Lösungsvorschläge zum Übungsskript
Einführung in die Informatik I

Stand: 19.02.2021

Stefan Berktold
www.stecrz.de

Falls du über einen QR-Code (oder den Link im Skript) hier gelandet bist, dann ist das die richtige Lösungsversion ☺ Anderenfalls überprüfe bitte deine **Skriptversion** (steht auf der zweiten Seite im Skript = Rückseite des Deckblatts).

08.06.20 bis heute: *Diese Datei!*
13.01.20 bis 30.02.20: *Klicke hier!*
10.01.20: *Klicke hier!*
19.12.19 bis 08.01.20: *Klicke hier!*
23.04.19 bis 18.12.19: *Klicke hier!*
11.03.19 bis 22.04.19: *Klicke hier!*
01.02.19 bis 10.03.19: *Klicke hier!*

Komm in mein Team!

EIDI bestanden? Zeit für ein echtes Projekt.

2020 hätte ich nicht gedacht, dass dieses Skript auch fünf Jahre später noch genutzt werden würde. Und noch weniger hätte ich gedacht, dass ich hierüber mal eine Stelle ausschreiben würde. Aber ich hätte auch erst gar nicht gedacht, dass einer der TUM-BWLer aus meinen EIDI-Crashkursen später mal mein Geschäftspartner wird.

Long Story short: Seit Ende 2023 sind Paul und ich Inhaber und Geschäftsführer von **VollCorner**, einem Münchner Bio-Lebensmittelhändler mit fast 400 Mitarbeitenden in 20 Biomärkten, der seit über 35 Jahren eine klare Mission verfolgt: Nachhaltige, faire und regionale Lebensmittel für München. Daran arbeite ich heute – mit derselben Energie, die ich während meines Informatik-Bachelors in Projekte wie dieses EIDI-Skript gesteckt habe.

Jetzt suche ich einen Allrounder, der Lust hat, unsere Daten und Prozesse auf das nächste Level zu bringen. Einen Denker, der kritisch hinterfragt, und Macher, der Lösungen findet und selbst entwickelt. Jemanden, der Micromanagement genauso hasst wie ich, stattdessen lieber frei, flexibel und eigenverantwortlich arbeitet, aber dafür so lange im Tunnel bleibt bis auch der letzte Edge-Case gefixt ist.

Wir sind ein mittelständisches Unternehmen im Wandel, den du aktiv mitgestalten kannst. Du übernimmst Themen, die unmittelbaren Mehrwert stiften – und wenn du willst gibt es auch jede Menge Freiraum für eigene Ideen und Projekte.

Klingt spannend? Dann packen wir‘s noch in **2025** an! Lebensläufe sind nett, aber viel mehr interessiert mich dein Drive.

Ich freu mich auf deine Nachricht!

Stefan

Kontakt: linkedin.com/in/berktold
Mail: stefan.berktold@vollcorner.de

Paul Pfaff & Stefan Berktold

Data- & Process-Engineer

Dein Profil:

- (Laufendes) Studium der (Wirtschafts-) **Informatik** o. ä.
- Hohes Maß an analytischem und **logischem Denkvermögen**
- Interesse an **Datenintegration** und Prozessoptimierung
- **Programmiererfahrung** (bestenfalls mit **Python**)
- Grundlegendes Verständnis von **Datenbanken** und **SQL**
- Frontend-Skills (Flask, JavaScript) sind nice, aber kein Muss

Mögliche Aufgaben:

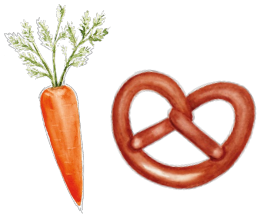
- **Data Warehouse** betreuen, **ETL-Prozesse** implementieren, Schnittstellen (z. B. Zeiterfassungssystem) per API anbinden, ggf. DWH-API entwickeln, um externe Tools auszurollen
- **Reporting** optimieren und Datenqualität verbessern
- **Digitalisieren und Automatisieren**, was das Zeug hält
- Neue Tools ausprobieren & integrieren (z.B. Web-Scraping)

Dein Team: 1 Geschäftsführer (also ich, wir arbeiten direkt zusammen), 1 IT-Systemadministrator, ca. 25 Mitarbeitende in der Zentrale (für dich insb. Controlling & Einkauf) und 20 Märkte

Dein Arbeitsplatz: Im Tunnel. Du entscheidest, wie, wo und wann du arbeitest. Kein Zeittracking, Hauptsache, du lieferst.

Umfang: Werkstudent, Teilzeit oder mehr. Es gibt genug zu tun.

Diese Beschreibung gilt für das Jahr **2025** (aber auch danach wird es spannende Projekte geben).



VollCorner Bio

Aufgabe auswählen

Hinweise:

- Nur durch das Lesen von Lösungen lernst du nicht viel. Bearbeite die Aufgaben davor selbst!
- Lerne keine Implementierungen auswendig! Es gibt unzählige mögliche Aufgabenstellungen. Die Wahrscheinlichkeit, dass eine Aufgabe genau so drankommt wie in diesem Skript geht gegen Null. Du solltest stattdessen die Konzepte verstehen, bspw. wie du eine Polymorphie-Aufgabe bearbeitest oder wie rekursives Suchen grundsätzlich funktioniert.
- Die hier gezeigten Implementierungen sind Lösungsvorschläge. Es gibt eine Vielzahl weiterer Lösungen. Teste deine Lösung in einer Entwicklungsumgebung, wenn du dir unsicher bist.

①

a) Ja. Es erfolgt ein impliziter Cast (Upcast) von `char` zu `int`.

b) Ja. Escape-Sequenz ist *ein* Zeichen (hexadezimal angegeben, 16 Bit = 2 B = 4 Stellen).

c) Nein. Präfix `0` steht für das Oktalsystem. Dort existiert das Symbol `9` nicht (nur `0` bis `7`).

d) Ja. Führt zu einem *Laufzeit*fehler (`ArithmeticException: / by zero`), kompiliert aber.

e) Ja. Es erfolgt ein impliziter Cast (Upcast) von `int` zu `long`.

f) Nein. Variblennamen dürfen nicht mit einer Ziffer beginnen.

g) Ja. In Variablennamen sind Unterstriche überall erlaubt (allerdings darf der Name nicht nur „_“ sein). Bei Zahlen sind mittige Unterstriche erlaubt (also nicht am Anfang/Ende).

h) Ja. Es erfolgt ein impliziter Cast (Upcast) von `int` zu `float` (ggf. ungenau).

i) Ja. `314e-2` entspricht $314 * 10^{-2}$, also 3,14.

j) Nein. Der maximale Wert von `short` ist $2^{15} - 1$, also 32767. Die Zahl ist also zu groß. Die Wertebereiche musst du für die Klausur nicht auswendig wissen (nur wie viele Bit).

k) Ja. Der maximale Werte von `char` ist $2^{16} - 1$. Variablennamen dürfen Ziffern enthalten.

l) Ja. Der größtmögliche `int` plus 1 ergibt den kleinstmöglichen `int`. *Nach* der Addition wird der entstehende `int` (-2147483648) durch einen impliziten Cast zu einem `long`.

m) Nein. Es ist kein impliziter Cast von `long` (`1L`) zu `int` möglich.

n) Nein. Es ist kein impliziter Cast von `double` zu `float` möglich.

o) Ja. Vorsicht: Das `e` steht nicht für einen etwaigen Exponenten, sondern ist ein gültiges Zeichen im Hexadezimalsystem (`0x`). `0x0e0` entspricht der Zahl 224,0.

p) Ja. Gleitkommazahlen ohne Suffix sind immer `double`s. Der Suffix `d` ist optional.

q) Ja. Die Oktalzahl (Präfix `0`) 17 entspricht dezimal 15. `int_` ist aufgrund des Unterstrichs ein gültiger Variablenname, während `int` ungültig wäre.

r) Ja. Die Binärzahl (Präfix `0b`) 10110 entspricht dezimal dem `int` 22.

s) Nein. Vorsicht: `028F` ist möglich, da das Oktalzahl-Präfix `0` nur bei Ganzzahlen anwendbar ist, d. h. `028F` entspricht dem `float` *28,0*. Allerdings wird anschließend ein double addiert. `float + double` ergibt `double` (impliziter Upcast von `float`) und dieser kann wiederum nicht implizit zu `float` geacastet werden, Speicherung unmöglich.

t) Nein. Der größtmögliche `char` passt nicht in ein `byte`, da `char` 16 Bit (2 Byte) breit ist.

u) Ja. `0xBCE2` entspricht dezimal 48354. Ein Exponent ist nicht vorhanden.

v) Nein. `super` ist ein reserviertes Schlüsselwort in Java. Mit einem anderen Variablennamen wäre die Zuweisung ok, denn der Vor- *oder* Nachkommateil einer Gleitkommazahl kann weggelassen werden, d. h. `.1` entspricht `0.1` und `1.` entspricht `1.0`.

w) Nein. Auf das Zeichen `e` muss eine Zahl (der Exponent folgen).

x) Nein. `char` ist vorzeichenlos, kann also nur positive Werte aufnehmen.

y) Nein. Der Unterstrich ist nur zwischen Ziffern bzw. Unterstrichen erlaubt (...`1_0_L`).

z) Ja. Währungssystembole sind erlaubt. Klicke *hier* für weitere Infos.

A) Ja. `0.e5` entspricht `0.0e5` entspricht $0{,}0 * 10^5$ entspricht 0,0.

B) Nein. Eine Zahl darf nicht mit einem Unterstrich beginnen oder enden. Außerdem ist der größtmögliche Wert für `byte` $2^7 - 1 = 127$, d. h. 250 kann nicht in `byte` gespeichert werden.

C) Nein. Bei Gleitkommazahlen im Hexadezimalsystem ist der Exponent (eingeleitet mit `p`) zwingend erforderlich, d. h. es fehlt bspw. `p0`.

D) Ja. Das Präfix `0` existiert nur für Ganzzahlen, d. h. `018.` entspricht 18,0. Das Währungssymbol für Yen (¥) ist erlaubt, vgl. *z)*.

E) Nein. `String+String` ergibt `String`. Ein `String` besteht aus `char`s, nicht andersrum.

F) Ja. `P` leitet den notwendigen Exponenten für hexadezimale Gleitkommazahlen ein. `0x.AP1d` entspricht somit `0.A`$_{16} * 2^1 = \frac{10}{16} * 2 = 1{,}25$.

G) Nein. Ein `char` muss aus einem Symbol bestehen. Das leere Symbol existiert nicht. Außerdem ist der Variablenname `G!` unzulässig, da er ein Ausrufezeichen enthält.

H) Ja. `float * int` ergibt `float`.

I) Nein. `float + double` ergibt `double` → kann nicht in `float` gespeichert werden.

J) Ja. `char - char + int` ergibt `int` (hier 1). Da die Berechnung nicht von Variablen abhängt (nur dann!), kann der Compiler erkennen, dass 1 in einen `char` passt. Wir wir gesehen haben (z. B. in Teilaufgabe *m)*) führt der Compiler diese Erkennung nicht immer durch, sondern nur für Zuweisungen von `int`s an `byte`/`char`/`short`. Daher sind diese Typen in der Abbildung heller dargestellt.

K) Nein. Der Exponent (eingeleitet mit `p`) muss immer im Zehnersystem angegeben werden.

L) Ja. Oktalzahl `0_7` entspricht dezimal ebenfalls 7. `0_8` wäre nicht möglich.

M) Nein. Vorsicht: Die rechte Seite wäre ok! Hier wird der Wert des Symbols @ negiert (entspricht −64). Allerdings ist der Variablenname ungültig, da er einen Operator enthält.

N) Nein. Der kleinstmögliche `int` passt nicht in einen `short`, daher kann die Optimierung aus J) nicht durchgeführt werden.

O) Ja. `0xa2.382ap3f` entspricht `(float)` (`A2.382A`$_{16} * 2^3$).

P) Nein. `#` ist kein gültiges Zeichen für Variablennamen. Klicke *hier* für weitere Infos.

②

a) Falsch. `a` könnte z. B. den Wert `0.0` annehmen und `0.0 / 0.0 == NaN`.

b) Wahr. Es kann aber zu einem *Laufzeit*fehler kommen (wenn `a` den Wert `0` annimmt).

c) Wahr. Da `long` eine Größe von 64 Bit hat, kann er zwar mehr unterschiedliche Zahlen speichern als ein `float` (32 Bit), allerdings haben Gleitkommazahlentypen durch die Angabe des Exponenten die Möglichkeit, deutlich größere Zahlen zu speichern.

d) Wahr. Bei einem Überlauf wird die Zählung einfach bei der kleinstmöglichen Zahl fortgesetzt, d. h. „größte Zahl“ + 1 = „kleinste Zahl“ (Grund: Bitweise Zahlendarstellung).

e) Falsch. Mit `float`s sind zwar größere, allerdings weniger Zahlen darstellbar (vgl. *c)*).

Beispiel: Wenn `a` den Wert 16777217 annimmt, wird b den Wert 16777216 annehmen.

f) Wahr (vgl. *d)*).

g) Falsch. Dies gilt z. B. nicht für `x = Integer.MAX_VALUE`.

h) Falsch. Die erste Anweisung zur Berechnung eines Mittelwerts aus zwei Zahlen sieht man häufiger als die zweite, allerdings sie eigentlich fehlerhaft. Durch die Addition kann es eventuell zu einem Überlauf kommen, sodass der Mittelwert aus zwei großen positiven Zahlen plötzlich eine negative Zahl wäre. Wenn wir davon ausgehen können, dass `right` größer ist als `left` und beide Zahlen positiv sind (das ist meistens der Fall), dann wird dies bei der zweiten Anweisung nicht passieren, da die Differenz dann auf jeden Fall eine nichtnegative Zahl zwischen beiden Zahlen ergibt und die Addition zu `left` einen Wert liefert, der definitiv kleiner ist als `right`. Zur Berechnung der Mitte zwischen zwei Werten solltest du daher am besten immer den zweiten Ausdruck verwenden.

i) Falsch. Das würde man vielleicht grundsätzlich nach Teilaufgabe *d)* erwarten, da `'\uFFFF'` der größtmögliche und `0` (`'\u0000'`) der kleinstmögliche Wert für einen `char` ist. Allerdings sind die arithmetischen Operatoren in Java nur auf `int`, `long`, `float` und `double` definiert. Daher stellen `char`, `short` und `byte` auch hier eine Ausnahme dar. Es erfolgt dann immer ein impliziter Upcast zu `int`, d. h. z. B. `char + char` ergibt `int`.

j) Wahr. Der Ausdruck `-1.0/0` (`double/int`) wird bspw. durch einen automatischen Upcast zu dem Ausdruck `-1.0/0.0`, welcher wiederum zu `-Infinity` auswertet (`0/0. == NaN`).

k) Wahr. Ein `double` hat für Mantisse und Exponent jeweils mehr Bits zur Verfügung.

l) Falsch. Die Genauigkeit von Gleichkommazahlen ist begrenzt, wobei Genauigkeit nicht den dezimalen Nachkommastellen entspricht. Daher kann es schon bei kleinen Berechnungen zu Ungenaugkeiten kommen, bspw. `0.2 + 0.1 == 0.30000000000000004 != 0.3`.

③

a) Nein. `int` passt nicht in `short`, daher ist `a2 = a1` unmöglich. Auch wenn der tatsächliche Wert von `a1` zur Laufzeit eigentlich gespeichert werden könnte, wird der Compiler die Übersetzung nicht zulassen. Werden also Variablen benutzt, sind keine Erkennungen wie in der vorherigen Aufgabe möglich.

b) Ja. `double + String` ergibt `String` (hier: `"5.2"`).

c) Nein. `long` passt niemals implizit in `int`.

d) Nein. `String` und `int` stehen in keinem Zusammenhang. Eine `String`-Variable kann nur `String`s (bzw. gar nichts, also `null`) speichern.

e) Ja. `0.5` wird explizit zu `char` gecastet (wird zu `'\u0000'`), `char + char` ergibt `int` und `int` passt ohne Probleme in `long` (impliziter Upcast).

f) Ja. Der Integer `0` wird zu `byte` und anschließend zu `short` gecastet. Denach wird er in einer `int`-Variable gespeichert (impliziter Upcast von `short` zu `int`).

g) Nein. Casting zwischen primitiven Datentypen (hier `char`) und Referenzdatentypen (hier `String`) ist unmöglich. Eine Ausnahme stellen Wrapper-Klassen dar, d. h. es ist bspw. möglich, einen `int` zu `Integer` zu castet, nicht aber `int` zu `Character` o. ä.

h) Nein. Der arithmetische Operator bindet schwächer als der Cast-Operator, daher wird zunächst `5` zu `float` gecastet und anschließend `5f * 5d` gerechnet, was den `double 25.0` ergibt. Dieser kann nicht (ohne Cast) als `float` abgespeichert werden.

i) Nein. Ein `String` kann nicht multipliziert werden. Nur `+` ist auf `String`s definiert.

j) Ja. Wir haben hier zwei verschiedene Variablen `_j` und `j_`, wobei der Wert `5` zunächst in `j_` und anschließend auch in `_j` gespeichert wird. Später mehr zum Hintergrund dazu.

k) Nein. `'World'` ist kein gültiger `char`.

l) Ja. Eine `short`-Variable kann grundsätzlich nicht an eine Variable vom Typ `byte` zugewiesen werden. Da es sich hier aber um einen konkreten Wert handelt (keine Variable), erkennt der Compiler dies und erlaubt die Zuweisung von `1` an `l`.

m) Ja. Die Addition `float + double` ergibt einen `double`. Dieser wird explizit zu `float` gecastet und anschließent in `m` gespeichert. Ohne Cast wäre die Zuweisung nicht möglich.

n) Ja. Der Cast bindet wieder stärker, d. h. zunächst wird `Integer.MAX_VALUE` zu `char` gecastet (entspricht `Character.MAX_VALUE`). Anschließend wird `nn` (also `1`) addiert. Die Summe (`int`) wird implizit zu `float` gecastet und `n` erhält den Wert `65536.0`.

o) Nein. `o` ist ein `float`. Der Cast bindet stark, d. h. es wird zunächst `2` (`int`) zu einem `int` gemacht (keine Änderung). Die Multiplikation ergibt einen `float`, dieser kann nicht in `int` gespeichert werden.

p) Ja. `_p_` wird zu einem `int` gemacht und mit sich selbst mutipliziert (`int * int = int`).

q) Nein. `8` ist keine gültige Oktalziffer.

r) Nein. `int` kann nicht zu `boolean` gecastet werden.

s) Ja. `s += 5` entspricht `s = s + 5`, also der Konkatenation des `String`s `s` mit `5`.

t) Ja. Der `double` wird zu `float` gecastet und mit dem `int` `13` multipliziert (= `float`).

u) Nein. `double` passt nicht implizit in `long` (Cast zu `long` wäre nötig).

v) Ja. *float + (int * int) = float*.

w) Ja. `w` hat anschließend den Wert `"714w6"`, da der `char w` den ASCII-Wert 119 hat.

x) Ja. `0.5 > 2.0` wertet zu `false` aus und dieser Wert wird in `x` gespeichert.

y) Ja. *int + char - int = int*. Durch das Casten einer Fließkommzahl in einen Ganzzahldatentyp (z. B. `char` oder `int`) gehen die Nachkommastellen verloren.

z) Ja. Vorsicht: Das `+` ist kein Operator, sondern das Vorzeichen des Exponenten, d. h. es handelt sich hier um einen einzigen `double` `2.00e2 == 2e2 == 200.0`.

A) Nein. `o` wird nur deklariert aber nicht initialisiert (hat keinen Wert). `o.toString()` ist daher nicht möglich. Hier entsteht keine Exception! Es kompiliert erst gar nicht.

B) Ja. *double * char + float = double + float = double*.

C) Nein. *byte + byte = int*. `int` kann nicht implizit in `byte` gespeichert werden, d. h. es wäre ein Cast der gesamten Addition nötig, statt die beiden Operanden zu casten.

D) Nein. Der Inkrementoperator ist für `String`s nicht definiert.

E) Nein. *short + char = int*.

F) Nein (vgl. *g)*).

G) Ja. Der Cast ist überflüssig, denn `15 - 3 > 1 || false == true || false == true`.

H) Ja. *int + int – char = int* und jeder `int` passt implizit in `long`.

I) Ja. Der rechte Ausdruck wertet zu 2 aus, daher hat `I` anschließend den Wert `"I2"`.

J) Ja. Der Ausdruck wird zur Compile Zeit berechnet (vgl. *a)* und *L)*).

K) Nein. Die Zuweisung von `short` an `char` ist nicht implizit (ohne Cast) möglich.

L) Ja. Wie bei *J)* erkennt der Compiler sofort, dass der Ausdruck auf der rechten Seite der Zuweisung in den Wertebereich eines `char`s passt (anders als wenn Var. vorkommen!).

4

a) Ja. Der im `Integer`-Objekt gespeicherte Wert 8 wird automatisch entpackt (*unboxing*). Dann erfolgt ein impliziter Upcast von `int` zu `double`.

b) Ja. Eine als `Object` deklarierte Variable kann alles speichern.

c) Ja. Der primitive Wert wird automatisch in ein Objekt Wrapper-Klasse des primtiven Typs (hier `Integer` für den primitiven Typ `int`) gepackt (*autoboxing*).

d) Nein. Der `char 'd'` würde in ein Objekt der Klasse `Character` gepackt werden, allerdings stehen die Klassen `Integer`, `Double`, `Character` usw. in keiner Vererbungsrelation, d. h. z. B. `Character` ist keine Unterklasse von `Integer`. Daher kann eine als `Integer` deklarierte Variable auch kein `Character`-Objekt speichern.

e) Nein. Generische Typen müssen Referenzdatentypen sein. Das ist eigentlich der einzige Grund, warum man überhaupt Wrapper-Klassen benötigt. Später mehr dazu.

f) Ja (vgl. *e)*).

g) `g1` speichert eine Referenz auf ein `Integer`-Objekt mit dem Inhalt 1. `g2` speichert ebenfalls eine Referenz auf ein anderes `Integer`-Objekt mit dem Inhalt 1 (*autoboxing*). `g3` speichert den Wert 1 (*unboxing* von `g1`).

   „`g1 == g2`“: Nein. Referenzvergleich von `g1` und `g2`. Es handelt sich um zwei Objekte.
   „`g2 == g3`“: Ja. Wertevergleich durch Unboxing: Die Werte (Inhalt) sind gleich.
   „`g3 == g1`“: Ja. Wertevergleich durch Unboxing: Die Werte (Inhalt) sind gleich.

h) Ja (vgl. *b)*).

i) `i1`: Nein. Auto-Boxing von `int` nicht möglich (vgl. *d)*).

   `i2`: Ja. Der Konstruktor von `Float` erwartet einen `float` als Parameter. Hier wird ein `char` übergeben, wobei ein impliziter Cast zu `float` passiert (ASCII-Wert von `'?'`: 63). Aus dem `Float`-Objekt wird bei der Zuweisung anschließend wiederum der `float 63.0` entpackt und für die Speicherung in `i2` automatisch ein Upcast zu `double` durchgeführt.

   `i3`: Nein. Ein Variable vom Typ `Long` kann nur Objekte vom Typ `Long` speichern.

   `i4`: Ja. Aus dem `Integer` wird der `int 1` entpackt (*unboxing*), zu `float` gecastet und in ein `Float`-Objekt gepackt (*autoboxing*), welches die Variable `i4` speichert.

   `i5`: Nein. Der Konstruktor einer Wrapper-Klasse erwartet immer einen primtiven Wert des entsprechenden primitiven Typs, d. h. `Byte` erwartet ein `byte`, keinen `int`. Hier ist ein expliziter Cast `(byte)0` notwendig. `Double` erwartet bspw. einen `double`, allerdings kann man dafür `int` übergeben, denn jeder `int` lässt sich implizit zu `double` casten.

   `x`: Ja. Für den Vergleich zweier Objekte eines Wrapper-Typs (können auch unterschiedlich sein) mit >, <, >= und <=, werden diese automatisch entpackt und der Vergleich über die primitiven Werte durchgeführt.

