rolle, wenn Exception-Klassen voneinander erben. Speziellere Exceptions müssen immer zuerst gefangen werden, d. h. z. B. ein `catch(RuntimeException e)`-Block muss vor einem `catch(Exception e)`-Block kommen, da `RuntimeExceptions` auch über beide Blöcke gefangen werden würden. Ein `catch(MyEx e)` fängt also alle Exceptions der Klasse `MyEx` sowie alle Unterklassen von `MyEx`. Es wird pro `try` immer maximal ein `catch`-Block ausgeführt.

Tritt eine Ausnahme auf, die nicht durch ein `catch` gefangen wird, so wird kein `catch`-Block ausgeführt und die Ausnahme bleibt unbehandelt (wie sonst auch, d. h. roter Fehlertext und das Programm stoppt). Tritt innerhalb eines `catch`- oder `finally`-Blocks wiederum eine Exception auf, so wird diese nicht von ggf. nachfolgenden `catch`-Blöcken gefangen, sondern bleibt ebenfalls unbehandelt. Es wäre dann ein weiteres `try-catch` innerhalb des `catch`/`finally` nötig.

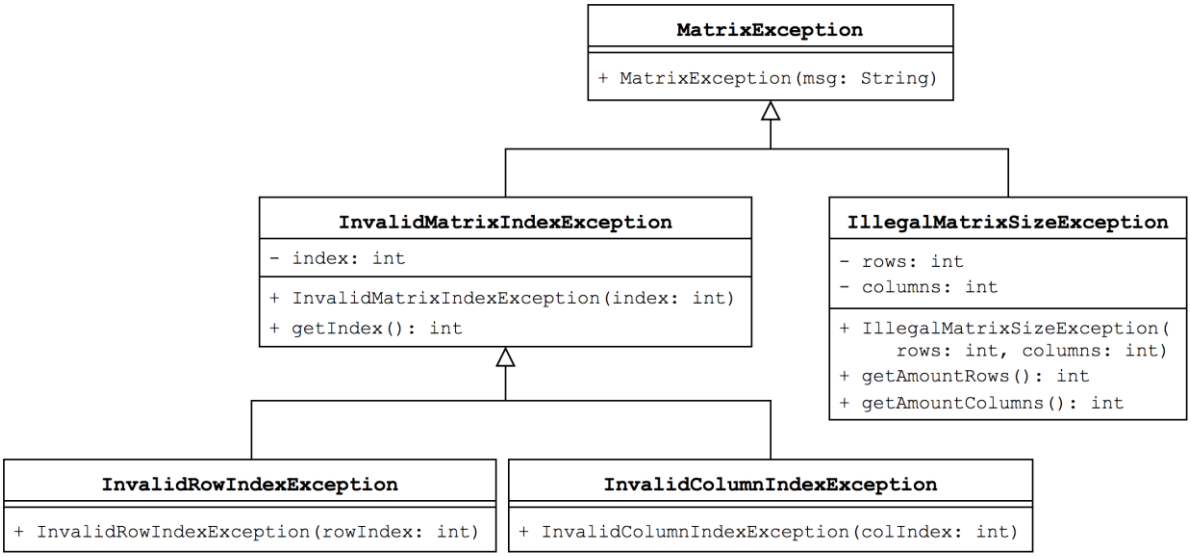
Der **`finally`**-Block folgt optional im Anschluss an alle `catch`-Blöcke. Es kann nur maximal einen `finally`-Block geben. Dieser wird immer ausgeführt! Einzige Ausnahme: Die JVM bricht zusammen. Das ist z. B. bei `System.exit()` der Fall.

Eine Exception kann wie folgt erzeugt und geworfen werden: `throw new MyExc1();`

(76) ★★☆

Für eine Matrix-Implementierung sollen Sie eine kleine, simple Hierarchie von Exceptions anlegen, die jeweils in einem bestimmten Fehlerfall geworfen werden können.

Erstellen Sie zunächst eine Klasse `MatrixException`, welche von `RuntimeException` erbt und als Superklasse für die verschiedenen Matrix-Exceptions dient. Sie soll einen Konstruktor bereitstellen, dem ein Fehlertext übergeben wird, welcher im Konstruktor wiederum an die Oberklasse von `MatrixException` übergeben wird (sodass der Fehlertext beim Werfen einer solchen Exception auf der Konsole angezeigt wird). Leiten Sie von dieser Klasse nun weitere Fehlerklassen entsprechend dem nachfolgenden UML-Diagramm ab.



Die im Konstruktor übergebenen Parameter sollen jeweils in einem aussagekräftigen Text an die Oberklasse `MatrixException` übergeben werden, sodass später beim Auftreten einer Ausnahme alle gespeicherten Werte im Fehlertext auf der Konsole ausgegeben werden. Würde die jeweilige Exception dann stattdessen gefangen werden, so soll dennoch Zugriff auf die auslösenden Werte bestehen, daher werden die Parameter außerdem gespeichert und über Getter zugreifbar gemacht.

(77) ★★☆ Betrachten Sie folgenden Code unter Zuhilfenahme des Klassendiagramms vom Anfang dieses Kapitels. Kleiner Hinweis: Der Code ist weder sinnvoll noch schön, aber gültig.

```
System.out.print("A");
try {
    System.out.print(" B1");
    fail();
    System.out.print(" B2");
} catch (FileNotFoundException fnfe) {
    System.out.print(" C");
} catch (ArrayIndexOutOfBoundsException | IOException e) {
    System.out.print(" D1");
    if (e instanceof RuntimeException)
        throw new NullPointerException("D2");
    System.out.print(" D3");
} catch (NullPointerException ex) {
    System.out.print(" E");
    return;
} catch (IndexOutOfBoundsException e) {
    int x = 1;    double y = x;   System.out.print(" F1");
    System.out.print(" " + y/0);  System.out.print(" F2");
    System.out.print(" " + x/0);  System.out.print(" F3");
} catch (RuntimeException ex) {
    System.out.print(" G1");
    try {
        throw ex;
    } catch (ArithmeticException e) {
        System.out.print(" G2");
    }
} finally {
    System.out.print(" H");
}
System.out.print(" I");
```

Geben Sie an, was bei Ausführung dieses Codes durch `System.out` auf der Konsole ausgegeben wird, wenn der Funktionsaufruf `fail()` die jeweils genannte Ausnahme wirft:

a) keine Ausnahme: _____

b) `FileNotFoundException`: _____

c) `NullPointerException`: _____

d) `EOFException`: _____

e) `ArithmeticException`: _____

f) `ArrayIndexOutOfBoundsException`: _____

g) `ClassCastException`: _____

h) `StringIndexOutOfBoundsException`: _____

i) `StackOverflowError`: _____

(78) ★☆☆

Die Klasse `BufferedReader` der Java-Bibliothek `java.io` kann z. B. genutzt werden, um eine Datei von der Festplatte zu lesen. Bei der Erzeugung eines `BufferedReaders` kann jedoch eine `FileNotFoundException` auftreten. Außerdem kann der Versuch, eine Zeile der geöffneten Datei zu lesen (`readLine`), in einer `IOException` enden. Jeder erfolgreich geöffnete `BufferedReader` muss nach dem Lesevorgang unbedingt wieder geschlossen werden! Das Schließen des `BufferedReaders` kann ebenfalls zu einer `IOException` führen.

Betrachten Sie den folgenden **fehlerhaften Code** zum Lesen der ersten Zeile einer Datei. Korrigieren Sie die darin enthaltenen Probleme sinnvoll.

Hinweis: Das Vereinigen der ersten beiden `try`-Blöcke ist bspw. nicht sinnvoll.

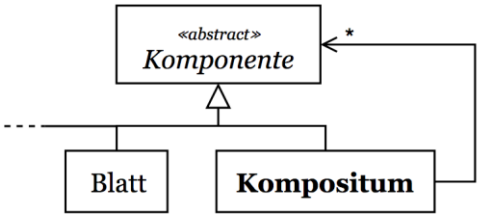
```
public static String readFirstLine(String filePath) {
    // BufferedReader öffnen:
    try {
        BufferedReader reader = new BufferedReader(
                                            new FileReader(filePath));
    } catch (FileNotFoundException e) {
        return null;   // Datei existiert nicht
    }

    // Erste Zeile lesen:
    try {
        return reader.readLine();
    } catch (IOException e) {
        return null;   // nicht lesbar
    }

    // BufferedReader schließen:
    try {
        reader.close();
    } catch (IOException e) {
        // hier sind wir machtlos...
    }
}
```

15. Entwurfsmuster

Entwurfsmuster werden eigentlich erst in der Veranstaltung „Einführung in die Softwaretechnik“ behandelt. Ein paar wenige spielen aber schon hier eine Rolle. Wie der Name bereits sagt sind *Entwurfsmuster* Vorlagen (Muster) für den Lösungsentwurf eines Problems. Bei *Strukturmustern* werden ähnliche Probleme bspw. mit einem ähnlichen Klassenentwurf gelöst.

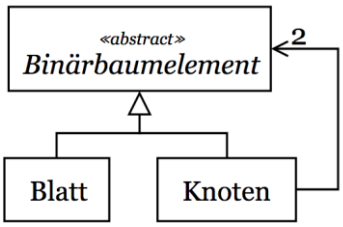
15.1. Kompositum (*composite pattern*)

Dieses Strukturmuster lässt sich typischerweise bei baumartigen Strukturen wie Hierarchien anwenden. Es gibt zwei Komponentenarten: Blätter und das Kompositum. Das Kompositum ist eine Klasse, die selbst wiederum aus Komponenten (also Blättern oder Komposita) besteht. Es können auch mehrere verschiedene Blatt oder Kompositum-Klassen existieren.

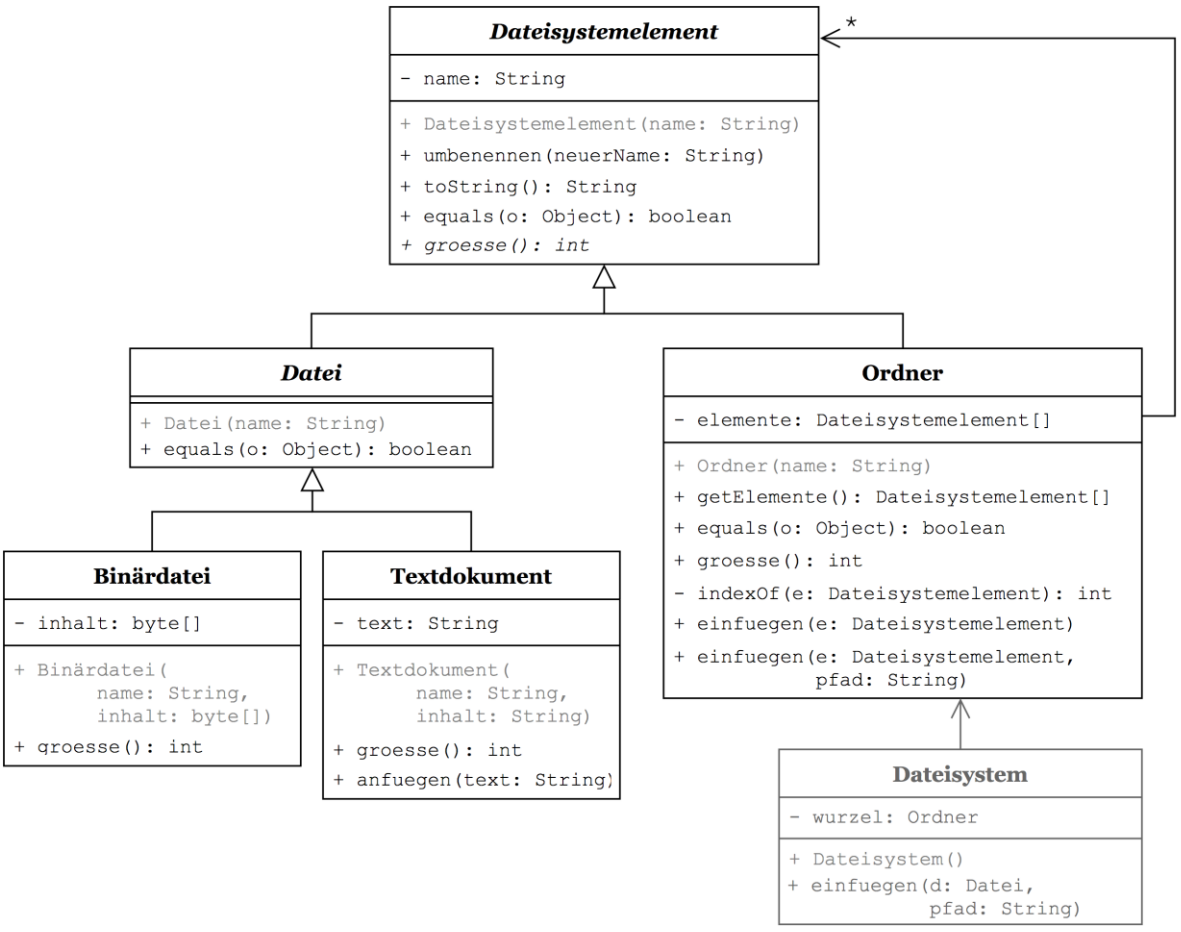


Beispiel: **Binärbaum** (→ Kap. 18)

Es gibt innere Knoten und Blattelemente. Blätter stehen auf der unterste Ebene, während jeder Knoten exakt zwei Nachfolger hat.



Beispiel: Dateiverwaltungssystem eines Computers (UML-Diagramm)



(79) ★★★ Implementieren Sie ein **Dateisystem** genau wie im UML-Diagramm in 15.1 dargestellt. Eine *kursive* Schreibweise steht in UML für eine abstrakte Implementierung. Private Member wurden mit „-“, öffentliche mit „+“ gekennzeichnet.

Die Methoden sollen wie nachfolgend beschrieben implementiert werden:

(a) `umbenennen` ändert den Namen eines Dateisystemelements.

(b) `toString` (in `Dateisystemelement`) gibt den Namen und die Größe (→ (d)) des Dateisystemelements in einem `String` zurück, z. B. `"OrdnerXyz (128B)"` oder `"A.java (19B)"`.

(c) `equals` gibt **`true`** zurück, wenn das Dateisystemelement *äquivalent* ist zu dem als Parameter übergebenen Objekt, anderenfalls **`false`**. Zwei Dateisystemelemente werden dabei genau dann als *äquivalent* angesehen, wenn sie den gleichen Namen haben. Binärdateien können also äquivalent zu Textdokumenten sein. Dateien sollen jedoch nie *äquivalent* zu Ordnern sein (oder umgekehrt)!

(d) `groesse` gibt die Größe des Dateisystemelements *in Bytes* zurück. Die Größe einer Datei ergibt sich unmittelbar aus ihrem Inhalt. Die Größe eines Ordners entspricht der Summe der Größen aller gespeicherten Dateien (ggf. in Unterordnern!).

(e) `anfuegen` fügt den als Parameter übergebenen Text an den Inhalt eines Textdokuments hinten an.

(f) `indexOf` gibt den Index des übergebenen Dateisystemelements im `elements`-Array zurück, also an welcher Stelle im Array der Parameter `e` steht. Ist das Element nicht enthalten, so soll `-1` zurückgegeben werden. Nutzen Sie hier `equals` (→ (c)).

(g) `einfuegen` ist eine in der Klasse `Ordner` überladene Methode:
- Die erste (ohne `pfad`) nimmt das als Parameter übergebene Dateisystemelement in das `elemente`-Array auf. Ist bereits ein äquivalentes Element enthalten (→ (f)), so soll dieses überschrieben werden. Es dürfen also z. B. nicht mehrere Dateien mit demselben Namen in einem Ordner existieren.
- Die zweite bekommt zusätzlich einen Pfad (`pfad`) als Parameter übergeben, über welchen spezifiziert wird, in welchen Unterordner (bzw. Unter-unter-u...-Ordner) das Dateisystemelement eingefügt werden soll.* Sie dürfen davon ausgehen, dass der Dateipfad immer mit einem Slash („/“) endet (bspw. `"Bilder/TUM/"`). Wird lediglich ein einzelner Slash (`"/"`) übergeben, so soll die Methode wie die erste (ohne `pfad`) arbeiten. Existiert ein Unterordner nicht, soll dieser erzeugt werden.

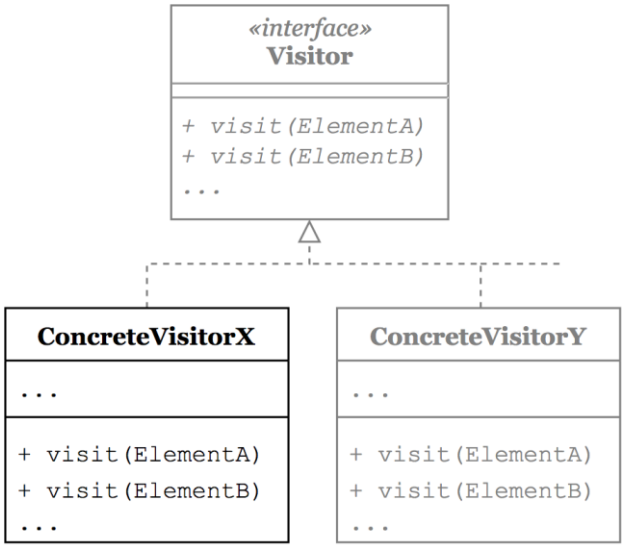
(h) Implementieren Sie anschließend die Klasse `Dateisystem`, welche das Wurzelverzeichnis eines Computers (*root*) verwaltet. Das Dateisystem speichert einen Ordner mit dem Namen „`.`“ (Wurzelverzeichnis) und ermöglicht das Einfügen von Dateien in ein beliebiges Unterverzeichnis mit der `einfuegen`-Methode (→ (g)).

* Sollte dir diese Aufgabe große Schwierigkeiten bereiten, so nutze für die folgenden Aufgaben bitte die Musterlösung.

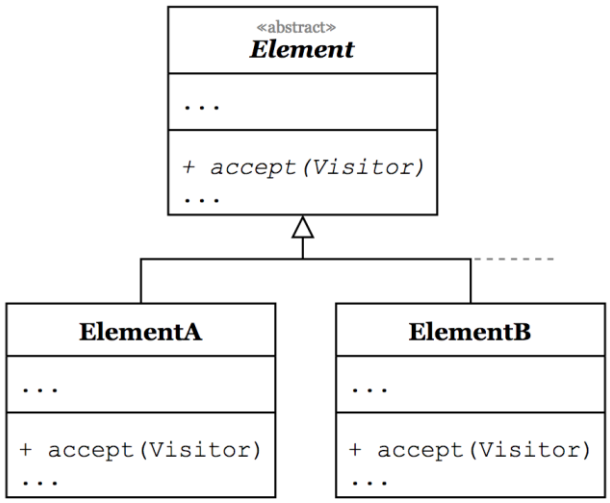
15.2. BESUCHER (*visitor pattern*) ***Hinweis:*** *Vsl. nicht klausurrelevant (WS 19/20)*

Da es sich hierbei um ein *Verhaltensmuster* handelt, ist der Klassenentwurf nicht immer genau gleich, sieht aber zumeist sehr ähnlich aus.

Das zugrundeliegende Probem ist: Es gibt mehrere Klassen (hier: `ElementA` und `ElementB`) und ein Hauptprogramm (hier nicht eingezeichnet), welches in Abhängigkeit vom Typ eines Elements irgendetwas tun möchte. Der einfachste Ansatz wäre, mit vielen `instanceof`-Vergleichen jeweils zu überprüfen, ob das gegebene Objekt nun ein `ElementA` oder `ElementB` oder ... ist. Das ist unschön und unübersichtlich. Man könnte den Code auch in die Element-Klassen auslagern. Das Problem wäre dann aber, dass wir bei Änderungen alle Element-Klassen einzeln verändern müssten.



Die Lösung ist daher, den Code in einer Klasse – dem *Visitor* – zu behalten (hier: `ConcreteVisitorX`), aber auf *visit*-Methoden auszuteilen – eine für jeden konkreten Elementtyp. Möchte man mehrere verschiedene solcher Besucher implementieren (z. B. zusätzlich `ConcreteVisitorY` für eine andere Aufgabe), so kann man diese entweder in einer abstrakten Oberklasse oder über ein Interface (hier: `Visitor`) zusammenfassen. **Kurz**: In den *visit*-Methoden steht der Code, der für eine bestimmte Aufgabe je nach Elementtyp ausgeführt werden soll.



Möchte man die *visit*-Methode nun mit einem als `Element` deklarierten `ElementA` aufrufen, so stößt man auf ein Problem: `visit(Element)` ist undefiniert. Die *visit*-Methode muss dynamisch gebunden werden, daher erweitern wir alle Elementklassen um eine Methode, die einen Visitor entgegennimmt (*accept*) und an dessen *visit*-Methode delegiert. Diese Methode sieht eigentlich immer (hier: in `ElementA` und `ElementB`) so aus:

```
public void accept(Visitor v) { // ohne das Interface hieße es ConcreteVisitorX
    v.visit(this);
}
```

Oft berechnet der Visitor etwas, speichert es in einem privaten Attribut und macht es über einen Getter zugreifbar. Das Hauptprogramm könnte z. B. so aussehen:

```
ConcreteVisitorX visitor = new ConcreteVisitorX();
someElement.accept(visitor); // visitor.visit(someElement) wäre falsch!
System.out.println(visitor.getCalculatedValue());
```

(80) ★★★ Aufbauend auf der vorherigen Aufgabe möchten wir nun eine Möglichkeit bereitstellen, Dateisystemelemente in einen `String` hierarchischer Form umzuwandeln (UNIX: *tree*). Dieser `String` soll in unserem Fall neben der Datei bzw. dem Ordnernamen auch dessen Größe ausgeben und könnte für eine beispielhafte Ordnerstruktur in etwa so aussehen:

```
. (51B)
|__ a.txt (4B)
|__ dirX (23B)
|   |__ a.txt (2B)
|   |__ 6byte.txt (6B)
|   |__ src (13B)
|   |   |__ Hello.txt (10B)
|   |   |__ binfile (3B)
|   |__ Runner.exe (2B)
|__ dirY (0B)
|   |__ empty.txt (0B)
|__ README (24B)
```

Die hierarchischen Einrückungen erschweren die Implementierung, daher greifen wir hier auf das ***Visitor-Pattern*** zurück. Implementieren Sie einen Besucher für Dateisystemelemente, der die Umwandlung in einen `String` übernimmt, sodass Sie letztlich die `toString()`-Methode der Klasse `Dateisystem` auf diesem Besucher abstützen können. Gehen Sie bei der Implementierung schrittweise vor:

1. Definieren Sie in einem Interface `Visitor` die nötigen `visit`-Methoden. Überlegen Sie sich genau, welche `visit`-Methoden notwendig sind (Redundanzvermeidung).
2. Erweitern Sie alle Klassen, denen Sie eine `visit`-Methode dediziert haben, um eine entsprechende `accept`-Methode.
3. Schreiben Sie eine Klasse `FormatVisitor`, die das `Visitor`-Interface implementiert. Die Klasse verfügt über ein privates Attribut vom Typ `String`, über welches das Ergebnis von den `visit`-Methoden aufgebaut werden kann. Zugriff auf dieses Attribut besteht schließlich über die Methode `getFormattedString()`.
4. Überlegen Sie sich, wie Sie Einrückungen in den `visit`-Methoden umsetzen können. Sie dürfen weitere private Methoden und Attribute hinzufügen.
5. Implementieren Sie die `toString()`-Methode der Klasse `Dateisystem`, welche einen derart formatierten `String` beginnend beim Wurzelverzeichnis zurückgeben soll. Lassen Sie die Formatierung dabei von einem `FormatVisitor` durchführen.

Folgendes Beispielprogramm sollte die obige Ausgabe produzieren:

```
public static void main(String[] args) {
    Dateisystem fs = new Dateisystem();
    fs.einfuegen(new Textdokument("a.txt", "ab"),  "/");
    fs.einfuegen(new Textdokument("a.txt", "a"),  "dirX/");
    fs.einfuegen(new Textdokument("6byte.txt", "wird überschrieben"),  "dirX/");
    fs.einfuegen(new Textdokument("Hello.txt", "world"),  "dirX/src/");
    fs.einfuegen(new Binärdatei("binfile", new byte[] {65, 66, 67}), "dirX/src/");
    fs.einfuegen(new Textdokument("empty.txt", ""), "dirY/");
    fs.einfuegen(new Binärdatei("Runner.exe", new byte[] {65, 67}), "dirX/");
    fs.einfuegen(new Textdokument("README", "nothing here"),  "/");
    fs.einfuegen(new Textdokument("6byte.txt", "new"),  "dirX/");
    System.out.println(fs);
}
```

16. Collections: List, Stack & Set

Eine ***Liste*** (*list*) ist eine endliche, geordnete Sequenz von Elementen/Objekten, d. h. jedes Element hat eine eindeutige Position in der Liste.

Beispiele sind die generischen Klassen `java.util.ArrayList<E>` und `java.util.LinkedList<E>`, welche jeweils das Interface `java.util.List<E>` implementieren, sich aber in ihrer Implementierung unterscheiden. Eine `ArrayList` ist der `LinkedList` aus Gründen der Effizienz i. d. R. dann vorzuziehen, wenn häufig auf Elemente an bestimmten Indizes zugegriffen werden muss, während sich eine `LinkedList` eignet, wenn häufig Elemente am Ende der Liste hinzugefügt und entfernt werden müssen.