5

a)

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

b)

			A	F	E	0	5	D	C	9	16
		3	B	4	C	D	F	0	3	1	16
+		1	1	1		1					
		4	6	4	A	E	4	D	F	A	16

c)

	5	A	B	1	C	3	A	4	13
		7	B	A	B	4	9	2	13
-		1	1		1				
	5	2	C	4	0	C	1	2	13

d)

	1	1	0	1	1	0	0	0	1	1	2
		1	1	0	0	1	0	1	1	0	2
-	1	1			1	1	1				
	0	1	1	1	0	0	1	1	0	1	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
+	1	2	1	2	1	1			
	1	0	2	3	0	3	1	4	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
+	1	1	10	10	10	1	1	1			
	1	0	1	1	0	0	1	0	0	1	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
-	1	2	2	1	1				
	0	1	0	2	1	1	0	1	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
-		2		2	1		1				
	1	5	B	1	6	3	0	A	1	2	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
+	2	1	2	3	2	1	1		
	2	2	0	6	6	1	A	8	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
-	1	10	11	10	1	10	1	1	1		
	0	0	0	1	1	0	1	0	0	1	2

·	**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)

	E	5	A	7	$_{16}$	·	B	$_{16}$
						4	D	
					6	E		
				3	7			
			9	A				
+					^{1}			
		9	D	E	2	D	$_{16}$	

b)

	F	0	1	F	6	E	$_{16}$	·	C	$_{16}$
								A	8	
							4	8		
						B	4			
					0	C				
				0	0					
		B	4							
+					^{1}		^{1}			
		B	4	1	7	9	2	8	$_{16}$	

c)

	1	2	3	$_{16}$	·	9	0	$_{16}$
						0	0	
					0	0		
				0	0			
					1	B		
				1	2			
			0	9				
+								
				A	3	B	0	$_{16}$

d)

	5	B	7	3	·	1	0	2	0	$_{16}$
					0	0	0	0	0	
							0	6		
						0	E			
					1	6				
				0	A					
			5	B	7	3	0			
+				^{1}						
			5	C	2	9	E	6	0	$_{16}$

e)

	1	0	1	1	1	0	1	0	1	$_{2}$	·	1	0	0	0	1	0	$_{2}$
								1	0	1	1	1	0	1	0	1	0	
				1	0	1	1	1	0	1	0	1	0	0	0			
+					^{1}	^{1}	^{1}		^{1}	^{1}	^{1}							
				1	1	0	0	0	1	1	0	0	0	1	0	1	0	$_{2}$

f)

5	F	0	2	C	$_{16}$	·	7	1	D	$_{16}$
								9	C	
							1	A		
						0	0			
					C	3				
				4	1					
			+				1			
				4	D	3	2	3	C	
				5	F	0	2	C		
						5	4			
					0	E				
				0	0					
			6	9						
		2	3							
	+				1					
		2	9	9	1	3	4			
+			1	1						
	2	A	3	D	6	8	F	C	$_{16}$	

g)

f	e	e	d	$_{16}$	·	c	0	f	4	$_{16}$
								3	4	
							3	8		
						3	8			
					+	3	C			
						3	F	B	B	4
								C	3	
							D	2		
						D	2			
				+	E	1				
					E	E	F	E	3	
						9	C	0		
					A	8				
			A	8						
		+	B	4						
		B	F	3	1	C				
+		1	1	1	2	1				
	C	0	2	4	B	9	E	4	$_{16}$	

h)

1	1	1	0	0	1	0	1	0	0	1	1	0	$_2$	·	1	0	0	1	1	0	1	1	1	$_2$
											1	1	1	0	0	1	0	1	0	0	1	1	0	
										1	1	1	0	0	1	0	1	0	0	1	1	0		
									1	1	1	0	0	1	0	1	0	0	1	1	0			
								1	1	1	0	0	1	0	1	0	0	1	1	0				
							1	1	1	0	0	1	0	1	0	0	1	1	0					
					1	1	1	0	0	1	0	1	0	0	1	1	0							
+			1	1	1	1	10	10	10	11	11	10	10	10	1	10	10	1	1	1	1			
		1	0	0	0	1	0	1	1	0	0	1	1	0	1	1	0	1	0	1	0	1	0	$_2$

i)

C	B	F	6	8	F	E	$_{16}$	·	E	2	D	5	9	3	C	$_{16}$
								9	8	F	8	E	B	E	8	
							2	6	3	E	3	A	F	A		
						7	2	B	A	B	0	E	E			
					3	F	B	D	0	C	F	6				
				A	5	B	8	5	4	E	6					
			1	9	7	E	D	1	F	C						
		B	2	7	7	B	D	E	4							
+			1	1	3	3	3	3	5	2	2	2	1			
	B	4	B	9	D	4	E	1	0	2	E	9	8	8	$_{16}$	

(7)

a) **C794F3**

b) **54DD0B**

c) **1000 1100 0101 0001 1111**

d) $1 * 2^0 + 1 * 2^1 + 1 * 2^2 + 0 * 2^3 + 1 * 2^4 + 0 * 2^5 + 0 * 2^6 + 1 * 2^7 + 0 * 2^8 + 1 * 2^9 =$
$= 2^0 + 2^1 + 2^2 + 2^4 + 2^7 + 2^9 = 1 + 2 + 4 + 16 + 128 + 512 =$ **663**

e) $5 * 6^0 + 2 * 6^1 + 4 * 6^2 + 1 * 6^3 = 5 + 2 * 6 + 4 * 36 + 1 * 216 =$
$= 5 + 12 + 144 + 216 =$ **377**

f) $2573 : 16 = 160$ Rest $13 \triangleq D_{16}$
$160 : 16 = 10$ Rest $0 \triangleq 0_{16}$
$10 : 16 = 0$ Rest $10 \triangleq A_{16}$
$\Rightarrow$ **A0D**

g) $18293 : 5 = 3658$, Rest 3. $\rightarrow 3658 : 5 = 731$, Rest 3. $\rightarrow 731 : 5 = 146$, Rest 1. $\rightarrow$
$\rightarrow 146 : 5 = 29$, Rest 1. $\rightarrow 29 : 5 = 5$, Rest 4. $\rightarrow 5 : 5 = 1$, Rest 0. $\rightarrow 1 : 5 = 0$, Rest 1.
$\Rightarrow$ **1041133**

h) $2120112_3 = 2_3 * 10_3^0 + 1_3 * 10_3^1 + 1_3 * 10_3^2 + 0_3 * 10_3^3 + 2_3 * 10_3^4 + 1_3 * 10_3^5 + 2_3 * 10_3^6 =$
$= 2_7 * 3_7^0 + 1_7 * 3_7^1 + 1_7 * 3_7^2 + 0_7 * 3_7^3 + 2_7 * 3_7^4 + 1_7 * 3_7^5 + 2_7 * 3_7^6 =$
$= 2_7 + 3_7 + 3_7^2 + 2_7 * 3_7^4 + 3_7^5 + 2_7 * 3_7^6 =$
$= 2_7 + 3_7 + 12_7 + 2_7 * 144_7 + 465_7 + 2_7 * 2061_7 =$
$= 2_7 + 3_7 + 12_7 + 321_7 + 465_7 + 4152_7 = 5321_7 \Rightarrow$ **5321**

i) Zur Basis 6: $0 * 10^0 + 2 * 10^1 + 2 * 10^2 + 1 * 10^3 + 2 * 10^4 + 5 * 10^5 + 4 * 10^6$

10 ins Zielsystem umwandeln (via Dezimalsystem): $10_6 = 6_{10} = 2 * 3^1 + 0 * 3^0 = 20_3$

Zur Basis 3: $0 * 20^0 + 2 * 20^1 + 2 * 20^2 + 1 * 20^3 + 2 * 20^4 + 12 * 20^5 + 11 * 20^6$

Schriftlich Potenzen (z. B. 20^5), dann Teilsummen ($12 * 20^5$) berechnen und aufaddieren:

$2 * 20 + 2 * 1100 + 22000 + 2 * 1210000 + 12 * 101200000 + 11 * 2101000000 =$
$= 110 + 2200 + 22000 + 10120000 + 1222100000 + 100111000000 =$ **102121022010**

j) Darstellung zur Basis 7: $3 * 10^0 + 1 * 10^1 + 5 * 10^2 + 0 * 10^3 + 4 * 10^4$

10 ins Zielsystem umwandeln (über das Dezimalsystem): $10_7 = 7_{10} = 7 * 12^0 = 7_{12}$

Darstellung zur Basis 12: $3 * 7^0 + 1 * 7^1 + 5 * 7^2 + 0 * 7^3 + 4 * 7^4$

Schriftlich Potenzen (z. B. $7^2 = 41$), dann Teilsummen berechnen und aufaddieren:

$3 * 1 + 1 * 7 + 5 * 41 + 4 * 1481 = 3 + 7 + 185 + 5684 =$ **5857**

⑧

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

Erklärungen (chronologisch)**:**

- Zunächst werden die Variablen `b1`, `b5`, `b10` und `b12` lediglich deklariert. Die Variablen erhalten keinen Wert, da es sich nicht um Attribute handeln kann (weil wir uns in einer Funktion bzw. Methode befinden müssen, schließlich sind außerhalb von Methoden bspw. keine Zuweisungen möglich wie sie in den nächsten Zeilen folgen (außer Initialisierungen)).
- Der Zuweisungsoperator ist rechtsassozitativ, d. h. es wird zunächst `b5` auf `true` gesetzt und der zugewiesene Wert anschließend auch in `b1` gespeichert, d. h. beide sind `true`.
  Analog dazu werden `b10` und `b12` mit `false` initialisiert.
- `b2` wird mit `false` initialisiert. `b7` ebenso.
- `b1` ist `true`, `b2` ist `false`, daher wertet der Ausdruck wie folgt aus und `b3` ist somit `true`: `!(!b1 && b2)` → `!(!true && false)` → `!(false && false)` → `!(false)` → `true`
- `b2` ist `false`, weshalb auch die rechte Seite des Ausdrucks `b2 || (b1 = false)` ausgewertet werden muss (`||` würde abbrechen und zu `true` auswerten, falls die linke Seite `true` wäre; hier nicht der Fall). Bei dem Ausdruck `(b1 = false)` handelt es sich nicht etwa um einen Vergleich (==), sondern um eine Zuweisung (=), daher bekommt `b1` den Wert `false` zugewiesen und der Ausdruck wertet zum zugewiesenen Wert - also `false` - aus. Der Gesamtausdruck ist folglich `false||false` → `false`, weshalb `b4` den Wert `false` annimmt.
- `b2` ist `false`, weshalb die linke Seite des Ausdrucks `b2 && (b2 = true)` zu `false` auswertet. Da es sich bei `&&` (genau wie bei `||`) um einen Kurzschlussoperator handelt, wird die rechte Seite nicht ausgewertet. Die linke Seite ist bereits `false`, also spielt es keine Rolle, wozu die rechte Seite auswerten würde (weil *beides* `true` sein müsste, damit der Gesamtausdruck `true` ist). Der Gesamtausdruck ist somit `false`, was in `b6` gespeichert wird.
- Der Ausdruck `(b7 = false) || true` wird von links nach rechts ausgewertet. Im linken Teil wird die Variable `b7` auf `false` gesetzt (wieder: hier wird nicht verglichen (==) sondern zugewiesen (=)!) und zum zugewiesenen Wert, also `false`, ausgewertet, was in `b8` gespeichert wird, da `false || true` wiederum `false` ergibt.
- Die *`if`*-Bedingung `b5 = false` ist kein Vergleich, sondern eine Zuweisung, d. h. `b5` wird auf `false` gesetzt, der Gesamtausdruck ist `false` und der *`if`*-Block wird nicht ausgeführt.
- `b9` wird anfangs auf `true` gesetzt. In der *`if`*-Bedingung wird `b10` auf `true` gesetzt, die Zuweisung wertet selbst wiederum zum zugewiesenen Wert, also `true`, aus, weshalb der `if`-Fall eintritt und `b9` mit `false` überschrieben wird.
- Der ternäre Operator prüft, ob `b12` gilt. `b12` wurde nie verändert, ist also `false`, weshalb die Bedingung falsch ist und der ternäre Operator zum *`else`*-Teil (also nach dem Doppelpunkt) auswertet. Auch hier handelt es sich um eine Kurzschlussauswertung, weshalb der *`if`*-Teil (zwischen Fragezeichen und Doppelpunkt) nicht ausgeführt wird. `b12` behält daher seinen Wert und `b11` wird auf `true` gesetzt.

⑨

`a = 1`	`j = 3L`	`s = "Afalse"`
`b` nicht initialisiert!	`k = 0`	`t = true`
`c = 'B'`	`l = 2`	`u = null`
`d = 1`	`m = 3.0`	`v = "Anull"`
`e = false`	`n = 2L`	`w = 0L`
`f = "345"`	`O = 7`	`x = 1.0`
`g = true`	`p = 9`	`y` nicht initialisiert!
`h = 65`	`q = 67`	`z = 3.0`
`i = 4.0`	`r = 3`	`A = 'A'`

Erklärungen (chronologisch)**:**

- Zunächst werden die statischen Variablen (außerhalb der main-Methode) initialisiert. Dabei erhalten `d` und `k` den Wert `0`, `e` und `t` den Wert `false` und `u` und `f` den Wert `null`.
- Die lokal deklarierten Variablen `a` und `b` werden im Gegensatz zu den Membervariablen nicht automatisch initialisiert und speichern deshalb zunächst keinen Wert.
- Es folgen einige Initialisierungen: `c` ist `'A'`, `l` ist `2`, `r` ist `2`, `O` ist `7`, `w` ist `0`, `A` ist `'c'`.
- Die Variablen `x`, `y` und `z` werden deklariert, wobei lediglich `z` mit `3.0` initialisiert wird. Wollte man alle Variablen initialsieren, so müsste man `double x = ..., y = ...;` schreiben.
- `g` wird mit `false` initialisiert, `s` ergibt sich aus `'A' + ""` zu `"A"`.
- Der Zuweisungsoperator ist rechtsassoziativ, d. h. es wird zunächst `3 / 2` zu `1` ausgewertet, was zunächst in der Variable `d` und anschließend in `a` gespeichert wird. Durch einen impliziten Cast zu `double` erhält `x` den Wert `1.0` zugewiesen.
- `p` ergibt sich aus der Summe der Variablen `O` und `l`, also `7+2` zu `9`.
- `A` erhält den Wert von `c`, also `'A'`, zugewiesen.
- Der Vergleich `r == 2` wertet zu `true` aus. Dieser Wert wird in `t` gespeichert.
- Bei der Zuweisung von `c` (Wert `'A'`) an `h` erfolgt ein impliziter Cast zu `int`. Der Hinweis zu Beginn verrät, dass `'A'` dem Zahlenwert `65` entspricht, was in `h` gespeichert wird. Erst anschließend (`c++` ist ein *Post*inkrement) wird `c` um eins erhöht und ist daher `'B'`.
- Die Division der Variablen `O` und `l` (also `7/2`) ergibt `3` (`int` durch `int` ist `int`), was nach einem impliziten Cast als `3.0` in `i` gespeichert wird. `i++` erhöht den Wert von `i` auf `4.0`.
- Bei `j=O/l` wird wieder `7/2` gerechnet, was `3` ergibt und als `long` in `j` gespeichert wird.

- `q` ergibt sich aus der Summe von `2` und `A`, also `2+'A'`, also `2+65` (impl. Cast) zu `67`.
- `m` speichert `3.0`, denn: `1/2 + 2/1` → `2/2 + 2/1` → `1+2` → `3` (impl. Cast zu `3.0`).
- `v` ist `s + u`, also `"A" + null` und somit `"Anull"`.
- Für die Berechnung `5 % ++r` wird `r` zunächst von `2` auf `3` erhöht. Für die Rechnung wird der neue Wert benutzt (weil *Prä*inkrement). Die Division $5/3$ ergibt 1 Rest 2, d. h. `n` ist `2`.
- `1 + 2 + "" + 4 + 5` wird von links nach rechts ausgewertet: `1+2` ergibt `3`, `3+""` ergibt `"3"`, `"3"+4` ergibt `"34"` und `"34"+5` ergibt schließlich `"345"`, was in `f` gespeichert wird.
- An den String `s` wird `g` angehängt, also `"A" + false` ergibt "Afalse".

  `g` wird negiert (also `!g` in `g` gespeichert): `!g` ist `!false` also `true`, d. h. `g` ist jetzt `true`.
- `t` wird auf das Ergebnis der Veroderung `t || (++i == 0)` gesetzt. `t` speichert zu diesem Zeitpunkt noch den Wert `true`, welcher zuvor durch den Vergleich `r == 2` gesetzt wurde. Damit ist der erste Teil der Bedingung bereits erfüllt und `(++i == 0)` wird nicht ausgewertet, weshalb `i` nicht erneut erhöht wird, sondern den Wert `4.0` behält. `t` bleibt `true`.

⑩

Bei primitiven Datentypen prüft == wie erwartet auf Gleichheit der Werte, daher evaluiert Ausdruck (2) zu `true`. Bei Objekten (kein primitiver Wert) wird auf Referenzgleichheit geprüft. In Ausdruck (1) existieren zwei `Integer`-Objekte mit äquivalentem *Inhalt*, allerdings handelt es sich nicht um ein und dasselbe Objekt, sondern wird der Wert `6` in zwei verschiedene `Integer`-Objekte gepackt. Damit auch (1) funktioniert müsste die `equals`-Methode benutzt werden. Mehr dazu im Kapitel zu Objektorientierung.

⑪

Wichtig ist der Unterschied zwischen `x++` und `++x` (bzw. analog dazu `x--` und `--x`). Wie bereits in der Operatorenübersicht beschrieben wird die Variable `x` mit beiden Operatoren (`x++` und `++x`) um 1 erhöht, allerdings wertet `x++` zum alten Wert (vor der Inkrementierung) und `++x` zum neuen Wert von `x` aus, d. h. bspw. bei a): `a++ - --b` ergibt `1 - 2`, also `-1`.

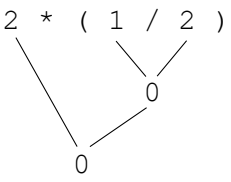
	Ergebnis	a	b
a)	`-1`	2	2
b)	`1`	2	3
c)	`4`	1	3
d)	`"9"`	1	3
e)	`1`	2	3
f)	`4`	2	1
g)	`10`	2	2
h)	`"-22"`	2	3
i)	Compiler-Fehler		
j)	`0`	2	2
k)	`4.5`	2	4
l)	`2`	2	3
m)	`3`	3	3
n)	`false`	3	2
o)	Compiler-Fehler: kann `boolean` nicht in `int` Variable `a` speichern		
p)	`-6`	1	`-6`

	Ergebnis	a	b
q)	`5`	2	5
r)	`2`	3	2
s)	`true`	2	2
t)	`false`	2	2
u)	`"316"`	2	3
v)	`2`	2	2
w)	`false`	2	4
x)	`false`	2	2
y)	`2`	2	3
z)	`true`	2	2
A)	`3`	3	2
B)	`false`	0	2
C)	`1`	1	3
D)	`21`	21	3
E)	`6`	4	2
F)	`true`	2	3

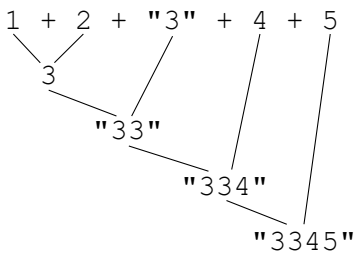
Die Ausdrücke werden immer von links nach rechts ausgewertet (außer Zuweisungen, da muss natürlich zuerst die rechte Seite berechnet werden, bevor man zuweisen kann; Zuweisungsoperatoren und ternärer Operator sind daher nicht links- sondern rechtsassoziativ). Außerdem ist darauf zu achten, dass der `&&`-Operator mit `false` abbricht, falls die linke Seite bereits `false` ist, und der `||`-Operator mit `true` abbricht, falls die linke Seite bereits `true` ist.

12

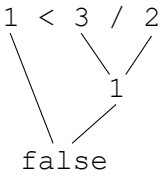
a)


2 * (1 / 2)
0
0

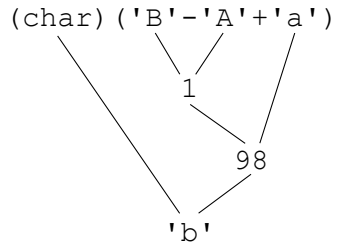
b)


1 + 2 + "3" + 4 + 5
3
"33"
"334"
"3345"

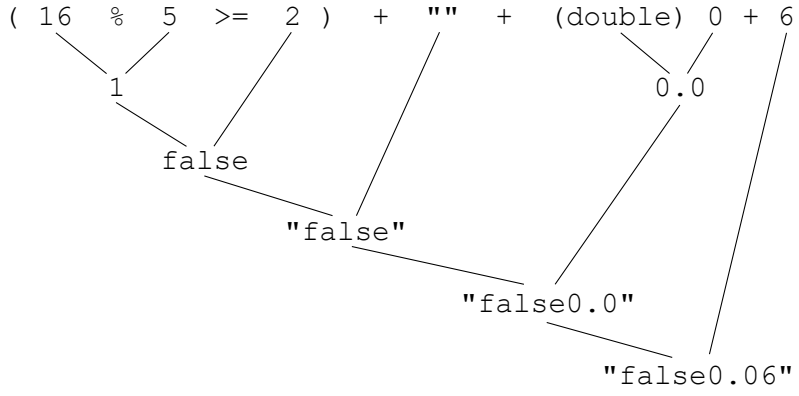
c)


1 < 3 / 2
1
false

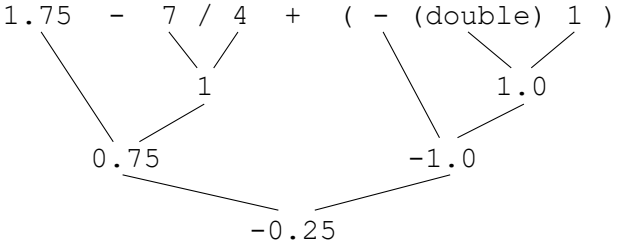
d)


(char)('B'-'A'+'a')
1
98
'b'

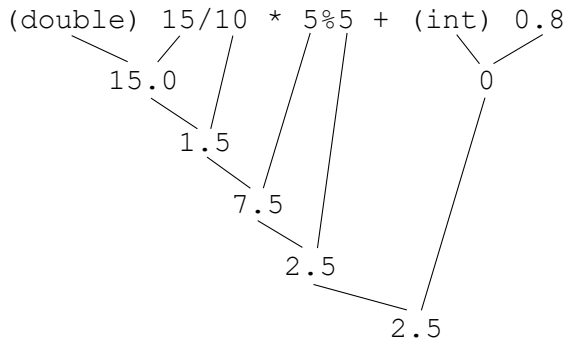
e)


(16 % 5 >= 2) + "" + (double) 0 + 6
1
0.0
false
"false"
"false0.0"
"false0.06"

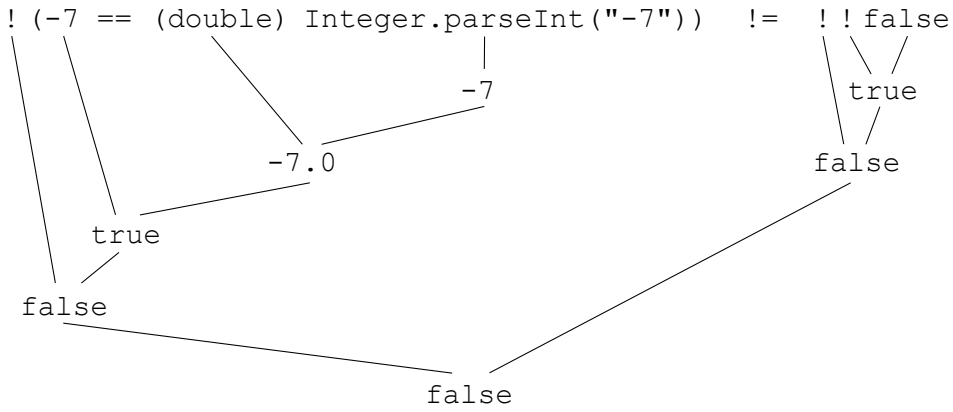
f)


1.75 - 7 / 4 + (- (double) 1)
1
1.0
0.75
-1.0
-0.25

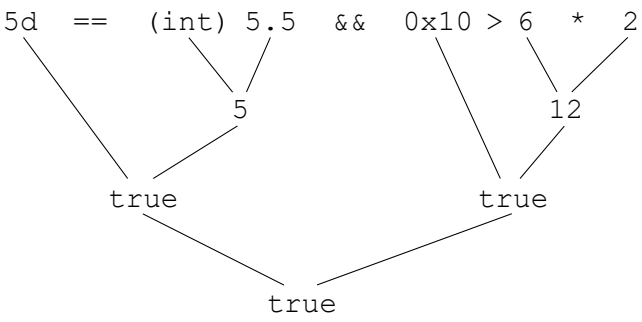
g)



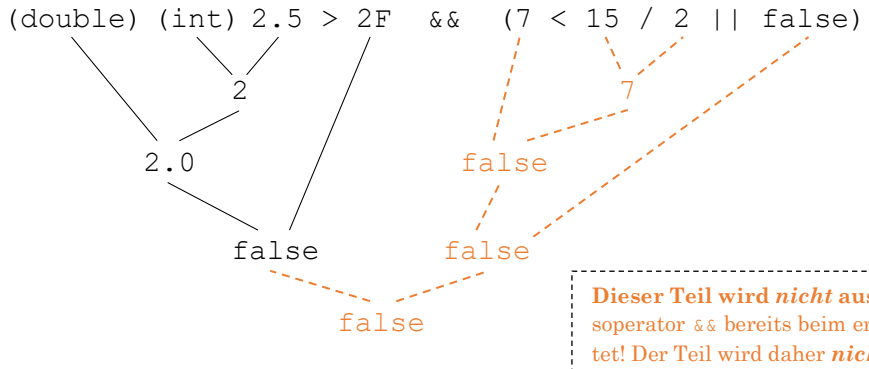
h)



i)



j)



Dieser Teil wird *nicht* ausgewertet, da der Kurzschlussoperator `&&` bereits beim ersten `false` zu `false` auswertet! Der Teil wird daher ***nicht* gezeichnet** und wurde hier nur eingezeichnet, um die eigene Lösung vergleichen zu können. Der Teil würde ausgewertet werden, wenn anstelle des `&&`-Operators („boolesches Und“) der `||`- („boolesches Oder“) oder `&`-Operator („binäres Und“) stünde.