Die wichtigsten Methoden sind:

- `boolean add(E element)`: Hängt ein Element am Ende der Liste an. Gibt `true` zurück.
- `E get(int index)`: Gibt das Element am jeweiligen Index zurück (wie bei Arrays).
- `boolean remove(Object o)`: Entfernt das erste Vorkommen eines Elements (`o`) in der Liste, wobei die Gleichheit mit der `equals`-Methode überprüft wird (nicht Referenzgleichheit!). Gibt `true` zurück, falls das Element entfernt wurde, sonst `false`.
- `int size()`: Gibt die Länge der Liste zurück (Anzahl der Elemente)
- `boolean contains(Object o)`: Gibt `true` zurück, falls das Element enthalten ist.

Eine (*Warte-*)***Schlange*** (*queue*) ist eine Liste mit eingeschränkten Operationen und eingeschränktem Zugriff auf die Elemente, d. h. es existiert eine Einfügeoperation (*enqueue*), welche ein Element hinten einfügt, und eine Entnahmeoperation (*dequeue*), die das vorderste Element aus der Schlange entnimmt (**F**irst-**I**n, **F**irst-**O**ut). Außer `get` sind die oben genannten Methoden ebenfalls vorhanden. *Queues* können bspw. über eine Liste implementiert werden, können intern aber auch Arrays o. Ä. nutzen.

Beispiele: `java.util.PriorityQueue<E>` oder `java.util.ConcurrentLinkedQueue<E>`, welche jeweils das Interface `java.util.Queue<E>` implementieren.

Ein ***Stapel*** (*stack*) ist ebenfalls eine weniger flexible, listenähnliche Struktur (evtl. in der Implementierung auf einer Liste basierend). Hier werden die Elemente nach dem „**L**ast-**I**n, **F**irst-**O**ut"-Schema eingefügt und entnommen, d. h. ein Element wird oben (bzw. hinten) eingefügt (*push*) und von oben (bzw. hinten) entnommen (*pop*). Sie enthält alle der für Listen genannten wichtigsten Methoden. Beispiel ist die Klasse `java.util.Stack<E>`.

Alle diese Klassen implementieren das Interface `java.util.Collection<E>`, welches das Interface `java.util.Iterable<E>` erweitert und bspw. besagt, dass jede „*Collection*" nicht nur einen Iterator zurückgeben (`iterator()`), sondern insb. Elemente einfügen (`add(e)`) und entfernen (`remove(e)`) können muss, siehe *Javadoc** bzw. Grafik auf der nächsten Seite.

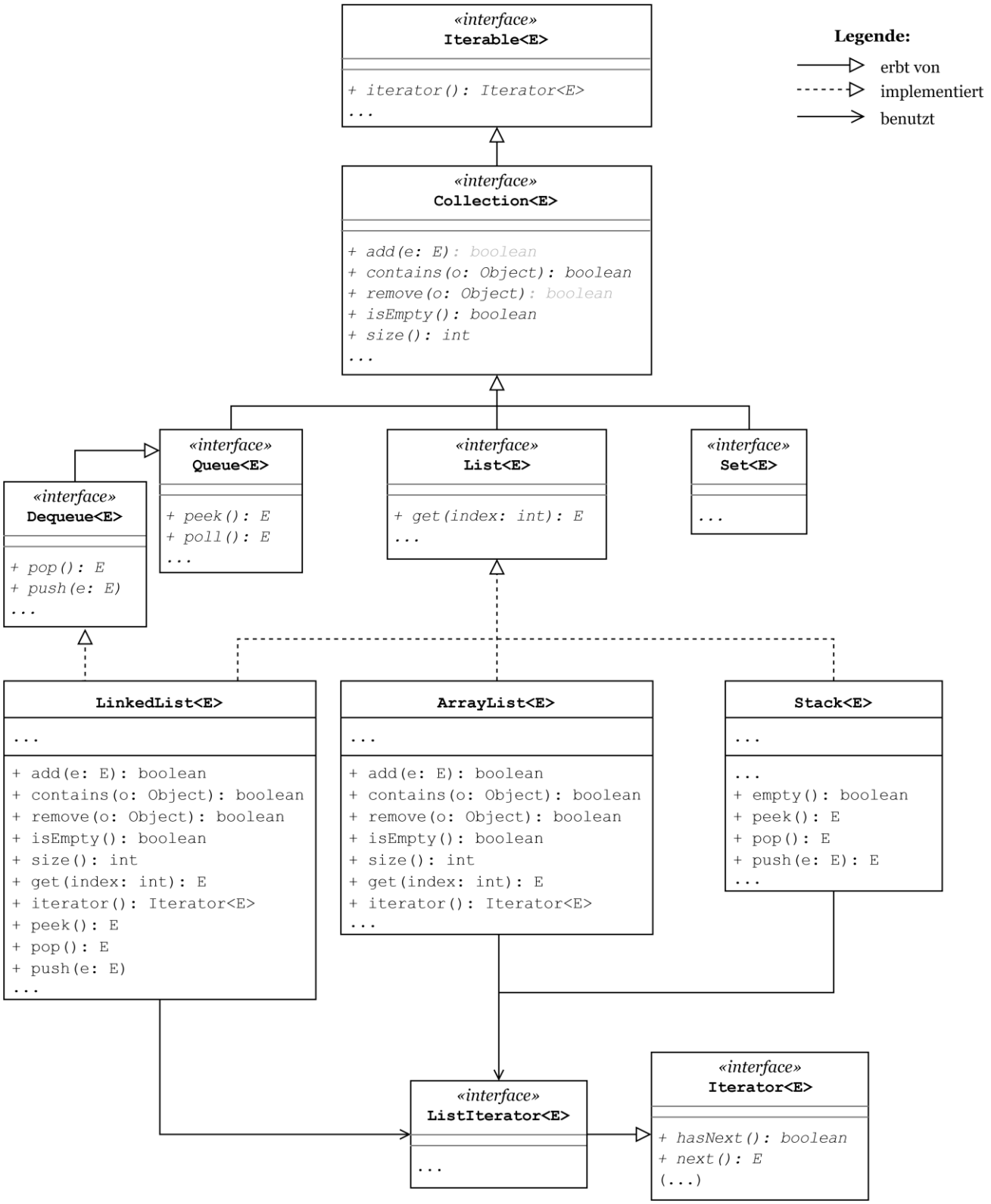
Beispiel: Welche Ausgabe produziert folgender Code?

```
List<Integer> liste = new LinkedList<>();
liste.add(7);
liste.remove(new Integer(7));
System.out.println(liste.size());
```

* *https://docs.oracle.com/javase/8/docs/api/java/util/Collection.html*

Im Gegensatz zu Listen können **Mengen** (*Sets*) Objekte nicht mehrfach enthalten.

Unterhalb ist ein Auszug aus der Java-Standardbibliothek `java.util` grafisch dargestellt. Viele Klassen und Interfaces sind nicht eingezeichnet (z. B. die Oberklassen von `LinkedList`).



Alle konkreten Klassen geben in ihrer `iterator()`-Methode einen `ListIterator` zurück. Dieser kann z. B. in einer inneren (oder anonymen) Klasse implementiert worden sein.

(81) ★★☆ **Eigene Listen 1:** Vervollständigen Sie die nachfolgende Implementierung einer rekursiv definierten, einfach verketteten Liste, *ohne* Membervariablen oder Methoden hinzuzufügen. Implementieren Sie alle Methoden ebenfalls *rekursiv*!

(a) `void einfuegen(int wert)` fügt einen neuen Wert hinten ein.

(b) `Liste entfernen(int wert)` löscht das erste Vorkommen des als Parameter übergebenen Werts und gibt den (ggf. neuen) Ersten (*head*) zurück.

(c) `String toString()` gibt eine kommagetrennte String-Repräsentation aller in der Liste enthaltenen Werte zurück, bspw. `"1, 3, 4"` (kein Komma am Ende).

(d) `boolean equals(Object o)` gibt **`true`** zurück, wenn es sich bei dem als Parameter übergebenen Objekt um eine Liste handelt, die die gleichen Elemente in der gleichen Reihenfolge enthält. Ihre Implementierung muss insb. die Symmetrie- und Reflexivitätseigenschaft besitzen (vgl. *Javadoc* zu `Object.equals`).

```
public class Liste {
    private final int wert;
    private Liste naechster;

    public Liste(int e) {
        this(e, null);
    }

    public Liste(int e, Liste naechster) {



    }

    public void einfuegen(int wert) {







    }

    public Liste entfernen(int wert) {









    }
```

```
    @Override
    public String toString() {









    }

    @Override
    public boolean equals(Object o) {









    }

    public static void main(String[] args) {
        Liste x = new Liste(1);
        x.einfuegen(6);
        x.einfuegen(4);
        x.einfuegen(2);
        x.entfernen(6);
        x.einfuegen(5);
        System.out.println(x);
        Liste y = new Liste(1);
        y.einfuegen(4);
        y.einfuegen(2);
        y.einfuegen(5);
        System.out.println(y);
        System.out.println(x.equals(y));
    }
}
```

(82) ★★★

Eigene Listen 2: Implementieren Sie eine einfach verkettete Liste in einer generischen Klasse `List<E>`, welche Objekte der Klasse `E` (bzw. einer Unterklasse von `E`) speichern kann. Objekte der inneren Klasse `Entry<E>` repräsentieren einzelne Listenknoten. Vervollständigen Sie das Codegerüst auf den folgenden Seiten. Sie dürfen *keine* Membervariablen oder Methoden hinzufügen. Implementieren Sie außerdem folgende Methoden:

a) `void add(E e)`: Fügt das Element `e` hinten ein.

b) `E removeFirst()`: Entfernt das vorderste (erste) Element aus der Liste und gibt es zurück.

c) `int size()`: Gibt die Größe der Liste, d. h. die Anzahl in der Liste enthaltener Elemente, zurück.

d) `String toString()`: Gibt eine String-Repräsentation der Liste zurück, wobei zwischen den Elementen Kommata stehen (ohne Komma am Ende oder am Anfang). Ist die Liste leer, so soll ein leerer String zurückgegeben werden.
Beispiel: `"Hans, Norbert, Peter"`

e) `boolean remove(Object o)`: Entfernt das als Parameter übergebene Objekt (einmal) aus der Liste, wenn es denn enthalten ist. Gibt **`true`** zurück, wenn die Operation erfolgreich durchgeführt wurde, sonst (d. h. wenn das Objekt nicht enthalten ist) **`false`**.

f) `boolean equals(Object o)`: Gibt **`true`** zurück, wenn es sich bei dem als Parameter übergebenen Objekt um eine Liste handelt, welche alle Elemente in der gleichen Reihenfolge enthalten sind. Dabei sollen die einzelnen Objekte auf Objektgleichheit überprüft werden (nicht auf Referenzgleichheit), sodass die beiden Listen also nicht dieselben, aber die gleichen Elemente enthalten müssen.

(83) ★★★

Erweitern Sie Ihre Listenimplementierung aus Aufgabe 82 um einen **Iterator**. Implementieren Sie dafür eine weitere innere Klasse `MyListIterator<E>`, welche das Interface `Iterator<E>` implementiert. Die Klasse muss entsprechend Implementierungen für die Methoden `boolean hasNext()` und `E next()` bereitstellen. Betrachten Sie die *Import*-Anweisung „`import java.util.Iterator;`“ als gegeben.

Machen Sie Ihre Klasse `List<E>` anschließend iterierbar, indem Sie auf diese Iterator-Implementierung zurückgreifen. `List<E>` muss dazu das Interface `Iterable<E>` implementieren und die Methode `Iterator<E> iterator()` bereitstellen, welche ein Objekt Ihrer Iterator-Klasse zurückgibt.

Testen Sie die Implementierung, indem Sie mit einer *„for-each“*-Schleife* (→ Kap. 5.5) über eine beispielhafte Liste aus Aufgabe 82 iterieren und die Elemente ausgeben.

* In Zusammenhang mit Iteratoren lässt sich eine *for-each*-Schleife übrigens wie folgt als *while*-Schleife schreiben. Annahme in folgendem Beispiel: `myIntList` wurde als `List<Integer>` initialisiert:

```
Iterator<Integer> iter = myIntList.iterator();
while (iter.hasNext()) {
    Integer elem = iter.next();  // niemals mehrfach aufrufen!
    // tue etwas mit <elem>, bspw.: System.out.println(elem);
}
```

```
public class List<E> {

    private Entry<E> first;   // vorderstes Element (Knoten)

    private class Entry<E> {   // E könnte auch anders genannt werden









    }

    public void add(E e) {









    }

    public E removeFirst() {









    }

    public int size() {









    }
```

```
    public String toString() {

    }

    public boolean remove(Object o) {

    }
```

```
    public boolean equals(Object o) {

    }

    // Hier folgt die Implementierung des Iterators aus Aufgabe 83.
}
```

(84) ★★★ **Eigene Stacks:** Häufig basieren Stacks intern auf Listen, wobei man nur zwei Operationen zugreifbar macht: Hinten einfügen (*push*) und hinten entnehmen (*pop*). Statt „hinten“ und „vorne“ verwenden wir bei Stacks die Bezeichnungen „oben“ und „unten“.

Schreiben Sie nun eine generische Klasse `SimpleStack<E>`, welche intern nicht auf einer Liste, sondern einem Array (welches nur Elemente vom Typ `E` speichert) basiert. Beachten Sie, dass die Erzeugung eines generischen Arrays mittels `new E[size]` in Java nicht ohne Weiteres möglich ist. Weichen Sie daher auf ein `Object`-Array aus.

Implementieren Sie ...

... einen Konstruktor `SimpleStack(int maxSize)`, welcher einen neuen Stack fester Größe erzeugt. Der Stack kann maximal `maxSize` Elemente speichern und soll daher ein Array dieser Größe erzeugen, d. h. das interne Array ändert die Größe niemals.

... die Methode `push(E element)` bzw. `E pop()`, welche ein Element auf den Stack legt bzw. das oberste Element vom Stack entnimmt und zurückgibt. Hinweis: Sie dürfen weitere Attribute hinzufügen (Das gilt immer, wenn nicht explizit verboten).

... die Methode `clear()`, welche den Stack leert.

... die Methode `int size()`, welche die Größe des Stacks zurückgibt.

... die Methode `boolean isEmpty()`, welche `true` zurückgibt, wenn der Stack leer ist.

Legen Sie eine Hierarchie von **Exceptions** für Ihre Klasse `SimpleStack` an. Unterhalb ist jeweils die Sigatur des Konstruktors der geforderten Exception angegeben. Stellen Sie sicher, dass jede Exception einen aussagekräftigen Fehlertext liefert, wenn sie auftritt.

- `SimpleStackException(String)` ist die Oberklasse für alle nachfolgend genannten Ausnahmen. Sie erbt von `RuntimeException` und übergibt den übergebenen Fehlertext als Parameter an den Konstruktoraufruf der Oberklasse.
- `SimpleStackOverflowException(int)` wird geworfen, wenn versucht wird, etwas auf den vollen Stack zu legen. Erwartet die maximale Stack-Größe als Parameter.
- `SimpleStackEmptyException()` wird geworfen, wenn versucht wird, etwas vom bereits leeren Stack zu entnehmen.
- `InvalidSimpleStackSizeException(int)` wird geworfen, wenn versucht wird, einen Stack ungültiger Größe zu erzeugen. Die Größe wird als Parameter übergeben.

Werfen Sie diese Ausnahmen an geeigneter Stelle in `SimpleStack`.

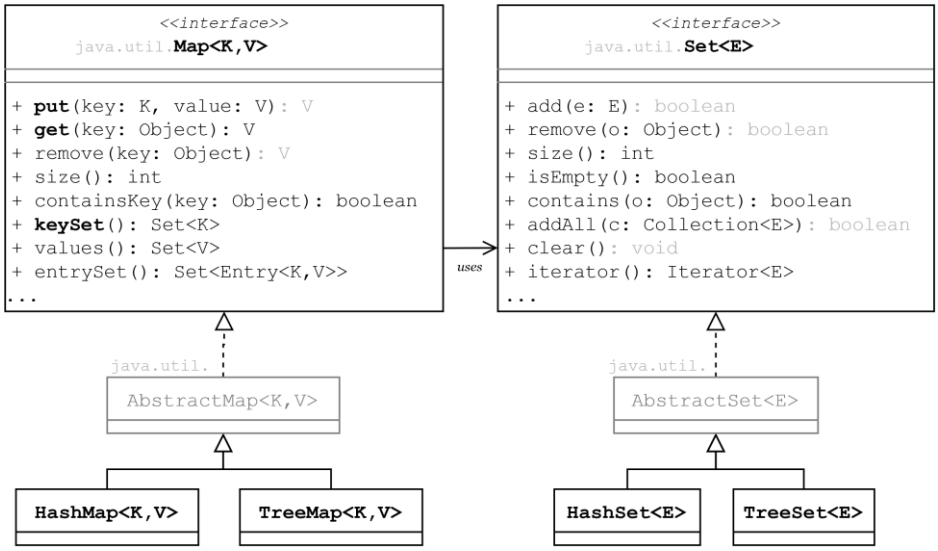
Erweitern Sie Ihren `SimpleStack<E>` nun um einen **Iterator**, welcher die Elemente des Stacks von oben nach unten zurückgibt (also das oberste Element zuerst). Schreiben Sie dafür eine private innere Klasse `SimpleStackIterator`. Der Iterator darf den Stack wie üblich nicht verändern. Ergänzen Sie die Klassen um nötige `implements`-Zusätze, sodass problemlos mit einer *for-each*-Schleife über den Stack iteriert werden kann.

`Map` und `Set`: Weiters stellt Java über das Interface `Map<K,V>` Klassen zur Speicherung von Zuordnungspaaren bereit. Objekte von einem beliebigen Typ `K` werden auf Objekte eines beliebigen Typs `V` abgebildet. Jedem Schlüssel (***Key***) ist ein Wert (***Value***) zugeordnet.

Man kann sich bspw. Arrays als Maps vorstellen: So ordnet z. B. ein String-Array (`String[]`) Strings zu Zahlen zu, d. h. jedem Index (`int`) ist ein `String`-Objekt zugeordnet. Ein String-Array ist also eine Abbildung von Zahlen (Schlüssel) auf Strings (Werte). Ein Integer-Array wäre eine Abbildung von Zahlen auf Zahlen. Arrays sind Abbildungen von Zahlen (den Indizes) auf Werte.

Maps bieten die Möglichkeit, nun auch den Typ der Schlüssel zu bestimmen. Wir können bspw. `Strings` (Schlüssel) auf Zahlen (Werte) abbilden, also über einen String auf einen bestimmen Zahlenwert zugreifen (umgekehrt wie bei einem String-Array). Wie in einem Array müssen die Schlüssel aber eindeutig sein, d. h. jeder Schlüssel ist einzigartig (genau wie es in einem Array nicht zweimal den Index 0 gibt), während Werte mehrfach zugeordnet werden können.

`Map` ist ein Interface! Wir benötigen zur Instanziierung immer eine konkrete Map, z. B. `TreeMap` oder `HashMap` (für uns erstmal egal welche, aber ein Import ist nötig, z. B. `import java.util.HashMap`).



Beispiel:

```
Map<String, Integer> goals = new HashMap<>();
goals.put("Schweinsteiger", 2);
goals.put("Hummels", 1);
goals.put("Müller", 2);
System.out.println("Tore von Hummels: " + goals.get("Hummels"));
```

Diamond-Operator (hier könnte auch nochmal `<String, Integer>` stehen).

Über eine **Zuordnung** (`Map`) können wir nicht einfach so iterieren. Zum Iterieren benötigen wir immer eine **Menge** (`Set`). `Set` implementiert wiederum das Interface `Iterable`. Mengen sind wie *ungeordnete* Listen, die keine Duplikate enthalten können.

```
Set<String> players = goals.keySet();
Iterator<String> playersIter = players.iterator();
while (playersIter.hasNext()) {
    String player = playersIter.next();   // nur ein next() in der Schleife!
    System.out.println("Tore von " + player + ": " + goals.get(player));
}
```

Oder kurz mittels *for-each*-Schleife über die Schlüsselmenge:

```
for (String player : goals.keySet())
    System.out.println("Tore von " + player + ": " + goals.get(player));
```

Oder Menge von Eintragspaaren: `Map.Entry<String,Integer> e : goals.entrySet()`

(85) ★★☆

Rekursion in objektorienterter Programmierung: Biologische Vererbung

In dieser Aufgabe gehen wir davon aus, dass jeder Mensch exakt eine biologische Mutter und einen biologischen Vater hat. Außerdem kann jeder Mensch beliebig viele Kinder haben. Die Klasse `Mensch` modelliert einen Menschen. Sie dürfen davon ausgehen, dass die Hierarchie der Generationen vollständig ist, d. h. die Attribute `vater` und `mutter` sind nie `null`. Ein Mensch, der (noch) keine Kinder hat, speichert im Attribut `kinder` eine leere Menge, d. h. `kinder` ist ebenfalls nie `null`.

Implementieren Sie die fehlenden Methoden entsprechend der *Docstrings* rekursiv.

```
public class Mensch {
    private Mensch vater, mutter;
    private Set<Mensch> kinder;

    /**
     * Mit dieser Methode kann ein bestimmter Vorfahr ermittelt werden. Über den
     * Parameter wird eine Folge von Elternteilen spezifiziert, wobei gültige Zeichen
     * ausschließlich m (mütterlicherseits) und v (väterlicherseits) sind.
     * Beispiele: • vorfahr("mv") gibt den Großvater mütterlicherseits zurück.
     *            • vorfahr(""): Der Mensch gibt sich selbst zurück.
     *            • vorfahr("m") gibt die eigene Mutter zurück.
     *            • vorfahr("vmm") gibt die Großmutter des eigenen Vaters zurück.
     *            • vorfahr("mk") gibt null zurück (ungültiger Parameter).
     */
    public Mensch vorfahr(String folge) {









    }
```

```java
    /**
     * Diese Methode gibt die Menge aller Vorfahren einer bestimmten Generation
     * zurück. Da sich die Vorfahren überschneiden können, eignet sich eine Liste
     * (oder Array) hier nicht, daher verwenden wir Mengen. Über den Parameter
     * wird die Generation spezifiziert*: 1 entspricht den Eltern, 2 den Großeltern,
     * 3 den Urgroßeltern, usw. Ein negativer Parameter entspricht der entgegen-
     * gesetzten Richtung, d. h. -1 sind die eigenen Kinder, -2 die Enkel, usw.
     * Beispiele: • vorfahren(0) enthält nur sich selbst.
     *            • vorfahr(2) enthält die Großeltern (i. d. R. 4 Menschen).
     *            • vorfahr(-3) enthält die Urenkel (mind. 0, max. unbegrenzt).
     * Hinweis: Verwenden Sie nicht die Methode vorfahr (aus Effizienzgründen).
     * * https://de.wikipedia.org/wiki/Generationsbezeichnungen
     */
    public Set<Mensch> vorfahren(int generation) {

    }
}
```

Zusatzfrage: Ist Ihre Implementierung von `vorfahr` bzw. `vorfahren` endrekursiv?

(86) ★★★

Effizientes Zählen: Finden Sie die häufigsten Elemente des Arrays!

Schreiben Sie eine Methode `mostFrequentElements(int[] arr)`, welche das am öftesten im Array `arr` vorkommende Element findet. Da es mehrere häufigste Elemente geben könnte, soll die Methode nicht nur ein einzelnes Element sondern eine Menge der häufigsten Elemente zurückgeben.

Beispiele:
- $mostFrequentElements([19, 3, 4, 19, 4, 7, 3, 5, 4]) \Longrightarrow \{4\}$
- $mostFrequentElements([1, 2, 3, 2, 1]) \Longrightarrow \{1, 2\}$
- $mostFrequentElements([5, 7, 11]) \Longrightarrow \{5, 11, 7\}$
- $mostFrequentElements([8, 3, 5, 7, 3, 3, 7, 8, 7, 2, 2]) \Longrightarrow \{3, 7\}$
- $mostFrequentElements([\,]) \Longrightarrow \{\,\}$

Ihre Implementierung soll möglichst effizient sein (insb. für große Arrays), daher dürfen Sie *keine geschachtelten Schleifen* verwenden (da Ihr Ansatz dann vermutlich in $\mathcal{O}(n^2)$ liegen würde)! Mehrere einfache Schleifen hintereinander sind natürlich erlaubt.