(13)

	Vollständig geklammerter Ausdruck	Ergebnis
a)	`(( 4 - 2 ) + 3 )`	5
b)	`( 5 + (( 5 - 5 ) * 2 ))`	5
c)	`(( 3 * ( 5 - 1 )) + 6 )`	18
d)	`(( 2 * ( 7 % 5 )) / 2 )`	2
e)	`(( 4 + "" ) + ( 3 * 2 ))`	"46"
f)	`((( 1 + "2" ) + 6 ) / 2 )`	Fehler
g)	`((( 4 - 3 ) / 2 ) + 1 )`	1
h)	`( "6" + (( 3 - 2 ) * 3 ))`	"63"
i)	`(( 2 * 3 ) / ( 5 % 3 ))`	3
j)	`(( 3 / ( 3 - 1 )) + 1 )`	2
k)	`(( 6d + ( 0x1 % 4 )) / 2 )`	3.5
l)	`(( 3d / 2 ) + "" )`	"1.5"
m)	`( "" + ( .5 * ( 3 - 1 )))`	"1.0"
n)	`(( 1e2 * ( 10 - 2 )) + "" )`	"800.0"
o)	`(( "" + ( 4 - 2 )) / 1 )`	Fehler
p)	`( 0x0A - ( 0b10 % 3 ))`	8

14

```
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;
```

Der „`else`“- bzw. „`else if`“-Teil gehört immer zum innersten „`if`“. Es existiert hier daher kein „`else`“ oder „`else if`“ zum äußersten „`if`“.

15

	f(2)	**f(5)**
a)	5	30
b)	-2	-3
c)	1	2
d)	"xx"	"xxxxx"
e)	2	-2
f)	-2	-3
g)	"2.0"	"2.25"
h)	false	true
i)	"220"	"553-1"
j)	3	6
k)	3	Stack Overflow
l)	-1	2
m)	5	Endlosschleife

16

a) Wahr.

b) Falsch. Die Größe des Arrays ist 9, der Index des letzten Elements ist 8.

c) Wahr. Die *Elemente* eines Arrays werden bei der Erzeugung eines Array (z. B. `new int[5]`) aber immer automatisch initialisiert.

d) Falsch. Eine lokale Variable, die ein Array speichert, wird nicht automatisch initialisiert, eine globale Variable (Member) würde hingegen den Wert `null` speichern.

e) Falsch. Auf Arrays werden wir nie Methoden benutzen. Der Zugriff auf die Länge eines Arrays erfolgt mit `arr.length` (keine Klammern, weil keine Methode).

f) `a:` Wahr. Erzeugt das Array `[0,0,0,0,0]`.

`b:` Falsch. Implizite Casts funktionieren bei Arrays nicht. Explizite Casts sind bei Arrays über primitiven Werten ebenfalls nicht möglich, d. h. auch

```
int[] b = (int[]) new short[5];
```

wäre nicht möglich.

Aber: Explizite Casts bei Arrays über Referenzdatentypen sind möglich! So wäre bspw. folgendes möglich:

```
String[] arr = (String[]) new Object[5];
```

Man müsste dann aber sicherstellen, dass alle Objekte des gecasteten Arrays auch tatsächlich Strings sind (hier der Fall, da das Array nicht gespeichert, sondern ohne weitere Initialisierungen direkt gecastet wird, d. h. jeder Wert ist `null` und damit passend).

Solche Casts kommen praktisch nicht vor. Sie finden aber Anwendung, wenn man ein generisches Array erzeugen will. Generics behandeln wir erst später. Vorgriff: Möchte man ein Array eines generischen Typs erzeugen (z. B. `T[]`), so würde das z. B. mit

```
T[] genArr = (T[]) new Object[5];
```

funktionieren.

`c:` Falsch.

`d:` Wahr. Erzeugt das Array `[1,2,3,4,5]`, die Größe steht also durch die Anzahl der Elemente fest.

`e:` Wahr. Erzeugt das Array `[1,2,3]`. Vorsicht: Derartige Array-Erzeugungen funktionieren im Gegensatz zu `d` nur direkt neben der Deklaration der Variable, da sonst der Array-Typ nicht eindeutig ist.

`f:` Wahr.

`g:` Falsch. Entweder Größe oder Elemente angeben, beides geht nicht.

`h:` Wahr. Erzeugt das Array `[]` (keine Elemente, entspricht `new String[0]`).

`i:` Falsch. Alle Elemente des Arrays müssen den angegebenen Typ (hier `int[]`) haben. `'c'` und `'d'` haben den Typ `char` bzw. `int`, aber nicht `int[]`.

`j:` Wahr. Es wird ein zweidimensionales `int`-Array erzeugt, also ein Array, das Array von `int`s enthält. Die enthaltenen Arrays können unterschiedliche Längen haben, hier: `[[1,2],[0,0],[]]`

`k:` Falsch. `double[]` fehlt (`new double[] {1.0, 0.2}`)

`l:` Wahr. Es erfolgt ein automatischer Upcast von `char` zu `double`, d. h. es wird das einelementige Array `[65.0]` erzeugt.

`m:` Falsch. Falsches Trennzeichen (`;` statt `,`).

`o:` Falsch. Solche Initialisierungen sind nur unmittelbar nach der Deklaration möglich (vgl. `e`).

`p:` `q:` Wahr. Es werden zwei Variablen vom Typ `int`-Array deklariert. Anschließend erfolgt eine Zuweisung des Arrays `[1,66,3]` an `q`, wobei *dasselbe* Array auch in `p` gespeichert wird. Hier würde also `p == q` gelten. Es existieren zwei Variablen, welche auf dasselbe Array-Objekt zeigen.

Normalerweise müssen wir die Arrays elementweise vergleichen. Dazu können wir die Methode `Arrays.equals(arr1, arr2)` nutzen. Wie bereits in *e)* gesagt, benutzen wir auf Arrays keine Methode, daher greifen wir hier auf `java.util.Arrays` zurück.

`r:` Falsch. `r` ist als `int` deklariert (nicht als `int`-Array).

`s:` Falsch. Auch Auto-Boxing/Unboxing funktioniert bei Array-Typen nicht (vgl. `b`).

`t:` Wahr. Die *Elemente* können weiterhin automatisch gewrappt werden.

`u:` Falsch.

`v:` Wahr. `Number` ist die Oberklasse von `Integer`, `Double` usw. Und: Vererbung funktioniert bei Arrays! Auch Folgendes würde funktionieren:

```
Object[] arr = new String[] {"hi"};
```

`x:` Wahr. vgl. `t`

`y:` Falsch. Autoboxing kann `1` nur in einen `Integer` wrappen und `Integer` ist keine Unterklasse von `Double` (vgl. Aufgaben zu Wrapper-Klassen).

g) Wahr. Das ist bei Arrays genauso wie bei allen anderen Referenzdatentypen der Fall. Statische Variablen und Attribute werden also immer mit `null` vorinitialisiert. Variablen eines primitiven Datentyps wie `int` oder `boolean` werden mit `0` bzw. `false` vorinitialisiert.

h) Wahr. Die lokale Variable `elem` nimmt alle Werte des Arrays genau einmal an. Die Zuweisung von `char` an `int` ist möglich. Die *for-each*-Schleife entspricht folgender *for*-Schleife:

```
for (int i = 0; i < arr.length; ++i) {
    int elem = arr[i];  // automatischer Upcast von char zu int
    System.out.println(elem);
}
```

i) Falsch. Die Zuweisung von `int` an `char` ist nicht (ohne expliziten Cast) möglich.

j) Falsch. Durch die Neuzuweisung an `e` wird lediglich die lokale Variable `e` verändert. Diese wird in der Schleife dann nicht mehr das Array-Element, sondern `true` speichern. Dadurch wird das Array aber nicht verändert! Die *for-each*-Schleife entspricht folgender *for*-Schleife:

```
for (int i = 0; i < arr.length; i++) {
    boolean e = arr[i];  // kopiert den Wert arr[i] in die Variable e
    e = true;  // arr[i] bleibt unverändert (arr[i] = true wäre nötig)
}
```

k) Wahr. Es muss sich um ein `Object`-Array (`Object[]`) handeln. Nur dieses kann z. B. gleichzeitig `Strings` und `ints` aufnehmen (und auch sonst alles!).

l) Falsch. Wie bei *j)* wird nur die Iterationsvariable `elem` verändert, nicht das Array `arr`.

17

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

18

Das Beispielarray sieht so aus: `double[] t = {1.2, 7.7, 3.1, 10.3, -2.4, 8.1, 0.9};`

a)

```
public static double avgTemp(double[] t) {
    double sum = 0;
    for (double e : t)
        sum += e;
    return sum / t.length;
}
```

Alternativ geht natürlich auch eine normale *for*-Schleife:

```
public static double avgTemp(double[] t) {
    double summe = 0;
    for (int i = 0; i < t. length; i++) {
        summe = summe + t[i];
    }
    double ergebnis = summe / t.length;
    return ergebnis;
}
```

b)

```
public static double minTempNoon(double[] t) {
    double min = Integer.MAX_VALUE; // nicht auf 0 setzen!
    for (int i = 1; i < t.length; i += 2)
        if (t[i] < min)
            min = t[i];
    return min;
}
```

Alternativ kann man bspw. mit `t[1]` starten und/oder mit `i++` und Modulo arbeiten:

```
public static double minTempNoon(double[] t) {
    double min = t[1];
    for (int i = 3; i < t.length; i++) {
        if (i % 2 == 1 && t[i] < min) {
            min = t[i];
        }
    }
    return min;
}
```

c)

```
public static double[] insert(double[] t, double value) {
    double[] tNew = new double[t.length + 1];
    for (int i = 0; i < t.length; i++)
        tNew[i] = t[i];
    tNew[tNew.length-1] = value;
    return tNew;
}
```

Alternativ kann man die folgende Funktion nutzen (und nicht nur hier...): `System.arraycopy(src, srcPos, dest, destPos, length)`

```
public static double[] insert(double[] t, double value) {
    double[] t2 = new double[t.length + 1];
    // Schreibe t.length viele Elemente (also alle) aus Array t
    // beginnend beim Index 0 in das Array t2 beginnend bei Index 0.
    System.arraycopy(t, 0, t2, 0, t.length);
    t2[t.length] = value;
    return t2;
}
```

d)

```
public static int dayWithHighestVariation(double[] t) {
    int day = -1;
    for (int i = 0; i < t.length-1; i += 2) {
        double diff = Math.abs(t[i] - t[i+1]);
        if (day == -1 || diff >= Math.abs(t[day*2] - t[2*day+1]))
            day = i/2 + 1;
    }
    return day;
}
```

Alternativ speichert man sich nicht nur den Tag zwischen (an dem die maximale Differenz gefunden wurde), sondern zusätzlich auch noch die Differenz für diesen Tag (dann muss man es nicht immer neu berechnen). Statt mit `Math.abs()` kann man den Absolutbetrag auch einfach manuell berechnen, indem man einen negativen Wert invertiert. Zu einem negativen Wert würde es kommen, wenn der Mittag-Wert kleiner ist als die Morgen-Messung. Der Operator muss >= sein, damit der letzte größte Wert genommen wird, falls es mehrere gibt:

```
public static int dayWithHighestVariation(double[] t) {
    double maxDiff = 0;
    int dayWithMaxDiff = 0;

    for (int i = 0; i < t.length-1; i += 2) {
        double diff = t[i+1] - t[i];   // Temperaturschwankung
        if (diff < 0)
            diff = -diff; // ggf. negativen Wert positiv machen

        if (diff >= maxDiff) {
            maxDiff = diff;
            dayWithMaxDiff = i / 2 + 1;
        }

    }

    return dayWithMaxDiff;
}
```

19

Ausgabe	Erklärung
2	Die Variable `a` wird mit `new int[] {2}` initialisiert und referenziert daher ein einelementiges Array (Größe 1) mit dem Inhalt [2]. Die *print*-Funktion ist links definiert und gibt `a[0]` mittels `System.out.println` aus.
3	***set3*** bekommt eine Referenz auf das Array `a` übergeben und speichert diese Referenz wiederum in einer (eigenen) Variable `a` (diese könnte auch anders heißen). Durch die Anweisung `a[0] = 3` wird der Wert `3` in dem von `a` referenzierten Array abgelegt. Nachdem das lokale `a` dasselbe Array referenziert wie das ursprüngliche `a`, wurde damit „implizit" das ursprüngliche `a` verändert.
3	***set5*** bekommt (wie *set3*) eine Referenz auf `a` übergeben und speichert diese in einer eigenen Variable `a`. Anschließend bekommt diese Variable `a` ein neues einelementiges Array mit dem Inhalt `5` zugewiesen. Dadurch wird die ursprüngliche Variable `a` jedoch *nicht verändert* (und insb. auch nicht das Array, welches dadurch referenziert wird). Es existieren zwei Variablen `a`, welche zwei verschiedene Arrays referenzieren und keinen Zusammenhang mehr haben. Grundsätzlich ist es nie möglich, innerhalb von einer Methode Variablen zu verändern, die in anderen Methoden (außerhalb) definiert wurden (Ausnahme: Attribute und Klassenvariablen).
4	Trivial wegen der Zuweisung `a[0] = 4`.
4	***set1*** speichert (wie *set3*) eine Referenz auf das durch `a` referenzierte Array in einer eigenen Parametervariable namens `a`. Ähnlich zu *set5* wird ein neues Array `{1}` erzeugt und gleichzeitig von `b` und `a` referenziert. Wieder wird das ursprüngliche Array `a` dadurch nicht verändert (und die Variable `a` sowieso nicht). Der Aufruf *set3* verändert lediglich das neu erzeugte Array, nicht aber das ursprüngliche `a`, da das lokale `a` ja `b` zugewiesen bekommen hat. Die `return`-Anweisung ist schön, tut aber ebenfalls nichts zur Sache, da das zurückgegebene Objekt nicht benutzt wird.
0	Es wird ein neues Array der Größe 2 erzeugt. Es wurde kein Inhalt, sondern eine Größe angegeben, daher wird das `int`-Array automatisch mit Nullen (`null` bei Objekten, `false` bei booleans) initialisiert. Es gilt `a = {0, 0}`, also `a[0] = 0`.
3	***set1***-Aufruf wie zuvor, mit dem Unterschied, dass der Rückgabewert des Aufrufs an `a` zugewiesen wird. *set1* gibt `b` zurück, wobei `b` das lokal erzeugte Array `{1}` referenziert. Dieses Array wurde durch den Aufruf *set3* innerhalb von *set1* verändert, sodass es das Element `3` speichert. Das ursprüngliche `a` bekommt durch das `return` mit anschließender Zuweisung `a = ...` ebendieses Array zugewiesen und speichert somit eine Referenz auf das einelementige Array `{3}`.

20

```java
public static int[] removeDuplicates(int[] arr) {
    // Zuerst zählen wir, wie groß das Ergebnisarray sein muss:
    int size = 0;
    for (int i = 0; i < arr.length; i++) {
        // Prüfen, ob arr[i] bereits gezählt wurde:
        boolean alreadyCounted = false;
        for (int j = 0; j < i; j++) {  // alle davor liegenden Elemente
            if (arr[j] == arr[i]) // ein gleiches gefunden?
                alreadyCounted = true;

            // Kein else-Fall! Einer der häufigsten Anfängerfehler ist es, in einer
            // Schleife sowohl im if- als auch im else-Fall bspw. ein return oder
            // (wie hier) einen bestimmten boolean zu setzen. Viele würden jetzt im
            // else-Fall gerne alreadyCounted auf false setzen. Tut man das,
            // dann ist das so, als würde man immer nur das allerletzte Element
            // betrachten; schließlich ist es egal, was vor dem letzten Element passiert,
            // wenn dieses den boolean dann sowieso überschreibt... Nur weil das
            // Element arr[i] nicht an der Stelle j steht, heißt das noch lange nicht,
            // dass es nicht vielleicht schon zuvor gefunden wurde. Überlege dir also
            // immer in welche Richtung du pauschalisieren kannst: Hier kann man
            // sagen: Wenn arr[i] an der Stelle j steht, dann wurde es bereits gezählt.
        }

        if (!alreadyCounted)
            size++;
    }

    // Jetzt Array anlegen und befüllen:
    int[] result = new int[size];
    int index = 0;  // result-Array benötigt ebenfalls einen Indexzähler
    for (int i = 0; i < arr.length; i++) {
        boolean alreadyInserted = false;
        for (int j = 0; j < i; j++) {
            if (arr[j] == arr[i]) {
                alreadyInserted = true;
            }
        }

        if (!alreadyInserted) {
            result[index] = arr[i];
            index++;
        }
    }

    return result;
}
```

Alternativer Lösungsvorschlag:

```
public static int[] removeDuplicates(int[] arr) {
    int[] result = new int[0];

    outer: for (int i = 0; i < arr.length; i++) {
        for (int j = 0; j < result.length; j++)
            if (result[j] == arr[i])
                continue outer;

        int[] tmp = new int[result.length + 1];
        System.arraycopy(result, 0, tmp, 0, result.length);
        tmp[result.length] = arr[i];
        result = tmp;
    }

    return result;
}
```

21

```
public static int[] extractMultiplesOf(int[] array, int num) {
    // Entweder wieder erst zählen und dann Array erzeugen bzw. Array
    // immer größer machen (wie in den Lösungsvorschlägen zur vorherigen
    // Aufgabe), oder alternativ Array erstmal zu groß erzeugen ...
    int[] mutiples = new int[array.length];  // evtl. zu groß
    int resultCounter = 0;

    for (int i = 0; i < array.length; i++) {
        if (array[i] % num == 0) {  // array[i] ist Vielfaches von num
            mutiples[resultCounter] = array[i];
            resultCounter++;
        }
    }

    // ... und nach dem Befüllen auf die richtige Länge kürzen
    // (es wurden resultCounter viele Elemente eingefügt):
    int[] multiplesNew = new int[resultCounter];
    for (int i = 0; i < resultCounter; i++)
        multiplesNew[i] = mutiples[i];

    return removeDuplicates(multiplesNew);  // Duplikate löschen
}
```

22

```
static void reverse(Object[] a) {
    for (int i = 0; i < a.length/2; i++) {
        Object tmp = a[i];
        a[i] = a[a.length-1-i];
        a[a.length-1-i] = tmp;
    }
}
```

Wichtig ist, dass du **keine zwei geschachtelten *for*-Schleifen** benutzt. Falls das dein Ansatz war, dann wolltest du wohl eher zwei Zähler, aber nur eine Schleife benutzen:

```
static void reverse(Object[] a) {
    int zaehler1 = 0;
    int zaehler2 = a.length - 1;

    while (zaehler1 < zaehler2) {
        Object tmp = a[zaehler2];
        a[zaehler2] = a[zaehler1];
        a[zaehler1] = tmp;

        zaehler1++;
        zaehler2--;
    }
}
```

23

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

    if(n < 0)   // nicht nötig, aber schön :)
        throw new IllegalArgumentException();

    long[] ret = new long[n+1];
    ret[0] = 1;
    for (int i = 1; i <= n; i++)
        ret[i] = ret[i-1] * x;
        // bzw. alternativ (weniger effizient):
        // ret[i] = (long) Math.pow(x, i);

    return ret;
}
```

(24)

Die geschweiften Klammern der Kontrollstrukturen (`if`/`for`/`while`) können weggelassen werden, da der zugehörige Block jeweils aus nur einem einzigen Statement besteht. Ein `if`-Statement und sein Block ist bspw. ein einzelnes Statement (vgl. Syntaxbäume).

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

    static int indexOf(Object[] arr, Object elem) {
        int i = arr.length;
        while (i-- > 0)
            if (arr[i] == elem)
                return i;
        return -1;
    }

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

        // Alle Elemente (außer elem) in Ergebnisarray kopieren:
        for (int a = arr.length - 1; a >= 0; a--)
            if (arr[a] != elem)
                result[--b] = arr[a];
        return result;
    }
}
```

25

```java
// Diese Methode ist laut Angabe gegeben (liest einen Wert vom Nutzer ein):
public static int read() {
    System.out.print("Bitte Wert eingeben: ");
    return new java.util.Scanner(System.in).nextInt();
}

// Hauptmethode (nutzt die Hilfsmethoden darunter):
public static void userSum() {
    int sum = 0;
    int[] addedValues = new int[0];

    int userInput = -1;  // irgendein Wert ungleich 0 (oder do-while)
    while (userInput != 0) {
        userInput = read();  // read nur einmal in der Schleife

        if (!contains(addedValues, userInput)) {
            sum += userInput;
            addedValues = append(addedValues, userInput);
        }
    }

    System.out.println("Die Summe ist " + sum);
}

// (Man kann es auch ohne Hilfsmethode machen, so wie in den anderen Aufgaben)
/** Gibt true zurück, falls value im Array array enthalten ist, sonst false. */
private static boolean contains(int[] array, int value) {
    for (int i = 0; i < array.length; i++)
        if (array[i] == value)
            return true;

    return false;  // erst wenn nie true zurückgegeben wurde
}

// Da der Nutzer beliebig viele Werte eingegeben können soll, können wir das Array
// nicht mit einer bestimmten Größe vorinitialisieren (wie in den vorherigen
// Aufgaben). Wir müssen dieses Mal mit einem Array der Größe 0 starten und
// immer vergrößern (hier: immer exakt um 1). Hier in eine Methode ausgelagert:
/** Erzeugt ein neues Array bestehend aus array und (zusätzlich) value. */
private static int[] append(int[] array, int value) {
    int[] newArray = new int[array.length + 1];

    // arraycopy kopiert hier array.length viele Elemente aus array
    // (ab Index 0) nach newArray (ebenfalls beginned bei Index 0).
    // Alternativ kann man einfach wieder mit einer for-Schleife arbeiten.
    System.arraycopy(array, 0, newArray, 0, array.length);
    newArray[newArray.length - 1] = value;
    return newArray;
}
```

26

☑	`String a = "Hello World";` ⇒ ganz normaler String `"Hello World"`
☑	`String b = 5 + "5";` ⇒ `int` wird für die Konkatenation zu String → `b` ist `"55"`
☐	`String c = '5' + 5;` ⇒ *char* + *int* = *int*, kann `int` nicht zu `String` machen
☑	`String d = null;` ⇒ `null` ist zwar kein String, aber die Var. `d` speichert nichts (`null`)
☑	`String e = "" + -1 * 5;` ⇒ Opertorpräzedenz beachten (Punkt vor Strich) → `"-5"`
☑	`String f = null + " World";` ⇒ Konkatenation zu `"null World"`
☑	`String g = new String("unknown");` ⇒ Konstruktoraufruf möglich → `"unknown"`
☐	`String h = 'null';` ⇒ `'null'` ist kein gültiger `char` (nur ein Symbol erlaubt)
☐	`String i = (String) 'Test';` ⇒ `'Test'` ist kein gültiger `char`
☑	`String j = '5' + 5.0 + "5" + 5;` ⇒ → `"58.055"`, denn `(int)'5'` ist 53
☐	`String k = 'J';` ⇒ kann `char` nicht implizit zu `String` casten
☐	`String l = (String) 'K';` ⇒ kann `char` auch nicht explizit zu `String` casten
☐	`String m = new String(5);` ⇒ Konstruktor erwartet `String` **oder char-Array**
☑	`String n = new String();` ⇒ leerer String `""`
☑	`String o = new String("Hello") + new String("World");` ⇒ `"HelloWorld"`
☐	`String p = "Beispiel" * 3;` ⇒ `*`-Operator ist auf Strings nicht definiert
☑	`String[] q = new String[2];` ⇒ erzeugt String Array `{null, null}`
☑	`String r[] = null;` ⇒ Variable `r` vom Typ `String[]` referenziert nichts (`null`)
☐	`String[] s = new String[]{1, 2, 3};` ⇒ kann `int` nicht zu `String` casten
☑	`String t = new Integer(5).toString();` ⇒ `Integer` ist Ref.dat.typ → hat `toString`
☐	`class U extends String { }` ⇒ Klasse `String` ist `final` (Wrapper-Klassen auch)
☑	`String v = (String) "V";` ⇒ Cast zu `String` bringt nichts, ist aber möglich
☑	`Object[] w = new Object[]{"A", "B", "A"};` ⇒ jeder `String` ist ein `Object`
☑	`Object x = "XxX";` ⇒ jede Klasse erbt von `Object`, daher kann `x` alles speichern
☑	`String y = "Y" + new Object();` ⇒ Konkat. von String mit Objekt (→ `toString`)
☐	`String z = new Object();` ⇒ kann `Object` nicht implizit zu `String` casten
☑	`Object[] A = {"A".toString(), "B"};` ⇒ wie w)
☐	`boolean B = 'A' == "B";` ⇒ `char` und `String` sind nicht vergleichbar
☑	`String C = (String) new Object();` ⇒ **Laufzeit**fehler (`ClassCastException`)
☑	`String[] D = (String[]) new Object[17];` ⇒ **Laufzeit**fehler (`ClassCastExc.`)
☑	`boolean E = new String("x") == new String("x");` ⇒ ergibt `false` (2 Objekte)
☑	`boolean F = "x" == "x";` ⇒ ergibt `true` (gleiche Referenz wegen String-Pool)
☑	`String G = String.valueOf("rot".toUpperCase() == "ROT");` ⇒ Der Vergleich ergibt `false` (Methodenaufruf erst zur Laufzeit, daher zwei verschiedene String-Objekte im String-Pool). `String.valueOf(...)` wandelt den `boolean` in einen String um, d. h. `G` speichert `"false"`.