Hinweise:
- Verwenden Sie eine `Map<Integer,Integer>`, um die Elemente zunächst zu zählen (Elemente werden auf ihre Häufigkeit abgebildet).
- Extrahieren Sie die häufigsten Elemente anschließend in ein `Set`.
- Mengen (`Sets`) verfügen über keine Reihenfolge, daher ist es egal, in welcher Reihenfolge Sie die Elemente im Ergebnis zusammenstellen.
- Methoden zu `Maps` und `Sets`: Siehe Seite 124.

```
public static Set<Integer> mostFrequentElements(int[] arr) {




}
```

17. POLYMORPHIE

In diesem Kapitel wollen wir uns mit dem Unterschied zwischen statischem und dynamischem Typ sowie deren Bestimmung beschäftigen. Dieses Verständnis ist die Grundlage und Handwerkszeug für Polymorphie. Polymorphie ist ein Konzept der objektorientierten Programmierung, bei dem es im Grunde darum geht, dass eine Variable Objekte unterschiedlichen Typs speichern kann. Schuld daran ist Vererbung (bzw. die damit einhergehende Überschreibung) und Überladung. Eine Methode kann in einer Unterklasse überschrieben werden (gleiche Signatur). Gleichzeitig kann eine Methode überladen sein (gleicher Methodenname, unterschiedliche Parametertypen). Die Signatur der aufgerufenen Methode ist statisch eindeutig bestimmt (durch die statischen Typen), nicht aber die tatsächlich aufgerufene Methode (da ggf. überschrieben)...

I. **Bestimmung des *statischen Typs*** (↝ Typ der Variable):

Der statische Typ einer Variable kann einfach an der *Deklaration* abgelesen werden, d. h. er steht immer links neben dem ersten Vorkommen des Variablennamens (im Gültigkeitsbereich der Variable). Er kann nicht geändert werden (nur „vorübergehend" durch Casten), denn jede Deklaration ist eindeutig.

Sonderfälle: Der stat. Typ von `this` entspricht der Klasse, in der es benutzt wird;
`super` entspricht der Oberklasse der Klasse, in der es benutzt wird.

Besonderheit: *„Casting"* *(vgl. Kap. 2)*

Liegt ein Cast (bspw. `(A)b`) vor, so „ändert" sich der stat. Typ *vorübergehend* (zu `A`).
Wir müssen jedoch zuerst überprüfen, ob der Cast überhaupt funktioniert:

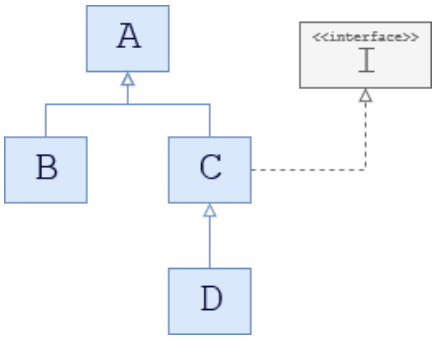
1. *Compiler-Sicht*: Steht die Klasse des *stat. Typs* der Variable, die gecastet wird, in einer Relation (d. h. Vererbungshierarchie) mit der *„Cast"-Klasse*? Wenn die beiden Klassen etwas miteinander zu tun haben, also eine von der anderen erbt (ggf. indirekt) oder die Klassen identisch sind, dann könnte der Cast theoretisch möglich sein. Anderenfalls gibt es einen ***Compiler-Fehler***.
2. *Dynamische Sicht*: Ist das Objekt der Variable, die wir gerade versuchen zu casten, auch tatsächlich vom Typ des „Casts" (oder spezieller)? Wir sehen uns jetzt also den *dyn. Typ* der Variable an (siehe unten) und prüfen, ob diese Klasse (des dyn. Typs) von der „Cast"-Klasse erbt (ggf. indirekt, bzw. identisch sind). Ist der Typ des Objekts nicht mindestens so speziell wie der „Cast-Typ", so tritt ein ***Laufzeitfehler*** (`ClassCastException`) auf.

II. **Bestimmung des *dynamischen Typs*** (↝ Typ des Objekts):

Den tatsächlichen (dyn.) Typ des Objekts, auf das eine Variable verweist, ermittelt man durch *„Zurückverfolgen"* bis zur Objekterzeugung. Wird der Variable `b` bspw. `a` zugewiesen (`b = a;`), so müssen wir nachsehen, auf welches Objekt `a` *zu diesem Zeitpunkt* verweist, um den dyn. Typ von `b` zu bestimmen. Ist die letzte Zuweisung an `a` bspw. `a = new C()`, so zeigt `b` auf ebendieses `C`-Objekt, d. h. `b` den dyn. Typ `C`. Hier interessieren uns also nicht die Deklarationen, sondern die tatsächlichen Zuweisungen der Objekte. Wie die Bezeichnung „dynamisch" schon sagt, ist der dyn. Typ abhängig von der letzten Zuweisung, folglich veränderlich und nur zur Laufzeit des Programms bestimmbar.

(87) ☆☆☆

Bestimmen Sie für alle Teilaufgaben den statischen und dynamischen Typ aller Variablen nach der Ausführung *aller* Statements der Teilaufgabe. Sollte ein Statement nicht übersetzt werden können, so schreiben Sie als Antwort für diese Teilaufgabe „*Compiler-Fehler*“. Tritt ein Fehler bei der Ausführung auf, so schreiben Sie „*Laufzeitfehler*“ mit Zeilenangabe.



Folgende Hierarchie sei gegeben:

```
interface I { }
class A { }
class B extends A { }
class C extends A implements I { }
class D extends C { }
```

		Variable	stat. Typ	dyn. Typ

a)	`A a = new A();`
	`A b = new C();`
	`B c = new B();`
		a
		b
		c		

b)	`B a = new C();` `A b = new A();`	a b		
c)	`I i = new C();` `C a = new D();`	a i		
d)	`D d = new D();` `I c = new I();`	c d		

e)	`C b = new C();`
	`D a = new D();`
	`A c = b; b = a;`
		a
		b
		c		

f)	`I c = new C(); I d = new D();` `c = d; d = c;`	c d		
g)	`A d = new D();` `C c = d;`	c d		

h)	`A a = new B();`
	`B b = (B)a;`
	`Object o = (A)b;`
		a
		b
		o		

i)	`C c = new D();`
	`I i = c;`
	`c = (C)i;`
		c
		i		

j)	`C a = new C();` `B c = (B)a;`	a c		
k)	`I c = new C();` `C d = c;`	c d		
l)	`A a = new D();` `B b = (B)a;`	a b		

m)	`I c = new C();` `C b = (C)c; c = new D();`	b c		
n)	`Object o; B a = new B();` `o = (C)a;`	a o		
o)	`A c = new C();` `C a = c;`	a c		
p)	`Object c, d; c = new D();` `d = (I)c;`	c d		
q)	`A a = (C)(A)new D();`	a		
r)	`B b = (A)new C();`	b		
s)	`I c = new D();` `C d = c;`	c d		

t)	`A c;`
	`I b = new C();`
	`c = b;`
		b
		c		

u)	`A b = (D)(A) new C();`	b		

v)	`Object d = new D();`
	`I c = (C) d;`
	`d = c;`
		c
		d		

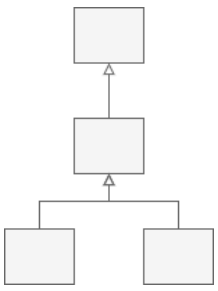
w)	`I i = new D();`
	`Object c, a;`
	`c = (A)i; a = (C)c;`
	`c = new A();`
		a
		c
		i		

x)	`Object d = new D();` `A c = (C)d;`	c d		

y)	`B c = new B();`
	`A a = c;`
	`C b = (C)a;`
		a
		b
		c		

(88) ☆☆☆ Füllen Sie die Vererbungshierarchie auf der rechten Seite so aus, dass folgender Code zwar kompiliert, zur Laufzeit jedoch eine `ClassCastException` in Zeile 3 entsteht.

```
1    R p = new K();
2    R a = (F) p;
3    F y = (W) a;
```



III. **Bestimmung der aufgerufenen Methode** / Membervariable für Aufrufe der Form $e_0.method(e_1, \dots, e_n)$ bzw. $e_0.member$, wobei e_i Ausdrücke (häufig einfach Variablen) sind:

❶ ***Compiler-Sicht*** *(bei allen Methoden oder Attributen)*

1. Bestimme den statischen Typ das Ausdrucks (bzw. der Variable), auf dem die Methode aufgerufen wird (e_0). Eventuell gibt es einen Fehler durch einen Cast.
2. Bestimme – falls nötig – auch die statischen Typen aller Parameter ($e_{1\dots n}$).
3. Wir suchen nun
   - in der in Schritt 1 gefundenen Klasse (oder einer Oberklasse davon)
   - eine Methode mit passendem Namen (wir nehmen logischerweise nicht `k(...)` wenn wir `m(...)` suchen) und ausreichender Sichtbarkeit, die
   - die Parametertypen aus Schritt 2 akzeptiert. Beachte, dass man einer Methode, die als Parameter z. B. ein `Fahrzeug` erwartet, auch alle Typen (Klassen) übergeben kann, die von `Fahrzeug` erben (bspw. `Fahrrad`).

   Gibt es mehrere passende Methoden (bspw. `m(Fahrzeug)` und `m(Fahrrad)`, wenn wir ein `Mountainbike` übergeben und `Mountainbike` von `Fahrrad` von `Fahrzeug` erbt), so wählen wir immer die bzgl. der Parametertypen speziellste (hier `m(Fahrrad)`). Existiert keine bzgl. *aller* Parameter speziellste Methode (bei mehreren Parametern möglich), so ist der Aufruf *mehrdeutig* → Compiler-Fehler.

   Achtung: Die speziellste Methode könnte auch in der Oberklasse stehen, schließlich erben wir alle nicht-privaten Methoden der Oberklasse! Wir würden im obigen Beispiel also auch dann `m(Fahrrad)` wählen, wenn diese nicht in der aktuellen Klasse aber in der Oberklasse definiert worden wäre. Beachte wiederum, dass wir keinen Zugriff auf private Methoden einer Oberklasse haben.

   Keine passende Methode gefunden bzw. kein Zugriff? → Compiler-Fehler.

❷ ***Dynamische Sicht*** *(nur bei nicht-statischen Methoden)*

Wir führen direkt die in Schritt ❶ gefundene Methode aus, wenn diese `static` (statischer Kontext) oder `final` (sowieso nicht überschreibbar) ist oder wir auf dem Schlüsselwort `super` operieren (also nicht wieder in der Unterklasse kucken wollen). Auch beim Zugriff auf eine Membervariable/Attribut (statt Methode) entfällt der Schritt ❷ (da Variablen wie statische Methoden *nicht* überschrieben werden).

1. Bestimme den dynamischen Typ des Ausdrucks, auf dem die Methode aufgerufen wird (e_0). Sind dynamischer und statischer Typ identisch, so kann man aufhören und die Methode aus Schritt ❶ ausführen. Anderenfalls:
2. Schau nun in der Klasse des dyn. Typ nach, ob die in Schritt ❶ gefundene Methode überschrieben. Dazu muss die Signatur (d. h. der Methodenname und alle Parametertypen) *exakt* übereinstimmen! Ist das der Fall, so führen wir diese Methode aus, anderenfalls die Methode aus Schritt ❶.

   Achtung: Auch hier könnte die überschriebene Methode geerbt worden sein; nämlich von einer Oberklasse des dyn. Typs, die der Klasse des stat. Typs wiederum untergeordnet ist (dyn. Typ ist immer spezieller oder gleich speziell wie stat. Typ).

Wie du siehst ist die Vorgehensbeschreibung ziemlich umfangreich, was insb. den ganzen Sonderfällen geschuldet ist. Daher findest du unterhalb noch **Kurzbeschreibungen** für das Vorgehen bei Polymorphie, welche die grundlegenden Fälle abdecken. Diese Beschreibungen sind im Gegensatz zur vorherigen Erklärung unvollständig, können dir aber beim Einstieg in dieses Thema helfen, falls du auf den vorherigen Seiten den Überblick verloren hast.

Für einen Methodenaufruf der Form **`ausdruck.methode()`**:

1. Bestimme den statischen Typ von **`ausdruck`**, nachfolgend ***StatTyp*** genannt.
2. Suche in der Klasse ***StatTyp*** (bzw. anschließend den Oberklassen von ***StatTyp***) nach einer Methode mit dem Namen **`methode`**, die keine Parameter erwartet. Wird keine passende Methode gefunden, so sprechen wir von einem ***Compiler-Fehler*** und sind fertig.
3. Ist die gefundene Methode statisch (`static`), so bist du fertig und führst die gefundene Methode aus. Anderenfalls bestimme den dynamischen Typ von **`ausdruck`**, also den Typ des Objekts, zu dem **`ausdruck`** auswertet, nachfolgend ***DynTyp*** genannt.
4. Suche in der Klasse ***DynTyp*** (bzw. anschließend den Oberklassen von ***DynTyp***) nach einer Methode mit dem Namen **`methode`**, die keine Parameter erwartet, und führe die gefundene Methode aus. Du findest diese Methode sicher (spätestens die in Schritt 2 gefundene).

Für einen Attributzugriff der Form **`ausdruck.attribut`**:

1. Bestimme den statischen Typ von **`ausdruck`**, nachfolgend ***StatTyp*** genannt.
2. Suche in der Klasse ***StatTyp*** (bzw. anschließend den Oberklassen von ***StatTyp***) nach einer Variable mit dem Namen **`attribut`**. Das Ergebnis ist der Wert dieser Variable.

Für einen Methodenaufruf der Form **`ausdruck0.methode(ausdruck1, ausdruck2, ...)`**:

1. Bestimme den statischen Typ von **`ausdruck0`** und **`ausdruck1`** (und allen weiteren). Diese Typen werden im Folgenden mit ***StatTyp0***, ***StatTyp1***, usw. bezeichnet.
2. Suche in der Klasse ***StatTyp0*** (bzw. danach in den Oberklassen von ***StatTyp0***) nach einer Methode mit dem Namen **`methode`**, die als Parameter die Typen ***StatTyp1***, ***StatTyp2***, usw. „akzeptiert“. Dazu muss die Anzahl der Parameter übereinstimmen. Eine Methode `methode(ParamTyp1 param1, ...)` „akzeptiert“ bspw. ***StatTyp1*** als Parameter genau dann, wenn `ParamTyp1` eine Oberklasse von ***StatTyp1*** ist (oder beide gleich sind). Wird keine passende Methode gefunden, so sprechen wir von einem ***Compiler-Fehler*** und sind fertig. Bei mehreren passenden Methoden: Wähle die speziellste, also die, deren Parametertypen in der Vererbungshierarchie möglichst nah an ***StatTyp1***, ***StatTyp2***, usw. liegen.
3. Ist die gefundene Methode statisch, so bist du fertig und führst die Methode aus. Anderenfalls bestimme den dynamischen Typ von **`ausdruck0`**, nachfolgend ***DynTyp0*** genannt.
4. Ist ***DynTyp0*** gleich zu ***StatTyp0***, so bist du ebenfalls fertig. Anderenfalls suche in der Klasse ***DynTyp0*** (bzw. anschließend den Oberklassen von ***DynTyp0***), ob die zuvor gefundene Methode (aus Schritt 2) hier überschrieben wird. Dazu müssen die Signaturen (Methodenname und Parametertypen) exakt übereinstimmen. Ist dies der Fall, so führe die neue (überschreibende) Methode aus, anderenfalls die in Schritt 2 gefundene.

(89) ★★★ Gegeben seien folgende Klassen (jeweils in einer eigenen Datei):

```
public class F {
    String m = "F.m";
    String m() { return "F.m()" + this.s(); }
    String m(F f) { return "F.m(F)" + s(); }
    private String m(G g) { return "F.m(G)"; }
    String k() { return "F.k()" + this.m(this); }
    String k(G g) { return "F.k(G)"; }
    static String s() { return "F.s()"; }
    static String t() { return "F.t()"; }
}

public class G extends F {
    String m = "G.m";
    final static String s = "G.s";
    String m() { return "G.m()" + super.s(); }
    String m(F f) { return "G.m(F)"; }
    String m(G g) { return "G.m(G)"; }
    String k(F f) { return "G.k(F)" + this.m((G) f); }
    static String s() { return "G.s()"; }
}

public class H extends G {
    String k(F f) { return "H.k(F)"; }
    String k(A x) { return "H.k(A)"; }
    String k(G g) { return "H.k(G)" + super.m((F) g); }
}

public class A extends G {
    F m;
    A(F m) { this.m = m; }
    String k(H t) { return "A.k(H)" + t(); }
    String k(G x) { return "A.k(G)" + x.k(this) + m.m(x); }
}
```

Auf der nächsten Seite finden Sie einige Aufrufe, welche zu `String`s auswerten. Geben Sie für jeden Aufruf an, zu welchem `String` der Aufruf auswertet oder ob der Aufruf zu einem *Compiler-Fehler* bzw. *Laufzeitfehler* führt. Nennen Sie den ggf. auftretenden Fehler. Alle Aufrufe erfolgen aus einer anderen Klasse im selben Package.

Hinweis: Du bist schon geübt im Umgang mit Polymorphie? Schau dir zumindest die Aufrufe ab 57 an!

Folgende Deklarationen und Initialisierungen seien zu Beginn bereits gegeben:

```
F a = new F();
G b = new G();
F c = b;
H h = new H();
A y = new A(h);
```

	Aufruf	Ergebnis
1.	`b.m()`	
2.	`b.m(a)`	
3.	`b.m(b)`	
4.	`b.m(c)`	
5.	`b.k()`	
6.	`b.k(a)`	
7.	`b.k(b)`	
8.	`b.k(c)`	
9.	`b.s()`	
10.	`b.t()`	
11.	`a.m()`	
12.	`a.m(a)`	
13.	`a.m(b)`	
14.	`a.m(c)`	
15.	`a.k()`	
16.	`a.k(a)`	
17.	`a.k(b)`	
18.	`a.k(c)`	
19.	`a.s()`	
20.	`a.t()`	
21.	`c.m()`	
22.	`c.m(a)`	
23.	`c.m(b)`	
24.	`c.m(c)`	
25.	`c.k()`	
26.	`c.k(a)`	
27.	`c.k(b)`	
28.	`c.k(c)`	
29.	`c.s()`	
30.	`c.t()`	
31.	`a.m`	
32.	`b.m`	
33.	`c.m`	

	Aufruf	Ergebnis
34.	`((F)b).m((F)a)`	
35.	`((G)c).m((F)b)`	
36.	`((F)c).m(c)`	
37.	`((F)b).k((F)b)`	
38.	`((G)a).k(c)`	
39.	`((G)c).k((G)c)`	
40.	`h.k()`	
41.	`h.k(c)`	
42.	`h.k(b)`	
43.	`h.k(g)`	
44.	`((F)h).k(b)`	
45.	`((F)h).k(c)`	
46.	`((G)h).k(a)`	
47.	`((G)h).k(b)`	
48.	`((G)h).k(c)`	
49.	`((G)h).k((G)c)`	
50.	`h.k(h)`	
51.	`h.s()`	
	`a = h; // gilt ab jetzt`	
52.	`((H)a).k(b)`	
53.	`((H)a).m(h)`	
54.	`((G)a).k(h)`	
55.	`a.k(a)`	
56.	`a.m(h)`	
57.	`A.t()`	
58.	`H.s`	
59.	`A.k(h)`	
60.	`y.k(h)`	
61.	`new H().k(y)`	
62.	`((H)y).k(c)`	
63.	`y.k(b)`	
64.	`y.k(y)`	
65.	`((G)y).k(h)`	

(90) ★★☆

Sei `C` eine Klasse, die von `B` erbt, und `B` eine Klasse, die von der Klasse `A` erbt. Die Klasse `MiniJava` stellt eine `write(int)`-Methode zur Verfügung, welche die übergebene Zahl auf der Konsole ausgibt. Weiters seien die Klassen `X` und `Y` wie folgt implementiert:

```
class X extends MiniJava {
    protected final int z = 1;
    public void write(A a) { write(1*z); }
    public void write(B b) { write(3*z); }
}
final class Y extends X {
    private final int z = 2;
    public void write(A a) { write(5*z); }
    public void write(B b) { write(7*z); }
    public void write(C c) { write(9*z + super.z); }
}
```

Welche Ausgaben produzieren die folgenden Methodenaufrufe. Welche Statements sind nicht ausführbar (*Compiler-Fehler*) und welche führen zu einem *Laufzeitfehler*?

```
A a = new A();
B b = new B();
C c = new C();

Y y;
X x = y = new Y();
```

	Statement	Ausgabe
1.	`y.write(c);`	
2.	`x.write(c);`	
3.	`new X().write(c);`	
4.	`y.write((A) c);`	
5.	`x.write((A) c);`	
6.	`y.write(x);`	
7.	`((X) y).write(b);`	
8.	`((Y) x).write(c);`	
9.	`y.write((B) a);`	
10.	`((X) y).write(c);`	
11.	`((Y) (X) new Y()).write((A) b);`	
12.	`x.write((B) (Object) c);`	
13.	`((Y) new X()).write(a);`	
14.	`((X) y).write(x);`	
15.	`y.write((A) new Object());`	

(91) ★★☆ Gegeben seien folgende Klassen (jeweils in einer eigenen Datei):

```
public class A {
    int method(A a) { return 1; }
    static int method(B b) { return 2 + b.method((A)b); }
}

public class B extends A {
    final int method(A a) { return 4 + super.method(a); }
    int method(C c) { return 8 + this.method((B)c); }
}

public class C extends B {
    int method(C c) { return 16 + method((A)c); }
}
```

Welche der folgenden Methodenaufrufe führen zu einem *Compiler-Fehler*, welche zu einem *Laufzeitfehler*? Wozu werten die funktionierenden Aufrufe aus?

```
A a = new A();
B b = new B();
C c = new C();
```

	Statement	Ergebnis
1.	`a.method(a);`	
2.	`a.method(c);`	
3.	`b.method(b);`	
4.	`b.method(c);`	
5.	`c.method(a);`	
6.	`c.method(c);`	
7.	`a.method((A)c);`	
8.	`b.method((B)c);`	
9.	`b.method((A)b);`	
10.	`c.method((A)c);`	
11.	`a.method((B)a);`	
12.	`((A)c).method(a);`	
13.	`((A)c).method(c);`	
14.	`((B)c).method(c);`	
15.	`((B)c).method((A)c);`	
16.	`((A)b).method((A)c);`	
17.	`((C)c).method((C)b);`	
18.	`((C)c).method((A)new Object());`	
19.	`((C)c).method((Object)b);`	

(92) ★★★ Welche Ausgaben produzieren die unterhalb aufgeführten Aufrufe? Betrachten Sie die Aufrufe unabhängig voneinander (d. h. alle anderen seien jeweils auskommentiert).

```
class L {
    public String toString() { return this.getClass().getName(); }
    static void write(String s) { System.out.println(s); }
    public void some()     { write("A"); }
    public void some(L k)  { write("B"); }
    public void some(N n)  { write("C"); }
    public L another()     { return this; }
    public static void someStatic() { write("D"); }
}

class N extends L {
    public void some()     { write("E"); }
    public void some(L l)  { write("F"); l.some(); }
    public N another()     { return this; }
    public static void someStatic() { write("G"); }
}

class P extends L {
    public void some()     { write("H"); }
    public void f(N k) { ((L) k).some(super.another()); }
}

L l = new L();   N n = new N();   P p = new P();
```

	Statement	Ausgabe
1.	`l.someStatic();`	
2.	`n.someStatic();`	
3.	`((L) n).someStatic();`	
4.	`N.someStatic();`	
5.	`P.someStatic();`	
6.	`((L) n).some();`	
7.	`((N) l).some();`	
8.	`((P) n).some();`	
9.	`l.some(p);`	
10.	`n.some(n);`	
11.	`l.some((L) n);`	
12.	`(l = n).some(n);`	
13.	`new N().some(l);`	
14.	`p.some(n);`	
15.	`(n = l).some(n);`	
16.	`p.f(n);`	
17.	`System.out.println(n.another());`	
18.	`System.out.println(l.another());`	
19.	`System.out.println(((L) n).another());`	

(93) ★★☆

Gegeben seien folgende Klassen:

```
class X {
    int a = 4;
    int get() { return a; }
}

class Y extends X {
    static int a = 7;
    int get() { return a; }
    static void set(int x) { a = x; }
    static void set(char c) { a = 2 * c; }
}

class Z extends Y {
    static int b = 3;
    int get() { return b + a; }
    static int get(X x) { return x.a; }
    static void set(int i) { a = 3 * i; }
    static void set(X x, int i) { a = i; }
}
```

Welche Ausgaben produzieren die unterhalb aufgeführten Aufrufe? Die Statements werden in der angegebenen Reihenfolge nacheinander ausgeführt, d. h. sie **bauen aufeinander auf**! Könnte die Klasse `X` zusätzlich `final` markiert werden (ohne Fehler)?

```
Z z = new Z();
Y y = z;
```

	Statement	**Ausgabe**
1.	`System.out.println(z.a);`	
2.	`System.out.println(z.get(z));`	
3.	`System.out.println(((X) z).get());`	
	`z.set('c' - 'a' - 1);`	
4.	`System.out.println(z.get(z));`	
5.	`System.out.println(z.get());`	
	`y.set(2);`	
6.	`System.out.println(z.get());`	
	`z.set(y, 0);`	
7.	`System.out.println(y.get());`	

(94) ★★★ Geben Sie unter Betrachtung des nachfolgenden Codes an, welche Ausgaben die in der `main`-Methode markierten Aufrufe produzieren. Sollte einer der Aufrufe nicht kompilieren oder zu einem Laufzeitfehler führen, so erklären Sie, welcher Fehler warum und an welcher Stelle auftritt. Die Aufrufe sind isoliert zu betrachten, d. h. alle anderen Aufrufe seien jeweils auskommentiert.

```
public class BasicPoly {
    static void write(String s) { System.out.println(s); }

    static class A {
        void m(A a) { write("A.m(A)"); a.m((B)a); }
        void m(B b) { write("A.m(B)"); this.m((A)b); }
    }
    static class B extends A {
        void m(B b) { write("B.m(B)"); }
        void m(C c) { write("B.m(C)"); this.m(this); c.m(this); }
    }
    static class C extends A {
        void m(A a) { write("C.m(A)"); }
        A m(C c)    { write("C.m(C)"); return c; }
        A m(D d)    { write("C.m(D)"); return d; }
    }
    static class D extends B {
        void m(A a) { write("D.m(A)"); }
        void m(B b) { write("D.m(B)"); b.m(new B()); }
        void m(C c) { write("D.m(C)"); c.m(c); super.m(c); }
    }

    public static void main(String[] args) {
        B b = new B(); C c = new C(); D d = new D(); A x = b; A z = d;

        z.m(c);         // Aufruf 1
        x.m(d);         // Aufruf 2
        ((A)c).m(x);    // Aufruf 3
        c.m(b);         // Aufruf 4
        x.m(x);         // Aufruf 5
        z.m(b);         // Aufruf 6
        new A().m(b);   // Aufruf 7
        x.m(c);         // Aufruf 8
        z.m(d);         // Aufruf 9
        b.m(c);         // Aufruf 10
        ((B)c).m(x);    // Aufruf 11
        x.m(z);         // Aufruf 12
        b.m(a);         // Aufruf 13
        d.m(c);         // Aufruf 14

        c.m(c.m(d));       // Aufruf 15
        c.m(c).m(c);       // Aufruf 16
        c.m(d).m(c.m(c)); // Aufruf 17
    }
}
```

(95) ★★☆ Geben Sie an, welche der in der Tabelle unterhalb genannten Anweisungen nicht übersetzt werden können (*Compiler-Fehler*), welche *zur Laufzeit* zu einem Fehler führen und welche Ausgaben von den funktionierenden Statements produziert werden. Die Ausführung der Teilaufgaben erfolgt *unabhängig voneinander* (also nicht sequenziell). Begründen Sie Fehler kurz.

```
class K extends Object {
    public void  b()    { b(this); }
    private void b(K k) { write("K.b"); }
    public void  c()    { this.c(new L()); }
    public void  c(K k) { write("K.c"); }
}
class L extends K {
    public void a()     { super.c(this); }
    public void b(K k)  { write("L.b"); }
    public void c(L l)  { write("L.c"); }
}
class M<N> extends K {
    public void c(M<N> m) { write("M<N>.c"); }
}
```

	Statement	Ergebnis
a)	`K k = new L();` `k.c(k);`	
b)	`(new L()).a();`	
c)	`((K) (new L())).c(new L());`	
d)	`L l = (L) (new K());` `l.c();`	
e)	`((K) new L()).b();`	
f)	`(((new L()))).b(new L());`	
g)	`L l = new L();` `((K) l).c(new L());`	
h)	`new K().c(new L());`	
i)	`K l = new L();` `((K) new L()).b(l);`	
j)	`new L().a();`	
k)	`((K) (L) new K()).c(new L());`	
l)	`(K) new L().c();`	
m)	`K l = new L();` `l.c(l);`	
n)	`L l = (L) (K) (new L());` `l.b();`	
o)	`((K)(K)(K) new K()).c(new L());`	
p)	`L l; K k = l = new L();` `k.c(l);`	

q)	`M<K> m = new M<K>(); m.c(m);`	
r)	`M<K> m = new M<L>(); m.c(m);`	
s)	`M<L> m = new M<L>(); m.c(m);`	
t)	`(new M<K>()).c(new L());`	
u)	`(new M<new K()>()).c();`	
v)	`new M<M<K>>().b();`	
w)	`M<K> m = new M<K>();` `((K)m).c(m);`	
x)	`new M<K>().c(new M<L>());`	

y)	`class Y extends L {`
	`  private void b(K k) {}`
	`}`	

z)	`class Z<R> extends M<R> { }`	
A)	`class A extends M<N> { }`	
B)	`class B<R,S> extends M<R> { }`	
C)	`class C extends M<K> { }`	
D)	`abstract class D extends M<L> { }`	
E)	`new L().c((L) ((K) new L()));`	
F)	`class F<P> extends M<M<P>> { }`	
G)	`class G extends M { }`	

(96) ★☆☆ **Polymorphie im Visitor-Pattern**: Gegeben seien die folgenden Klassen.

```
class Parent {
    public void accept(Visitor v)  {  v.visit(this);  }
}

class Child1 extends Parent {}

class Child2 extends Parent {
    public void accept(Visitor v)  {  v.visit(this);  }
}

class Visitor {
    private int x = 0;
    public int getX() { return x; }

    public void visit(Parent p)  {  x = 1;  }
    public void visit(Child1 p)  {  x = 2;  }
    public void visit(Child2 p)  {  x = 3;  }
}
```

Welche Ausgabe produziert die nachstehende Codesequenz?

```
Visitor v = new Visitor();  System.out.print(v.getX());
new Parent().accept(v);     System.out.print(v.getX());
new Child1().accept(v);     System.out.print(v.getX());
new Child2().accept(v);     System.out.print(v.getX());
```

(97) ★★★ **Polymorphie mit mehreren Parametern und Mehrdeutigkeit (*ambiguity*):**

Geben Sie für jeden der markierten Aufrufe an, welche Ausgabe dieser produziert. Sollte ein Aufruf Ihrer Meinung nach zu einem Compiler-Fehler oder Laufzeitfehler führen, so ist dies zu begründen. Gehen Sie für die übrigen Aufrufe davon aus, dass alle nicht funktionierenden Aufrufe auskommentiert sind.

Hinweis: Die Funktion *write* sei gegeben und äquivalent zu `System.out.println`.

```
public class AmbiguousPoly {
    static class A {
        void m(A a1, A a2) { write("A.m(A,A)"); a2.m(this, (C)a1); }
        void m(B b, A a) { write("A.m(B,A)"); }
        void m(A a, C c) { write("A.m(A,C)"); }
    }
    static class B extends A {
        void m(A a, B b) { write("B.m(A,B)"); a.m(b, this); }
    }
    static class C extends B {
        void m(B b, A a) { write("C.m(B,A)"); m((A)this, a); }
        void m(B b, C c) { write("C.m(B,C)"); this.m((A)b, (B)c); }
    }

    public static void main(String[] args) {
        A a = new A(); B b = new B(); C c = new C();

        a.m(c, b);        // Aufruf 1: _____
        c.m(a, (B)c);     // Aufruf 2: _____
        b.m(c, b);        // Aufruf 3: _____
        ((A)b).m(b, b);   // Aufruf 4: _____
        c.m(b, b);        // Aufruf 5: _____
        ((A)c).m(c, b);   // Aufruf 6: _____
        a.m(a, b);        // Aufruf 7: _____
        ((A)b).m((A)c, b); // A. 8: _____
        c.m(b, c);        // Aufruf 9: _____
        ((B)c).m(c, a);   // Aufruf 10: _____
        b.m(c, c);        // Aufruf 11: _____
        c.m(c, c);        // Aufruf 12: _____
        ((A)c).m(b, c);   // Aufruf 13: _____
        c.m((A)c, b);     // Aufruf 14: _____
        ((C)a).m(b, a);   // Aufruf 15: _____
    }
}
```

(98) ★★★ **Polymorphie mit Attributen (+ Konstruktoren):**

Betrachten Sie den folgenden Code und geben Sie für jeden der markierten Aufrufe an, welche Ausgabe durch dessen Ausführung produziert wird. Falls Sie der Meinung sind, dass ein Aufruf zu einem Fehler führt, so begründen Sie Ihre Entscheidung ausführlich (Art des Fehlers + Ursache). Alle Aufrufe sind unabhängig voneinander zu betrachten, d. h. alle anderen Aufrufe seien jeweils auskommentiert.

```
public class AttributPoly {

    static class A {
        protected B a;
        public A() { a = new C(this); }
        public A(B b) { set(b); }
        public void set(B b) { a = b; }
        void f(B b) { System.out.println("A.f(B)"); a.f(this); }
    }

    static class B extends A {
        public B(B b) { super(b); }
        public B() { }
        void f(A a) { System.out.println("B.f(A)"); this.a.f(a); }
        void f(B b) { System.out.println("B.f(B)"); }
    }

    static class C extends B {
        private A a;
        public C(A a) { super(null); this.a = a; }
        void f(A a) { System.out.println("C.f(A)"); }
        void f(B b) { System.out.println("C.f(B)"); a.f(this); }
        void f(C c) { System.out.println("C.f(C)"); }
    }

    public static void main(String[] args) {
        B b = new B(); A a = new A();
        B c = a.a;
        c.set(b);
        b.a.set(c);

        b.f(a);          // Aufruf 1: _____
        c.f((C)c);       // Aufruf 2: _____
        a.f(a);          // Aufruf 3: _____
        b.a.f(c);        // Aufruf 4: _____
        a.f(b);          // Aufruf 5: _____
        ((C)c).a.f(c);   // Aufruf 6: _____
        ((C) a.a.a.a).a.a.a.f(a);   // Aufruf 7: _____
    }
}
```

(99) ★★★ **Polymorphie mit Generics** (+ Attribute + Konstruktoren):

```
public class GenericsPoly {
    static void print(String s) {  System.out.println(s);  }

    static class A<T extends B> {
        protected T e;
        public void set(T elem) {  e = elem;  }
        void f(B b)     {  print("A.f(B)");  }
        void f(C<E> c) {  print("A.f(C)"); e.f(e); c.f(e);  }
        void g(F f)     {  print("A.g(F)");  }
    }
    static class B extends A<B> {
        public B() {  set(this);  }
        void f(C<E> c) {  print("B.f(C)");  }
        void f(F f)     {  print("B.f(F)"); e.g(f);  }
    }
    static class C<R extends E> extends B {
        public R e;
        public C(R r) {  this.e = r;  }
        void f(B b)        {  print("C.f(B)");  }
        void f(A<C<E>> a) {  print("C.f(A)"); a.f(this); e.g(e);  }
        void g(F f)        {  print("C.g(F)"); e.g(f);  }
    }
    static class D extends A<C<E>> implements E {
        protected A<B> e;
        public D() {  e = new C<D>(this);  }
        public void f(D d) {  print("D.f(D)"); e.f(super.e);  }
        public void g(F f) {  print("D.g(F)");  }
        public void g(E e) {  print("D.g(E)");  }
    }
    interface E {
        void g(E e);
    }
    static class F extends D {
        void f(A<C<F>> a)  {  print("F.f(A)"); f(a.e); e.f(e.e); }
        public void f(D d) {  print("F.f(D)"); super.f(this);  }
    }
}
```

Geben Sie für jeden der rechts dargestellten Aufrufe an, welche Konsolenausgabe durch dessen Ausführung produziert wird. Ggf. auftretende Fehler sind zu begründen. Betrachten Sie die folgenden Statements vor der Ausführung jedes Aufrufs als gegeben. Die Aufrufe erfolgen aus der `main`-Methode einer Klasse im Package der Klasse `GenericsPoly`.

```
A<C<F>> a = new A<>();
C<F> c = new C<>(new F());
a.set(c);
D d = c.e;
```

Aufrufe:

1. `d.f(c);`
2. `c.f(a.e);`
3. `d.f(a);`
4. `d.f(d);`
5. `new B().f(c.e);`
6. `c.f(d);`
7. `d.e.f(c);`
8. `a.f((C<E>)c);`
9. `((F)d).f(a);`
10. `c.f((F)d);`
11. `a.f(((A<C<E>>)d).e);`

Polymorphie EXTREM

Aufgrund der Steigerung des Schwierigkeitsgrades der Polymorphie-Klausuraufgabe über die letzten Jahre soll diese Aufgabe bereits vorab einen Nervenzusammenbruch garantieren, um diesen während der Klausur zu vermeiden.

Du solltest zuvor die anderen Polymorphie-Aufgaben aus dem Skript bearbeitet haben und die einfacheren Altklausuraufgaben (d. h. bis inkl. WS 16/17) gut lösen können, bzw. dich sehr sicher bei Polymorphie fühlen. Diese Aufgabe wird dir selbst mit perfektem Wissen über Polymorphie noch offene Wissenslücken aufzeigen. Solltest du diese Aufgabe fehlerfrei lösen können, dann ... niemand kann diese Aufgabe fehlerfrei lösen. Solltest du diese Aufgabe aber zumindest größtenteils lösen können, dann bist du wirklich gut auf Polymorphie vorbereitet. Die Lösungen sind getestet (auch wenn du es an mancher Stelle nicht wahrhaben wollen wirst).

In dieser Aufgabe ist neben den für Polymorphie typischen Kenntnissen auch das Verständnis folgender Java-Konzepte notwendig:

- *Generics, generische Vererbung und Type Erasure (Übersetzen von Generics)*
- *Überladung und Mehrdeutigkeit bei Methoden und Konstruktoren (ambiguity)*
- *Ketten polymorpher Zugriffe (`a().b().c()`)*
- *primitive Datentypen, implizites Typecasting, Wrapper-Klassen, Autoboxing (→ Kap. 9)*
- *Sichtbarkeitsmodifier `private` (→ wird nicht vererbt, aber ggf. über `super` zugreifbar)*
- *Ausdrücke und Operatoren (Bindungsstärken), insb. Zuweisungen der Form:*
  `(a = b).m()` (→ stat. Typ von „`(a = b)`“ ≙ stat. Typ von `a`; dyn. Typ ≙ dyn. Typ von `b`)

Die nachfolgende Aufgabenstellung bezieht sich auf die Klasse `Poly`, welche du auf den folgenden Seiten findest. `Poly` enthält mehrere statische innere Klassen. Statische innere Klassen sind im Gegensatz zu nicht-statischen inneren Klassen nicht an eine Instanz der umliegenden Klasse gebunden. `Poly` stellt die Methode `out` bereit. Diese erwartet beliebig viele Objekte (mind. eines) als Parameter und gibt diese in der Form `o1(o2, o3, ...)` auf der Konsole aus. Beachte, dass jede Art von Zahlen (außer `char`) in Java bekanntlich stets dezimal und Gleitkommazahlen immer in Punktnotation ausgegeben werden. Beispiele:

- *`out("f")` gibt `f()` aus*
- *`out("g", 1)` gibt `g(1)` aus*
- *`out("F.m", 'a', 1.1, 0x20, .1)` gibt `F.m(a,1.1,32,0.1)` aus*

Ich empfehle dir (insb. bei Teilaufgabe 3), verschiedene Skizzen – z. B. ähnlich zu einem Objektdiagramm - anzufertigen, um den Überblick zu behalten. Die Lösung enthält eine solche Skizze.

(100) ★★☆ **Polymorphie mit allem (und scharf)**: Diese Aufgabe ist in mehrere unabhängige Teilaufgaben gegliedert. Geben Sie jeweils für jeden der markierten Aufrufe an, welche Ausgabe dieser produziert. Sollte ein Aufruf Ihrer Meinung nach zu einem Compiler- oder Laufzeitfehler führen, so ist mit Begründung anzugeben, wo und weshalb dieser auftritt. Außerdem sind alle bis zum Auftreten einer Exception produzierten Ausgaben trotzdem anzugeben, bspw. in der Form: „*`Y.m() Z.k(14)` - dann ArithmeticException (division by zero)*“. Gehen Sie für die übrigen Aufrufe davon aus, dass die nicht funktionierenden Aufrufe auskommentiert sind.

```
public class Poly {
 public static void out(Object m, Object... o) {
   System.out.println(m + "(" + java.util.Arrays.stream(o)
        .map(Object::toString)
        .collect(java.util.stream.Collectors.joining(",")) + ')');
 }
 static class T<Gen> {
   public Gen t;
   public T(Gen t) { out("T","Gen"); this.t = t; }
   public T(Integer x) { out("T","Integer"); this.t = null; }
   public void m(short x) { out("T.m","(short)"+x);
                            this.m(x - 0b10); m((int)x, (int)x); };
   public void m(double x) { out("T.m",x); };
   public void m(double x, int y) { out("T.m",x,y); }
   private void m(S<Gen> s) { out("T.m",s); m((T<Gen>)s); }
   public void m(T<Gen> t) { out("T.m","T"); }
   public T<Gen> m() { out("T.m"); return this instanceof S ?
                                         ((S<Gen>)this).t : this; }
 }
 static class U extends T<Character> {
   public U(char c) { super(c); out("U","char"); }
   public U() { super('A' + 1); out("U"); }
   public void m(int x) { out("U.m",x); this.m(x * .1, x); }
   public void m(int x, double y) { out("U.m",x,y); }
   void m(S<Character> s) { out("U.m","S"); super.m(s); s.m(t); }
 }
 static class X<Z> extends S<Z> {
   public X(Integer i) { super(i); out("X","Integer");
                         t = (T<Z>) new U(); }
   public X(Object o, T<Z> t) { super(o); out("X","Object","T");
                                this.t = t; }
   public void m(int x, int y) { out("X.m",x,y); };
   public void m(int x) { out("X.m",x); super.m(x, x); }
   public void m(double x) { out("X.m",x); };
   public void m(T<Z> t) { out("X.m","T"); }
   public void m(S<Z> s) { out("X.m", "S"); super.m(s);
                           this.t.m(s); s.m(this); }
   public void m(U u) { out("X.m", "U"); m(u.t); t.m().m(u.t); }
 }
 static class S<P> extends T<P> {
   protected T<P> t;
   public S() { this(null); out("S"); }
   public S(int i) { super(i); out("S","int"); t = new X<P>(i, this);
   public S(Object o) { super((P)o); out("S","Object"); }
   public void m(double x, int y) { out("S.m",x,y); m(y); }
   public void m(U u) { out("S.m","U"); this.m(u.t); t.m(this); }
   public T<P> m() { out("S.m"); this.m(this); return super.m(); }
   public String toString() { return "S"; }
 }
```

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

        {    // Teilaufgabe (1) – Primitve Datentypen & Ambiguous Calls

             T<Integer> t1 = new S<>();
             X<Integer> x1 = new X<>(null, null);
             T<Integer> t2 = x1;
             U u1 = new U();
             T<Character> t3, t4 = u1;
             S<Integer> s1;

             byte b1 = 1, b2 = 2;
             char c = 'A';
             short sh1 = 3, sh2 = 4;
             int in = 5;
             float f = sh1 * 2;
/* Aufruf */
    /* 1: */   t1.m(c);
    /* 2: */   t1.m(b1);
    /* 3: */   t1.m(b1 + 1);
    /* 4: */   t1.m(2, 4);
    /* 5: */   t1.m(sh2 - sh1);
    /* 6: */   ((X<Integer>) t1).m(in);
    /* 7: */   ((S<Integer>) t1).m((long)in);

    /* 8: */   t2.m(b2);
    /* 9: */   t2.m(in);
   /* 10: */   t2.m(in == 1);
   /* 11: */   t2.m(in, in);
   /* 12: */   t2.m(sh2, b1);

   /* 13: */   x1.m(5/2);
   /* 14: */   x1.m(b1);
   /* 15: */   x1.m(1l, 07);
   /* 16: */   x1.m(sh1, in);
   /* 17: */   ((S<Integer>) t2).m(0x9);
   /* 18: */   ((S<Object>) t2).m(b1);
   /* 19: */   ((S<Integer>) t2).m(2, 4);
   /* 20: */   ((X<Integer>) t2).m(c);
   /* 21: */   ((S<Integer>) t2).m((byte)f);
   /* 22: */   ((U) t2).m(in);
   /* 23: */   (t3 = t2).m(in);
   /* 24: */   (s1 = x1).m(in, in);
   /* 25: */   (t3 = x1).m(in);
   /* 26: */   ((T<Integer>)(s1 = x1)).m(in, new Integer(9));

   /* 27: */   u1.m(b1 + sh1);
   /* 28: */   u1.m((int) 2*f);
   /* 29: */   u1.m(sh1, in);
   /* 30: */   u1.m(3/2, 1L);
   /* 31: */   u1.m(f, f);
   /* 32: */   u1.m(f, b1);
   /* 33: */   t4.m(in, f);
   /* 34: */   t4.m(in, in);
   /* 35: */   ((S<Character>) u1).m(in);
   /* 36: */   ((S<Character>) t4).m(5d);
   /* 37: */   (t3 = u1).m(in);
   /* 38: */   u1.m(in = (short)sh1);
   /* 39: */   t4.m(13, f = (int)in);
        }  // end: innerer Block zu Teilaufgabe 1
```

```
                {       // Teilaufgabe (2)
                        // (a) Konstruktoren bei generischer Vererbung
/* Aufruf */            Integer i1 = 13;
   /* 1: */             T<Character> t1 = new T<>('A');
   /* 2: */             T<Float> t2 = new T<>(11);
   /* 3: */             T<Object> t3 = new T<>((short)5);
   /* 4: */             T<Integer> t4 = new T<>(i1);
                        T<Character> t5;

   /* 5: */             S<Integer> s1 = new S<>(3 + 1);
   /* 6: */             S<Character> s2 = new S<>();
   /* 7: */             S<Long> s3 = new S<Long>('B');
   /* 8: */             S<Float> s4 = new S<Float>(i1);
                        S<Character> s5 = new S<Character>(15);
   /* 9: */             X<Float> s6 = new X<Float>(4F);

  /* 10: */             U u1 = new U();
  /* 11: */             U u2 = new U('C');

  /* 12: */             X<Character> x1 = new X<>(7);
  /* 13: */             X<Integer> x2 = new X<>(1.5, s1);

  /* 14: */             System.out.println(t1.t);
  /* 15: */             System.out.println(t2.t);
  /* 16: */             System.out.println(t3.t);
  /* 17: */             System.out.println(1 + t4.t);
  /* 18: */             System.out.println(x1.t.t);
  /* 19: */             System.out.println(((T<Integer>)s1).t);
  /* 20: */             System.out.println(((T<Character>)s2).t);
  /* 21: */             System.out.println(((T<Long>)s3).t);
  /* 22: */             System.out.println(((T<Float>)s4).t);
  /* 23: */             System.out.println(((T<Character>)x1).t);
  /* 24: */             System.out.println(((T<Integer>)x2).t);

/* Aufruf */            // (b) Polymorphie bei privaten Methoden und gen. Attributen

   /* 1: */             t1.m(s2);
   /* 2: */             ((S<Float>) t2).m(s4);

   /* 3: */             s1.m(x2);
   /* 4: */             s1.m(x1);
   /* 5: */             ((T<Character>) s2).m(u2);

   /* 6: */             x1.m(s2);
   /* 7: */             ((S<Integer>) x2).m(s1);
   /* 8: */             x1.m(u2);
   /* 9: */             x1.m(u1);
  /* 10: */             x1.m((T<Character>) u1);
  /* 11: */             x2.m((S<Character>) u2);
  /* 12: */             x2.m(u2);

  /* 13: */             (t5 = u1).m((S<Character>) x1);
  /* 14: */             u1.m(x1);
  /* 15: */             u2.m(s2);
  /* 16: */             u1.m(u2);

  /* 17: */             s3.m((T<Character>) u2);
  /* 18: */             s3.m(u2);
  /* 19: */             s2.m(u2);
```

```
                // (c) Polymorphe Aufruf-Ketten
/* Aufruf */
    /* 1: */   s1.m(u2.t);
    /* 2: */   s1.m(u1.t);
    /* 3: */   s3.m().m(t1.t);
    /* 4: */   x2.m(s1.t.t);
    /* 5: */   System.out.println(s1.t.t.t);
    /* 6: */   s4.t.m();
    /* 7: */   System.out.println(((T<Integer>) s1.m()).t);
    /* 8: */   s5.m().m(u2);
    /* 9: */   u1.m((X<Character>)s5.m());
   /* 10: */   s5.t.m().m(x1);
   /* 11: */   x1.m().m(s1.m().t);
   /* 12: */   t1.m(((T<Integer>)x2).t);
   /* 13: */   s1.t.m(1 + (int)(double)((T)x2).t);
   /* 14: */   t3.m(t3.t);
   /* 15: */   x1.m().m().m().m(1,5 % 3);
   /* 16: */   x1.m(s3.t.t);
   /* 17: */   ((X<Integer>) s1.t).m(x2);
   /* 18: */   (s1 = s2).m(x2);
   /* 19: */   ((S<Byte>) new X<Byte>("r",
                          new X<Byte>(7))).m(new U((char)6));
        }  // end: innerer Block zu Teilaufgabe 2

        {  // Teilaufgabe (3) – Gen. Klassen als Typparam. & Type Erasure
            S<Character> s1 = new S<>("str");
            X<Character> x1 = new X<>(3.0, s1);
            s1.t = new U('a');
            S<S<Character>> s2 = new S<>(-7);
            T<S<Character>> s3 = s2.t;
            Object o = s3.t;
            s3.t = x1;
            ((T) s2).t = o;
            s3 = new T<>(s1);
            s1 = x1 = null;
/* Aufruf */
    /* 1: */   s2.t.m(s3);
    /* 2: */   System.out.println(s2.t.t.t.t);
    /* 3: */   s2.t.t.m((char)65);
    /* 4: */   ((T<S<Character>>) s2.t.m()).m(1, s3.t.t.t);
    /* 5: */   s2.m().t.m((U) s3.t.t);
    /* 6: */   s3.t.t.m(s2);
    /* 7: */   s2.m((U) s3.m().t.m());
    /* 8: */   System.out.println(((T<Character>)s2.t.t).t);
    /* 9: */   s3.m(((T<Character>)s2.t.t).t);
   /* 10: */   System.out.println(((T<S<Character>>) s2.t).t.t.m().t);
        }  // end: innerer Block zu Teilaufgabe 3
    }  // end: main-Methode
}  // end: Poly-Klasse
```

18. Sortier- und Suchalgorithmen

In der Vorlesung werden i. d. R. einige der Algorithmen aus folgenden Aufgaben behandelt, wobei derselbe Algorithmus auf unterschiedlichste Arten implementiert werden kann, ob objektorientiert oder prozedural, auf Listen oder Arrays, *in-place* oder mit Rückgabe, usw. Ein Algorithmus wird in der Klausur manchmal, aber nicht zwangsläufig abgefragt. Außerdem ist es möglich, dass ein Algorithmus implementiert werden soll, der nicht in der Vorlesung besprochen wurde. In diesem Fall ist der Algorithmus in der Aufgabenstellung beschrieben. Der Fokus sollte bei der Vorbereitung nicht darauf liegen, eine bestimmte Implementierung auswendig zu lernen, sondern verschiedene Algorithmen grundsätzlich zu verstehen, um sich in der Klausur schneller in eine Aufgabe (ggf. mit einem frei erfundenen Algorithmus) eindenken zu können.

(101) ★★☆ **Insertionsort**: Implementieren Sie die folgende Methode, welche ein Array von Zahlen als Parameter übergeben bekommt und *in-place* (d. h. direkt auf der Datenstruktur, ohne ein neues Array zu erzeugen) in *aufsteigender* Reihenfolge sortiert.

Der Algorithmus verhält sich wie folgt: Anfangs wird das erste Elemente als „korrekt sortiert“ angesehen, alle darauffolgenden als unsortiert. Das Array wird nun sortiert, indem man von vorne nach hinten über das Array iteriert und jedes Element so lange nach vorne „schiebt“ (d. h. mit den Vorgängern vertauscht), bis es korrekt einsortiert ist.

Implementieren Sie zuvor außerdem die Hilfsmethode `swap`, welche die Elemente an den Indizes `a` und `b` im Array vertauscht.