27

```java
public static int countVowels(String text) {
    int count = 0;

    for (int i = 0; i < text.length(); i++) {
        // Zeichen an der Stelle i zwischenspeichern (nicht nötig):
        char c = text.charAt(i);

        // Zeichen zunächst kleinschreiben, falls es groß ist. Das ist nicht
        // nötig, erspart uns unterhalb aber einige Vergleiche.
        if (c >= 'A' && c <= 'Z')
            c = (char) (c - 'A' + 'a');  // Groß- zu Kleinbuchstabe

        // Prüfen, ob es sich um einen Vokal handelt (falls du
        // das Großschreiben nicht gemacht hast, musst du auch
        // Großbuchstaben alle einzeln prüfen, z. B. c == 'A'):
        if (c == 'a' || c == 'e' || c == 'i' ||
            c == 'o' || c == 'u' || c == 'y')
            count++;  // Zähler erhöhen, falls Vokal
    }

    return count;
}
```

28

```java
static String invert(String s) {
    String result = "";  // leerer String

    // Iteriere rückwärts über s:
    for (int i = s.length()-1; i >= 0; i--)
        result += s.charAt(i);

    // Alternativ: Vorwärts über s iterieren und
    // stattdessen: result = s.charAt(i) + result

    return result;
}
```

29

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

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

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

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

    return result;
}

public static void main(String[] args) {
    for (int i = 0; i < args.length; i++)
        System.out.println(filterAlphaToUpper(args[i]));

    // oder alternativ mit for-each:
    for (String s : args)
        System.out.println(filterAlphaToUpper(s));
}
```

30

```java
public static boolean contains(String s, String sub) {
    // Wir dürfen nicht zu weit hinten in s suchen, sonst bekommen
    // wir evtl. eine StringIndexOutOfBoundsException:
    int maxIndex = s.length() - sub.length();

    for (int start = 0; start <= maxIndex; start++) {

        // Prüfe, ob sub ab dem Index start in s steht
        boolean found = true;   // Annahme: Ja.
        for (int j = 0; j < sub.length(); j++) {
            if (sub.charAt(j) != s.charAt(start + j))
                found = false;

            // Hier bloß kein else (häufiger Fehler)!
            // Wenn sich ein Zeichen unterscheidet, dann wissen wir, dass der
            // der gesamte String sub nicht an dieser Stelle in s befindet und
            // können found daher auf false setzen. Umgekehrt gilt aber:
            // Nur weil ein einzelnes Zeichen übereinstimmt wissen wir noch
            // lange nicht, ob der gesamte String ab dieser Stelle vorkommt,
            // daher können wir found hier nicht auf true setzen!
        }

        // found nie auf false gesetzt? Dann stimmt alles überein!
        if (found)
            return true;
    }

    // sub an keiner Stelle gefunden, dann:
    return false;
}
```

```java
public static long hexToDez(String hex) {
    hex = hex.toLowerCase(); // kleinschreiben damit z. B. 0X3A == 0x3a

    // Fehlerprüfung: Hexadezimalzahl muss mit 0x beginnen, sonst ungültig:
    if (!hex.startsWith("0x"))
        return -1;

    hex = hex.substring(2);

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

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

    // Führende Nullen entfernen:
    hex = hex.substring(zerosIndex);

    // 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 || hex.length() == 16 && hex.charAt(0) >= '8')
        return -1;

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

    // Iteriere über alle Ziffern der Hex-Zahl, beginnend bei der niederwertigsten:
    for (int i = hex.length() - 1 ; i >= 0 ; 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 = digit - 'a' + 10;
        else
            return -1;

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

    return result;
}
```

Nicht an `toLowerCase` gedacht oder am Anfang dieses Kapitels überlesen? Wenn dir so etwas in der Klausur passiert, lass dich nicht verunsichern, sondern fahre trotzdem mit den anderen Lücken fort.

Alternativ mit `charAt` oder `substring`, dann aber zuvor Längenprüfung nötig:

- `hex.length() < 2 || hex.charAt(0) != '0' || hex.charAt(1) != 'x'`
- `!(hex.length() >= 2 && hex.substring(0, 2).equals("0x"))`

(jeweils entweder mit `||` oder mit `&&` und `!`, aber Länge immer *zuerst* prüfen!)

Optional kann man der `substring`-Methode noch ein oberes Limit übergeben (exklusive): `hex.substring(zerosIndex, hex.length())`

Man muss `'a'` zu 10, `'b'` zu 11, usw. umwandeln, daher addiert man 10 und subtrahiert `'a'` (bzw. 97). Alternativ kann man direkt `digit-87` schreiben.

Fehlerfall: Ungültiges Zeichen

(32)

```java
/**
 * Hilfsmethode für cutOut, die für ein einzelnes Element
 * arbeitet, also alle Vorkommen von <search> aus <s>
 * entfernt und das Ergebnis zurückgibt. Hilfsmethoden
 * schreibe ich nur, um den Überblick zu behalten.
 */
private static String cutOutHelp(String s, String search) {
    // Sonderfall: search ist der leere String
    if (search.equals(""))
        return s;  // keine Änderung
    // else:

    // Baue den Ergebnisstring in neuer Variable auf:
    String result = "";

    // Iteriere über den String:
    int i = 0;
    while (i < s.length()) {
        // Enthält <s> ab dem Index <i> den Suchstring?
        boolean beginsWithSearch;

        // Hat <s> überhaupt noch genügend Zeichen?
        final int remainingChars = s.length() - i;
        if (remainingChars < search.length()){  // Nein
            beginsWithSearch = false;
        } else {
            // Ja, dann schaue, ob <search> hier steht:
            beginsWithSearch = true;  // Annahme
            for (int j = 0; j < search.length(); j++)
                if (search.charAt(j) != s.charAt(i + j))
                    beginsWithSearch = false;
        }

        if (beginsWithSearch) {
            // Ab Index <i> steht der gesuchte String, also
            // überspringe diese Indizes:
            i += search.length();
        } else {
            // Ansonsten übernehme das Zeichen
            result += s.charAt(i);
            i++; // gehe zum nächsten Zeichen
        }
    }

    return result;
}
```

Fortsetzung auf der nächsten Seite...

```java
public static int cutOut(String[] elements, String search) {
    int counter = 0;  // Anzahl veränderter Elemente

    // Iteriere über alle Elemente (String-Array):
    for (int i = 0; i < elements.length; i++) {
        // Betrachte nun das i-te Element (String):
        String cut = cutOutHelp(elements[i], search);

        if (!elements[i].equals(cut)) { // hat sich geändert
            counter++;
            elements[i] = cut;
        }
    }

    return counter;
}

// Test:
public static void main(String[] args) {
    String[] arr = {"ab","aabab","acba", "abababab","a","baba"};
    int changes = cutOut(arr, "ab");
    System.out.println("Veränderte Elemente: " + changes);
    System.out.println("=> " + java.util.Arrays.toString(arr));
}
```

```java
public class BaseConversion {

    /** Wirft Exception, falls Basis im Allgemeinen ungültig ist. */
    private static void checkBase(int base) {
        if (base < 2 || base > 10+26)
            throw new IllegalArgumentException("Basis ungültig: " + base);
    }

    /** Wandelt z. B. 'A' in 10 und '6' in 6 um. */
    private static int digitToInt(char c) {
        if (c >= '0' && c <= '9')
            return c - '0';
        else if (c >= 'a' && c <= 'z')
            return c - 'a' + 10;
        else if (c >= 'A' && c <= 'Z')
            return c - 'A' + 10;
        throw new IllegalArgumentException("Zeichen ungültig: " + c);
    }

    /** Wandelt eine Zahl aus einer beliebigen Basis in eine Zahl um. */
    private static long convertBaseToDecimal(String number, int base) {
        checkBase(base);

        long result = 0;  // Ergebnis im Zehnersystem
        for (int i = 0; i < number.length(); i++) {
            // Zeichen holen und in Zahl umwandeln
            int digit = digitToInt(number.charAt(i));
            if (digit >= base)
                throw new IllegalArgumentException("Ungültige Eingabe" +
                                    " für Basis " + base + ": " + number);
            // Horner-Schema anwenden:
            result *= base;
            result += digit;
        }
        return result;
    }

    /** Wandelt z. B. 10 in 'A' und 6 in '6' um. */
    private static char intToDigit(int d) {
        if (d >= 0 && d <= 9)
            return (char) (d + '0');
        else if (d >= 10 && d <= 36)
            return (char) (d - 10 + 'A');
        throw new IllegalArgumentException("ASCII-Code ungültig: " + d);
    }
```

Fortsetzung auf der nächsten Seite...

```java
/** Wandelt Dezimalzahl in eine Zahl zu einer beliebigen Basis um.*/
private static String convertDecimalToBase(long number, int base) {
    checkBase(base);

    if (base == 10)  // dann reicht auch Folgendes:
        return "" + number;  // Long.toString(number)

    String result = "";
    while (number > 0) {
        int remainder = (int) (number % base);  // Rest (< base)
        number /= base;  // abgerundete Division
        result = intToDigit(remainder) + result;  // vorne anfügen
    }
    return result;
}

public static String convert(String number, int baseA, int baseB) {
    return convertDecimalToBase(
                convertBaseToDecimal(number, baseA),
                baseB);
}

// Rechenoperationen über den Zwischenschritt des 10er-Systems:
public static String add(String[] numbers, int base) {
    long result = 0;
    for (String number : numbers)
        result += convertBaseToDecimal(number, base);
    return convertDecimalToBase(result, base);
}
public static String sub(String[] numbers, int base) {
    long result = convertBaseToDecimal(numbers[0], base);
    for (int i = 1; i < numbers.length; i++)
        result -= convertBaseToDecimal(numbers[i], base);
    return convertDecimalToBase(result, base);
}
public static String mul(String[] numbers, int base) {
    long result = 1;
    for (String number : numbers)
        result *= convertBaseToDecimal(number, base);
    return convertDecimalToBase(result, base);
}
public static String div(String[] numbers, int base) {
    long result = convertBaseToDecimal(numbers[0], base);
    for (int i = 0; i < numbers.length; i++)
        result /= convertBaseToDecimal(numbers[i], base);
    return convertDecimalToBase(result, base);
}
```

Fortsetzung auf der nächsten Seite...

```
    // War nicht verlangt: Test für die Aufgaben 5, 6 und 7
    public static void main(String[] args) {
        System.out.println("5 a) " + add(new String[] {"6352016",
                                                       "436560"}, 7));
        System.out.println("5 b) " + add(new String[] {"3B4CDF031",
                                                       "AFE05DC9"}, 16));
        System.out.println("5 c) " + sub(new String[] {"5AB1C3A4",
                                                       "7BAB492"}, 13));
        System.out.println("5 d) " + sub(new String[] {"1101100011",
                                                       "110010110"}, 2));
        System.out.println("5 e) " + add(new String[] {"2334301",
                                                       "414223",
                                                       "1421240"}, 5));
        System.out.println("5 f) " + add(new String[] {"101110110",
                                                       "11111001",
                                                       "1011010"}, 2));
        System.out.println("5 g) " + sub(new String[] {"11002112",
                                                       "221210",
                                                       "1212101"}, 3));
        System.out.println("5 h) " + sub(new String[] {"8A9C16B3A7",
                                                       "73E4AD0652",
                                                       "E5E5A243"}, 15));
        System.out.println("5 i) " + add(new String[] {"A32A590",
                                                       "4679342",
                                                       "906A81A",
                                                       "A8667"}, 11));
        System.out.println("5 j) " + sub(new String[] {"1110010101",
                                                       "1100110",
                                                       "111101001",
                                                       "11011101"}, 2));

        System.out.println("6 a) " + mul(new String[] {"E5A7", "B"}, 16));
        System.out.println("6 b) " + mul(new String[] {"F01F6E", "C"}, 16));
        System.out.println("6 c) " + mul(new String[] {"123", "90"}, 16));
        System.out.println("6 d) " + mul(new String[] {"5B73", "1020"}, 16));
        System.out.println("6 e) " + mul(new String[] {"101110101",
                                                       "100010"}, 2));
        System.out.println("6 f) " + mul(new String[] {"5F02C", "71D"}, 16));
        System.out.println("6 g) " + mul(new String[] {"feed", "c0f4"}, 16));
        System.out.println("6 h) " + mul(new String[] {"1110010100110",
                                                       "100110111"}, 2));
        System.out.println("6 i) " + mul(new String[] {"CBF68FE",
                                                       "E2D593C"}, 16));

        System.out.println("7 a) " + convert("1100011110010100111110011", 2, 16));
        System.out.println("7 b) " + convert("10101001101110100001011", 2, 16));
        System.out.println("7 c) " + convert("8C51F", 16, 2));
        System.out.println("7 d) " + convert("1010010111", 2, 10));
        System.out.println("7 e) " + convert("1425", 6, 10));
        System.out.println("7 f) " + convert("2573", 10, 16));
        System.out.println("7 g) " + convert("18293", 10, 5));
        System.out.println("7 h) " + convert("2120112", 3, 7));
        System.out.println("7 i) " + convert("4521220", 6, 3));
    }

}
```

34

a)
```
letterUpper  ::= A | ... | Z
letter       ::= a | ... | z | letterUpper
ersterTeil   ::= letterUpper (. letter)? (letter (. letter)?)*
zweiterTeil  ::= (letter (. letter)?)+
benutzername ::= ersterTeil _ zweiterTeil
```

b)
```
pdigit    ::= 1 | ... | 9
digit     ::= 0 | pdigit
vorkomma  ::= 0 | pdigit digit*
nachkomma ::= digit digit
zahl      ::= vorkomma , nachkomma
preis     ::= zahl (€ | Euro)  |  EUR zahl
```

c)
```
pdigit ::= 1 | ... | 9
digit  ::= 0 | pdigit
tag    ::= 0 pdigit | (1|2) digit | 3 (0|1)
monat  ::= 0 pdigit | 1 (0|1|2)
jahr   ::= (digit digit)? digit digit
datum  ::= tag . monat . jahr?
```

d)
```
digit ::= 0 | ... | 9
stundeOhneNull ::= digit | 1 digit | 2 (0|...|3)
stunde ::= stundeOhneNull | 0 digit
minute ::= 0 digit | (1|...|5) digit
name ::= Uhr
uhrzeit ::= stunde (.|:) minute name? | stundeOhneNull name
```

35

a) `(b | ... | y)*`

b) `(letter letter)*`

c) `(b | ... | z) letter* (b | a)  |  b`

d) `a letter* a  |  (b | ... | z) letter*  |  a?`

e) `letter letter (letter letter)?`

f) `(b | ... | z)*  a a  (b | ... | z)*`

g) `a* ((b | ... | z) a?)*`

36

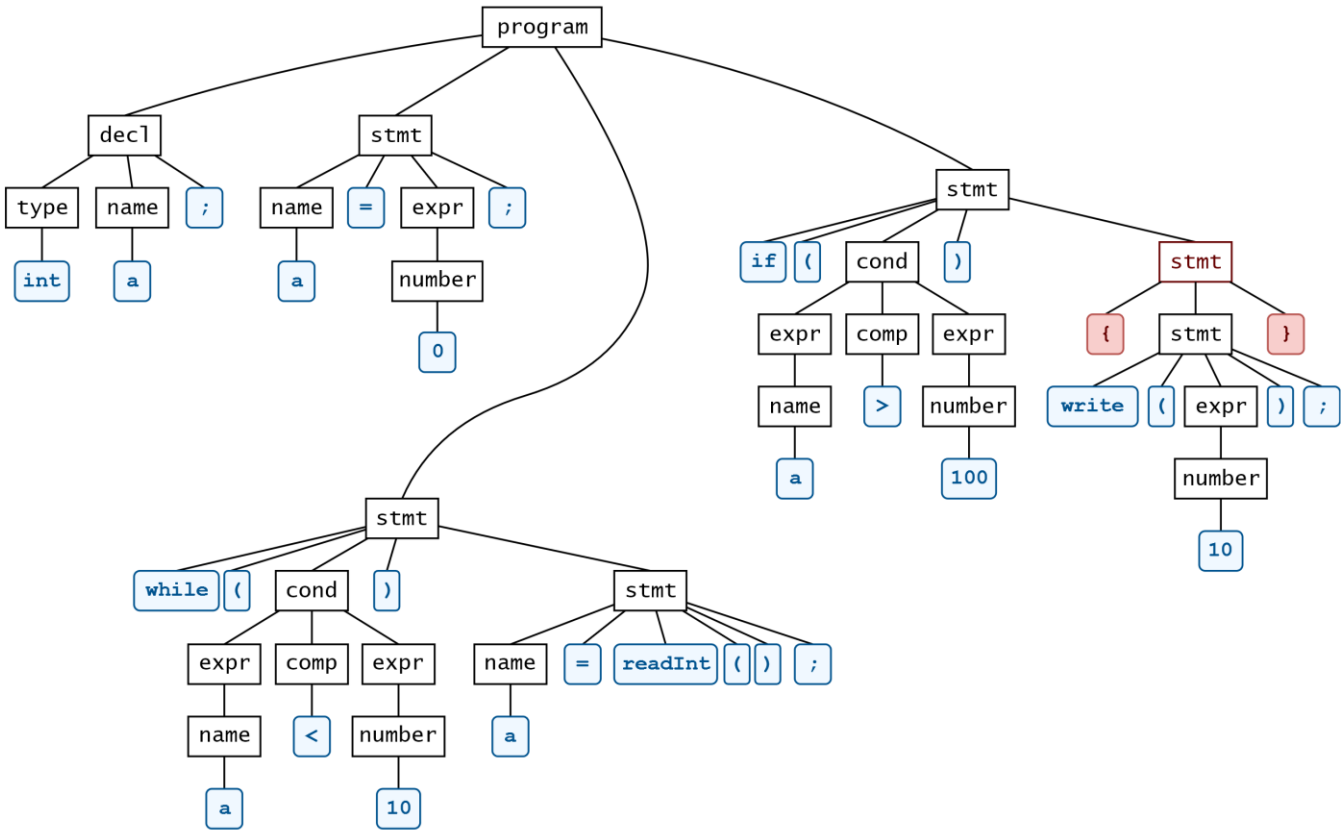
In den folgenden Lösungen wurden Terminalsymbole blau dargestellt.
Damit du deine Fehler schneller findest wurden häufige Fehlerquellen zudem rot markiert.

Tipp: Ich zeiche den Syntaxbaum stets von oben nach unten (wie an den folgenden Lösungvorschlägen erkennbar ist), d. h. bspw.: Ich erkenne ein Statement mit *if*- und *else*-Teile und schreibe wegen der Definition von `stmt` dann direkt die ganze Zeile `if(cond) stmt else stmt` nebeneinander auf. Dann schaue ich mir die einzelnen Teile an, also bspw. wie ich das `stmt` im *if*-Teil weiter unterteilen muss. Das ist meiner Meinung nach einfacher und sicherer als von unten nach oben zu arbeiten, da sich z. B. nie Frage stellt, ob `name` zu `expr` geht oder nicht (weil `name` nie zu `expr` wird, sondern immer nur umgekehrt `expr` zu `name`). Deshalb würde ich empfehlen, dass du dir das Zeichnen von oben nach unten angewöhnst, auch wenn es anfangs ein bisschen mehr Übung erfordert (nach 2-3 Syntaxbäumen geht es dann).

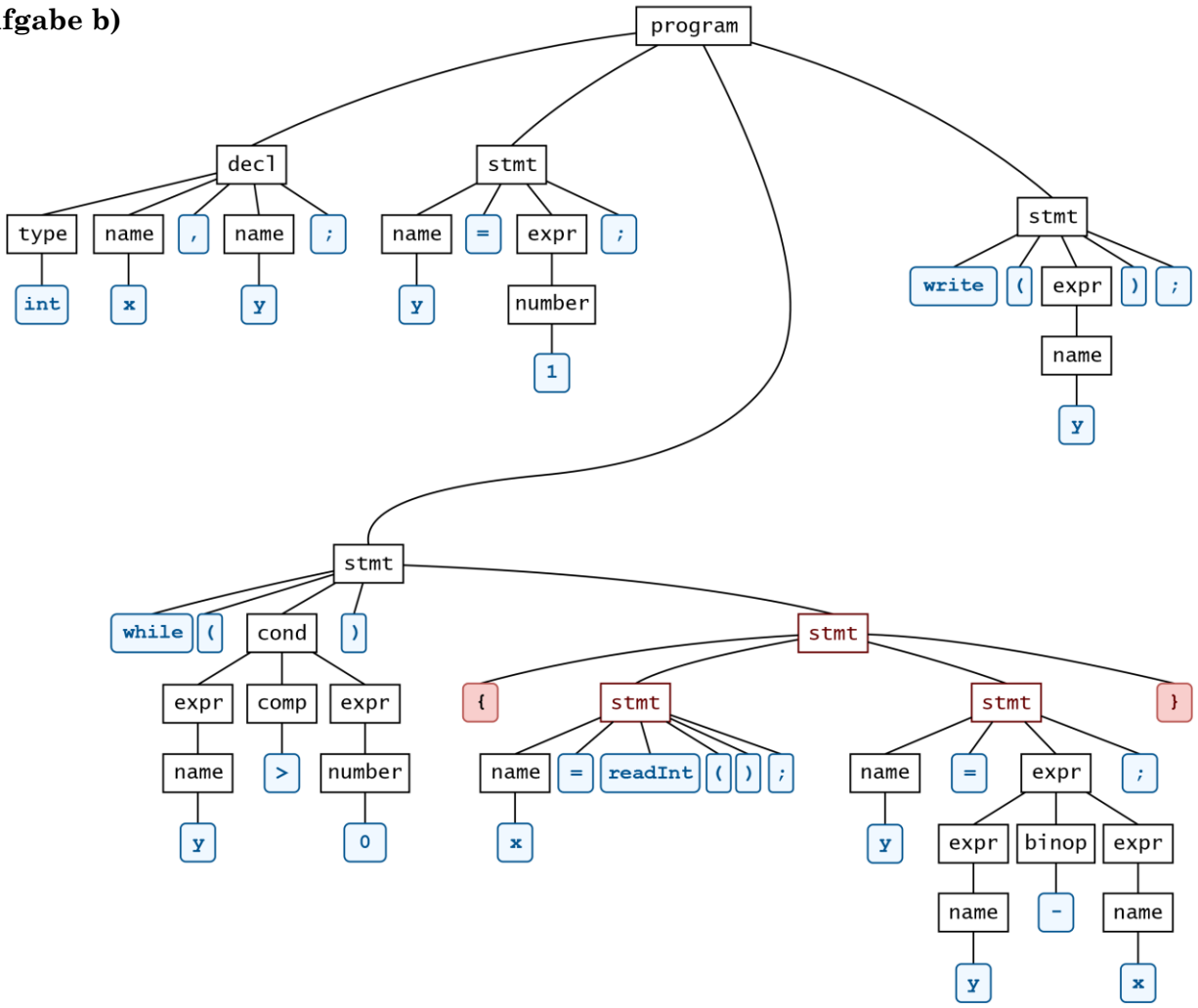
Tipp: Falls du einen Syntaxbaum für das PGdP zeichnen musst, empfehle ich dir draw.io oder den Syntaxbaum-Generator von Ludwig Stecher.

Hinweis: In Klausuren werden aus Leichtsinn häufig Verbindungslinien vergessen. Außerdem sollten sich Kanten niemals überkreuzen, sonst ist irgendwo ein Fehler. Falls dir der Platz ausgehen sollte, markiere eine Kante einfach mit *[1], *[2], ... und zeichne sie anderswo weiter.

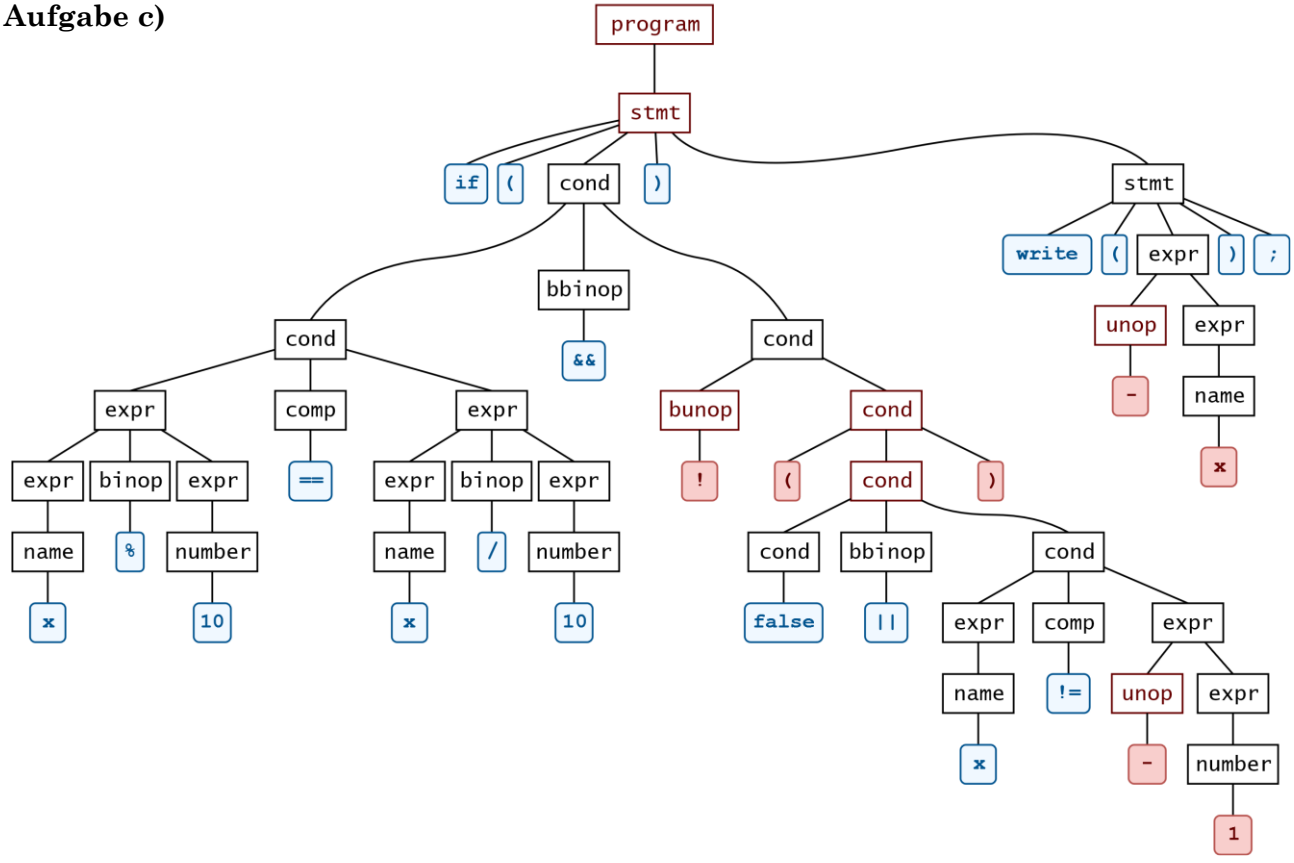
Aufgabe a)



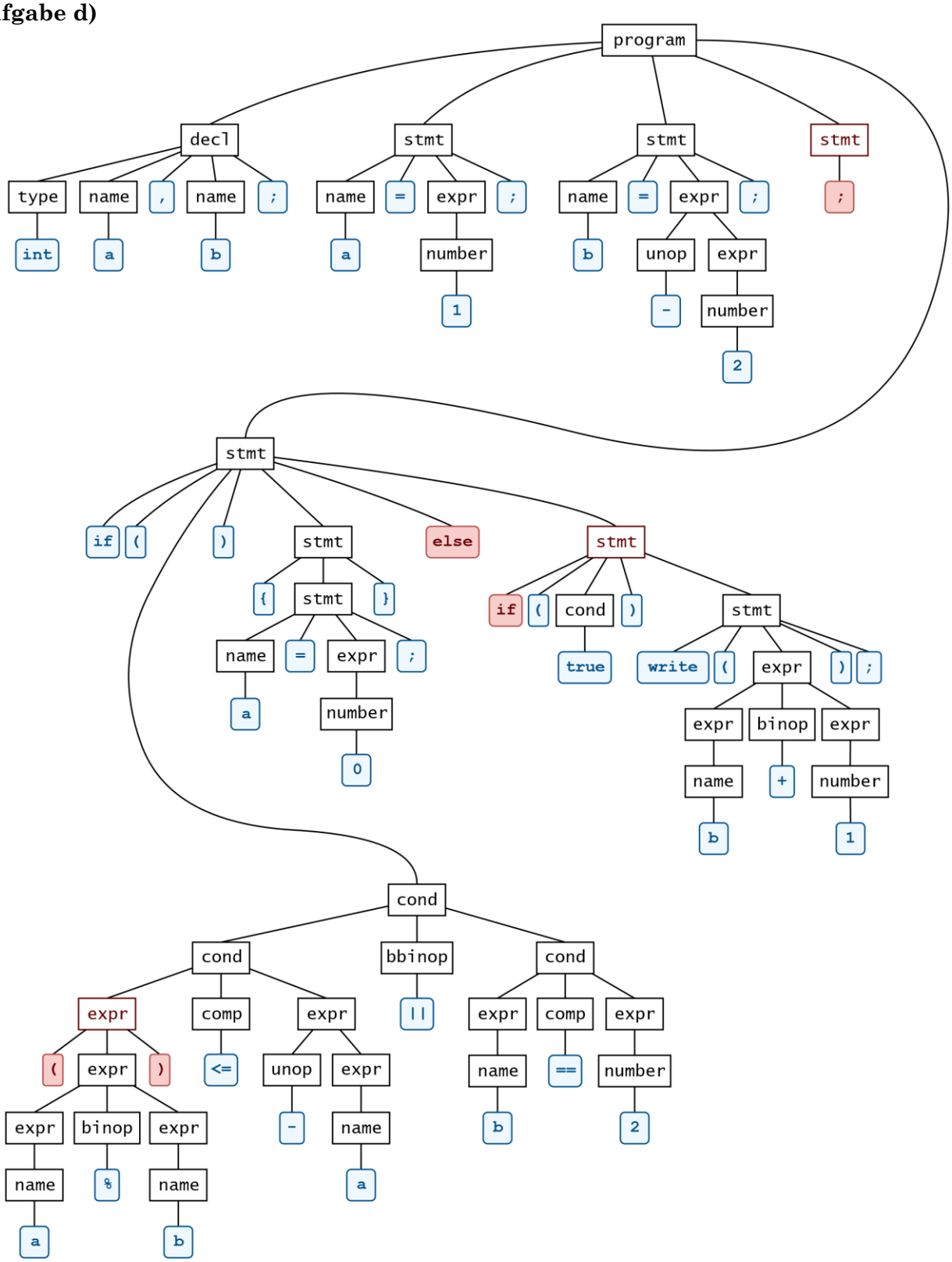
Aufgabe b)



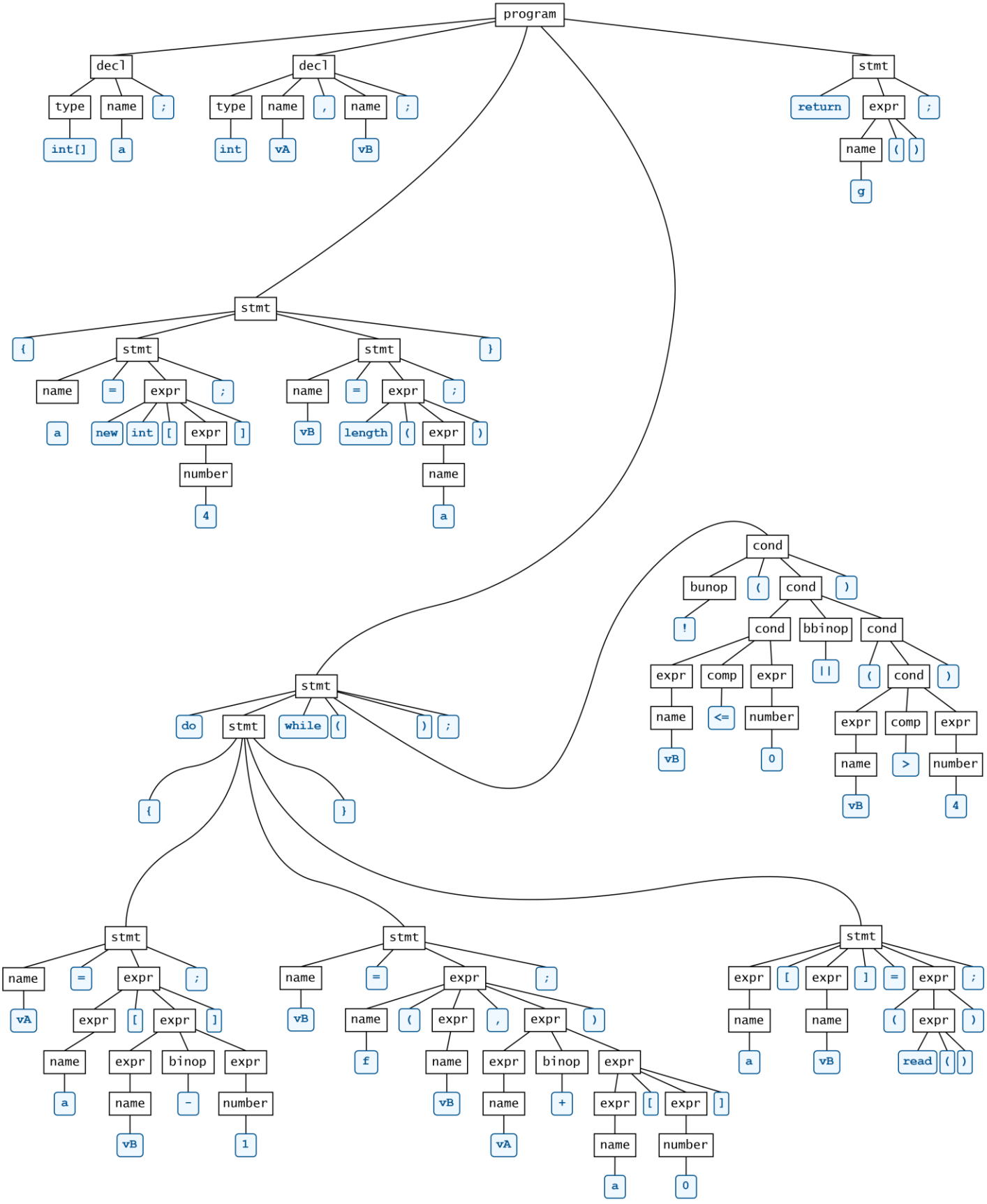
Aufgabe c)



Aufgabe d)

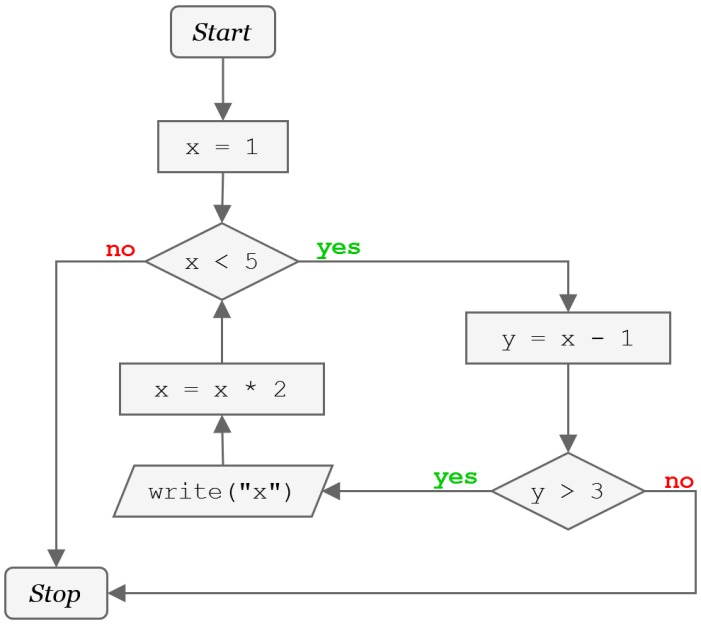

program
decl
type
int
name
a
,
name
b
;
stmt
name
a
=
expr
number
1
;
stmt
name
b
=
expr
unop
-
expr
number
2
;
stmt
;
stmt
if
(
)
stmt
{
stmt
}
name
a
=
expr
number
0
;
else
stmt
if
(
cond
true
)
stmt
write
(
expr
expr
binop
expr
name
b
+
number
1
)
;
cond
cond
expr
(
expr
expr
name
a
binop
%
expr
name
b
)
comp
<=
expr
unop
-
expr
name
a
bbinop
||
cond
expr
name
b
comp
==
expr
number
2

Aufgabe e)

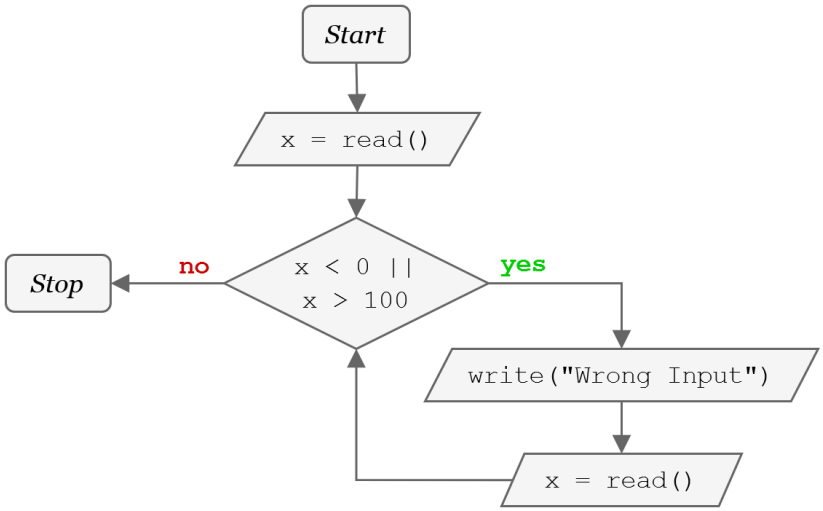

program
decl
type
int[]
name
a
;
decl
type
int
name
vA
,
name
vB
;
stmt
return
expr
name
g
( )
;
stmt
{
stmt
name
a
=
expr
new
int
[
expr
number
4
]
;
stmt
name
vB
=
expr
length
(
expr
name
a
)
;
}
stmt
do
stmt
{
}
while
(
cond
bunop
!
(
cond
)
cond
expr
name
vB
comp
<=
expr
number
0
bbinop
||
cond
(
cond
)
expr
name
vB
comp
>
expr
number
4
;
stmt
name
vA
=
expr
expr
name
a
[
expr
name
vB
binop
-
expr
number
1
]
;
stmt
name
vB
=
expr
name
f
(
expr
name
vB
,
expr
name
vA
binop
+
expr
expr
name
a
[
expr
number
0
]
)
;
stmt
expr
name
a
[
expr
name
vB
]
=
expr
(
expr
read
( )
)
;

37

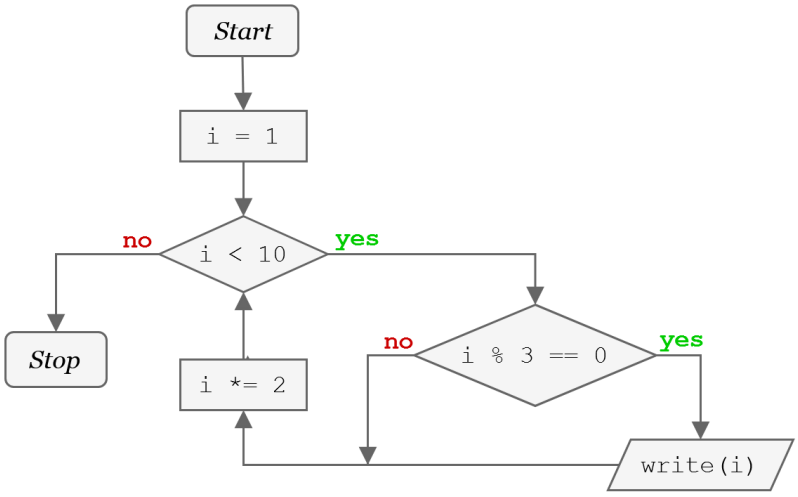
a)


Start
x = 1
x < 5
no
yes
y = x - 1
x = x * 2
write("x")
yes
y > 3
no
Stop

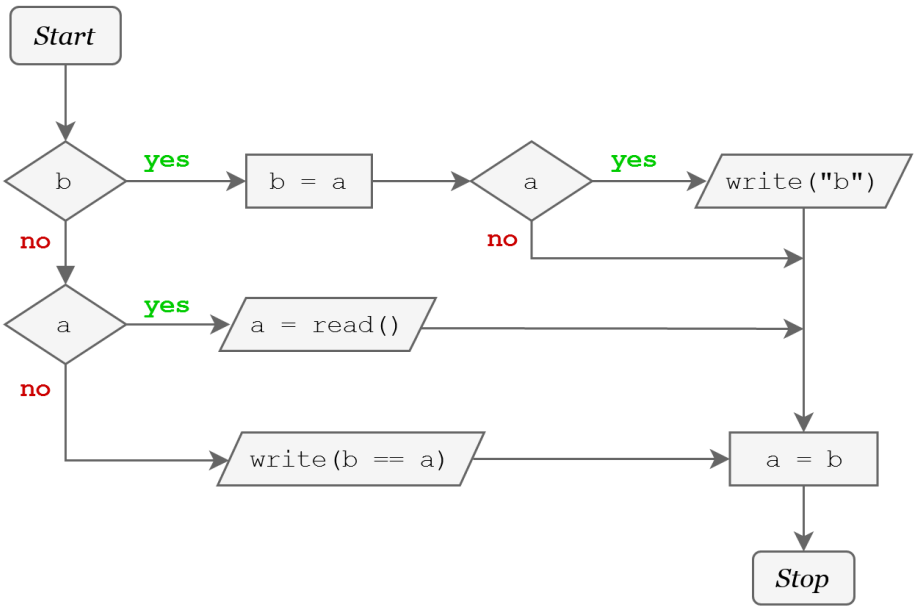
b)


Start
x = read()
Stop
no
x < 0 ||
x > 100
yes
write("Wrong Input")
x = read()

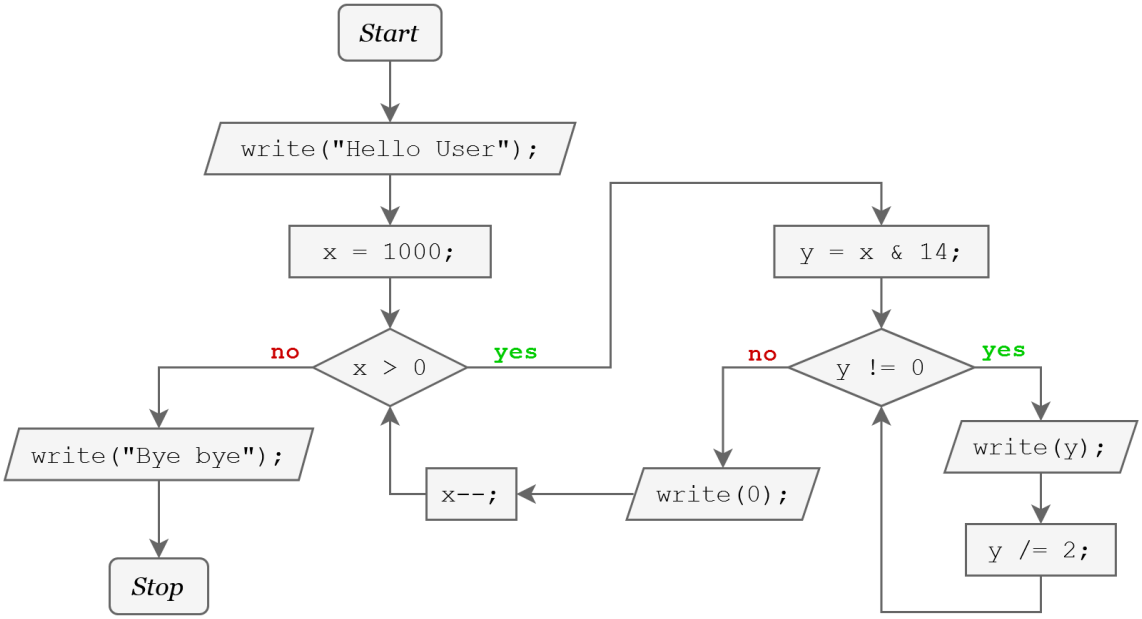
c)


Start
i = 1
no
i < 10
yes
Stop
i *= 2
no
i % 3 == 0
yes
write(i)

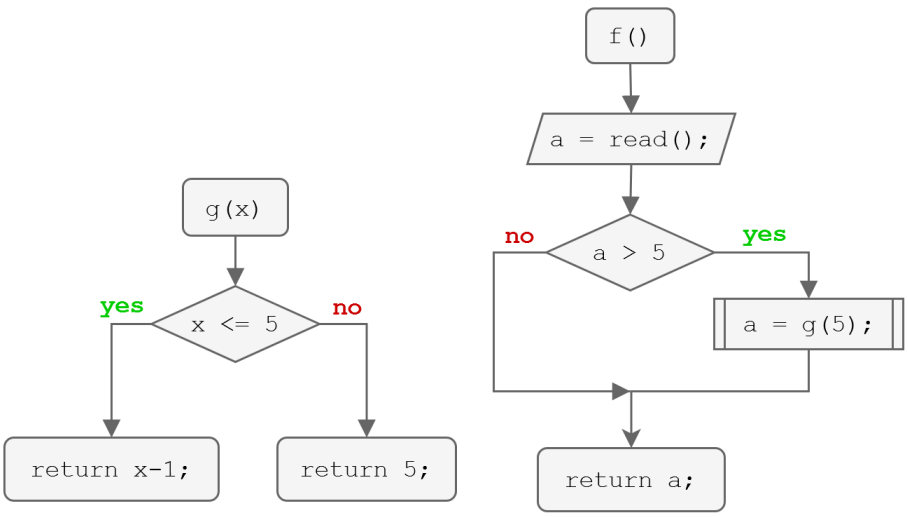
d)


Start
b
yes
b = a
a
yes
write("b")
no
no
a
yes
a = read()
no
write(b == a)
a = b
Stop

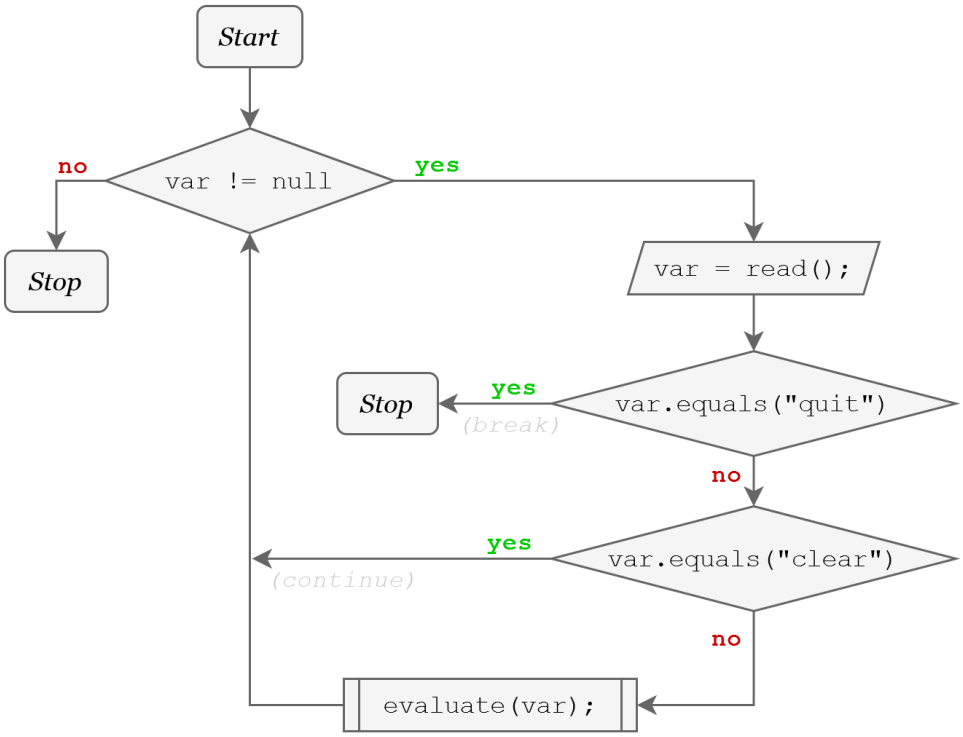
e)


Start
write("Hello User");
x = 1000;
x > 0
no
yes
write("Bye bye");
Stop
x--;
y = x & 14;
y != 0
no
yes
write(0);
write(y);
y /= 2;

f)

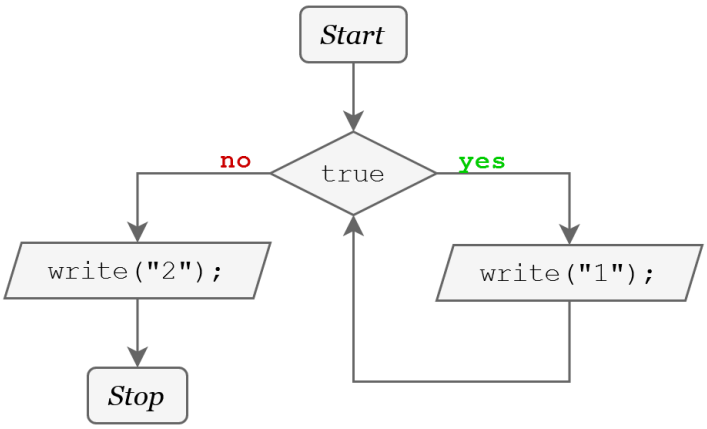

g(x)
x <= 5
yes
no
return x-1;
return 5;
f()
a = read();
a > 5
no
yes
a = g(5);
return a;

g)

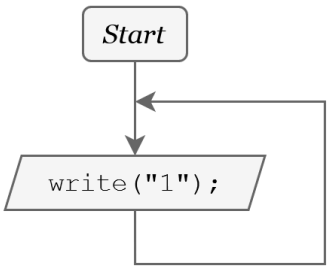


(Bemerkungen in hellgrau sind nicht Teil des Diagramms)

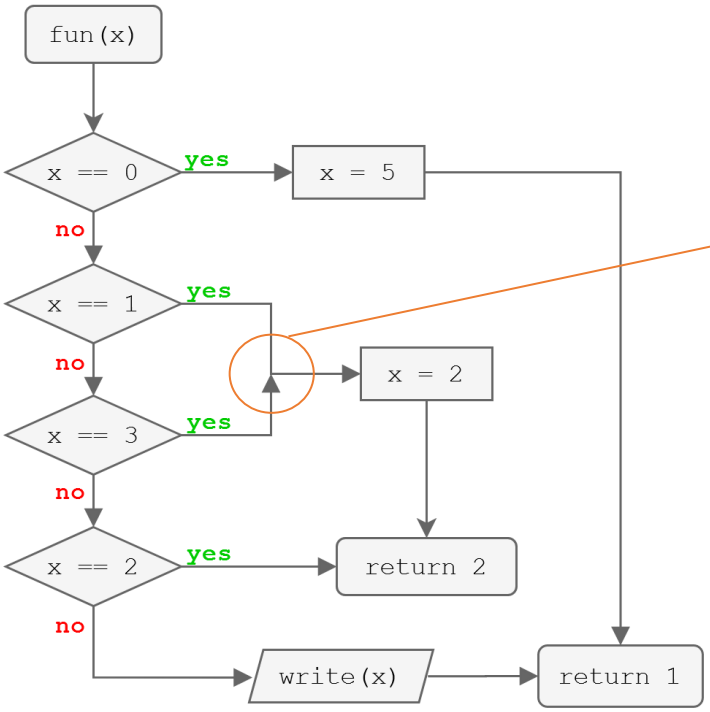
h)



Alternativ: Gekürzter Graph (falls Vereinfachen erlaubt ist):



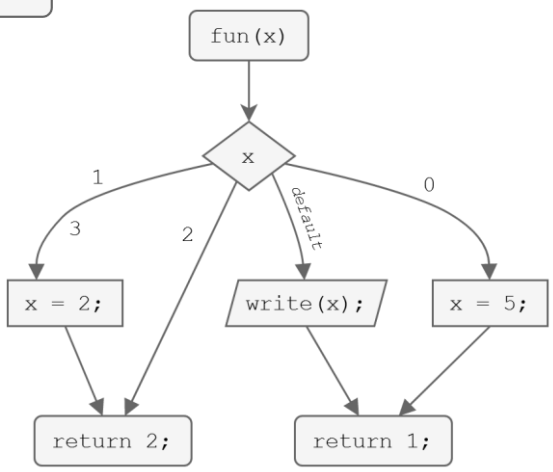
i)



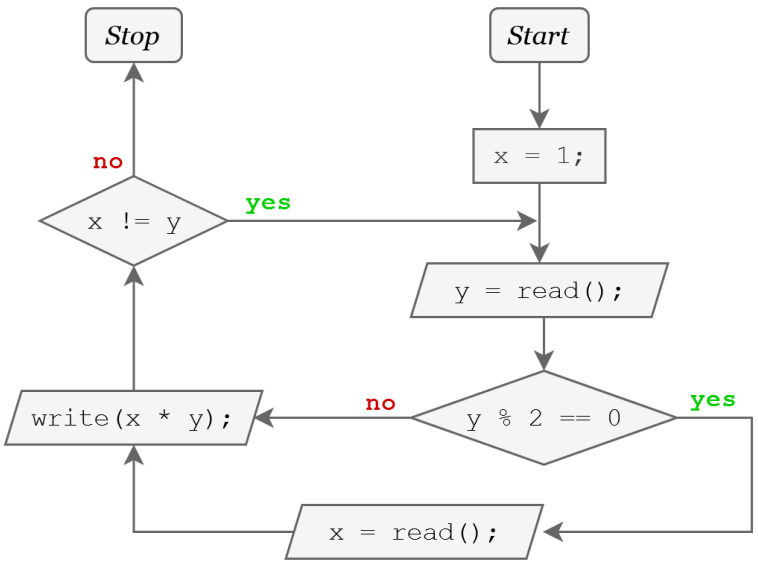
Beachte, dass die **Pfeilspitze** auch beim Kantenzusammenlauf nötig ist. Eine Kante darf entweder zwei Knoten (hier z. B. `<x==1>` und `[x=2]`) oder einen Knoten mit einer Kante (hier z. B. `<x==3>` mit der *yes*-Kante von `<x==1>`) verbinden.

Alternativ:

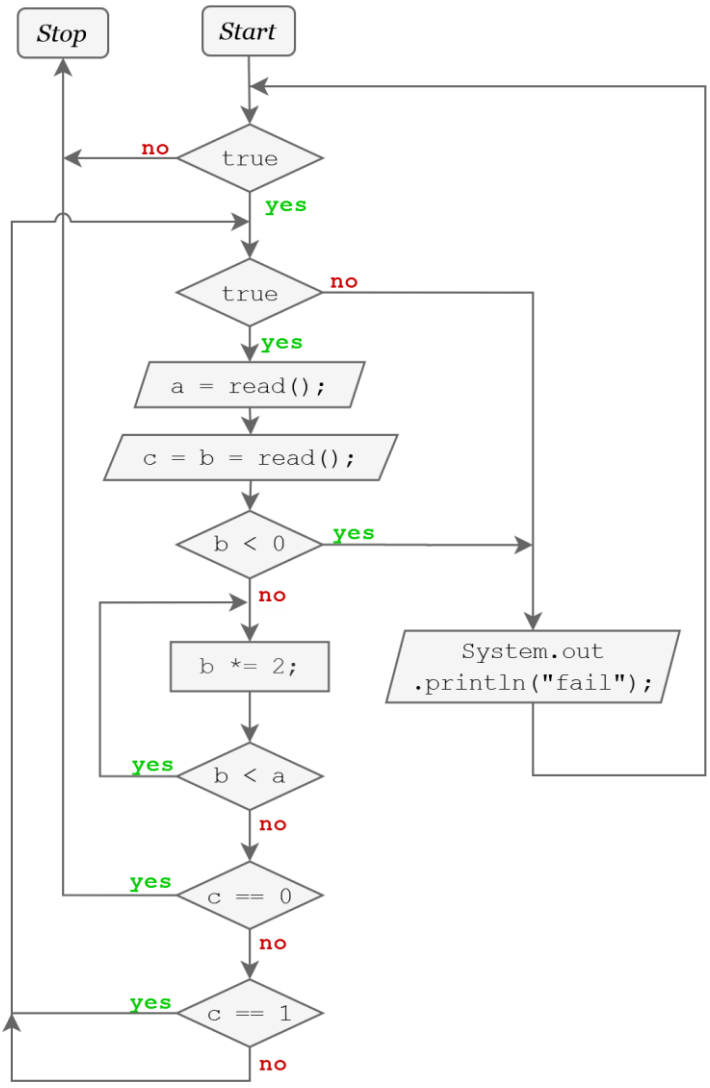
switch wird typischerweise nicht in der Vorlesung behandelt (für Kontrollflussdiagramme), wurde aber dennoch schon in Klausuren abgeprüft. Für ein *switch* wird neben der obigen Lösung anscheinend auch die rechtsseitige Lösung akzeptiert (je nach Übungsleitung).



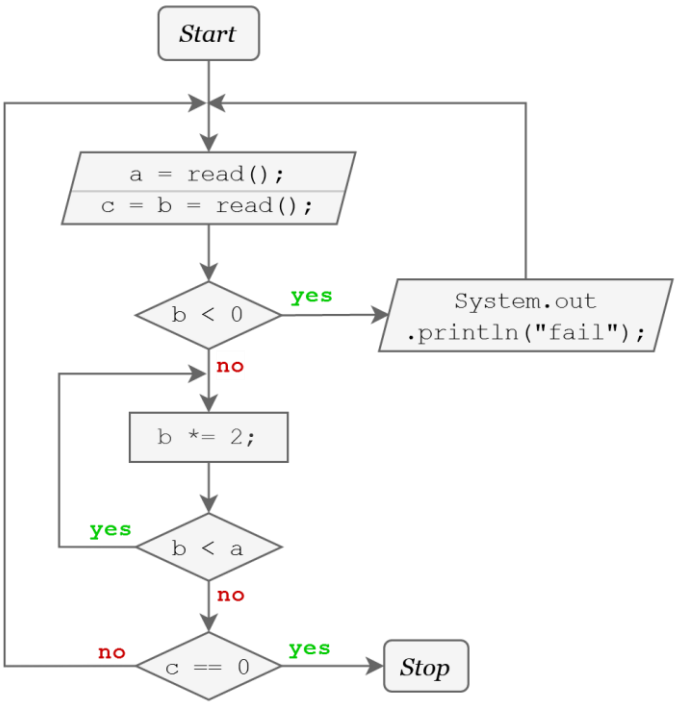
j)



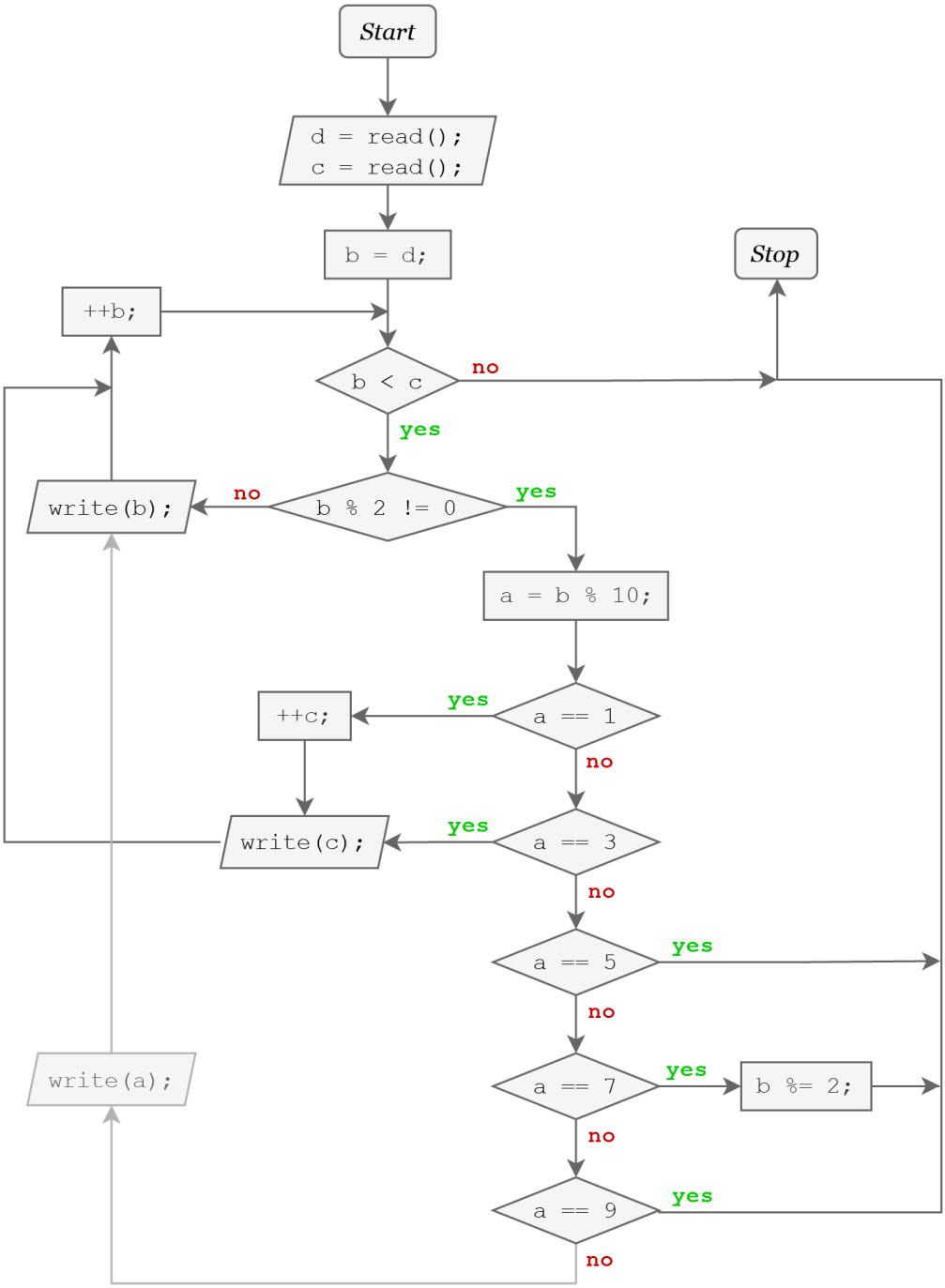
k)



Alternativ: Gekürzter Graph (sofern Vereinfachen erlaubt ist)



l)



Der hell eingezeichnete Teil ist unerreichbar, weil `a` nur ungerade Ziffern als Wert annehmen kann und diese alle in eigenen Cases behandelt werden, d. h. wenn `a` keinen der Werte 1, 3, 5 oder 7 hat, muss der Wert von `a` 9 sein. Im Graph muss der helle Pfad trotzdem eingezeichnet werden, da jede Verzweigung sowohl *yes*- als auch *no*-Fall braucht (und `a == 9` hätte sonst keinen *no*-Fall). Man könnte das optimieren, indem man diesen Bedingungsknoten einfach entfernt, d. h. das *No* von `a == 7` entspricht dem *Yes* von `a == 9`.

Wichtig: `continue` setzt die *for*-Schleife bei der Veränderung (hier: `++b`) fort.

Alternativ: `switch(a)` als <**a?**>-Knoten mit mehreren ausgehenden Pfeilen, beschriftet mit *1*, *3*, *5*, *7* und *9* (statt *yes* und *no*), ähnlich zu Teilaufgabe i).

Durch eine Zeile durchgeführte Änderungen der Variablen sind hier rot hervorgehoben, damit du deine Lösung schneller vergleichen kannst.

38

a) `6` und `8`:

Zeilennr.	a	b	c	d
2	–	–	–	6
3	–	–	8	6
4i	–	6	8	6
4m	–	7	8	6
6	7	7	8	6
16	7	1	8	6

b) `-1` und `2`:

Zeilennr.	a	b	c	d
2	–	–	–	-1
3	–	–	2	-1
4i	–	-1	2	-1
6	-1	-1	2	-1
4m	-1	0	2	-1
4m	-1	1	2	-1
6	1	1	2	-1
9	1	1	3	-1
4m	1	2	3	-1
4m	1	3	3	-1

c) `0` und `5`:

Zeilennr.	a	b	c	d
2	–	–	–	0
3	–	–	5	0
4i	–	0	5	0
4m	–	1	5	0
6	1	1	5	0
9	1	1	6	0
4m	1	2	6	0
4m	1	3	6	0
6	3	3	6	0
4m	3	4	6	0
4m	3	5	6	0
6	5	5	6	0

39