```
private static void swap(long[] numbers, int a, int b) {



}

public static void insertionSort(long[] numbers) {









}
```

102 ★★☆ **Selectionsort**: Wir wollen ein Array im Folgenden *absteigend* sortieren. Dieser Algorithmus teilt das Array gedanklich in einen „bereits sortierten“ (anfangs leeren) und in einen „noch unsortierten“ (anfangs alles) Arrayteil. Im noch unsortierten Arrayteil wird nun jeweils das größte Element gesucht und mit dem ersten Element dieses Arrayteils vertauscht, sodass dieses Element nun ganz vorne im unsortierten Arrayteil steht. Das Element gilt nun jedoch als „sortiert“ und stellt somit das letzte Element des sortierten Arrayteils dar. Die Hilfsmethode *swap* zum Vertauschen zweier Elemente sei gegeben.

```
public static void selectionSort(int[] array) {
    for (int i = 0; i < array.length-1; i++) {
        int maxIndex = ____________;
        for (int j = ____________; j < array.length; j++)
            if ( ____________________________ )
                maxIndex = j;
        swap(array, ____________, ____________);
    }
}
```

103 ★★☆ **Bubblesort**: Implementieren Sie die folgende Methode, welche ein Array von Zahlen als Parameter übergeben bekommt und *in-place* (d. h. direkt auf der Datenstruktur, ohne ein neues Array zu erzeugen) in *aufsteigender* Reihenfolge sortiert.

Der Algorithmus verhält sich wie folgt: Große Elemente werden nach und nach ans Ende der Liste gebracht. Dazu wird das Array mehrmals von vorne bis hinten durchlaufen und zwei benachbarte Elemente jeweils genau dann vertauscht, wenn dies nötig ist. Dadurch erreicht man, dass nach jedem Schleifendurchlauf ein weiteres Element am Ende der Liste richtig einsortiert ist. Im ersten Schleifendurchlauf muss das Array somit vollständig durchlaufen werden, im zweiten Durchlauf bis auf das letzte (da dieses im ersten Durchlauf richtig einsortiert wurde), im dritten Schleifendurchlauf bis auf die beiden letzten usw. Der Algorithmus ist zu Ende, wenn in einem Schleifendurchlauf keine Vertauschung durchgeführt wurde. Im schlechtesten Fall sind für ein Array der Länge n somit $n-1$ Durchläufe nötig. Die Methode `bubbleSort` soll zurückgeben, wie viele Vertauschungen nötig waren. `swap` sei wieder gegeben.

```
public static int bubbleSort(int[] arr) {




}
```

104 ★★★ **Mergesort rekursiv**: Der Algorithmus unterteilt das zu sortierende Array in zwei ungefähr gleich große Teilhälften, welche rekursiv mittels *Mergesort* sortiert werden. Anschließend werden die beiden sortierten Teile wieder zusammengeführt (*merging*).

Implementieren Sie diesen Algorithmus in mehreren Schritten:

1. Hat das zu sortierende Array die Länge 0 oder 1, so ist nichts zu tun.
2. Erzeuge ein neues Array für beide Hälften (disjunkt).
3. Sortiere die beiden Hälften mittels *Mergesort*.
4. *Merging*: Um eine *aufsteigende* Sortierung zu generieren, wird über die beiden Arrays iteriert und immer das nächstkleinere Element in das Ursprungsarray zurückkopiert. Verwende dazu für jede Hälfte einen eigenen Indexzähler, welcher erhöht wird, sobald ein Element aus der entsprechenden Hälfte in `array` zurückgeschrieben wurde. Ist eine Hälfte bereits vollständig eingefügt worden, so entfällt dieser Vergleich natürlich und man wählt direkt das Element aus der anderen Hälfte.

```
public static void mergeSort(int[] array) {
    // (1.) Abbruchbedingung: nichts zu sortieren




    // (2.) Array in linke und rechte Hälfte teilen:




    // (3.) Beide Hälften sortieren:
```

```
        // (4.) Die sortierten Hälften in das Ursprungsarray mergen:



}
```

Der Mergesort-Algorithmus lässt sich auch iterativ implementieren. Da es dafür aber unzählige verschiedene Umsetzungen gibt, ist es sehr schwierig, hierfür einen Lösungsvorschlag anzubieten, daher habe ich diese Aufgabe nicht ins Skript aufgenommen. Du kannst dich trotzdem daran versuchen und deine Ergebnisse mit der rekursiven Implementierung vergleichen.

105 ★★☆ **Quicksort rekursiv**: Dieser Algorithmus verfolgt folgende Strategie, wenn es um das Sortieren eines Arrays (hier: `double`-Array) geht.

Zunächst nehmen wir an, es sei neben dem Array auch eine linke (`from`) sowie rechte (`to`) Grenze des zu sortierenden Bereichs gegeben. Diese Grenzen seien inklusive. Hat der zu sortierende Bereich nicht mindestens die Länge 2, so ist nichts zu tun – der Bereich ist offensichtlich bereits sortiert. Anderenfalls wählen wir ein sog. *Pivot*-Element aus, welches wir im gegebenen Bereich so positionieren wollen, dass alle Zahlen links vom Pivot-Element kleiner als dieses, und alle Elemente rechts vom Pivot-Element größer oder gleich dem Pivot-Element sind. Als Pivot-Element wählen wir dabei immer das rechteste Element des Teilbereichs (also das Element am Index `to`; diese Wahl könnte in einer anderen Implementierung jedoch auch anders sein).

Fortsetzung auf der nächsten Seite.

Beispiel: Angenommen wir möchten den Teilbereich [5 1 3 9 1 5 3] mit Quicksort sortieren, so muss das Array zunächst irgendwie beliebig umsortiert werden, sodass das Pivot-Element (hier: 3) nur noch kleinere Elemente links von sich stehen hat, in diesem Beispiel also [1 1 3 _ _ _ _], wobei die Unterstriche für die verbleibenden Zahlen stehen.

Für die Einsortierung des Pivot-Elements gehen wir wie folgt vor: Wir verwenden zwei Indexzähler, wobei einer ganz rechts und der andere ganz links (Bereichsgrenzen) beginnt. Den linken Zähler erhöhen wir solange, bis wir auf ein Element stoßen, das größer/gleich dem Pivot-Element ist. Den rechten Zähler erniedrigen wir solange, bis wir auf ein Element stoßen, das kleiner ist als das Pivot-Element. Dann tauschen wir die beiden gefundenen Elemente. Dieses Vorgehen wiederholen wir solange, bis der linke Zähler gleich dem rechten Zähler ist. Anschließend wird das Pivot-Element mit dem ersten Element, das größer/gleich dem Pivot-Element ist, vertauscht (dieses steht am rechten Indexzähler). Für das Vertauschen stehe wieder `swap(double[], int, int)` bereit (`swap(arr, i, j)` ist äquivalent zu `swap(arr, j, i)`).

Nachdem das Pivot-Element nun an der richtigen Stelle steht, wird der Teil rechts bzw. links vom Pivot-Element rekursiv mittels Quicksort sortiert.

```
private static void quickSort(double[] array, int from, int to) {
    int length = ____________________;
    if (____________________) {
        int left = from;
        int right = to;

        while (____________________) {
            while (________________________________________)
                left++;
            while (________________________________________)
                right--;
            swap(array, ____________________, ____________________);
        }

        swap(array, ____________________, ____________________);

        int pivotIndex = ____________________;

        quickSort(array, from, pivotIndex-1);
        quickSort(array, pivotIndex+1, to);
    }
}

public static void quickSort(double[] array) {
    quickSort(________________________________________);
}
```

106 ★★★

Binärbäume: Als *Binärbaum* wird eine baumartig angeordnete Datenstruktur bezeichnet, in der die Daten in sog. *Knoten* gespeichert sind, wobei ein Knoten neben einem *Element* maximal zwei Nachfolgerknoten besitzt. Der oberste Knoten wird als *Wurzel* bezeichnet. *Blätter* sind Knoten, die keinen Nachfolger mehr besitzen.

Wir definieren nun einen Binärbaum zur *sortierten* Speicherung von Zahlen rekursiv in einer Klasse `Binaerbaum`. In dieser Implementierung repräsentiert die Klasse `Binaerbaum` gleichzeitig Knoten und Blätter, daher sind rechter und linker Nachfolger optional (können jeweils **`null`** sein).

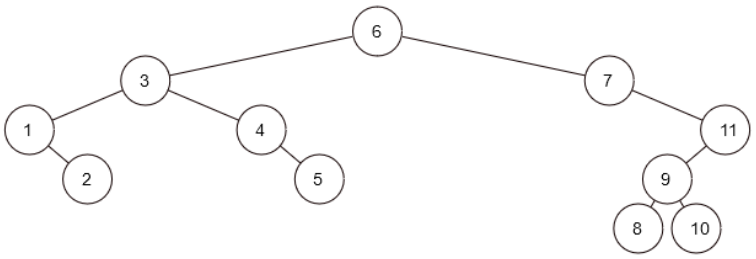
Invariante: Alle Elemente des linken Teilbaums (falls vorhanden) sind kleiner, die des rechten Teilbaums (falls vorhanden) sind größer/gleich dem Elternknoten. Der Konstruktor erzeugt einen einelementigen Baum bzw. Knoten.

Implementieren Sie die Methode **`binaereSuche(int)`**, welche das als Parameter übergebene Element im Baum sucht und **`true`** zurückgibt, falls es gefunden wurde, **`false`** sonst. Die Methode soll Gebrauch von der Invariante des Baums machen. Schreiben Sie außerdem die Methode `laenge()`, die die Anzahl der Elemente im Baum ermittelt.

```
public class Binaerbaum {
    protected int element;
    protected Binaerbaum links, rechts;
    public Binaerbaum(int element) {
        this.element = element;
    }
    public boolean binaereSuche(int wert) {









    }
    public int laenge() {









    }
}
```

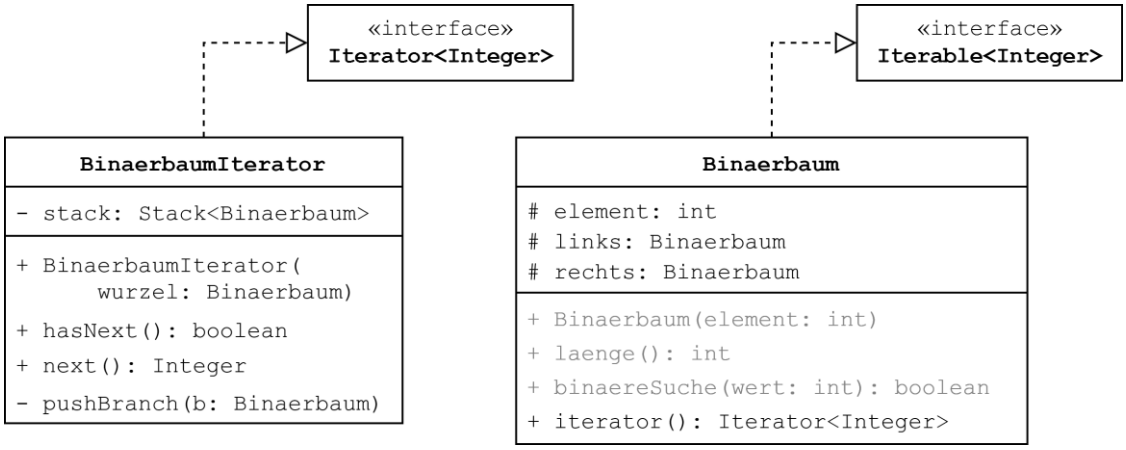
(107) ★★★ Sie sollen nun einen **Iterator** für die Binärbaumimplementierung aus Aufgabe 106 anbieten, durch welchen die Elemente des Binärbaums in *absteigender* Reihenfolge zurückgegeben werden. Gehen Sie davon aus, dass die Invariante wie zuvor beschrieben gilt.

Beispiel:



Schreiben Sie eine neue Klasse `BinaerbaumIterator`, welche das Interface `Iterator<Integer>` implementiert. Die Methode `next()` soll wie üblich das jeweils nächste Element des Binärbaums zurückgeben, wobei das am weitesten rechts stehende Element zuerst zurückgegeben werden muss (hier: 11, 10, 9, 8, 7, ...). Nachdem es keine Möglichkeit gibt, von einem Knoten (z. B. 11) zurück zum Elternknoten (7) zu kommen, muss der Iterator Knoten zwischenspeichern. Verwenden Sie daher einen `Stack` zur Verwaltung der noch zu bearbeitenden (übersprungenen) Knoten. Die Klasse `Stack<E>` stellt die Methoden `push(E e)`, `pop()` und `isEmpty()` bereit (→ vgl. Kap. 16).

Hilfestellung: **Nur bei Bedarf** (erst ohne oder nur mit dem Diagramm versuchen)**!**



Gehen Sie schrittweise vor:

1. Im Konstruktor wird der Stack initialisert und der gesamte rechte Teilpfad sowie die Wurzel (oberstes Element, als Parameter erhalten) gepusht (→ 2.).
2. `pushBranch(b)` legt `b` sowie dessen rechtesten Teilzweig auf den Stack. Im obigen Beispiel würden bei Knoten 3 erst 3, dann 4 und dann 5 gepusht werden. Für die Wurzel würde 6, 7, 11 in dieser Reihenfolge gepusht werden (→ 11 ist *next* Element).
3. `next()` entnimmt das oberste Element vom Stack und gibt dessen Inhalt zurück (anfangs: 11). Vor der Rückgabe muss der linke Nachfolger sowie dessen gesamter rechtester Pfad auf den Stack gelegt werden (→ 2.), im Beispiel für 11 also erst 9, dann 10. `next()` wirft eine `NoSuchElementException`, falls keine Elemente mehr im Stack.
4. `hasNext()` gibt `false` zurück, falls der Stack leer ist, anderenfalls `true`.
5. Die Klasse `Binaerbaum` muss um den `implements`-Zusatz sowie die `iterator()`-Methode erweitert werden, sodass jeder Binärbaum einen Iterator für sich zurückgibt.

```
import java.util.*;
public class BinaerbaumIterator implements Iterator<Integer> {









}

// Ergänzung zur Klasse Binaerbaum:
```

(108) ★★★ **Generische Binärbäume**: Der Binärbaum aus Aufgabe 106 (Seite 155) kann nur ganze Zahlen speichern und bietet nur wenige Methoden. In dieser Aufgabe soll es ermöglicht werden, Binärbäume für beliebige Typen (z. B. *Integer*, *Float*, *Double* oder einen selbst definierten Typ) zu erzeugen. Wir verwenden daher einen Typparameter `T`.

Um zwei beliebige Objekte gleichen Typs vergleichen zu können (was für die Prüfung nach Invariante nötig ist), darf der Typparameter eingeschränkt werden. Dieser implementiert das Interface `Comparable<T>` und somit die Methode `int compareTo(T o)`, die bei einem Aufruf der Form `a.compareTo(b)` einen negativen Wert zurückgibt, falls `a < b` gilt, und einen positiven Wert liefert, falls `a > b`.

Gehen Sie die Implementierung der generischen Klasse `BinaryTree` mit dem wie oben beschriebenen Typparameter `T` schrittweise an:

1) Implementieren Sie die öffentlichen Klassen `Node`, `Leaf` und `InnerNode` mit Typparameter `E` (gleiche Einschränkung wie `T`). `Node` stellt eine abstrakte Basisimplementierung dar, von der `Leaf` und `InnerNode` erben.

   `Node` verfügt über ein nicht-öffentliches, für beide Unterklassen zugreifbares aber nach erfolgter Zuweisung nicht mehr veränderliches Attribut `E elem`, welches im Konstruktor übergeben wird (jede der Klassen benötigt einen solchen Konstruktor). Ein `InnerNode` repräsentiert einen Knoten mit mindestens einem und maximal zwei direkten Nachfolgern. Indirekte Nachbarn oder Vorgänger werden nicht gespeichert. Ein `Leaf` (*Blatt*) hat keinen Nachfolger. Jeder Knoten muss und kann genau ein Element (`elem`) speichern. Wird versucht, ein Objekt mit dem Inhalt **`null`** als Element zu speichern, so soll eine *`IllegalArgumentException`* geworfen werden.

   `Node` bietet eine abstrakte Definition oder eine Standardimplementierung (entscheiden Sie sinnvoll, um Redundanz zu vermeiden) für folgende öffentliche Methoden an. In `Leaf` und `InnerNode` werden diese ggf. überschrieben (oder erstmalig implementiert).

   a) `boolean contains(E e)`: Gibt **`true`** zurück, wenn das Element `e` in dem jeweiligen Knoten (oder einem direkten/indirekten Nachfolger, falls vorhanden) gespeichert ist. Gibt **`false`** zurück, falls `e` im gesamten unter dem aktuellen Knoten aufgespannten Baum nicht enthalten ist.

   b) `append(E e)`: Fügt `e` richtig (d. h. so, dass die Invariante erhalten bleibt) in den Baum ein. Den Rückgabewert dürfen Sie selbst wählen. **`null`** soll nicht ohne Fehler eingefügt werden können.

   c) `boolean equals(Object o)`: Gibt **`true`** zurück, falls der Knoten äquivalent ist zu `o`, sonst **`false`**. Äquivalent sind zwei Konten dann, wenn ihr Inhalt (Element) und der gesamte (inkl. indirekten Nachfolgern) unter Ihnen aufgespannte Baum gleich aufgebaut ist.

   d) `String toString()`: Gibt eine String-Repräsentation des unter dem aktuellen Knoten aufgespannten Baums im *Inorder*-Format (d. h. erst linker Baum, dann Knoten, dann rechter Baum) zurück. Beispiel: Die Rückgabe für einen Baum mit der Wurzel 2 (linkem Kind 1, rechtem Kind 3) wäre `"1, 2, 3, "`, also sortiert.

```
public abstract class Node<E extends Comparable<E>> {

}

public class Leaf<E extends Comparable<E>> extends Node<E> {
    public Leaf(E elem) {

    }

}
```

```
    public class InnerNode<E extends Comparable<E>> extends Node<E> {

    }
```

2) Implementieren Sie nun die Klasse `BinaryTree` wie oben beschrieben unter Zuhilfenahme der bereits implementierten Klassen und Methoden. Das einzige Attribut speichert eine Referenz auf die Wurzel. Es existiert kein Konstruktor.

```
public class BinaryTree<T extends Comparable<T>> {

    private Node<T> root;

    public boolean contains(T element) {









    }

    public void append(T element) {









    }

    public boolean equals(Object o) {









    }

    public String toString() {









    }
}
```

19. Lambdas & Streams

Streams kommen aus der funktionalen Programmierung. Ein ***Stream*** repräsentiert eine geordnete Sequenz von Elementen, ähnlich zu einer Liste. Während eine Liste aber eine „echte" Datenstruktur ist, die die *konkreten* Daten (i. d. R. Referenzen auf Objekte) speichert und uns deren direkte Veränderung erlaubt, ist ein Stream eher als eine *abstrakte* Kapselung von Daten zu sehen. Streams erlauben uns, mit nur wenig Code (ohne aufwendige Schleifen) viele Operationen hintereinander auf alle Daten anzuwenden. Der Stream selbst wird dabei allerdings eigentlich nie *echt* verändert. Wenden wir eine der zahlreichen Stream-Methoden an, die die Daten des Streams *scheinbar* verändert (sog. *intermediäre* Operationen), wird tatsächlich ein *neuer* Stream erzeugt, der nun einfach andere Daten repräsentiert. Auf diesem neuen Stream können wir nun wiederum eine Stream-Operation ausführen, bspw. um den Stream weiter zu „verändern". Dies erlaubt uns die Verkettung mehrerer „verändernder" Methoden durch Punktnotation. Man kann sich sozusagen Daten formen. Zuletzt möchte man die im Stream gekapselten Daten jedoch zumeist verwenden/ausgeben/modifizieren. Hierfür gibt es *abschließende* Stream-Operationen.

Beispiel:

Gegeben sei ein Array `names`, das die Namen von Nutzern speichert, wobei die Namen nicht in korrekter Rechtschreibung eingegeben wurden. Wir möchten nun die Namen aller Nutzer, deren Name mit „C" beginnt, in alphabetischer Reihenfolge und korrekt geschrieben (d. h. erster Buchstabe groß, Rest klein) ausgeben. Die ursprünglichen Daten (Array `names`) sollen dabei jedoch nicht verändert werden, da sie später noch benötigt werden.