```
int facRec(int n) {
    if (n <= 1) // Abbruchbedingung
        return 1;
    // rekursiver Aufruf (else):
    return n * facRec(n-1);
}
```

40

```
int fibRec(int n) {
    if (n < 0) // ungültiger Parameter
        return -1;

    if (n == 0 || n == 1) // Anker per Def.
        return n;
    else // rek. Aufruf per Def.
        return fibRec(n-1) + fibRec(n-2);
}
```

Zusatzaufgabe:

```
int fibIter(int n) {
    if (n < 0) return -1;
    if (n == 0) return 0;

    int a = 0, b = 1;
    while (n > 1) {
        int tmpA = a;
        a = b; b = tmpA+b;
        n -= 1;
    }
    return b;
}
```

41

```
private static String stars(int amount) {
    if (amount <= 0)
        return "";
    return "*" + stars(amount-1);
}
public static void printTriangle(int size) {
    if (size == 0)
        return;
    printTriangle(size-1);
    System.out.println(stars(size));
}
```

Rechtsbündig könnte der Code bspw. so aussehen (aber es gibt unzählige andere Lösungen, teste deine Lösung also einfach in der IDE):

```
public static void printTriangle(int size) {
    printTriangle(size, 0);
}

public static void printTriangle(int stars, int spaces) {
    if (stars == 0)
        return;
    printTriangle(stars - 1, spaces + 1);
    System.out.println(spaces(spaces) + stars(stars));
    // bzw. rek. Aufruf mit sysout tauschen für absteigende Ausgabe
}

private static String spaces(int amount) {
    if (amount <= 0)
        return "";
    return " " + spaces(amount - 1);
}

private static String stars(int amount) {
    if (amount <= 0)
        return "";
    return "*" + stars(amount - 1);
}
```

42

```
static int ggt(int a, int b) {
    if (a > b)
        return ggt(a-b, b);
    if (a < b)
        return ggt(a, b-a);
    return a;
}
```

43

1.

a) Ja, die Methode ist *rekursiv*, weil sie sich selbst aufruft. Sie ist aber nicht *endrekursiv*.

b) Ja, warum nicht? Es sind keine Syntax- oder Typfehler enthalten.

c) Ja, der Methodenaufruf terminiert immer, allerdings nicht immer in einem Wert:

Für alle `x` ≤ 0 wertet `kleiner(x)` offensichtlich zu `x+1` aus und terminiert somit.

Für alle `x` > 0 wird die „Schleife“ betreten und es folgt der rek. Aufruf `kleiner(x--)`. Der Dekrementoperator reduziert `x` zwar um 1, allerdings wird der alte Wert für `x` als Parameter übergeben (anders als bei `--x`). Da jeder Methodenaufruf seine eigenen lokalen Variablen hat, hat jeder rekursive Aufruf seine eigene Variable `x`. Das nachträgliche Verringern von `x` durch `x--` hat also keinen Einfluss auf den Parameter des rekursiven Aurufs (*Call by Value*). Im rekursiven Aufruf passiert wiederum das Gleiche.

Beispiel: `kleiner(1)` ruft `kleiner(1)` auf, ruft `kleiner(1)` auf, ... („ewig“ so weiter)

Nun könnte man behaupten, dass ein Aufruf mit `x` > 0 nie terminiert. Das wäre korrekt, wenn die Maschine, auf der der Code ausgeführt wird, unbegrenzt viel Speicher (Stack) hätte. Praktisch wird der Aufruf in diesem Fall in einem `StackOverflowError` terminieren.

Fazit: Anders als Schleifen, welche tatsächlich ewig laufen können (→ Endlosschleife), terminiert jede Rekursion irgendwann, d. h. es gibt keine Endlosrekursion. Probier es einfach mal in der Entwicklungsumgebung aus. Eine scheinbare Endlosrekursion wird in einem `StackOverflowError` ihr Ende finden.

Hinweis: Wenn die rekursive Methode außerdem endrekursiv ist, dann könnte der Compiler Optimierungen durchführen (weil nichts zwischengespeichert werden muss), sodass die Rekursion tatsächlich nicht terminieren würde. Diese Optimierung wird bei uns aber nicht durchgeführt (sprengt den Rahmen dieses Moduls).

2. Die Ausgabe ist „2“. Begründung: Das `x` aus `main()` wird an keiner Stelle verändert. Jede Variable hat ihre eigenen lokalen Variablen. Ber der Parameterübergabe wird der Wert von `x` (aus `main`) kopiert (*Call by Value*). Möchte man Variablenwerte unter Methoden teilen, so nutzt man Membervariablen (z. B. Attribute).

   Ja, die Methode `happy` ist endrekursiv, da nach dem rekursiven Aufruf keine weitere Anweisung mehr ausgeführt werden muss.

3. `f(2)` → `-4`, `f(-1)` → `2` und `f(0)` → Laufzeitfehler (`StackOverflowError`). Jeder Funktionsaufruf hat eigene lokale Variablen: `f` wird durch rekursiven Aufruf nicht verändert.

4. `3, 6, 12, 6`

   Vorab zum Mitschreiben: **Was bedeutet *Call by Value*?**

   Ist der Typ des Parameters primitiv (wie in den vorherigen Teilaufgaben), so wird bei einem Aufruf immer eine *Wertkopie* übergeben, d. h. die Parametervariable erhält eine Kopie des Werts der übergebenen Variable.
   Ist der Typ des Parameters hingegen ein Referenzdatentyp (wie hier, denn Arrays sind Objekte), so wird bei einem Aufruf ebenfalls immer eine *Wertkopie* übergeben, allerdings ist der *Wert* eine Referenz. Eine Variable speichert nie ein Objekt, sondern entweder einen primitiven Wert oder eine Referenz auf ein Objekt (aka Speicheradresse).

   Erklärung: In der main-Methode wird zunächst ein Array-Objekt erzeugt, welches durch die Variable `b` zugreifbar ist. Durch den Methodenaufruf `a(b)` mit `b = {3}` bekommt die Parametervariable `a` den Wert von `b`, also eine Referenz auf das Array `{3}`, zugewiesen. Die Parametervariable `a` referenziert also dasselbe Objekt wie die lokale Variable `b` aus der `main`-Methode, d. h. `a[0]` entspricht dem Wert `3` und es wird zunächst die Ausgabe `3` produziert. Anschließend wird eine lokale Variable `b` angelegt, welche komplett unabhängig von dem `b` aus `main()` ist und `{6}` speichert. Ich nenne das `b` des ersten Aufrufs der Methode `a()` deshalb von nun an b_1 und die Variable `a` stattdessen a_1. Wegen der Zuweisung `a[0] = b[0]` wird `6` (aus b_1) in a_1 gespeichert. Da a_1 aber dasselbe Array referenziert wie das `b` aus `main()`, wird damit ja „gleichzeitig“ auch `b` verändert! Die Zuweisung `a = b` weist eine Kopie der Referenz, die b_1 speichert, an a_1 zu. Damit wird die Variable `a` überschrieben! Sie referenziert nun nicht mehr das Array `b` aus `main()` sondern b_1 (also `{6}`). Dadurch wird aber `b` nicht verändert. Wegen `6 < 15` folgt der rekursive Aufruf `a(`a_1`)`. Jeder Aufruf hat seine eigenen lokalen Variable, d. h. es gilt zunächst $a_2 = a_1$ und es folgt die Ausgabe `6`. b_2 speichert ein neues Array mit dem Inhalt `{12}`. Durch `a[0] = b[0]` speichert a_2 nun ebenfalls den Wert `12` im ersten Feld. Die Zuweisung `a = b` weist eine Kopie der Referenz, die b_2 speichert, an a_1 zu. Damit wird die Variable `a` überschrieben. Es folgt im dritten Aufruf von `a()` die Ausgabe `12`. Zuletzt wird in der `main`-Methode `6` ausgegeben, was der ersten Überschreibung `a[0] = b[0]` geschuldet ist. Dass die Ausgabe nicht noch einmal `12` ist liegt an der Überschreibung `a = b` (siehe oben).

5. Ja. Notfalls könnte man einen eigenen Stack verwenden (z. B. `java.util.Stack`) und damit die „normale“ Rekursion nachbauen.

44

a)
```
int powerIter(int a, int b) {
    int result = 1;
    while (b > 0) {
        result *= a;
        b -= 1;
    }
    return result;
}
```

b)
```
int powerRec(int a, int b) {
    if (b <= 0)
        return 1;
    return a * powerRec(a, b-1);
}
```

45

```
public void reverse(int x) {
    if (x >= 1) {
        // x % 10 = letzte Ziffer
        System.out.print(x % 10);
        // x / 10 = alles außer letzte Ziffer
        reverse(x / 10);
    }
}
```

oder alternativ bspw. über einen String:

```
public void reverse(int x) {
    String r = "" + x;
    if (x <= 0)
        return;
    System.out.print(r.charAt(r.length()-1));
    reverse(x / 10);
}
```

46

```
int price(int n) {
    if (n >= 100)
        return 250 + price(n - 100);
    else if (n >= 50)
        return 130 + price(n - 50);
    else if (n >= 10)
        return 28 + price(n - 10);
    else if (n >= 1)
        return n*3;
    else
        return 0;
}
```

oder alternativ bspw.:

```
int price(int n) {
    if (n <= 0)
        return 0;
    if (n < 10)
        return 3 + price(n-1);
    if (n < 50)
        return 28 + price(n-10);
    if (n < 100)
        return 130 + price(n-50);
    return 250 + price(n-100);
}
```

47

a)

```
int binomialRec(int n, int k) {
    if (k < 0 || k > n)
        return -1;

    if (k == 0 || k == n)
        return 1;

    return binomialRec(n-1, k-1) + binomialRec(n-1, k);
}
```

b)

```
int binomialDef(int n, int k) {
    return facRec(n) / (facRec(k) * facRec(n-k));
}
```

48

```
public static void multiplesOfThree(int n) {
    multiplesOfThree(n, 0);
}
private static void multiplesOfThree(final int n, int i) {
    if (i > n)
        return;
    if (i != 0)
        System.out.print(", ");
    System.out.print(i);
    multiplesOfThree(n, i + 3);
}
```

49

```
public static String replaceVowels(String s) {
    return replaceVowels(s, 0);
}

private static String replaceVowels(final String s, int i) {
    if (i >= s.length()) // Abbruchbedingung (am String-Ende angekommen)
        return "";

    char c = s.charAt(i);   // nächsten Buchstaben holen
    if (c == 'a' || c == 'e' || c == 'i' ||
        c == 'o' || c == 'u' || c == 'y')   // durch kleines x ersetzen
        c = 'x';
    else if ("AEIOUY".indexOf(c) != -1)   // alternativ einfach wie im if
        c = 'X';

    return c + replaceVowels(s, i+1);
}
```

50

```
static int sum(int a, int b) {
    if (b == 0) return a;
    if (a == 0) return b;
    if (a > 0)
        return 1 + sum(a-1, b);
    if (a < 0) // else
        return sum(a+1, b) - 1;
    return -1; // kann nicht eintreten
}
```

51

```
public long replaceDigit(long number, byte digit, byte newDigit) {
    byte currentDigit = (byte) (number % 10);  // letzte Ziffer
    if (currentDigit == digit)  // ersetze die Ziffer, falls nötig
        currentDigit = newDigit;
    if (number < 10)  // nichts mehr zu tun (Rekursionsanker)
        return currentDigit;