```
String[] names = {"norbert", "CORNELIUS", "cOrDuLa", "BEN", "", "Carmen"};
```

Lösung ohne Streams:

```
LinkedList<String> modifiedNames = new LinkedList<>();

for (String name : names) { // iteriere über das names-Array
    if (!name.equals(""))
        name = Character.toUpperCase(name.charAt(0)) // erster Buchstabe groß
               + name.substring(1).toLowerCase();    // Rest klein

    if (name.startsWith("C"))
        modifiedNames.add(name);   // Zwischenspeichern ist für das Sortieren nötig!
}

java.util.Collections.sort(modifiedNames); // alphabetisch sortieren

for (String name : modifiedNames) // verbleibende Elemente einzeln ausgeben:
    System.out.println(name);     // Ausgabe: Carmen, Cordula, Cornelius
```

Lösung mit Streams (werden in diesem Kapitel erläutert):

```
java.util.Arrays.stream(names)                    // erzeuge Stream, der die ...
  .map((name) -> name.equals("") ? ""             // ... names-Elemente kapselt
                 : Character.toUpperCase(name.charAt(0))   // erster B. groß
                   + name.substring(1).toLowerCase())      // Rest klein
  .filter(name -> name.startsWith("C"))           // nur Vornamen mit C
  .sorted()                                       // alphabetisch sortieren
  .forEach(System.out::println);                  // Elemente einzeln ausgeben
```

19.1. *Exkurs*: Anonyme Klassen & Lambda-Ausdrücke (->)

Manche Methoden erwarten als Parameter ein Objekt eines Interface-Typs. Wie wir wissen, können Interfaces jedoch nicht instanziiert werden. An solchen Stellen übergeben wir dann typischerweise eine Instanz einer Klasse, welche dieses Interface implementiert. Ein Beispiel dafür ist der Konstruktor der Klasse `Thread`, welcher ein `Runnable` (Interface) erwartet. Dummerweise müssen wir nun extra eine eigene Klasse schreiben, die die Methoden des Interfaces implementiert, oder? Ja, das geht, aber in so einer Situation können wir auch eine *anonyme Klasse* verwenden (siehe Beispiele in Kap. 21!), d. h. wir geben wie sonst auch in geschweiften Klammern eine Implementierung für unsere „Klasse" an, die das entsprechende Interface implementiert. Anstelle des üblichen „`public class MyClass`" setzen wir unmittelbar vor den Klassenrumpf jedoch eine Instanziierung mit dem Namen des jeweiligen Interfaces. Die Klasse trägt keinen Namen, es existiert aber eine Instanz von ihr – man nennt sie daher *anonym*.

Ist das gegebene Interface außerdem *funktional* (d. h. es definiert nur *eine* abstrakte Methode), so können wir die Instanziierung noch weiter abkürzen. Wir schreiben:

```
(nameParam1, nameParam2, ...) -> { /* Methodenrumpf */ }
```

Diese Form heißt *Lambda*-Ausdruck. Die Parametertypen entfallen, da sie eindeutig aus der einzigen Methodensignatur im Interface hervorgehen. Der Methodenrumpf kann beliebigen Code enthalten. Fordert die Signatur der Methode jedoch eine Rückgabe (ist also nicht `void`), so ist wie üblich ein `return`-Statement nötig. Besteht der Methodenrumpf wiederum nur aus dem `return`-Statement, so kann man auf das `return`-Schlüsselwort und die geschweiften Klammern verzichten. Bei einem einzelnen Parameter sind außerdem die runden Klammern nicht nötig.

1. **Allgemeines Beispiel** für Lambdas:

```
interface Function {
    int perform(int a, int b);
}

Function adder = (a, b) -> a + b;
Function divider = (x, y) -> {
    if (y == 0) return -1;  // division by zero
    return x / y;
};
System.out.println(adder.perform(2, 6));   // 8
System.out.println(divider.perform(7, 2)); // 3
```

Dieser Lambda-Ausdruck entspricht folgender Objekterzeugung mittels anonymer Klasse:

```
Function adder = new Function() {
   public int perform(int a, int b) {
      return a + b;
   }
};
```

2. **Konkretes Beispiel** `Stream.iterate(0, x -> x+2)` aus Kapitel 19.2:

   Der Lambda-Ausdruck „`x -> x+2`" entspricht: `(x) -> { return x+2; }`
   `Stream.iterate` erwartet als zweiten Parameter einen `UnaryOperator<T>`, was ein funktionales Interface mit der abstrakten Methode `T apply(T t)` ist. Der Lambda-Ausdruck ist also eine Kurzschreibweise für Instanziierung durch folgende anonyme Klasse:

```
Stream.iterate(0, new UnaryOperator<Integer>() {
    public Integer apply(Integer x) {
        return x+2;
    }
})
```

19.2. Erzeugen eines Streams ...

... über die Elemente eines Arrays: `Arrays.stream(myArray)`
... über die Elemente einer Collection (z. B. Listen): `myCollection.stream()`
... aus beliebig vielen einzelnen Objekten: `Stream.of(myObj1, myObj2, ..., myObjN)`
... durch iterative Generierung der Elemente: `Stream.iterate(initialVal, myFunction)`
... durch eine versorgende Funktion: `Stream.generate(myFunctionSupplyingValues)`

Mit `Arrays` ist `java.util.Arrays` und mit `Stream` die Klasse `java.util.stream.Stream` gemeint. Es gibt noch einige weitere Möglichkeiten, für uns sollten aber insb. die ersten drei ausreichen. Ein Beispiel für `iterate` wäre `Stream.iterate(0, x -> x+2)`, was einen unendlich langen Integer-Stream über die Zahlen 0, 2, 4, 6, 8, ... erzeugt. „`x -> x+2`“ ist ein Lambda-Ausdruck und wurde zuvor erklärt. Du fragst dich nun vielleicht, wie ein Stream unendlich lang sein kann, schließlich geht das bei Listen nicht... Das liegt daran, dass der Stream nicht die konkreten Daten 0, 2, 4, ... speichert, sondern die Funktion, die diese Zahlen erzeugt. Häufig möchte man jedoch nicht mit einem unendlich langen Stream arbeiten, daher nutzt man in diesem Zusammenhang meist die Methode `limit(maxSize)`, um den Stream auf maximal `maxSize` viele Elemente zu beschränken (bspw. `Stream.iterate(0, x->x+2).limit(5)`).

Die zuvor genannten Methoden geben grundsätzlich einen **`Stream<T>`** zurück. So würde bspw. der Aufruf `Stream.of(1)` einen `Integer`-Stream (`Stream<Integer>`) zurückgeben, welcher lediglich die Zahl 1 kapselt. Eine Ausnahme bildet `Arrays.stream(a)`, welche sich abhängig vom Array-Typ anders verhält. Ist `a` das `Integer`-Array `{1}`, so kreiert `Arrays.stream(a)` ebenfalls einen `Integer`-Stream. Ist `a` aber ein `int`-Array `{1}`, so kreiert `Arrays.stream(a)` einen *IntStream*, denn für die primitiven Datentypen `int`, `long` und `double` (und nur für diese) existieren eigene Stream-Klassen **`IntStream`**, **`LongStream`** und **`DoubleStream`**.

Erzeugen eines *IntStream*s ...

... über die Elemente eines `int`-Arrays: `Arrays.stream(myIntArray)`
... aus beliebig vielen einzelnen `int`s: `IntStream.of(myInt1, myInt2, ..., myIntN)`
... durch iterative Generierung der Elemente: `IntStream.iterate(initialVal, myFunc)`
... über einen definierten Zahlenbereich: `IntStream.range(startIntIncl, endIntExcl)`
... aus einem gegebenen `Stream<T>`: `myTStream.mapToInt(myFunctionMappingTtoInt)`

Für *Long-* und *DoubleStream*s geht das ganze analog, allerdings ist die `range`-Funktion nicht für *DoubleStream*s definiert. `range` ist besonders nützlich, um Schleifen zu simulieren. So würde bspw. `IntStream.range(0,6)` einen *IntStream* mit den ganzen Zahlen von 0 bis 5 erzeugen. *Int-, Long-* und *DoubleStream*s sind nützlich, da sie die Methoden `sum()` und `average()` bereitstellen, um die Summe bzw. den Mittelwert über den Stream zu berechnen. Während `sum()` unmittelbar zu einer Zahl auswertet, gibt `average()` ein `OptionalDouble`-Objekt zurück (da der Stream leer sein könnte), aus welchem der Zahlenwert mittels `getAsDouble()` gewonnen werden kann (sofern vorhanden).

(109) ★☆☆

1. Nenne mind. vier Beispiele zur Erzeugung eines *IntStream*s über 3, 4, 5, 6, 7.
2. Nenne Beispiele zur Erzeugung eines String-Streams über `a`, `aa`, `aaa`, `aaaa`, `aaaaa`.
3. Erzeuge mittels `Stream.iterate` einen Stream über alle Großbuchstaben (A bis Z).
4. Gegeben sei ein Array `double[] noten = {1.7, 1.3, 2.0, 1.3, 4.0, 2.7};` Berechne den Notendurchschnitt, ohne Schleifen zu verwenden.

19.3. *Exkurs*: Methodenreferenzen (`::`)

Lambda-Expressions sind bereits ziemlich kurze Ausdrücke zur Definition von Methoden. Noch schneller/kürzer geht es, wenn eine Methode bereits existiert, denn dann kann man die bestehende Methode referenzieren. Mit Methodenreferenzen können also bereits definierte Methoden identifiziert und wiederverwendet werden. Sie haben folgende Form:

`Klassenname::methodenname`	(bei statischen Methoden)
`objekt::methodenname`	(bei nicht-statischen Methoden (selten))

Betrachten wir wieder das **Beispiel** `Stream.iterate(0, x -> x+2)` aus Kapitel 19.2 und nehmen zusätzlich an, wir hätten bereits folgende Klasse definiert:

```
public class Funktionen {
    public static int inkrementUmZwei(int x) {
        return x + 2;
    }
}
```

Dann können wir `Stream.iterate(0, x -> x + 2)` auch schreiben als:

```
Stream.iterate(0, Funktionen::inkrementUmZwei)
```

Besseres Beispiel (Vorgriff auf `forEach` → Kap. 19.4):

Wir möchten alle Elemente eines Streams `s` auf der Konsole ausgeben.

Grundsätzlich ginge das so: `s.forEach((elem) -> System.out.println(elem))`
Wir können aber auch schreiben: `s.forEach(System.out::println)`

→ `System.out` ist hier ein Objekt, dessen Methode `println(Object o)` wir referenzieren.

19.4. Stream-Operationen

Wir unterscheiden zwischen *intermediären* und *abschließenden* Operationen. Eine *intermediäre* Methode ist i. d. R. nicht die letzte, die aufgerufen wird, sondern wird genutzt, um den Stream zu „verändern“, d. h. derartige Methoden geben wiederum einen Stream zurück. Nach dem Aufruf einer *abschließenden* Methode wird der Stream geschlossen und kann nicht weiter verwendet werden. Vereinfacht gesagt *muss* jeder Stream abgeschlossen werden, sonst werden die intermediären Methoden gar nicht erst ausgeführt (da ja scheinbar nicht benötigt). Im Folgenden werden die wichtigsten Stream-Methoden beschrieben.

QR-Code: *https://stecrz.de/files/skript/Spicker_Streams.pdf*

Die echten Parametertypen sind in den nachfolgenden Beschreibungen an manchen Stellen bewusst weggelassen, da sie dich vermutlich mehr verwirren als aufklären würden. Die meisten Parametertypen sind funktionale Interfaces (z. B. `Consumer<T>`). Entsprechend können wir Lambdas verwenden, was wir in Zusammenhang mit Streams auch eigentlich ausschließlich tun.

Aus diesem Grund werden einige Parameter im Folgenden direkt als „Funktion“ bezeichnet, obwohl es sich streng genommen um eine Instanz einer Klasse handelt, welche die entsprechende Methode (Methodensignatur gegeben) implementiert. Behalte außerdem im Hinterkopf, dass keine Methode den Stream wirklich verändert. Intermediäre Methoden geben stattdessen immer einen (ggf. veränderten) *neuen* Stream zurück.

Dieses Beispiel-Array sei für die Code-Beispiele in den folgenden Unterkapiteln gegeben:

```
Spieler[] mannschaft = {
    new Spieler("Max", 2),
    new Spieler("Luis", 3),
    new Spieler("Ed", 2),
    new Spieler("Tom", 1),
};
```

```
public class Spieler {
    public int tore;  public String name;
    public Spieler(String n, int t) {
        name = n; tore = t;
    }
    public String toString() {
        return name + "[" + tore + "]";
    }
}
```

19.4.1. Abschließende Methoden (für einen `Stream<T>`):

`long count()` gibt die Länge des Streams zurück (und beendet ihn).

`void forEach(`*`myConsumer`*`)`
mit `myConsumer` ist Funktion `void accept(T t)`

Wendet die übergebene Funktion *`myConsumer`* auf jedes Element des Streams an.

1. Beispiel: Gibt die Strings des Arrays in Großschreibung aus.

```
String[] lang = {"de", "En", "es"};
Arrays.stream(lang).forEach(s -> System.out.println(s.toUpperCase()));
```

2. Beispiel: Setzt die Anzahl an Toren aller Spieler, die 2 Tore haben, auf 0 (hier für Max und Ed).

```
Arrays.stream(mannschaft).forEach(spieler -> {
    if (spieler.tore == 2)
        spieler.tore = 0;
});
```

`boolean anyMatch(`*`myPredicate`*`)` *analog:* **`allMatch`**, **`noneMatch`**
mit `myPredicate` ist Funktion `boolean test(T t)`

Gibt **`true`** zurück, wenn die Funktion *`myPredicate`* für mind. ein Element des Streams wahr ist (also zu **`true`** auswertet), **`false`** sonst. (Kurzschlussauswertung)

1. Beispiel: Prüft, ob *mind. ein* Spieler mehr als 3 Tore geschossen hat (hier nicht, also *keine* Ausgabe).

```
if (Arrays.stream(mannschaft).anyMatch(spieler -> spieler.tore > 3))
    System.out.println("Es gibt einen Spieler mit über 3 Toren");
```

2. Beispiel: Prüft, ob *alle* Elemente aus 4, -6, 2, 8 gerade sind (ja, also folgt die Ausgabe).

```
if (IntStream.of(4, -6, 2, 8).allMatch(x -> x % 2 == 0))
    System.out.println("Alle Elemente sind gerade.");
```

`A[] toArray(`*`arrayConstructor`*`)`
mit `arrayConstructor` ist z. B. `Spieler[]::new` *(bei primitiven Streams kein `arrayConstructor`)*

Packt die Elemente des Streams in ein neues Array des übergebenen Typs, wobei dieser dem Typ aller Elemente (bzw. einem Obertyp) entsprechen muss. Ein Stream von `Spieler`-Objekten kann bspw. nicht in ein `String`-Array gepackt werden.

1. Beispiel: Packt die Elemente des Streams (1, 6, 3) in ein Integer-Array (`int[]` wäre nicht möglich!).

```
Integer[] arr = Stream.of(1,6,3).toArray(Integer[]::new);
System.out.println(Arrays.toString(arr));  // Ausgabe: [1, 6, 3]
```

2. Beispiel: Erzeugt ein `String`-Array aus den Spielernamen (also `["Max","Luis","Ed","Tom"]`)

```
String[] names = Arrays.stream(mannschaft).map(s->s.name).toArray(String[]::new);
```

3. Beispiel: Erzeugt ein `int`-Array aus den Toren der Spieler (Vorgriff auf primitive Streams).

```
int[] spielerTore = Arrays.stream(mannschaft).mapToInt(s -> s.tore).toArray();
```

R collect(*someCollector*)

mit `someCollector` *ist z. B.* `Collectors.toList()` *oder* `Collectors.toSet()`
oder `Collectors.joining(String)` *oder* `Collectors.toMap(myKeyMapper, myValueMapper)`
oder `Collectors.groupingBy(myGroupMapper)` *oder …*

Packt die Elemente des Streams in eine neue Datenstruktur, je nachdem was für *someCollector* übergeben wird. Die Klasse `java.util.stream.Collectors` stellt einige Möglichkeiten bereit, wobei die häufigsten nachfolgend beschrieben sind:

» `Collectors.`**`toList()`** erzeugt eine neue Liste, die Elemente des Streams enthält (analog dazu: `toSet()` speichert die Elemente in einer Menge).

» `Collectors.`**`joining(`***`sep`***`)`** kann lediglich auf einem `Stream<String>` benutzt werden und verknüpft die Elemente (also Strings) zu einem einzelnen String, jeweils getrennt durch den `String` *sep* (optionaler Parameter).

» `Collectors.`**`toMap`**`(`*keyMapper*`, `*valueMapper*`)` erzeugt eine neue `Map<K,V>`. Es werden zwei Zuordnungsfunktionen erwartet, wobei die erste zur Generierung der Schlüssel dient, d. h. *keyMapper* bildet je ein Element des Streams (Typ `T`) auf genau einen Schlüssel ab (hier wird der Typ `K` der Map implizit festgelegt), wobei Schlüssel eindeutig sein müssen (keine Duplikate erlaubt). Die Ergebnis-Map ist folglich genauso lang wie der Stream. Der *valueMapper* bildet analog dazu je ein Element des Streams (Typ `T`) auf genau einen Wert ab (hier wird der Typ `V` der Map implizit festgelegt). Dies geschieht in der gleichen Reihenfolge wie beim *keyMapper*, wodurch die „Schlüssel → Wert"- Zuordnung eindeutig ist.

» `Collectors.`**`groupingBy`**`(`*groupMapper*`)` erzeugt eine neue `Map<K,List<T>>`. Es wird eine Zuordnungsfunktion erwartet, welche zur Generierung der Schlüssel dient, d. h. *groupMapper* bildet je ein Element des Streams (Typ `T`) auf genau einen Schlüssel ab (hier wird der Typ `K` der Map implizit festgelegt). Im Unterschied zu `toMap` dürfen mehrere Elemente auf denselben Schlüssel abbilden. Alle Elemente des Streams (Typ `T`), die auf einen Schlüssel `key` abbilden, werden in einer `List<T>` gruppiert und in der Ergebnis-Map unter diesem `key` gespeichert.

1. Beispiel: Wandelt den Stream von Spielern mittels `collect` in eine `java.util.List` um.

```
List<Spieler> spieler = Arrays.stream(mannschaft).collect(Collectors.toList());
```

2. Beispiel: Verknüpft die Elemente des String-Streams zu einem String (`"rotblaugelb"`).

```
String s = Stream.of("rot", "blau", "gelb").collect(Collectors.joining());
```

3. Beispiel: Verknüpft die Spielernamen zu dem kommagetrennten String `"Max, Luis, Ed, Tom"`.

```
String s = Arrays.stream(mannschaft).map(sp -> sp.name) // map kommt noch...
                                    .collect(Collectors.joining(", "));
```

4. Beispiel: Erzeugt für die Zahlen 1 bis 5 eine Abbildung der Zahl als String auf ihre Quadratzahl, d. h. das Ergebnis `myMap` ist die `Map` {`"1"` → `1`, `"2"` → `4`, `"3"` → `9`, `"4"` → `16`, `"5"` → `25` }. Anschließend wird die `Map` mittels `forEach` ausgegeben (Stream über Schlüssel-Menge). `boxed()` ist nötig, um den `IntStream` (vgl. Kap. 19.4.3) zu einem `Stream<Integer>` umzuwandeln, weil `collect` nicht für primitive Streams existiert (zum. nicht in dieser Form).

```
Map<String,Integer> myMap = IntStream.range(1, 6).boxed()
                                     .collect(Collectors.toMap(i -> ""+i, i -> i*i));
myMap.keySet().stream().forEach(k -> System.out.println(k + ": " + myMap.get(k)));
```

5. Beispiel: Gruppiert die Spieler nach der Anzahl ihrer geschossenen Tore in Listen, d. h. das Ergebnis `grouped` ist die `Map` { 1 → [*Tom*], 2 → [*Max*, *Ed*], 3 → [*Luis*] }.

```
Map<Integer, List<Spieler>> grouped = Arrays.stream(mannschaft)
                                    .collect(Collectors.groupingBy(sp -> sp.tore));
```

`T reduce(T myStartValue, myAccumulatorFunc)`

mit `myAccumulatorFunc` ist Funktion `T apply(T acc, T elem)`

Wendet die Funktion *`myAccumulatorFunc`* nacheinander auf alle Elemente des Streams an. Dabei wird über den ersten Parameter (`acc`) ein Ergebnis aufgebaut (vgl. Aufgaben zu Endrekursion), welches `reduce` am Ende auch zurückgibt. Der Akkumulator `acc` bekommt anfangs den Wert *`myStartValue`*, sodass die Funktion *`myAccumulatorFunc`* auf das erste Element des Streams angewendet werden kann. Das Ergebnis des Aufrufs ist dann der neue Akkumulatorwert (für das zweite Element).

`reduce` existiert auch ohne `myStartValue`. In diesem Fall wird direkt mit dem ersten Stream-Element begonnen (als Akkumulator). Rückgabe dann als `Optional<T>`.

1. Beispiel: Summiert die Zahlen des Streams auf (und gibt 10.899...99 aus).

```
System.out.println(Stream.of(6.1, 3.6, 1.2).reduce(0.0, Double::sum));
```

2. Beispiel: Verknüpft die Wörter, die mit *t* beginnen, zu einem Text (Ergebnis: `"#-ton-tee"`).

```
System.out.println(Stream.of("ton","rot","ab","tee").reduce("#", (acc, str) -> {
    if (str.startsWith("t"))
        return acc + "-" + str;
    return acc;
}));
```

3. Beispiel: Berechnet die Summe der Tore aller Spieler (also hier 8).

```
int anzahlTore = Arrays.stream(mannschaft).map(spieler -> spieler.tore)
        .reduce(0, (acc, elem) -> elem + acc);  // oder .reduce(0, Integer::sum)
```

`Optional<T> min(myComparator)` *analog:* **`max`**

mit `myComparator` ist Funktion `int compare(T o1, T o2)`

Ermittelt dasjenige Element des Streams, welches bzgl. der Vergleichsfunktion *`myComparator`* minimal (bzw. bei **`max`**: maximal) ist. Die Vergleichsfunktion bekommt zwei Elemente *`o1`* und *`o2`* übergeben. Sie muss zu einem negativen Wert auswerten, wenn *`o1`* < *`o2`* gilt, und zu einem positiven, wenn *`o1`* > *`o2`* gilt. Die Rückgabe erfolgt als `Optional`-Objekt `res`, wobei man den tatsächlichen Wert dann mittels `res.get()` erhält, sofern nicht `res.isEmpty()` (sondern `res.isPresent()`) gilt. Alternativ kann `res.orElse(otherVal)` benutzt werden, was wie `get()` zum tatsächlichen Wert (Typ `T`) auswertet, falls vorhanden, anderenfalls zu `otherVal`.

1. Beispiel: Ermittelt den Spieler mit den meisten Toren und gibt dessen Name aus (hier: Luis).

```
Optional<Spieler> bester = Arrays.stream(mannschaft)
   .max((sp1, sp2) -> sp1.tore > sp2.tore ? 1 : (sp1.tore < sp2.tore ? -1 : 0));
if (bester.isPresent())
    System.out.println("Bester: " + bester.get().name);
else
    System.out.println("Es gibt keine Spieler...");
```

2. Beispiel: Ermittelt das Minimum der gegebenen Zahlen (hier: 4).

```
System.out.println(Stream.of(8, 4, 9, 6).min((a, b) -> a - b).get());
```

3. Beispiel: Ermittelt den kürzesten Namen aller Spielernamen und gibt ihn aus (Ausgabe hier: Ed).

```
System.out.println( Arrays.stream(mannschaft).map(spieler -> spieler.name)
                        .min((str1, str2) -> {
                            if (str1.length() < str2.length())
                                return -1;  // str1 soll vor str2 kommen
                            else if (str1.length() > str2.length())
                                return 1;
                            else return 0;  // Namen sind gleich
                        }).orElse("Es gibt keine Spieler...") );
```

19.4.2. Intermediäre Methoden (für einen `Stream<T>`):

`Stream<T> peek(`*`myConsumer`*`)`
mit `myConsumer` ist Funktion `void accept(T t)`

Wendet die Funktion *`myConsumer`* auf alle Elemente des Streams an (wie `forEach`). Im Unterschied zu `forEach` wird der Stream hier aber nicht geschlossen, sondern kann weiterverwendet werden, d. h. der Ergebnisstream entspricht dem ursprünglichen Stream. Das ist sinnvoll, um mehrere Operationen hintereinander anzuwenden.

Hinweis: Streams werden über die abschließenden Methoden gesteuert, d. h. `peek` ist auf keinen Fall ein Ersatz für `forEach`! `peek` sollte möglichst sparsam eingesetzt werden.

Beispiel: Gibt jeden Spieler mittels `println()` (→ `toString`) auf der Konsole aus, bevor die Anzahl seiner Tore auf 0 gesetzt wird. Die Ausgabe ist also *Max[2], Luis[3], Ed[2], Tom[1]*. Anschließend haben alle Spieler 0 Tore. Danach: Spieler werden mit `collect` als Menge gespeichert.

```
Set<Spieler> players = Arrays.stream(mannschaft)
                            .peek(System.out::println)
                            .peek(sp -> sp.tore = 0)
                            .collect(Collectors.toSet());
```

`Stream<T> filter(`*`myPredicate`*`)`
mit `myPredicate` ist Funktion `boolean test(T t)`

Der Ergebnisstream enthält alle Elemente des Streams (auf dem die Methode aufgerufen wird), für die die Funktion *`myPredicate`* wahr ist (also zu **`true`** auswertet).

Beispiel: Ermittelt Spieler mit mehr als einem Tor und gibt sie aus (hier: *Max[2], Luis[3], Ed[2]*)

```
Arrays.stream(mannschaft).filter(s -> s.tore > 1).forEach(System.out::println);
```

`Stream<R> map(`*`myFunction`*`)`
mit `myFunction` ist Funktion `R apply(T t)`

Wendet die Funktion *`myFunction`* nacheinander auf alle Elemente des Streams an. Der von `map()` zurückgegebene Stream enthält dann die Ergebniswerte, also die Werte, zu denen *`myFunction`* auswertet, wenn man Element 1, 2, usw. als Parameter übergibt. Diese haben ggf. einen anderen Typ (deswegen Rückgabetyp `Stream<R>`).

Beispiel: Aus dem `Spieler`-Stream wird durch das erste `map` ein `String`-Stream von Spielernamen gewonnen. Das zweite `map` verändert nur die Elemente des Streams (nicht aber den Typ). Anschließend folgt die Ausgabe der einzelnen Elemente: *MAX, LUIS, ED, TOM*

```
Arrays.stream(mannschaft).map(sp -> sp.name)   // ab jetzt Stream<String>
                         .map(String::toUpperCase)
                         .forEach(System.out::println);
```

`IntStream mapToInt(`*`myToIntFunction`*`)` *analog:* **`mapToDouble`**, **`mapToLong`**
mit `myToIntFunction` ist Funktion `int applyAsInt(T value)`

Verhält sich wie `map()`, allerdings mit dem primitiven Datentyp `int` (bzw. `double` oder `long`, siehe Unterschied zur `apply`-Methode). Diese Methode kann bspw. genutzt werden, um einen `Integer`-Stream in einen `IntStream` umzuwandeln.

Beispiel: Berechnet die Summe der Tore aller Spieler durch einen Stream über die Einzel-Tore. Erg.: `8`

```
System.out.println(Arrays.stream(mannschaft).mapToInt(sp -> sp.tore).sum());
```

`Stream<T> distinct()`

Eliminiert doppelte Elemente, d. h. der Ergebnisstream enthält alle Elemente nur noch in einfacher Ausführung. Zur Prüfung, ob zwei Elemente gleich sind, wird die `equals`-Methode benutzt (bei primitiven Streams natürlich ==).

Beispiel: Eliminiert die zweite 2 und 3, d. h. der Stream enthält nur noch die Elemente 3, 1 und 2.