    // Restliche Zahl (vorderer Teil) rekursiv ersetzen, danach mich selbst hinten
    // dran hängen, indem die neue Zahl mit 10 multipliziert wird, wodurch
    // ein freier Platz (0) entsteht, auf die ich mich selbst addieren kann:
    return replaceDigit(number / 10, digit, newDigit) * 10
        + currentDigit;
}
```

52

```
public boolean isPalindrome(char[] c) {
    if (c == null)
        return false;
    if (c.length == 0)
        return true;
    return isPalindrome(c, 0);
}

private boolean isPalindrome(char[] c, int pos) {
    if (pos >= c.length/2)
        return true;
    if (c[pos] != c[c.length-1-pos])
        return false;
    return isPalindrome(c, pos+1);
}
```

Zusatzaufgaben:

```
public boolean isPalindromeIter(char[] c) {
    if (c == null)
        return false;

    for(int i = 0; i < c.length/2; i++) {
        if (Character.toLowerCase(c[i]) !=
            Character.toLowerCase(c[c.length-1-i]))
            return false;
    }
    return true;
}
```

53

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

private boolean isPrime(int n, int div) {
    if (div <= 1)
        return true;
    if (n % div == 0) // ohne Rest teilbar?
        return false;
    return isPrime(n, div-1);
}
```

54

```java
int ultraRec(short x) {
    if (x <= 1)
        return 0;
    return x * (x-1) +
      ultraRec((short)(x-2));
}
```

55

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

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

   int middle = (from+to)/2;
   Object[] left = reverse(a, from, middle);
   Object[] right = reverse(a,       middle + 1       , to);
    Object[]     result = new Object[     to - from + 1     ];

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

   return result;
}
```

besser: `from + (to-from)/2`

oder: `left.length + right.length`

56

a)

```
public static int anlagedauerBerechnen(double kapital,
                                        double zielkapital,
                                        double zinssatz) {
    if (kapital >= zielkapital) // Ziel erreicht
        return 0;

    return anlagedauerBerechnen(kapital + kapital * zinssatz/100,
                                zielkapital, zinssatz) + 1;
}
```

Alfi: `anlagedauerBerechnen(1_000, 1_000_000, 10)`
→ Alfi hat die Million nach 73 Jahren, also im Alter von 88 Jahren.

Henni: `anlagedauerBerechnen(500, 100_000, 5)`
→ Henni hat 100.000 erst nach 109 Jahren, also im Alter von 122 Jahren.

⇒ Beide sollten das Geld auf den Putz hauen (Hinweis: subjektive Einschätzung).

b)

```
public static double vermoegenBerechnen(double kapital,
                                         double monatsbeitrag,
                                         int jahre,
                                         double zinssatz,
                                         double dynamisierung) {
    if (jahre <= 0)   // keine Jahre zum Sparen mehr übrig
        return kapital;   // Guthaben zu diesem Zeitpunkt

    kapital += 12 * monatsbeitrag;   // monatliche Einzahlungen
    kapital += kapital * zinssatz/100;   // jährliche Rendite
    monatsbeitrag += monatsbeitrag * dynamisierung/100;   // Dynam.

    return vermoegenBerechnen(kapital, monatsbeitrag, jahre-1,
                              zinssatz, dynamisierung);
}
```

In diesem Beispiel hätte Alfi nach 50 Jahren (also im Alter von 65 Jahren) insgesamt Einzahlungen in Höhe von 50.243 Euro vollzogen und nun 230.986 Euro zur Verfügung:

`vermoegenBerechnen(0, 20, 50, 7, 5)`

Geduldet er sich um 10 weitere Jahre (also bis er 75 ist), dann hätte er schon mehr als 500.000 Euro zur Verfügung.

Hinweis: In realen Simulationen dieser Art Kosten/Steuern/Inflation beachten!

57

Die rekursiven Lösungen zu *multiplesOfThree*, *ggt*, *reverse*, isPalindrome und isPrime sind bereits endrekursiv. Es folgen die endrekursiven Implementierungen zu den Aufgaben, die noch nicht bereits endrekursiv waren.

Eine rekursive Methode lässt sich am einfachsten endrekursiv formulieren, indem man einen zusätzlichen Parameter als eine Art „Akkumulator“ einführt (wie bei *multiplesOfThree*) – das ist ein Parameter, über den wir das „bisherige“ Ergebnis zwischenspeichern und bspw. in jedem Methodenaufruf etwas aufaddieren, anhängen, o. Ä.. Dazu ist dann eine Hilfsmethode nötig (da man ja nicht einfach die Signatur einer verlangten Methode ändern darf und jetzt ein zusätzlicher Parameter existiert). Diese Methode wird dann wiederum von der ursprünglichen Methode aufgerufen, wobei der Akkumulator sinnvoll initalisiert wird.

```
int facRec(int n) {
    return facRec(n, 1);
}

private int facRec(int n, int acc) {
    if (n <= 1)
        return 1*acc;
    return facRec(n-1, n*acc);
}
```

```
int powerRec(int a, int b) {
    return powerRec(a, b, 1);
}

private int powerRec(int a, int b, int acc) {
    if (b <= 0)
        return 1*acc;
    return powerRec(a, b-1, a*acc);
}
```

```
int price(int n) {
    return price(n, 0);
}

private int price(int n, int acc) {
    if (n >= 100)
        return price(n-100, 250+acc);
    else if (n >= 50)
        return price(n-50, 130+acc);
    else if (n >= 10)
        return price(n-10, 28+acc);
    else if (n >= 1)
        return n*3 + acc;
    return 0 + acc;
}
```

```
public static String replaceVowels(String s) {
    return replaceVowels(s, 0, "");
}

private static String replaceVowels(final String s,
                                    int i, String neu) {
    if (i >= s.length())
        return neu;

    char c = s.charAt(i);
    if ("aeiou".indexOf(c) != -1) // alternativ wie zuvor
        c = 'x';
    else if ("AEIOU".indexOf(c) != -1)
        c = 'X';
    neu += c;

    return replaceVowels(s, i + 1, neu);
}
```

Mögliche (ineffizientere) Alternative: Jeweils einen neuen String anlegen, welcher bis auf das aktuelle Zeichen identisch ist zu `s`. Diesen dann als neues `s` übergeben (und am Ende einfach `s` zurückgeben).

```
int ultraRec(short x) {
    return ultraRec(x, 0);
}

private int ultraRec(short x, int acc) {
    if (x <= 1)
        return 0 + acc;
    return ultraRec((short) (x-2), x * (x-1) + acc);
}
```

58

```java
public long sumIt(long n) {
    long result = 0;
    for (int i = 1; i <= n; ++i)
        result += i;
    return result;
}
```

```java
public long sumRec(long n) {
    if (n < 1)
        return 0;
    return n + sumRec(n-1);
}
```

```java
public long sumTailRec(long n) {
    return sumTailRecHelper(n, 0);
}

private long sumTailRecHelper(long n, long acc) {
    if (n < 1)
        return acc;
    return sumTailRecHelper(n-1, acc+n);
}
```

59

```java
public int nonsenseRec(int x) {
    if (x >= 100)
        return 0;
    return x + nonsenseRec(x+3);
}
```

```java
public int nonsenseIter(int x) {
    int result = 0;
    for (int i = x; i < 100; i += 3)
        result += i;
    return result;
}
```

60

a)
```
public int quersumme(int a) {
    if (a != 0)
        return a % 10 + quersumme(a / 10);
    return 0;
}
```

b)
```
public int quersumme(int a) {
    int ret = 0;
    while (a != 0) {
        ret += a % 10;
        a /= 10;
    }
    return ret;
}
```

c)
```
private int quersumme_h(int a, int acc) {
    if (a != 0)
        return quersumme_h(a / 10, acc + a % 10);
    return acc;
}

public int quersumme(int a) {
    return quersumme_h(a, 0);
}
```

61

```
public String invert(String str)
{
    if (str == null)
        throw new IllegalArgumentException();
    return invert(str, str.length()-1, "");
}

private String invert(String str, int index, String result)
{
    if (index < 0)
        return result;
    result += str.charAt(index);
    return invert(str, index-1, result);
}
```

62

```
static int y(int x) {
    while(x < 100)
        x *= x;
    return x;
}
```

1.	**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. Die Ausgabe ist **24567**. Erklärung:

Bei der Initialisierung von `a` wird `"x"` in den String-Pool aufgenommen.

`"" + a` ist eine Operation mit einer Variable und wird daher erst zur Laufzeit ausgeführt, d. h. der Wert von `b` steht aus Compiler-Sicht noch nicht fest (weil der Compiler nur diese eine Zeile betrachtet und die Variable `a` – wie der Name schon sagt – theoretisch variabel ist, also hat `b` zu diesem Zeitpunkt theoretisch keinen fixen Wert).

`"" + "x"` kann hingegen schon vom Compiler zu `"x"` kombiniert werden. `"x"` ist jedoch bereits im String-Pool, daher wird `c` dasselbe String-Objekt referenzieren wie `a`, d. h. es gilt `a == c` (Ausgabe 2). Das gleiche gilt für `"x" + ""` (Ausgabe 4).

Es gilt weder `a == b` (Ausgabe 1) noch `b == c` (Ausgabe 3), da `b` erst zur Laufzeit erzeugt wird und daher nicht auf das Objekt im String-Pool zeigt.

Bei Ausgabe 5 werden primitive Werte verglichen und es gilt `'x'+0 == 'x'`.

Bei Ausgabe 6 werden die Strings zeichenweise verglichen.

Es folgt eine Zuweisung von `false` an die Variable `y`. Die Ausgabe 7 findet statt, obwohl `y` den Wert `false` hat, da hier kein Vergleich stattfindet (einfaches `=` statt `==`), sondern durch `y = true` der Wert `true` an `y` zugewiesen wird und der Ausdruck zu `true` auswertet.

```java
public class Book {

    // Globale Variable als Zähler für die ID-Nr. des nächsten Buches
    private static long NEXT_ID_NR = 1;

    // Attribute:
    private final String title, author;
    private int price;
    private final long idNr;

    /* Konstruktoren (Preis kann optional angegeben werden): */
    public Book(String title, String author) {
        this(title, author, 0);
    }
    public Book(String title, String author, int price) {
        this.title = title;
        this.author = author;
        this.idNr = NEXT_ID_NR++;
        this.setPrice(price);
    }

    /* Getter- und Setter-Methoden: */

    public String getTitle() {
        return title;
    }

    public String getAuthor() {
        return author;
    }

    public int getPrice() {
        return price;
    }

    public long getIdNr() {
        return idNr;
    }

    public void setPrice(int newPrice) {
        if (price < 0)
            throw new IllegalArgumentException("Preis ist negativ");
        price = newPrice;
    }
```

```
    // a)
    public String toString() {
        int euro = price/100;
        int cent = price%100;

        String centStr = "" + cent;
        if (cent < 10)
            centStr = "0" + centStr;

        return author + ": " + title + " (EUR " + euro + "," + centStr + ")";
    }

    // b)
    // Die Methode ist hier deshalb überflüssig, weil jedes Buch-Objekt eine eindeutige ID-
    // Nummer besitzt. Daher tut die equals-Methode dasselbe wie der ==-Operator.
    public boolean equals(Object o) {
        if (!(o instanceof Book))
            return false;

        return idNr == ((Book) o).idNr;
    }

    // c)
    public void increasePrice(float euro) {
        int cent = (int) (euro * 100);

        if (price + cent < 0)
            throw new IllegalArgumentException("Ungültige Erhöhung.");

        price += cent;
    }

    // d)
    public boolean hasSameAuthor(Book b) {
        return this.author.equals(b.author);
    }

    // e)
    public int authorLetters() {
        int letters = 0;

        for (int i = 0; i < author.length(); i++)
            if (author.charAt(i) != ' ')
                letters++;

        return letters;
    }

    // Statische Methode, die an Objektmethode delegiert:
    public static boolean equals(Book b1, Book b2) {
        return b1.equals(b2);
    }
}
```

```java
import java.util.LinkedList;

public class Library {
    private LinkedList<Book> lib;

    public Library() {
        lib = new LinkedList<>();
    }

    // a)
    public void add(Book b) {
        if (!lib.contains(b))
            lib.add(b);
    }

    // b)
    public Book get(long id) {
        for (Book b : lib)
            if (b.getIdNr() == id)
                return b;
        return null;
    }

    // c)
    public boolean delete(Book b) {
        return lib.remove(b);
    }

    // d)
    public LinkedList<Book> getBooksWrittenBy(String author) {
        LinkedList<Book> sameAuthor = new LinkedList<>();
        for (Book b : lib)
            if (b.getAuthor().equals(author))
                sameAuthor.add(b);
        return sameAuthor;
    }
}
```

```java
public abstract class Medium {
    private static long NEXT_ID_NR = 1;

    private final String title;
    private final long idNr;
    private int price;

    public Medium(String title) {
        this(title, 0);
    }
    public Medium(String title, int price) {
        this.title = title;
        this.idNr = NEXT_ID_NR++;
        this.setPrice(price);
    }

    public String getTitle() {
        return title;
    }
    public long getIdNr() {
        return idNr;
    }
    public int getPrice() {
        return price;
    }

    public final void setPrice(int newPrice) {
        if (price < 0)
            throw new IllegalArgumentException("Preis negativ");
        price = newPrice;
    }

    public void increasePrice(float euro) {
        int cent = (int) (euro * 100);
        if (price + cent < 0)
            throw new IllegalArgumentException("Ungültig.");
        price += cent;
    }

    public boolean equals(Object o) {
        if (!(o instanceof Medium))
            return false;
        return idNr == ((Medium) o).idNr;
    }
}
```

```java
public class Book extends Medium {
    private final String author;

    public Book(String title, String author) {
        this(title, author, 0);
    }

    public Book(String title, String author, int price) {
        super(title, price);
        this.author = author;
    }

    public String getAuthor() {
        return author;
    }

    public String toString() {
        int euro = getPrice() / 100;
        int cent = getPrice() % 100;
        String centStr = "" + cent;
        if (cent < 10)
            centStr = "0" + centStr;
        return author + ": " + getTitle() +
               " (EUR " + euro + "," + cent + ")";
    }

    public boolean hasSameAuthor(Book b) {
        return this.author.equals(b.author);
    }

    public int authorLetters() {
        int letters = 0;
        for (int i = 0; i < author.length(); i++)
            if (author.charAt(i) != ' ')
                letters++;
        return letters;
    }

    public static boolean equals(Book b1, Book b2) {
        return b1.equals(b2);
    }
}
```

```java
public class DVD extends Medium {
    private final String[] actors;

    public DVD(String title, String[] actors) {
        this(title, actors, 0);
    }

    public DVD(String title, String[] actors, int price) {
        super(title);
        this.actors = actors;
        this.setPrice(price);
    }

    public String[] getActors() {
        return actors;
    }
}
```

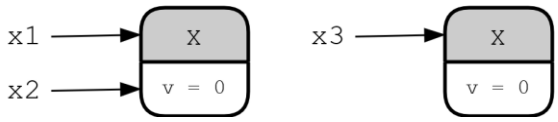
```java
import java.util.LinkedList;

public class Library {

    private LinkedList<Medium> lib;

    public Library() {
        lib = new LinkedList<>();
    }

    public void add(Medium m) {
        if (!lib.contains(m))
            lib.add(m);
    }

    public Medium get(long id) {
        for (Medium m : lib)
            if (m.getIdNr() == id)
                return m;
        return null;
    }

    public boolean delete(Medium m) {
        return lib.remove(m);
    }

    public LinkedList<Book> getBooksWrittenBy(String author) {
        LinkedList<Book> sameAuthor = new LinkedList<>();
        for (Medium m : lib) {
            if (m instanceof Book) {
                Book b = (Book) m;
                if (b.getAuthor().equals(author))
                    sameAuthor.add(b);
            }
        }
        return sameAuthor;
    }
}
```

(67)

Die Ausgabe lautet: **`1 1 0  2 3 3  2 4 4  6 5 2`**

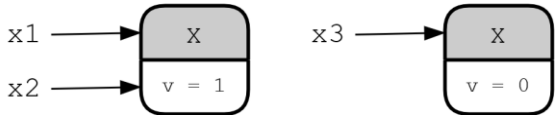
Die einzelnen Schritte werden im Folgenden mit Objektdiagrammen veranschaulicht. Dabei wird das Objektdiagramm nach jedem Methodenaufruf aktualisiert. Objekte werden dabei in Rechtecken mit abgerundeten Ecken gezeichnet.

1. Startzustand (nach `X x1, x2; x1 = x2 = new X(); X x3 = new X();`):



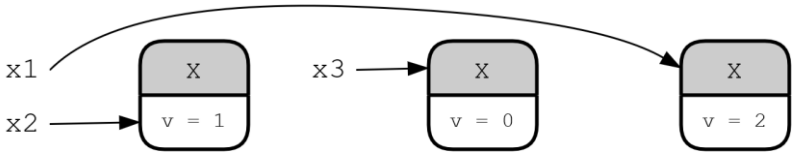
   Bedeutung: Es gibt drei Variablen (`x1`, `x2`, `x3`) aber nur zwei Objekte, da `x1` und `x2` dasselbe Objekt referenzieren. Die Objekte haben alle den `v`-Wert 0, da Attribute automatisch mit `null` (bei Referenzdatentypen) bzw. 0 bei `int`s initialisiert werden.

2. Zustand nach `x1.set(1)`: Setzt das `v`-Attribut von `x1` (und damit auch von `x2`) auf 1.

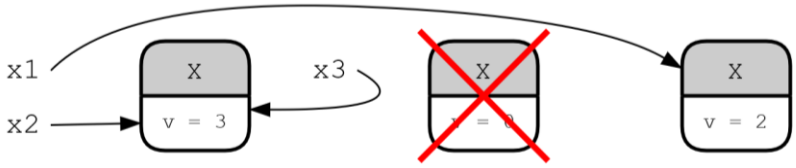


3. Ausgabe: **`1 1 0`**

4. Zustand nach `x1 = x1.set(x1)`: Erzeugt innerhalb des Methodenaufrufs ein neues `X`-Objekt und weist es an die Parametervariable `x` zu. Dessen `v`-Wert wird auf 2 gesetzt. Anschließend wird es zurückgegeben und an die Variable `x1` zugewiesen.

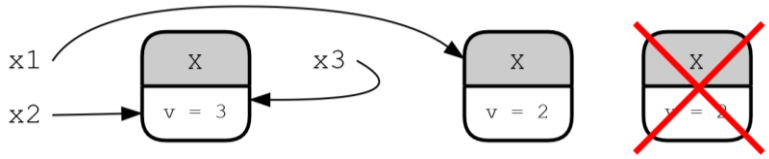


5. Zustand nach `x3 = x1.set(x2, 3)`: Setzt den `v`-Wert von `x2` innerhalb des Methodenaufrufs auf 3. Anschließend wird `x2` zurückgegeben (**`return`** `x;`) und an `x3` zugewiesen, d. h. `x3` zeigt nun auf dasselbe Objekt wie `x2`. Das alte `x3`-Objekt wird von der Garbage-Collection gelöscht, da es nicht mehr referenziert wird.

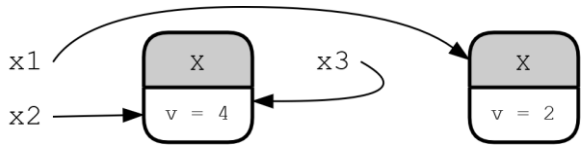


6. Ausgabe: **`2 3 3`**

7. Zustand nach `x1.set(x2)`: Wie bei Schritt 4 wird innerhalb des Methodenaufrufs ein neues `X`-Objekt erzeugt und in der Paramtervariable x gespeichert. Das Überschreiben der Variable `x` hat keinen Einfluss auf die übergebene Variable `x2` (*Warum?* Call by Value! `x` ist eine Kopie der Referenz `x2`. Die Zuweisung an `x` verändert nicht das von `x` referenzierte Objekt, sondern überschreibt `x` einfach). Das neu erzeugte Objekt wird zurückgegeben, es erfolgt jedoch keine Zuweisung. Daher wird das Objekt von keiner Variable mehr referenziert und somit gelöscht.

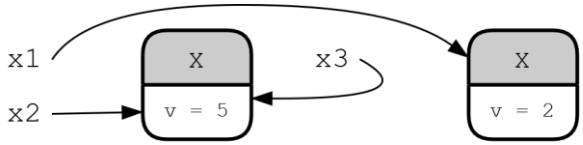


8. Zustand nach `x2.set(4)`: Setzt den `v`-Wert von `x2` (und somit auch `x3`) auf 4.

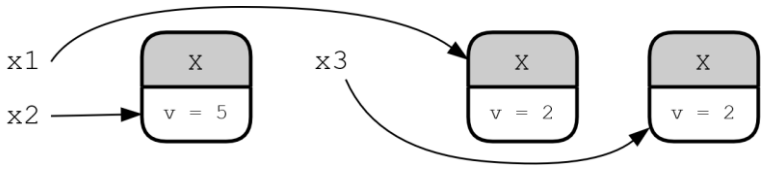


9. Ausgabe: **2 4 4**

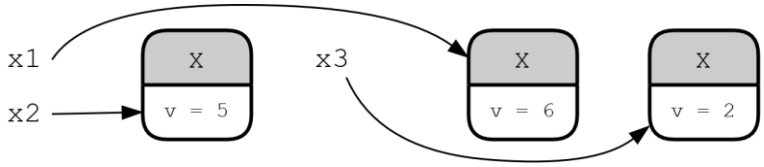
10. Zustand nach `x3.set(5)`: Setzt den `v`-Wert von `x3` (und somit auch `x2`) auf 5.



11. Zustand nach `x3 = x1.set(x2)`: Wie bei Schritt 7 wird ein neues `X`-Objekt mit dem `v`-Wert 2 erzeugt und zurückgegeben. Im Unterschied zu 7 wird das zurückgegebene Objekt weiterverwertet und an `x3` zugewiesen, deshalb auch nicht gelöscht.



12. Zustand nach `x3.set(x1, 6)`: Setzt den `v`-Wert von `x3` auf 6.



13. Ausgabe: **6 5 2**

Es können mehrere Methoden mit dem Namen `set` existieren, da sich diese in den Typen der Parameter unterscheiden und somit trotz des gleichen Namens verschiedene Signaturen besitzen. Dieses Konzept heißt Überladung.

68

Die Lösung zu dieser Aufgabe findest du unter ***stecrz.de/files/skript/forum_loesung.zip***

Ergänzende Anmerkungen:

Man sollte hier besonders darauf achten, Eigenschaften, die sich mehrere Klassen teilen, in Oberklassen zusammenzufassen. In dieser Aufgabe verfügen bspw. alle Benutzertypen über einen Namen und eine E-Mail-Adresse, also ist es sehr sinnvoll, diese Eigenschaften in der Oberklasse `User` zu definieren (statt in allen Unterklassen einzeln). Das muss nicht explizit in der Angabe stehen, d. h. bei Modellierungsaufgaben kann redundanter Code durchaus zu Punktabzug führen! Es wäre in dieser Aufgabe zwar nicht falsch, wenn jeder Nutzer zusätzlich ein Attribut zur Speicherung des Namens definiert, würde aber trotzdem zu Punktabzug führen, da das den Grundsätzen objektorientierten Modellierung widerspricht… So benötigt ein Moderator bspw. kein Attribut zur Speicherung des Namens, da er das Attribut ja von der Oberklasse erbt. Trotzdem muss man sich darum kümmern, dieses Attribut zu setzen. Nun hat ein Moderator aber keinen Zugriff auf die privaten Attribute der Oberklasse (wenngleich er dieses Attribut erbt). Man könnte das Attribut der Oberklasse `protected` definieren, um es im Konstruktor der Unterklasse zu setzen. Jedoch widerspricht auch das den Prinzipien objektorientierter Programmierung. Deutlich wird das am Attribut `email` der `User`-Klasse. Dieses Attribut ist `final`, muss daher im Konstruktor von `User` gesetzt werden und kann nachträglich nicht verändert werden. Es gibt somit keine andere Möglichkeit, als das Attribut im `User`-Konstruktor zu setzen. Es ist zwar richtig, dass man kein Objekt einer abstrakten Klasse erzeugen kann (d. h. man kann den `User`-Konstruktor nicht direkt aufrufen), allerdings ist das erste, was in jedem Konstruktor passiert (implizit oder explizit), der Aufruf eines anderen Konstruktors (in Form von `this(…)` oder `super(…)`) – dies ist die erste Zeile eines jeden Konstruktors (ob du es hinschreibst oder nicht…). Jede Unterklasse von `User` (bspw. `Moderator`) muss also in ihrem Konstruktor den Konstruktor von `User` aufrufen (mittels `super(…)`). Der `super`-Aufruf kann weggelassen werden, wenn es in der Oberklasse einen parameterlosen Konstruktor gibt – in diesem Fall geschieht der Aufruf `super()` automatisch. In `User` gibt es jedoch keinen solchen Konstruktor ohne Parameter, weshalb der `super(name,email)`-Aufruf zwingend nötig ist! Weil die Unterklassen einer abstrakten Klasse also den Konstruktor der abstrakten Klasse aufrufen müssen, verfügen abstrakte Klasse ebenfalls über Konstruktoren. Jede Klasse (auch abstrakte) verfügt übrigens über mindestens einen Konstruktor. Definieren wir keinen Konstruktor, so ist der *Default-Konstruktor* `public MyClass() { super(); }` implizit vorhanden.

Um jedem `Post` eine eindeutige ID zuzuweisen, können wir Gebrauch von einer statischen Variable machen. Eine statische Variable wird von allen Objekten einer Klasse geteilt (existiert also nur einmal, wohingegen bspw. jeder `Post` einen eigenen Autor hat). Diese Variable speichert in der Musterlösung die Anzahl erzeugter `Post`-Objekte. Bei der Erzeugung eines neuen `Post`-Objekts – also immer wenn ein Konstruktor der Klasse `Post` aufgerufen wird – wird dieser objektübergreifende Zähler erhöht. Wir können den Wert dieses Zählers daher als ID übernehmen. Das erste `Post`-Objekt setzt den Counter bspw. auf `1` und speichert sich den Wert `1` ab (eine Kopie davon). Das zweite `Post`-Objekt erhöht den Counter dann auf `2` und speichert sich den Wert `2` ab.

69

Gemische Aussagen zu Klassen und Interfaces		Wahr	Falsch
a)	Objektvariablen/Attribute können Klassenvariablen innerhalb derselben Klasse verschatten. Nein, aber Klasse kann geerbte verschatten	☐	☒
b)	Auf verschattete Membervariablen kann über den ~~Klassennamen~~ `this` zugegriffen werden.	☐	☒
c)	Auf verschattete statische Variablen kann über das Schlüsselwort `this` zugegriffen werden. Ja, aber normalerweise über den Klassenamen	☒	☐
d)	Ein formaler Parameter kann eine mit dem Modifier `public` deklarierte statische Variable verschatten. Modifier tut nichts zur Sache	☒	☐
e)	Eine Klasse kann unter Umständen `private` sein. Innere Klasse	☒	☐
f)	Eine *top-level* Klasse (keine innere Klasse) darf weder `private` noch `protected` sein. `public` oder package-private, sonst sinnlos	☒	☐
g)	Konstruktoren können überladen werden. Bei mehreren Konstr. immer	☒	☐
h)	Eine in einer Datei als `public` deklarierte *top-level* Klasse muss immer denselben Namen haben wie die Datei. nur 1x `public` pro 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. erbt von genau einer, implem. $0 - \infty$	☒	☐
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. „Inneres Interface“?	☒	☐
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. nicht-statisch → an Objekt von `A` gebunden	☐	☒
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: `if` ohne „{ ... }“-Block ist ok, allerdings macht dort eine Deklaration keinen Sinn, ist daher verboten. `public void method() { if (true) int x = 5; }`	☐	☒
t)	Innere Klassen können innere Klassen besitzen.	☒	☐
u)	Finale Methoden werden vererbt. dürfen nur nicht überschrieben werden	☒	☐
v)	Methodendeklarationen ohne Verwendung des Schlüsselwortes `abstract` sind in Interfaces möglich. `public abstract` ist da implizit	☒	☐
w)	In einem Interface können Variablen angelegt werden. statische	☒	☐
x)	Eine Methodendeklaration der Form „`public int x();`“ ist innerhalb einer abstrakten Klasse erlaubt. müsste `abstract` sein	☐	☒
y)	Von einer finalen Klasse kann man nicht erben.	☒	☐
z)	Eine abstrakte Klasse kann als `final` deklariert werden. sinnfrei	☐	☒
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: Blöcke („{ ... }“) können beliebig verwendet und geschachtelt werden. Sie kennzeichnen den Gültigkeitsbereich von Variablen. `public int method() { { return 5; } }`	☒	☐
F)	Interfaces können von ~~abstrakten Klassen~~ Interfaces erben.	☐	☒
G)	Abstrakte Klassen können von Interfaces ~~erben~~ implementieren.	☐	☒

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: x in A dürfte nur nichts zugewiesen bekommen
	`class A { final int x = 0; }`
	`class B extends A { int x = 5; }`
		☒	☐

P)	Eine finale Methode kann überladen werden. aber nicht überschrieben	☒	☐

Q)	Folgender Code kompiliert:
	`int method() {`
	`  if (false) {`
	`  int x = 5;` Gültigkeitsbereich von x an „{ ... }“ erkennbar
	`  } return x;` x ist an dieser Stelle nicht definiert/sichtbar
	`}`
		☐	☒

R)	Eine statische Variable kann als `final` deklariert werden.	☒	☐
S)	Interfaces können generisch* sein. . * Vorgriff	☒	☐

T)	Folgender Code kompiliert: Jede Klasse hat die `toString()`-Methode
	`class Foo<T, P> {`
	`  void m(P x) {`
	`  x.toString();`
	`  }`
	`}`
		☒	☐

U)	Folgender Code kompiliert: Generischer Typ ist an ein Objekt gebunden
	`class Foo<R> {`
	`  public static R m(R r) {`
	`  return r;`
	`  }`
	`}`
		☐	☒

V)	Von Klassen wie `String` und `Integer` kann man erben. `final`	☐	☒
W)	~~Variablen~~ Methoden können den Typ `void` haben.	☐	☒

X)	Folgender Code kompiliert: Z und W sind Klassen, V ist ein Platzhalter
	`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. Fahrzeug erwartet? Auto und Motorrad und Fahrzeug ok.	☒	☐
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. nur bei `protected`	☒	☒
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`. implizit	☒	☐
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); }` Verändert nur die Variable `a`, nicht das darin gespeicherte Objekt und damit auch nicht ↓ Folgt nach der Deklaration von `A a = new A(0);` der Aufruf `f(a)`, so hat `a.x` anschließend den Wert `1`. ist nach wie vor 0	☐	☒
λ)	Eine innere Klasse `class Foo` kann statische Variablen oder Methoden definieren. dazu müsste die Klasse `static` sein	☐	☒
μ)	Folgende Klasse kompiliert: Konstanten (`final`) dürfen nur einmal zugewiesen werden. Methoden können mehrfach aufgerufen werden, daher Zuweisung nur in Konstruktor oder direkt bei der Deklaration möglich `class A { final int a; void m() { a = 1; } }`	☐	☒

ν)	Klassennamen ~~müssen~~ sollten großgeschrieben werden.	☐	☒
ξ)	Konstruktoren können `static` sein.	☐	☒
ο)	Konstruktoren können `final` sein. sind sowieso nicht überschreibbar	☐	☒
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*: Doch, aber der Aufruf `m(1,2)` wäre nicht möglich, da nicht eindeutig! `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. Nein, kein `@Override` möglich. Das nennt sich dann „Überladen“.	☐	☒
φ)	Statische Methoden können `final` sein.	☒	☐
χ)	Statische Methoden können *überladen* werden.	☒	☐
ψ)	Gegeben sei die Klasse aus Teilaufgabe τ). • Folgender Aufruf kompiliert *nicht*: `X.m(0, 0d);` undefiniert • Folgender Aufruf kompiliert *nicht*: `X.m(0, 0);` nicht eindeutig	☒ ☒	☐ ☐
ω)	Folgender Code kompiliert: Reihenfolge der Member irrelevant `public class X { int m() { return x; } int x; }` Attribut automatisch mit 0 initialisiert	☒	☐
Δ)	Folgende Methode kompiliert: `int m(boolean b) { int x; if (b) x = 1; return x; }` x ist eventuell nicht initialisiert worden (else-Fall)	☐	☒

70

a) Falsch. Der `StackOverflowError` (*Stack-Überlauf*) entsteht bei zu vielen Methodenaufrufen, i. d. R. bei Rekursion, da für jeden Methodenaufruf Speicherplatz auf dem Stack benötigt wird (z. B. für die Rücksprungadresse).

b) Wahr. Bei der Erzeugung von zu vielen Objekten kommt es zu einem *Heap-Überlauf*, dem `OutOfMemoryError`.

c) Falsch. Stacks arbeiten nach dem *Last-In-First-Out*-Prinzip.

d) Falsch. Lokale Variablen werden auf dem Stack gespeichert (vgl. *a)*).

e)
```
public int a(int b) {
    if (b <= 0) return b;  // für b <= 0 nie Fehler
    if (b == 0) return 1 / b;  // wird nie aufgerufen
    return a(b--);  // ruft a(b) auf (mit altem Wert von b)
}
```

Der Aufruf $a(b)$ führt für ...	**Wahr**	**Falsch**
... $b < 0$ *nie* zu einem Fehler.	☒	☐
... $b > 0$ *nie* zu einem Fehler.	☐	☒
... $b > 0$ *immer* zu einem Fehler. → `StackOverflowError`	☒	☐
... $b = 0$ *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. → evtl. `OutOfMemoryError`	☐	☒
... $d > 0$ *nie* zu einem Fehler.	☐	☒
... $d > 0$ *immer* zu einem Fehler. → evtl. `StackOverflowError`	☐	☒
... $d = 0$ *nie* zu einem Fehler.	☒	☐

g) Folgender Code ...

... wertet beim Aufruf von `f()` zu 6 aus. → `void`	☐	☒
... ändert den Wert der Membervariable `x`.	☐	☒
... führt beim Aufruf von `f()` zu einem Laufzeitfehler.	☐	☒
... kompiliert nicht. → Was nicht kompiliert, kann nicht ausgeführt werden	☒	☐