```
Stream.of(3, 1, 2, 3, 2).distinct().forEach(System.out::println);
```

`Stream<T> limit(long maxSize)`

Beschränkt den Ergebnisstream auf die ersten *`maxSize`* Elemente (maximal). Diese Methode ist bspw. sinnvoll, um einen unendlichen Stream zu limitieren.

1. Beispiel: Beschränkt den unendlichen Stream 3, 4, 5, ... auf die Elemente 3 bis 7. ⇒ `3:4:5:6:7:`

```
Stream.iterate(3, x -> x + 1).limit(5).forEach(x -> System.out.print(x + ":"));
```

2. Beispiel: Erzeugt ein Array bestehend aus 5 Zufallszahlen im Bereich von 0 (inkl.) bis 10 (exkl.), wobei keine Zahl doppelt vorkommt (`ints` gibt einen unendlichen *IntStream* von Zahlen).

```
int[] a = new Random().ints(0, 10).distinct().limit(5).toArray();
```

`Stream<T> skip(long n)`

Beschränkt den Ergebnisstream auf *alle* Elemente *ohne* die ersten *`n`* vielen.

Beispiel: Überspringt das Element „`line 1`“, d. h. der Stream enthält nur noch `line 2`, `3` und `4`. Daher wertet `count()` zu `3` aus (Anzahl Stream-Elemente).

```
System.out.println(
   Arrays.stream("line 1\nline 2\nline 3\nline 4".split("\n")).skip(1).count());
```

`Stream<T> sorted(myComparator)`
mit `myComparator` ist Funktion **`int compare(T o1, T o2)`**

Ordnet den Ergebnisstream entsprechend der Vergleichsfunktion *`myComparator`* an. Die Vergleichsfunktion muss zu einem negativen Wert auswerten, wenn `o1` < `o2` gelten soll (d. h. `o1` kommt *vor* `o2`) und zu einem positiven, wenn `o1` > `o2` gelten soll.

Implementiert der Typ `T` das Interface `Comparable<T>`, so ist *`myComparator`* optional. Wird kein Parameter angegeben, so gibt der Aufruf `sorted()` einen entsprechend der natürlichen Ordnung sortierten Stream zurück. Z. B. die Klassen `Integer` und `String` implementieren `Comparable`, weshalb dafür kein Comparator nötig ist.

Tipp: Zum Umkehren der Reihenfolge eines Streams kann man `sorted(Collections.reverseOrder())` benutzen.

1. Beispiel: Gibt die Spieler sortiert nach der Anzahl ihrer Tore aus: `Tom[1]`, `Max[2]`, `Ed[2]`, `Luis[3]`

```
Arrays.stream(mannschaft).sorted((sp1, sp2) -> sp1.tore - sp2.tore)
                         .forEach(System.out::println);
```

2. Beispiel: Erzeugt einen sortierten *IntStream* von 6 Zufallszahlen im Bereich [−2; 4) und gibt sie aus.

```
new Random().ints(6, -2, 4).sorted().forEach(System.out::println);
```

3. Beispiel: Wie Beispiel 2, allerdings werden die Zahlen *absteigend* sortiert. Da `sorted` für primitive Streams keinen Comparator erlaubt (vgl. Kap. 19.4.3), wird zu `Stream<Integer>` geboxt.

```
new Random().ints(6, -2, 4).boxed().sorted((a, b) -> b - a)
                            .mapToInt(i -> i).forEach(System.out::println);
```

Du brauchst mehr Comparator-Beispiele? Siehe **`min`**`(myComparator)`*!*

19.4.3. **Methodenübersicht für primitive Streams** (am Beispiel von `IntStream`):

In manchen der zuvor genannten Beispielen wurden bereits primitive Streams benutzt. Für primitive Streams (*Int-, Long-* und *DoubleStream*) existieren alle genannten Methoden (außer das zuvor genannte `collect`) mit vergleichbarer Funktionalität, nur eben für primitive Daten. Es gibt kleine Unterschiede in den Signaturen: Z. B. erwartet `toArray()` für *IntStream*s keinen Parameter, da das Ergebnis immer fix ein `int[]` ist. Ebenso erwartet `sorted()` keinen Comparator mehr, da Zahlen nach ihrer natürlichen Ordnung sortiert werden. Wichtig: Durch `map()` kann sich die Art des Streams nicht mehr ändern, d. h. z. B. *IntStream*`.map()` gibt immer einen *IntStream* zurück.

Hier eine kurze Übersicht mit den wichtigsten Methoden für `IntStream`:

- `long` **`count()`**
- `void` **`forEach(`***`myIntConsumer`***`)`**
- `boolean` **`anyMatch(`***`myIntPredicate`***`)`** *und analog dazu* **`allMatch`**, **`noneMatch`**
- `int[]` **`toArray()`**
- `int` **`reduce(int`** ***`myStartValue`***`,` ***`myIntAccumulatorFunc`***`)`**
- `OptionalInt` **`min()`** *und analog dazu* **`max`**
- `IntStream` **`peek(`***`myIntConsumer`***`)`**
- `IntStream` **`filter(`***`myIntPredicate`***`)`**
- `IntStream` **`map(`***`myIntFunction`***`)`** *mit* `myIntFunction` *ist Fkt.* `int applyAsInt(int v)`
- `IntStream` **`limit(long`** ***`maxSize`***`)`** *und* `IntStream` **`skip(long`** ***`n`***`)`**
- `IntStream` **`distinct()`**
- `IntStream` **`sorted()`**

Zusätzlich gibt es für *IntStream*s folgende Methoden:

- `int` **`sum()`**: Berechnet die Summe über alle im `IntStream` enthaltenen Zahlen.
- `OptionalDouble` **`average()`**: Berechnet den Durchschnitt über alle Zahlen. Das Ergebnis ist ein `OptionalDouble opt`, welches mittels `opt.getAsDouble()` zu einem `double` entpackt werden, sofern der Stream nicht leer war (`opt.isEmpty()`). Wie bei `min` gibt es auch hier die Alternative `opt.orElse(otherDouble)`.
- `Stream<Integer>` **`boxed()`**: Wandelt den `IntStream` in `Stream<Integer>` um.
- `DoubleStream` **`asDoubleStream()`** *und analog dazu* **`asLongStream`**: Wandelt den `IntStream` durch implizite Upcasts in einen `DoubleStream` (bzw. `LongStream`) um. Wegen *int* → *long* → *double* existiert bspw. für `LongStream` nur `asDoubleStream`.
- `Stream<R>` **`mapToObj(`***`myIntFunction`***`)`** *mit* `myIntFunction` *ist Fkt.* `R apply(int v)`. Verhält sich so wie `map` bei nicht-primitiven Streams.

19.4.4. **Parallelisierung** von Streams (oberflächlich):

Um Operationen auf Streams mittels Threads (→ Kap. 21) zu parallelisieren, rufen wir auf dem entsprechenden Stream zuvor einfach die Methode `parallel()` auf.

Beispiel:
```
Arrays.stream(mannschaft).parallel()
      .peek(sp -> System.out.println(sp.name))
      .forEach(sp -> System.out.println(sp.tore));
```

Ausgabe variiert, z. B.: *Max 2 Luis 3 Ed 2 Tom 1* oder *Ed 2 Luis Tom 1 Max 3 2*

(110) ★★☆ Die unterhalb genannten Methoden aus vorherigen Aufgaben sollen nun unter Verwendung der Stream-API ohne Schleifen und Rekursion implementiert werden. Lösen Sie die folgenden Aufgaben (aus vorherigen Kapiteln) nun also mithilfe von Streams:

Hinweis: Solltest du Probleme bei einer Aufgabe haben, so kannst du dich an der ursprünglichen Lösung (mit Schleifen bzw. Rekursion) zu der jeweiligen Aufgabe orientieren.

Beachte, dass alle sonstigen Anweisungen (*if*, *return*, ...) weiterhin verwendet werden dürfen. Es soll lediglich auf jede Art von Schleifen und Rekursion verzichtet werden.

1. Aufgabe 58 (Seite 80): `long sum(long n)`
2. Aufgabe 18 (S. 40): (a) `avgTemp()` und (b) `minTempNoon()`
3. Aufgabe 20 (S. 42): `int[] removeDuplicates(int[] arr)`
4. Aufgabe 21 (S. 42): `int[] extractMultiplesOf(int[] array, int num)`
5. Aufgabe 22 (S. 43): `void reverse(Object[] a)`

   Hinweis: Mit einem Stream über ein Array (wie bei den vorherigen Teilaufgaben) kann das Array selbst nicht verändert werden, sondern nur die Elemente, die im Stream enthalten sind (wie bei einer *for-each*-Schleife). Daher benötigt man in solchen Fällen einen Stream über die Indizes des Arrays.
6. Aufgabe 23 (S. 43): `long[] powerOfX(int x, byte n)`
7. Aufgabe 27 (S. 48): `int countVowels(String text)`

   Hinweis: `text.chars()` gibt einen `IntStream` über die Zeichen von `text` zurück. `s.count()` gibt die Anzahl der Elemente des Streams `s` als `long` zurück.
8. Aufgabe 28 (S. 48): `String invert(String s)`
9. Aufgabe 41 (S. 68): (a) linksbündig (wie in der Angabe) (b) rechtsbündig
10. Aufgabe 52 (S. 75): `boolean isPalindrome(char[] c)`
11. Aufgabe 86 (S. 127): `Set<Integer> mostFrequentElements(int[] arr)`

    Hinweis: Schwierig. Bearbeite zuvor vielleicht erst die anderen Aufgaben zu Streams.

(111) ★★★ Ergänzen Sie den nachfolgenden Code, sodass die Summe der drei größten geraden Zahlen eines beliebigen Arrays `zahlen` ausgegeben wird (hier: 8+6+4). Wandeln Sie das Array dafür mittels `Arrays.stream()` in ein *IntStream*-Objekt um und nutzen Sie anschließend die Methoden `map(...)`, `filter(...)`, `limit(...)`, `sorted()` und `sum()`.

```
public static void main(String[] args) {
    int zahlen[] = {5, 8, 9, 2, -10, 3, 4, 7, 6, 1};
    System.out.println(/* TODO */); // Gibt 18 aus
}
```

Tipp: Sie können einen Stream einfach auf die drei größten Zahlen limitieren, nachdem Sie den Stream *absteigend* sortiert haben. Für die *absteigende* Sortierung können Sie die Elemente des Streams zunächst negieren und dann mit `sorted()` aufsteigend sortieren.

(112) ★★☆

Lösen Sie folgende Aufgaben ohne die Verwendung von Schleifen oder Rekursion:

1. Schreiben Sie eine Methode `int[] primes(int max)`, welche ein Array zurückgibt, das alle Primzahlen beginnend bei 2 bis höchstens `max` (inkl.) enthält.

   Hinweis: Eine Primzahl ist eine ganze Zahl größer als 1, die nur durch 1 und sich selbst teilbar ist.

2. Schreiben Sie eine Methode `long[] morePrimes(int count)`, welche ein Array zurückgibt, das die ersten `count` vielen Primzahlen enthält.

3. Schreiben Sie eine Methode `void fibonacci(int n)`, welche die ersten `n` Zahlen der Fibonacci-Folge kommagetrennt ausgibt. Die erste Zahl der Fibonacci-Folge sei 1.

   Beispiel: `fibonacci(7)` gibt „`1, 1, 2, 3, 5, 8, 13`“ aus.

   Tipp: Verwenden Sie den Ansatz `….iterate(new long[] {0, 1}, arr -> …)…`

(113) ★★★

Ergänzen Sie folgende Methode an den gekennzeichneten Stellen, sodass diese die als String übergebene Hexadezimalzahl als Dezimalzahl zurückgibt (angelehnt an Aufgabe 31). Gültige Hexadezimalzahlen bestehen lediglich aus den Ziffern `0` bis `9` und den Kleinbuchstaben `a` bis `f` und beginnen mit `0x`. Wird ein ungültiger Wert für `hex` übergeben (z. B. `0xA`), so soll eine `IllegalArgumentException` geworfen werden.

Beispiel: *`hexToDez`*`(0x13b)` gibt 315 (= $\mathbf{1} * 16^2 + \mathbf{3} * 16^1 + \mathbf{11}$) zurück.

Hinweis: `map` und `reduce`…

```
public static long hexToDez(String hex) {
    if (hex == null || !hex.startsWith("0x"))
        ┌──────────────────────────────────────┐
        │                                      │
        └──────────────────────────────────────┘

    return hex.chars()   // IntStream über alle Zeichen von hex
        ┌──────────────────────────────────────┐
        │                                      │
        │                                      │
        │                                      │
        │                                      │
        │                                      │
        │                                      │
        │                                      │
        │                                      │
        │                                      │
        └──────────────────────────────────────┘;
}
```

(114) **Datenbankauswertung (CSV-Dateien)** [65 Punkte]:

★★★

Eine Grundschule verwaltet eine aus drei Tabellen bestehende Datenbank, wobei jede Tabelle als eine Datei im CSV-Format gegeben ist. *Excel*-Tabellen können bspw. in diesem Format gespeichert werden. CSV-Dateien repräsentieren genau einen Datensatz pro Zeile. Jeder Datensatz besteht aus Werten, welche durch Strichpunkte getrennt sind. In der ersten Zeile werden hingegen die Namen der Spalten – ebenfalls getrennt durch Semikolons – gespeichert. Die Tabellen der Schule seien wie folgt gegeben:

Auszug aus *schueler.csv*:

```
1   nr;name;geschlecht;klasse
2   45;Tobias Tolk;m;4B
3   72;Elli Ebsen;w;3B
... ...
31  39;Marlene Miller;w;4A
```

Auszug *faecher.csv*:

```
1   nr;bezeichnung
2   4;Deutsch
3   2;Kunst
... ...
6   5;HSU
```

Auszug aus *noten.csv*:

```
1   schuelerNr;fachNr;note
2   45;2;3
3   76;4;2
... ...
410 45;2;1
```

Hinweis: Die Bezeichnung „Schüler“ ist im Folgenden geschlechtsneutral zu verstehen.

» *schueler.csv* speichert die Schüler der Jahrgangsstufen 3 und 4 mit Identifikationsnummer, Name, Geschlecht (*m*/*w*) und Name der Schulklasse (*3A, 3B, 3C, 4A, 4B, 4C*).

» *faecher.csv* ordnet jedem Schulfach eine Identifikationsnummer zu.

» *noten.csv* speichert die Noten der Schüler. Die oberhalb gegebenen Daten lassen somit darauf schließen, dass *Tobias Tolk* aus der *4B* im Fach *Kunst* die Noten *3* und *1* (ggf. noch weitere) erhalten hat. Nicht jeder Schüler belegt alle Fächer, d. h. es kann bspw. Schüler geben, die im Fach *Kunst* keine Note erhalten haben. Die Fächer *Deutsch*, *Mathematik* und *HSU* (Heimat- und Sachunterricht) werden jedoch von allen Schülern belegt. Jeder Schüler hat somit mindestens eine Note in jedem dieser Fächer erhalten.

Die Schulverwaltung beauftragt Sie nun zum Schuljahresende mit der Erzeugung der Zeugnisse für alle Schüler. Den Viertklässlern soll basierend auf der Zeugnisdurchschnittsnote außerdem eine Schullaufbahnempfehlung (Gymnasium, Realschule oder Mittelschule) ausgesprochen werden. Weiters ist die Schule interessiert:

- Sind Mädchen durchschnittlich signifikant besser oder schlechter als Jungen?
- Welcher Schüler hat die Note 1 am häufigsten erhalten (in einzelnen Tests)?
- Was ist die beste erreichte Zeugnisdurchschnittsnote und wer hat diese erreicht? Analog dazu: Wer ist der beste Viertklässler? Wer ist der Klassenbeste aus der *4A*?
- Wie vielen Schülern wurde Gymnasium/Realschule/Mittelschule empfohlen?

Lösen Sie die Aufgaben mit Streams, ohne Schleifen oder Rekursion zu verwenden. Die Beispieldateien sind zum Download unter *https://stecrz.de/files/skript/schule.zip* verfügbar. Die Datei `SchoolModel.java` stellt eine Methode `open(String filePath)` zum Lesen einer Datei bereit, welche einen `BufferedReader` zurückgibt. Aus einem solchen `BufferedReader br` kann über den Methodenaufruf `br.lines()` ein Stream über die Zeilen der Datei gewonnen werden. Beachten Sie, dass jeder `BufferedReader` *nach* der Verarbeitung wieder geschlossen werden muss. Dazu bietet die Klasse `SchoolModel` eine Methode `close(BufferedReader br)` an.

Hinweis: Diese Aufgabe solltest du nicht schriftlich lösen; nutze eine IDE! Die Teilaufgaben bauen aufeinander auf. Solltest du eine Aufgabe nicht lösen können, so nutze die Musterlösung, um die übrigen lösen und testen zu können.

Hinweis: Um aus einem Stream eine `Map` zu erzeugen, kann die `collect` Methode mit `Collectors.toMap(keyMapper, valueMapper)` aufgerufen werden (vgl. S. 167). Unter gewissen Umständen (Werte sind Listen) kann zur Erzeugung der `Map` auch `Collectors.groupingBy(groupMapper)` genutzt werden.

Hinweis: Wie bereits in Kapitel 16 erklärt, kann nicht *direkt* über eine `Map` iteriert werden (und genauso wenig kann man einen Stream über eine `Map` erzeugen). Einen Stream über eine `Map<K,V>` erzeugt man deshalb stattdessen ...

- ... *meistens* über die Menge der Schlüssel (`myMap.keySet()` → `Set<K>`) und holt sich den zugehörigen Wert falls nötig mit `myMap.get(myKey)`
- ... *oder* über die Eintragsmenge (`myMap.entrySet()` → `Set<Entry<K,V>>`) und holt sich Schlüssel/Wert mittels `myEntry.getKey()` bzw. `getValue()`
- ... *oder (eher selten)* über die Menge der Werte (`myMap.values()` → `Set<V>`). Das macht aber nur Sinn, falls man die Schlüssel nicht benötigt!

(a) 4P. `static List<Integer> parseGrades(int pupilNr, int subjectNr)`:
Liest für den Schüler mit Nr. `pupilNr` die Liste aller seiner Noten im Fach mit Nr. `subjectNr` aus der Datenbank, z. B. [3, 1, ...] für Schüler 45, Fach 2.

(b) 4P. `static Map<String, List<Integer>> parseGrades(int pupilNr)`:
Gibt für den Schüler mit Nr. `pupilNr` eine Zuordnung von Schulfachbezeichnungen auf Notenlisten zurück, z. B. {"Kunst" → [3, 1, ...], "Deutsch" → ...}.

(c) 1P. Implementieren Sie die Klasse `Pupil` zur Modellierung der Schüler. Ein Schüler speichert die Werte eines Datensatzes der *schueler.csv*-Tabelle als Attribute. Außerdem speichert er eine Zuordnung von Schulfächern auf Listen von Noten.

Stellen Sie die folgenden *Getter*-Methoden (und zugehörige Attribute) bereit:

- `int getNr()`
- `String getName()`
- `boolean isFemale()`
- `String getSchoolClass()`
- `Map<String, List<Integer>> getGrades()`

Die Attribute werden über den `Pupil`-Konstruktor initialisiert. Dieser erwartet die Initialwerte als Parameter (in der Reihenfolge wie auch die Methoden genannt sind).

(d) 5P. `static Set<Pupil> parsePupils()`: Erzeugt die Menge aller Schüler.
Beachten Sie: Die Datenbank kann mit `parsePupils` vollständig eingelesen werden, d. h. die nachfolgenden Methoden müssen keine Tabelle mehr einlesen.

(e) 5P. Erweitern Sie die Klasse `Pupil` um eine Methode `int countOnes()`, welche für einen Schüler zurückgibt, wie oft dieser die Note 1 erhalten hat.

(f) 4P. `static List<Pupil> mostOnes(Set<Pupil> pupils)`:
Ermittelt aus einer Menge von Schülern `pupils` den Schüler, der am häufigsten die Note 1 erhalten hat. Bei „Gleichstand“ sind das mehrere, daher Rückgabe als `List`.

(g) 1P. Implementieren Sie die Klasse `Report` zur Modellierung eines Jahreszeugnisses. Jedes Zeugnis speichert im Attribut `grades` vom Typ `Map<String,Integer>` eine Zuordnung von Fächernamen auf eine „Gesamtnote“ im jeweiligen Fach. Der Konstruktor bekommt `grades` als Parameter übergeben. Stellen Sie einen *Getter* bereit.

Jedem Schüler wird ein Zeugnis zugeordnet. Ergänzen Sie die Klasse `Pupil` um ein Attribut vom Typ `Report` mit dem *Getter* `getFinalReport` sowie einem *Setter*.

(h) 11P. `static Set<Report> createReports(Set<Pupil> pupils)`:
Erstellt für jeden Schüler der Menge `pupils` ein Zeugnis. Jedes Zeugnis enthält eine Zuordnung der vom jeweiligen Schüler „belegten“ Fächer (und nur diesen!) auf dessen „Gesamtnote“ im entsprechenden Fach. Ein Fach wird von einem Schüler genau dann *belegt*, wenn er darin mindestens eine Note erhalten hat. Die *Gesamtnote* entspricht dem Durchschnitt aller Noten des Schülers im jeweiligen Fach, mathematisch gerundet auf eine ganze Zahl. Dabei soll jedoch im Fall von *.5* zur besseren Note abgerundet werden (Beispiel: 1.5 wird zu 1, 1.7 wird zu 2). Ein jeder Schüler soll sein Zeugnis anschließend im dafür vorgesehenen Attribut speichern.

(i) 3P. Ergänzen Sie die Klasse `Report` um die `toString`-Methode, um ein Zeugnis als String darzustellen. Pro Zeile soll jeweils ein Fach mit zugehöriger Gesamtnote in diesem Fach stehen, wobei die Fächer alphabetisch sortiert sein sollen.

(j) 3P. `static double finalGrade(Pupil pupil)`: Berechnet für den Schüler `pupil` den Zeugnisdurchschnitt, also den Mittelwert aller seiner (gerundeter) Gesamtnoten.

(k) 3P. `static double averageFinalGrade(Set<Pupil> pupils, boolean female)`:
Berechnet die durchschnittliche Zeugnisgesamtnote (`finalGrade`) aller weiblichen (falls `female` gleich **`true`**) bzw. männlichen (falls `female` gleich **`false`**) Schüler.

(l) 8P. `static List<Pupil> bestOf(Set<Pupil> pupils, String context)`:
Ermittelt die Schüler mit der besten Zeugnisgesamtnote (`finalGrade()`) aus dem gegebenen Kontext `context`, welcher als einzelne Schulklasse (z. B. `"4A"`), alle Klassen einer Stufe (z. B. `"4"`) oder alle Stufen (`""` = alle Schüler) gegeben sein kann.

Beispiel: `bestOf(...,"4")` gibt eine Liste der besten Viertklässler zurück.

(m) 1P. Definieren Sie im Hinblick auf die nächste Teilaufgabe ein Enum `SchoolType` mit den Werten `GYMNASIUM`, `REALSCHULE` und `MITTELSCHULE` und ergänzen Sie die Klasse `Report` um ein Attribut des Typs `SchoolType` mit zugehörigem *Getter*/*Setter* `getRecommendation` bzw. `setRecommendation(SchoolType)`.

(n) 8P. `static void setRecommendations(Set<Pupil> pupils)`:
Trifft für jeden Viertklässler aus der Menge `pupils` eine Schullaufbahnempfehlung basierend auf dem Durchschnitt der Zeugnisnoten in den Fächern *Deutsch*, *Mathematik* und *HSU*. Das Gymnasium wird Schülern mit einem Durchschnitt bis *2,4* empfohlen, die Realschule bis *2,7* und die Mittelschule allen anderen (d. h. ab *3,0*).

(o) 4P. Erstellen Sie die Zeugnisse der Schüler und geben Sie diese aus.
Beantworten Sie die zu Beginn genannten Fragen der Schulverwaltung.

20. NETZWERKPROGRAMMIERUNG

Netzwerkprogrammierung wird in der Klausur ebenso wie GUIs mit sehr großer Wahrscheinlichkeit nicht abgeprüft und daher in diesem Skript nur sehr kurz bzw. gar nicht behandelt.

Für die Konsolenausgabe haben wir bisher die Methode `System.out.print()` bzw. `println()` genutzt. Für Eingaben über die Konsole stand bspw. `new Scanner(System.in).nextLine()` oder Wrapper wie `Terminal.readString()` bereit. Die Kommunikation über das Netzwerk basiert auf ähnlichen Grundlagen, welche wir in diesem Skript nur sehr oberflächlich behandeln. Alle nachfolgenden Aussagen sind also absichtlich simpel (also unsauber) formuliert. Wenn du dich bereits auf diesem Gebiet auskennst, dann ist dieser Abschnitt für dich ungeeignet.

Um über das Netzwerk zu kommunizieren werden sog. *Sockets* benötigt. Wir unterscheiden grundsätzlich zwischen *Server*- und *Client*-Sockets, wobei ein Server mit beliebig vielen Clients verbunden sein kann. Auf einem Rechner können wiederum mehrere Clients und/oder mehrere Server laufen, wobei der Rechner selbst eindeutig über eine Adresse (IP) identifiziert werden kann. Mit der Adresse `127.0.0.1` referenziert ein jeder Rechner sich selbst. Damit auf einem Rechner mehrere Server gleichzeitig Client-Verbindungen annehmen können, müssen die Server eindeutig angesprochen werden können. Daher ist ein Server-Socket an einen sog. *Port* gebunden (z. B. `8000`), das ist einfach irgendeine Zahl, wobei immer nur ein Server-Socket auf einem Port lauschen darf. Über die Kombination aus IP-Adresse und Port (z. B. `127.0.0.1:8000`) kann ein Client-Socket die Verbindung zu einem Server-Socket herstellen. Der Server-Socket muss diesen Verbindungswunsch aber erst annehmen (`accept`), bevor weitere Kommunikation stattfindet.

Sind Client und Server miteinander verbunden, so können diese z. B. Textnachrichten aneinander senden. Diese Kommunikation findet über einen `InputStreamReader` bzw. `OutputStreamWriter` statt. Die Ausgabe (Senden) erfolgt ähnlich zur Konsolenausgabe über die Methode `print()` bzw. `println()`, die Eingabe (Empfang) bspw. über die Methode `readLine()`. Wichtig zu wissen ist, dass das Senden nicht direkt erfolgt, sondern nur zwischengespeichert wird, denn wir werden mit einer Klasse `BufferedReader` arbeiten, welche alle über die `print`-Methoden ausgegebenen Symbole erst einmal nur zwischenspeichert. Über die Methode `flush()` werden alle zwischengespeicherten Symbole dann erst tatsächlich übermittelt. `readLine()` blockiert übrigens so lange, bis wir tatsächlich eine Nachricht empfangen.

Wenn wir einen Server implementieren, speichert sich dieser das Server-Socket ab. Über dieses Socket kann ein Server neue Verbindungen annehmen. Damit der Server mit dem oder den Client(s) kommunizieren kann, muss er sich diese(n) allerdings ebenfalls abspeichern (schließlich kann es ja mehrere verschiedene Clients geben). Ein Server merkt sich also eine Schnittstelle zu jedem Client (das ist das Client-Socket). Implementieren wir hingegen einen Client, so genügt diesem ein einziges Socket, schließlich ist ein Client immer eindeutig mit einem bestimmten Server verbunden. Kurz: Ein Server kann mit mehreren Clients verbunden sein, ein Client aber nur mit einem Server.

Hier ein Beispiel für einen Echo-Server. Dieser erlaubt nur die gleichzeitige Verbindung eines einzigen Clients. Trennt sich dieser, so kann der nächste Client eine Verbindung aufbauen. Der Echo-Server schickt dem Client das zurück, was er vom Client empfängt:

```
public class EchoServer {
   public static void main(String[] args) throws IOException {
      ServerSocket server = null;
      try {
         server = new ServerSocket(8000); // Server-Socket erzeugen
      } catch (BindException e) {
         System.out.println("8000 ist bereits durch einen Server belegt.");
         return;
      }
      try {
         while (true) {  // dauerhaft Clients akzeptieren (jeweils nur einer gleichzeitig)
            Socket client = server.accept();  // neuen Client annehmen

            BufferedReader in = new BufferedReader(new InputStreamReader(
                                                   client.getInputStream()));
            PrintWriter out = new PrintWriter(new OutputStreamWriter(
                                                   client.getOutputStream()));

            // Start: Ab jetzt kann mit dem Client kommuniziert werden (über in und out)
            while (true) {
               String line = in.readLine();  // empfange Daten vom Client
               if (line == null) // Client hat Verbindung geschlossen
                  break;
               out.println(line);  // dieselben Daten an den Client zurückschicken
               out.flush();  // Senden durchführen (println schreibt nur in den Buffer)
            }
            // Ende

            client.close();  // Verbindung zum zuvor angenommenen Client schließen
         }
      } finally {
         server.close(); // am Ende das Server-Socket wieder schließen (in jedem Fall)
      }
   }
}
```

Einfacher Client, der eine Nachricht sendet und die Server-Antwort auf der Konsole ausgibt:

```
public class Client {
   public static void main(String[] args) throws IOException  {
      Socket socket = new Socket("127.0.0.1", 8000);  // zum Server verbinden

      try {
         BufferedReader in = new BufferedReader(new InputStreamReader(
                                                socket.getInputStream()));
         PrintWriter out = new PrintWriter(new OutputStreamWriter(
                                                socket.getOutputStream()));

         out.println("Dieser Text wird an den Server geschickt");
         out.flush();

         System.out.println("Antwort des Servers: " + in.readLine());
      } finally {
         socket.close();
      }
   }
}
```

21. THREADS

Threads sind zeitlich parallel zueinander ausgeführte „Programme“. Bislang haben wir nur mit einem Thread gearbeitet, der `main`-Methode. Neben diesem Ausführungsstrang läuft eigentlich immer noch ein zweiter Thread, die *Garbage Collection* (vgl. Kap. 12). Um nun mehrere Methoden gleichzeitig auszuführen, gibt es die Klasse `Thread`, die das funktionale Interface `Runnable` (`run`-Methode) implementiert. Jeder Thread wird für sich betrachtet sequenziell ausgeführt.

Beispiele zur Erzeugung eines Threads zur Ausgabe der Zahlen von 1 bis 10:

```
private static void print(String senderName) {   // diese Methode sei gegeben
    for (int i = 1; i <= 10; i++) {
        System.out.println(senderName + " sagt: " + i);
    }
}
```

1. Als **Unterklasse von Thread** (meistens nicht schön):

```
public class PrinterThread extends Thread {
    public void run() {
        print(this.getName());
    }
}

Thread t0 = new PrinterThread();
t0.start();
```

2. Als eigene **Klasse, die das Interface `Runnable`** implementiert (eher besser als 1.):

```
public class PrinterRunnable implements Runnable {
    public void run() {
        print("Thread-2 (Runnable)");
    }
}

PrinterRunnable r = new PrinterRunnable();
Thread t2 = new Thread(r);
t2.start();
```

3. Mittels einer **anonymen Klasse** (verhält sich wie 2., vgl. Kap. 19.1):

```
Thread t3 = new Thread(new Runnable(){
    public void run() {
        print("Thread-3 (anonym)");
    }
});
t3.start();
```

4. Mittels eines **Lambda-Ausdrucks** (→ Kap. 19.1):

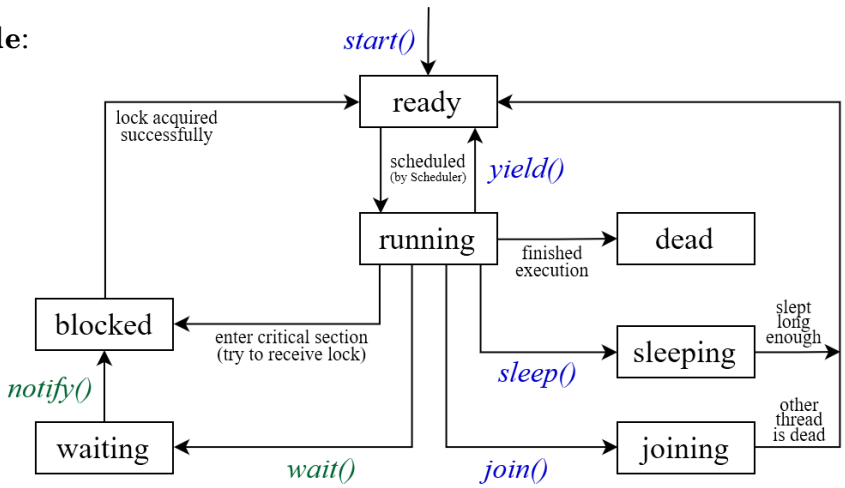
```
Thread t4 = new Thread(() -> {
    print("Thread-4 (Lambda)");
});
t4.start();   // oder direkt in einer Zeile ohne Zwischenspeichern:

new Thread(() -> print("Thread-5 (Lambda)")).start()
```

Die Klasse `Thread` stellt einige wichtige Methoden bereit, wobei wir die `run`-Methode entweder selbst implementieren oder – falls wir eine Klasse schreiben, die `Runnable` implementiert – die Standardimplementierung von `Thread` an die `run`-Methode des im Konstruktor übergebenen `Runnable`-Objekts delegiert. Die `run`-Methode ist an sich eine ganz normale Methode, die sequenziell ausgeführt wird. Ebenso ist unsere Unterklasse von `Thread` bzw. die Klasse, die `Runnable` implementiert, eine ganz normale Klasse (mit weiteren Methoden, Attributen usw.).

Wichtige Methoden für Threads:

- `public void start()`: Führt die Thread-Erzeugung durch, d. h. der Thread bekommt den Zustand „*ready*". Sobald der Scheduler den Thread in den Zustand „*running*" versetzt, führt dieser die `run`-Methode aus. Threads werden niemals mit `t.run()` ausgeführt, da sonst der entscheidende Schritt der Thread-Erzeugung fehlt!
- `public static void yield()`: Der aktuell ausgeführte Thread unterbricht seine Ausführung selbstständig und wird von „*running*" auf „*ready*" zurückgesetzt. Dies ist sinnvoll, wenn ein Thread weiß, dass er aktuell keine Rechenzeit benötigt (z. B. weil er auf etwas warten muss) und diese somit an einen anderen Thread abgeben kann.
- `public void interrupt()`: Beim Aufruf von `t.interrupt()` wird an den `Thread t` ein *Interrupt*-Signal gesendet. Dieser nimmt dieses Signal wahr, indem dort an manchen Stellen (z. B. im Zustand „*joining*" oder „*sleeping*" ist) eine *`InterruptedException`* geworfen wird. Der Thread kann darauf reagieren (festgelegt im jeweiligen `catch`-Block), muss aber nicht. Es handelt sich somit mehr um eine „Bitte" als um eine „Aufforderung".
- `public final void join() throws InterruptedException`: Der Thread, der `t.join()` aufruft, wartet anschließend auf Beendigung des Threads `t` und wechselt somit in den Zustand „*joining*". Er kehrt erst in den Zustand „*ready*" zurück und fährt mit der Ausführung fort (in der nächsten Codezeile), wenn `t` beendet wurde. Er kann beim Warten unterbrochen werden und sollte in einem `try-catch`-Block spezifizieren, wie er darauf reagieren möchte (ggf. gar nicht, indem einfach wieder `join` aufgerufen wird → Endlosschleife).
- `public static void sleep(int msec) throws InterruptedException`: Legt den aktuell in Ausführung (Zustand „*running*") befindlichen Thread für eine bestimmte Zeit schlafen. Der Thread kehrt frühestens nach der spezifizierten Zeit wieder in den Zustand „*ready*" zurück, beginnt die Ausführung aber nicht unbedingt zur selben Zeit wieder!
- `public final String getName()` und `public final void setName(String name)`

Threadzustände:



(115) ★★☆ **Reader-Writer-Szenario**: Gegeben sei folgender Code, welcher fünf *Reader*-Threads und zwei *Writer*-Thread erzeugt und startet. Alle Threads greifen dabei auf ein (gemeinsames) Objekt der Klasse `Shared` zu, wobei *Reader* den Wert des Attributs `value` (siehe Klasse `Shared`) lesen und *Writer* diese Variable beschreiben.

```
Shared share = new Shared();
for (int i = 1; i <= 5; i++)
    new Reader(share).start();
for (int i = 2; i <= 3; i++)
    new Writer(share, i).start();
```

Die Klasse `Reader` sei in einer Datei *Reader.java* und `Writer` in einer Datei *Writer.java* implementiert. Die Klasse `Shared` sei wie folgt gegeben (in einer Datei *Shared.java*):

```
public class Shared {
    public double value;
    public boolean changed = false;
}
```

Das Code-Gerüst der Klassen `Reader` und `Writer` finden Sie auf der nächsten Seite. Implementieren Sie nun die `run`-Methode in beiden Klassen:

- Ein *`Writer`* soll nacheinander die natürlichen Zahlen von 1 bis n in `share.value` speichern. n wird dem *Writer* über den Konstruktor übergeben. Die Überschreibung des alten Werts soll erst dann geschehen, wenn `share.changed` gleich **`false`** ist, der Wert also bereits gelesen wurde und demnach nicht mehr „neu verändert“ ist. Anderenfalls wartet der *Writer* bis diese Bedingung erfüllt ist (ein *Reader* also den Wert gelesen hat). Er muss dazu entsprechend von einem *Reader* aus dem Wartezustand „aufgeweckt“ werden. Wurde `share.value` vom Writer überschrieben, so wird `share.changed` auf **`true`** gesetzt und alle wartenden *Reader* aufgeweckt.
- Ein *`Reader`* liest den Wert der Variable `share.value` genau einmal ein und gibt ihn auf der Konsole aus, sofern `share.changed` gleich **`true`** ist. Anschließend wird `share.changed` auf **`false`** gesetzt und die *Writer* aufgeweckt. Solange diese Bedingung jedoch nicht erfüllt ist, wartet ein *Reader*.

Wird ein Thread an irgendeiner Stelle durch ein *Interrupt*-Signal unterbrochen, so soll er sich beenden. Synchronisieren Sie das oben beschriebene Szenario, sodass kein Wert doppelt gelesen oder ohne zu lesen nacheineinander geschrieben wird.

Vermeiden Sie „Busy-Waiting“ und verwenden Sie keine weiteren Membervariablen, Methoden oder Klassen. Das Code-Gerüst darf nicht verändert werden; implementieren Sie lediglich die `run`-Methoden.

Verwenden Sie für die Konsoleausgabe die Methode `System.out.println(...)`.

Zusatzfrage: Ist die Ausgabereihenfolge der Zahlen durch die Reader eindeutig?

```
public class Reader extends Thread {

    private final Shared share;

    public Reader(Shared s) {
        share = s;
    }

    public void run() {









    }
}

public class Writer extends Thread {

    private final Shared share;
    private final int n;

    public Writer(Shared s, int n) {
        share = s;
        this.n = n;
    }

    public void run() {









    }
}
```

(116) ★★☆ **Consumer-Producer**: Gegeben sei folgendes Szenario: In einer einfachen *Fabrik* gibt es unter anderem zwei Arten von *Mitarbeitern*, welche für den Transport gefährlicher Flüssigkeiten zwischen zwei Fabrikhallen zuständig sind: *Lieferanten* und *Abholer*.

Zur Modellierung sollen vier Klassen implementiert werden:

- **`Fabrik`**: Zwischen den beiden Hallen steht für den Transport ein 50-Liter-Fass bereit, welches als private Membervariable der Klasse `Fabrik` modelliert wird. Dort stehen außerdem *Getter* und *Setter* bereit, die Zugriff auf das Fass bieten. Die Mitarbeiter der Fabrik sollen in einer `LinkedList<Mitarbeiter>` verwaltet werden. Stellen Sie in der Klasse `Fabrik` eine Methode mit dem Namen `addMitarbeiter` zum Einfügen von Mitarbeitern bereit. Sie dürfen alle Methoden der Klasse `LinkedList` verwenden. Stellen Sie sicher, dass Mitarbeiter nicht doppelt eingefügt werden.
- **`Mitarbeiter`**: Alle *Mitarbeiter* besitzen eine *Identifikationsnummer*, einen *Namen* und ein fabrikinternes *Guthabenkonto* für den Lohn (wird in *Cent* gespeichert), auf welches an anderer Stelle in diesem Szenario Gehalt eingezahlt wird. Mit der Methode `double gehaltAusbezahlen()` wird das Gehalt in *Euro* ausbezahlt. Das Guthabenkonto soll zu Beginn leer sein. Identifikationsnummer und Name sollen nach der Erzeugung des Mitarbeiter-Objekts nicht mehr geändert werden können.
- **`Lieferant`**: *Lieferanten* sind Mitarbeiter, die die Flüssigkeit aus der einen Fabrikhalle, in der die Flüssigkeit ohne Begrenzung bereitsteht, ununterbrochen zum Fass bringen. Ihre Transportkübel fassen acht Liter. Das Fass soll auf keinen Fall überlaufen, d. h. die Lieferanten sollen nur so viel eingießen, bis die 50-Liter-Markierung *genau* erreicht ist. Anschließend müssen sie warten, bis ein Abholer Flüssigkeit abtransportiert.
- **`Abholer`**: *Abholer* sind Mitarbeiter, die die Flüssigkeit vom Fass in die andere Fabrikhalle bringen und dafür 10-Liter-Eimer nutzen. Ist das Fass leer, müssen sie warten. Die Flüssigkeit wird danach direkt von Maschinen verarbeitet und kann daher unbegrenzt abtransportiert werden.

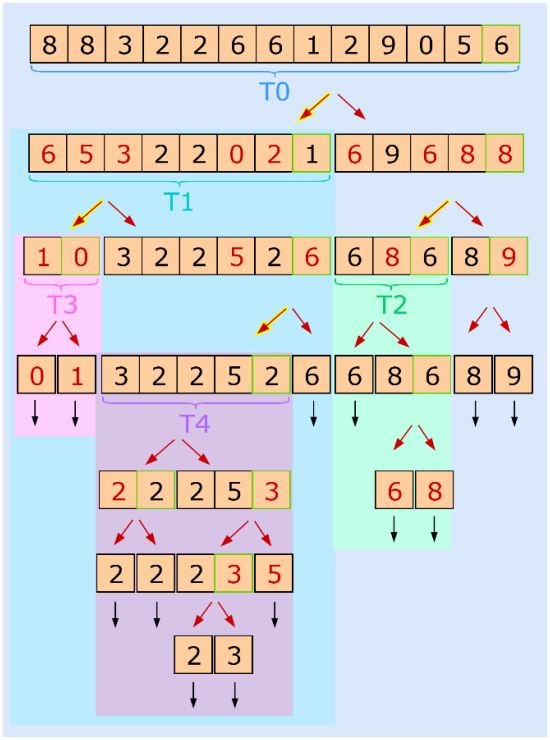
`Abholer` und `Lieferant` sollen das Interface `Runnable` implementieren und die oben beschriebene Arbeit so lange verrichten, bis sie durch ein *Interrupt*-Signal unterbrochen werden. Je transportiertem Liter erhalten sie als Vergütung eine Cent-Pauschale, die im Konstruktor übergeben wird. Synchronisieren Sie den Zugriff auf das Fass.

Implementieren Sie auch in der Klasse `Fabrik` die Methode `run()`, welche alle Mitarbeiter der Liste in jeweils einem eigenen Thread ausführt. Nach zehn Sekunden sollen alle Mitarbeiter-Threads mit einem *Interrupt* beendet werden.

Sie dürfen alle Klassen beliebig erweitern oder verändern, sofern das obige Szenario modelliert wird. Vermeiden Sie in jedem Fall „Busy-Waiting“.

(117) ★★☆ **Quicksort parallel:** Mit einer durchschnittlichen Laufzeit von $\mathcal{O}(n * \log(n))$ zählt Quicksort zu den effizientesten Sortierverfahren für *universelle** Szenarien.

In dieser Aufgabe ist der Großteil einer Quicksort-Implementierung gegeben, wobei diese (1.) mit beliebigen *Arrays* umgehen können soll und (2.) die Sortierung auf Wunsch parallelisiert. Das Pivot-Element wird in dieser Implementierung immer ganz rechts gewählt (vgl. Skizze). Der bereits implementierte Teil innerhalb der `quickSort`-Methode übernimmt die gedankliche Unterteilung des Bereichs von `start` bis `end` in zwei Teilbereiche (von `startLeft` bis `endLeft` bzw. `startRight` bis `endRight`, Grenzen inkl.), wobei der *linke* Teilbereich anschließend lediglich Elemente *kleiner gleich* dem Pivot-Element enthält (bzw. der *rechte* lediglich Elemente *größer gleich* dem Pivot-Element). Dadurch umsortierte Elemente sind in der Skizze rot markiert. Rekursive Aufrufe werden durch rote Pfeile repräsentiert, wobei diejenigen Aufrufe, welche in einem neuen Thread ausgeführt werden, zusätzlich gelb hinterlegt sind. Schwarze Pfeile illustrieren das Verlassen der Methode im Fall *end* ≤ *start*.



(1.) Nachdem die Vergleichbarkeit zwischen Elementen für Quicksort essenziell ist, setzen wir voraus, dass der Typ `E` das Interface `Comparable<E>` implementiert, d. h. ein jedes `E`-Objekt ist über die `compareTo(E)`-Methode mit einem beliebigen anderen `E`-Objekt vergleichbar. Der Aufruf `e1.compareTo(e2)` wertet zu einem negativen Wert aus, wenn `e1` *vor* `e2` steht, bzw. zu 0, falls `e1` gleich `e2` ist, anderenfalls positiv.

(2.) Parallelität ermöglichen wir über den Parameter `int nThreads`, welcher die maximale Anzahl an Threads spezifiziert, die für die Sortierung erzeugt werden dürfen (inkl. des Start-Threads). Wird für `nThreads` ein Wert kleiner gleich 1 übergeben, so sollen keine weiteren Threads erzeugt werden (= sequenzielle Ausführung). Bei einem Wert größer als 1 müssen weitere Threads erzeugt werden. Der rekursive Aufruf für die Elemente des *linken* Teilbereichs soll in diesem Fall in einem *neuen* Thread ausgeführt werden, während der *aktuelle* Thread die Sortierung des *rechten* Array-Abschnitts übernimmt. Die Aufteilung der verbleibenden Thread-Anzahl auf beide Threads geschieht proportional zur Größe des Teilbereichs entsprechend der bereits implementierten Hilfsmethode *`splitThreads`*. Diese erwartet als Parameter die Größe des linken Teilbereichs (um welchen sich der *neue* Thread kümmern soll), die Gesamtgröße des aktuellen Bereichs sowie die aktuelle Anzahl `nThreads` und gibt die max. Anzahl an Threads zurück, die für den linken Teilbereich bereitstehen (≥ 1).

* *Universell* bedeutet, dass keine Kenntnisse über die Elemente oder ihre ursprüngliche Anordnung vorliegen. Für *konkrete* Szenarien – wenn gewisse Umstände bekannt sind, bspw. dass nur Zahlen eines bestimmten Wertebereichs sortiert werden sollen – existieren effizientere Algorithmen (bspw. Radixsort).

(a) Implementieren Sie die generische Methode *swap* entsprechend ihrer Verwendung

(b) Vervollständigen Sie den mit *TODO* gekennzeichneten Teil entsprechend der vorherigen Beschreibung. Sie dürfen bestehende Methodensignaturen nicht verändern und keine weiteren Methoden oder Klassen hinzufügen (außer *swap*). Zudem erlauben wir keine Unterbrechungen, d. h. eine ggf. während eines Wartevorgangs auftretende `InterruptedException` soll ignoriert werden (→ Warten fortsetzen).

```
public static <T extends Comparable<T>> void quickSort(
                                                T[] array, int nThreads) {
    quickSort(array, 0, array.length - 1, nThreads);
}

private static <E extends Comparable<E>> void quickSort(
                        E[] array, int start, int end, int nThreads) {
    if (end > start) {
        int left = start, right = end;
        final E pivot = array[end];

        while (left <= right) {
            while (array[left].compareTo(pivot) < 0)  left++;
            while (array[right].compareTo(pivot) > 0)  right--;
            if (left <= right)  swap(array, left++, right--);
        }

        final int startLeft = start, endLeft = right;
        final int startRight = right+1, endRight = end;

        // TODO (b)









}}

private static int splitThreads(int sizeLeft, int size, int nThreads) {
    return 1 + Math.round(1F * sizeLeft / size * (nThreads-2));
}

// TODO (a)
```

DAS WAR'S...

Wie du dir vielleicht vorstellen kannst, ist der Zeitaufwand für die Konzeptionierung sinnvoller Übungsaufgaben, bei denen insbesondere auch diverse Sonderfälle aufgezeigt werden (welche in der Klausur leider genauso abgeprüft werden wie die Normalfälle), immens – ganz zu schweigen von der Erstellung von Grafiken und Lösungsvorschlägen/-erklärung (welche ich übrigens auch zu einigen Klausuren erstellt habe)... Aus diesem Grund war das Skript bislang nur käuflich zu erwerben. Leider wird meine Zeit nicht ausreichen, um das Skript optimal für kommendes Semester (2020/21) anzupassen, auch wenn die meisten Inhalte identisch bleiben werden. Da der wesentliche Sinn des Skripts aber immer war, ein möglichst gutes Hilfsmittel zur Klausurvorbereitung zu sein, habe ich mich dazu entschieden, es dir kostenlos bereitzustellen. Für die Beantwortung kleinerer Fragen zum Skript bin ich weiterhin unter *stefan@stecrz.de* erreichbar.

Wenn ich dir mit diesem Skript bei deiner Vorbereitung helfen konnte, würde ich mich sehr freuen, wenn du mich das im Gegenzug wissen lässt. Entscheide z. B. mit einer kleinen Spende über *paypal.me/stecrz* (oder obige E-Mail) selbst, wie viel dir dieses Skript wert ist. Auch ich bin Student, deshalb verstehe ich natürlich auch, wenn dein Konto nur 1-2 Euro übrig hat. Und falls nicht: Ich freue mich genauso über ein einfaches *Danke*! 😇