```
public class DynStat {
    int x;
    static void f() {
        this.x = 6;  // Versuchter Zugriff auf ein Attribut
    }                // aus einem statischen Kontext
}
```

71

☐ `public class A extends X { }` (Abstr.) Klasse kann nur (abst.) Klasse erweitern

☐ `class B implements X { }` `B` implementiert nicht die Methoden `x()` und `y(X)`.

☑ `abstract class C implements X { }` Abst. Klasse muss abst. Meth. nicht impl.

☐ `interface D implements X { }` Interface kann kein Interface implementieren

☑ `interface E extends X { }` Interface kann von Interface erben (nicht von Klasse)

☑
```
class F implements X {
    public int x() { return 0; };
    public void y(X x) { };
}
```
Korrekte Implementierung einer Klasse, die ein Interface implementiert, indem alle abstrakten Methoden überschrieben werden.

☑
```
class G extends Object implements X {
    public int x() { return 0; };
    public void y(X x2) { return; };
    int z() { return 1; };
}
```
`extends Object` ist ohne Angabe eines *extends* implizit. Sonst wie `F`. Name eines Parameters irrelevant. Zusätzliche Methoden dürfen implementiert werden

☐
```
abstract class H implements X {
    private int x() { return 1; }
    public void y(X x) { };
}
```
Sichtbarkeit darf nicht verringert werden. Wäre möglich.

☑
```
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() { }
}
```
Wie `H`. Interface-Objektethoden sind automatisch `public` und `abstract`! `x()` wäre hier jedoch *package-private*.

☑
```
abstract class K implements X {
    public abstract int x();
    public abstract void y(X k);
}
```
Abst. Klassen können abst. Methoden haben.
Genau gleich wie in `X` (dort implizit `public ab.`)

☑
```
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;
    }
}
```
Implementiert die beiden verlangten Methoden.
Zusätzliche Methode, die das Attr. `a` aus `L2` zurückgibt.
Hier kann es eine `ClassCastException` (Laufzeit) geben, wenn der Parameter `a` nicht wirklich ein Objekt der Klasse `L2` ist.

☐
```
abstract class M implements X {
    public int x();
}
```
Methode nicht als `abstract` deklariert, muss also einen Rumpf (Implementierung) besitzen (sonst wäre es richtig).

☑
```
abstract class N implements X {
    abstract void y(N n);
    abstract void z();
}
```
Methoden aus `X` müssen nicht implementiert werden, da `N` `abstract` ist. Die Methoden `y(N)` und `z()` sind zusätzliche Definitionen, wobei `y(N)` die Methode `y(X)` überlädt.

☐
```
public abstract class O implements X {
    abstract void y(X x);
    public int x() { return 1; }
}
```
Analog zu `J`:
Sichtbarkeit von `y` darf nicht eingeschränkt werden.

☑
```
abstract class P1 implements X {
    public int x() { return 0; }
}
class P2 extends P1 {
    public void y(X x) { };
}
```
Korrekte Abstrakte Klasse, die eine der beiden abstrakten Methoden mit einer Implementierung überschreibt (die andere bleibt `abstract`).
Muss nur noch die Methode `y()` implementieren, da diese in `P1` (durch das `implements`) `abstract` ist.

☑
```
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); }
}
```
Wie `P1`, nur mit zusätzlicher Methode, welche `y(X)` (bleibt abstrakt) überlädt.
`implements` wäre durch `Q1` implizit.
Implementierung sowieso notwendig.

☐
```
abstract class R implements X {
    abstract void r();
    public static void y(X x);
}
```
Statische Methode kann nicht abstrakt sein, muss also implementiert sein! Würde aber auch dann nicht funktionieren, da man `y(X)` bereits als Objektmethode definiert hat.

☑
```
public interface S extends X {
    int w();
    static void z() { };
}
```
Interface kann Interface um weitere abstrakte Methoden (ohne Impl.) und statische Methoden (mit Implementierung) erweitern.

☐
```
interface T1 extends X {
    int x();
}
class T2 implements T1 {
    public int x() { return 0; }
}
```
Wäre möglich (erbt alles von X).
Bringt nichts (bereits in X definiert)
Muss y(X) implementieren!

☑
```
class U implements X {
    public int x() { return 1; }
    public void y(X x) { y((X)this); }
}
```
wäre auch ohne Cast möglich

☐
```
abstract class V implements X {
    public int x() { return this.x(); }
}
class V2 extends V {
    public void y(V x) { }
}
```
Passt (hat alle abstrakten Methoden aus X).
Muss y(X) implementieren!

☑
```
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(); }
    }
}
```
Muss nichts impl., da abstrakt.
Muss x() und y(X) implementieren (denn W impl. X)
Überladung

☐
```
abstract class x implements X {
    public abstract char x();
    void y() {}
}
```
Klassenname wäre in Ordnung, aber Rückgabetyp muss identisch sein!
Wäre in Ordnung (Überladung)

☐
```
abstract class Y implements X {
    protected int x = 5;
    int x() { return x; }
}
```
Sichtbarkeit darf nicht zu *package-private* reduziert werden (wie in J).

☑
```
interface X2 {
    public abstract int x();
}
class Z implements X, X2 {
    public int x() { return -1; }
    public void y(X x) { }
}
```
Muss alle Methoden aus X und X2 implementieren.
In X und X2 definiert (beide abgedeckt)
In X2 definiert.

72

```java
public class Tuple {
    private final short x, y;

    public Tuple(short x, short y) {
        if (x < 0)
            throw new IllegalArgumentException();
        this.x = x;
        this.y = y;
    }

    public char combinedValue() {
        return (char) (x << 8 | y);
    }

    public void applyTo(char[] chars, boolean encode) {
        for (int i = 0; i < chars.length; i += (x+1))
            if (encode)
                chars[i] += (y+1);
            else
                chars[i] -= (y+1);
    }

    public static Tuple createTupleFromChar(char combined) {
        short x = (short) (combined >> 8);
        short y = (short) (combined & 0xFF);
        return new Tuple(x, y);
    }
}
```

```java
public class Key {
    private final Tuple[] parts;

    public Key(int length) {
        parts = new Tuple[length];
    }

    public String toString() {
        String result = "";
        for (Tuple part : parts)
            result += part.combinedValue();
        return result;
    }

    public void applyTo(char[] chars, boolean encode) {
        for (int i = 0; i < parts.length; i++)
            parts[i].applyTo(chars, encode);
    }

    public static Key createKeyFromString(String str) {
        Key key = new Key(str.length());
        for (int i = 0; i < str.length(); i++)
            key.parts[i] = Tuple.createTupleFromChar(str.charAt(i));
        return key;
    }
}
```

```java
public class Cryptograph {
    private final char[] text;
    private final Key key;

    public Cryptograph(String text, String key) {
        if (text == null || key == null)
            throw new IllegalArgumentException();

        //this.text = text.toCharArray();
        char[] chars = new char[text.length()];
        for (int i = 0; i < text.length(); i++)
            chars[i] = text.charAt(i);
        this.text = chars;

        this.key = Key.createKeyFromString(key);
    }

    public Cryptograph(String text, int keyLength) {
        this(text, createRandomString(keyLength));
    }
```

Fortsetzung auf der nächsten Seite...

```
    public Cryptograph(String text) {
        this(text, text.length() / 5);
    }

    public String getText() {
        String text = "";
        for (int i = 0; i < this.text.length; i++)
            text += this.text[i];
        return text;
    }

    public String getKey() {
        return key.toString();
    }

    public void encode() {
        key.applyTo(text, true);
    }

    public void decode() {
        key.applyTo(text, false);
    }


    public static String createRandomString(int length) {
        String str = "";
        while (length > 0) {
            str += (char) (1 + Character.MAX_VALUE * Math.random());
            length--;
        }
        return str;
    }


    public static void main(String[] args) {
        // Test:
        Cryptograph crypt = new Cryptograph("Beispiel Helloäöü", 200);
        System.out.println(crypt.getText());
        System.out.println(crypt.getKey());
        crypt.encode();
        System.out.println("Verschlüsselt: " + crypt.getText());
        crypt.decode();
        System.out.println("Entschlüsselt: " + crypt.getText());
    }
}
```

73

A: ☐ Die Definition des Konstruktors darf den Typparameter nicht enthalten, also müsste es stattdessen `public A()` heißen.

B: ☑ Der generische Typ kann in der Klasse mit einigen Einschränkungen (Instanziierung, Array) verwendet werden.

C: ☐ In den spitzen Klammern muss mindestens ein Typparameter angegeben werden. Der Typparameter ist ein Platzhalter für eine Klasse (analog sind Methodenparameter Platzhalter für Objekte bzw. Werte). Er kann beliebig heißen, wird aber i. d. R. großgeschrieben und meist `T` (für Typ) oder `E` (für Elementtyp) genannt.

D: ☑ Im Konstruktor von `D` wird ein weiteres `D`-Objekt erzeugt, wobei für den Typparameter `T` dort `Object` eingesetzt wird. Es wäre bspw. auch `new D<Integer>()` möglich. Beachte, dass dieses Beispiel zwar kompiliert, allerdings wird es nie möglich sein, ein Objekt des Klasse `D` zu erzeugen, da der Konstruktor „endlosrekursiv" ist, d. h. bei der Instanziierung eines `D`s erhalten wir immer einen `StackOverflowError`. Es macht nur sehr selten Sinn, im Konstruktor den Konstruktor selbst aufzurufen, d. h. rekursive Konstruktoren sind zwar möglich, aber so gut wie nie sinnvoll (wenn dann aber bitte mit Abbruchbedingung, nicht so wie hier).

E: ☑ Der Typparameter heißt hier `Example` statt `T` – ich kann ihn nennen wie ich will. Wie Variablen- und Klassennamen darf aber auch der Typparametername nicht mit einer Ziffer beginnen und nur aus Buchstaben, Ziffern, Unterstrichen und Währungssymbolen bestehen. Die Zuweisung `e = e` ist möglich. Sie weist den Wert des Parameters `e` an den Parameter `e` zu, tut also gar nichts. Das Attribut `e` wird immer den Wert `null` haben.

F: ☐ Es ist nicht möglich, ein Objekt des Klasse des Typparameters zu erzeugen. So etwas hätte man manchmal gerne, allerdings ist ja nicht klar, ob die tatsächlich eingesetzte Klasse überhaupt einen Konstruktor mit dieser Signatur bereitstellt!

G: ☑ Mehrere Typparameter sind möglich.

H: ☑ Beachte, dass der Typparameter `H1` nichts mit der Klasse `H1` zu tun hat. Es ist extrem unschön, einen Typparameter so zu nennen wie eine andere Klasse. Der Typparameter `H1` würde in diesem Fall die Klasse `H1` verschatten, ich kann also bspw. trotzdem `new H2<Integer>()` schreiben, um `Integer` für `H1` einzusetzen.

I: ☐

J: ☑ Statt `Float` könnte man auch `T` bei `J1` einsetzen, um den generischen Typ „durchzureichen". Aber auch ohne den Typparameter `T` wäre `J2` eine gültige Klasse, d. h. eine nicht generische Klasse kann von einer generischen Klasse erben, sofern ein fixer generischer Typ eingesetzt wird.

```java
import java.util.List; // nicht nötig, aber schöner (Interface)
import java.util.LinkedList;

public class Collection<M extends Medium> {
    private List<M> lib;

    public Collection() {
        lib = new LinkedList<>();
    }

    // 1 a)
    public void add(M m) {
        if (!lib.contains(m))
            lib.add(m);
    }

    // 1 b)
    public M get(long id) {
        for (M m : lib)
            if (m.getIdNr() == id)
                return m;
        return null;
    }

    // 1 c)
    public boolean delete(M m) {
        return lib.remove(m);
    }

    // 2)
    public List<M> getObjectsWithFirstLetter(char initial) {
        List<M> foundMedia = new LinkedList<>();
        for (M m : lib) {
            String title = m.getTitle();
            // Wir vermeiden Fehlermeldungen bei charAt, indem wir zuvor
            // sicherstellen, dass der Titel überhaupt existiert
            // (kein null-Pointer und kein leerer String)
            if (title != null && title.length() > 0
                              && title.charAt(0) == initial) {
                foundMedia.add(m);
            }
        }
        return foundMedia;
    }
}
```

75

☑ `X<A> a = new X<A>();` → A

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

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

☑ `X<A> d = new X<>(new D());` → D

☑ `X<C> e = new X<C>();` → A

☐ `X<T> f = new X<>();`

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

☐ `X<A> h = new X<C>();`

☑ `X<D> i = new X<D>(new D());` → D

☐ `X<C> j = new X<>(new B());`

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

☑ `new X<A>(new C());` → C

☐ `X<Object> m = new X<>();`

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

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

☑ `class P<T extends B> extends X<T> {}` → keine Ausgabe

☐ `class Q<T> extends X<T> {}`

☑ `class R<T extends B> extends X<T> {}; new R<>();` → A

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

☑ `class T<T extends D> extends X<T> {}` → keine Ausgabe

☐ `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<>();
```
→ erst wird A, dann eine Referenz auf X (z. B. `X@6d06d69c`) ausgegeben

76

Jede Fehlerklasse (wie z. B. `RuntimeException`) bietet standardmäßig mehrere Konstruktoren an, u. a. einen Konstruktor ohne Parametern (nicht aussagekräftig) und einen mit einem Fehlertext als Parameter, welchen wir hier benutzen. Da `MatrixException` von `RuntimeException` erbt, muss sie auch einen `RuntimeException`-Konstruktor aufrufen (durch `super()` in der ersten Zeile jedes Konstruktors). Dort übergeben wir den Fehlertext, der ausgegeben wird, wenn die Exception auftritt und nicht gefangen wird. Wird die Exception hingegen in einem `catch`-Block gefangen, dann sehen wir den Fehlertext nicht... Um dann trotzdem Zugriff auf die fehlerverursachenden Werte zu haben, nutzen wir in den Fehlerklassen Attribute und Getter.

```
public class MatrixException extends RuntimeException {
    public MatrixException(String msg) {
        super(msg);   // Fehlertext an die Oberklasse weitergeben

        // super(...) ist immer das erste, was in jedem Konstruktor passieren muss.
        // Wird dieser Aufruf nicht angegeben, so ist super() implizit. Die Oberklasse
        // muss dann aber auch einen solchen Konstruktor (ohne Parameter) anbieten.
    }
}
```

```
public class InvalidMatrixIndexException extends MatrixException {
    private int index;

    public InvalidMatrixIndexException(int index) {
        super("index " + index + "does not exist");
        this.index = index;
    }

    public int getIndex() {
        return index;
    }
}
```

```
public class InvalidRowIndexException extends InvalidMatrixIndexException
{
    public InvalidRowIndexException(int rowIndex) {
        super(rowIndex);
    }

    // Hier ist kein Attribut/Getter nötig, da die Klasse ja von
    // InvalidMatrixIndexException und damit auch getIndex() erbt.
}
```

```java
public class InvalidColumnIndexException
                                    extends InvalidMatrixIndexException {
    public InvalidColumnIndexException(int colIndex) {
        super(colIndex);
    }
}
```

```java
public class IllegalMatrixSizeException extends MatrixException {
    private int rows, columns;

    public IllegalMatrixSizeException(int rows, int columns) {
        super("cannot instantiate " + rows + "x" + columns + " matrix");
        this.rows = rows;
        this.columns = columns;
    }

    public int getAmountRows() {
        return rows;
    }
    public int getAmountColumns() {
        return columns;
    }
}
```

77

a) `A B1 B2 H I`

b) `A B1 C H I`

c) `A B1 E H`

d) `A B1 D1 D3 H I`

e) `A B1 G1 G2 H I`

f) `A B1 D1 H` (dann unbehandelte `NullPointerException`)

g) `A B1 G1 H` (dann unbehandelte `ClassCastException`)

h) `A B1 F1 Infinity F2 H` (dann unbehandelte `ArithmeticException`)

i) `A B1 H` (dann `StackOverflowError`)

78

```java
public static String readFirstLine(String filePath) {
    // BufferedReader deklarieren, damit später sichtbar (Scope):
    BufferedReader reader;
    // BufferedReader öffnen:
    try {
        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
    } finally {
        // Den BufferedReader in jedem Fall (→ finally) (auch bei
        // IOException) wieder schließen. Dazu muss er geöffnet sein,
        // daher kann man dieses try-catch nicht mit dem erstem try-catch
        // zusammenfassen (weil er dort nicht geschlossen werden kann).
        try {
            reader.close();
        } catch (IOException e) {
            // Hier sind wir leider wirklich machtlos...
            // Wir müssen sie aber fangen (alternativ throws-Klausel)...
        }
    }
}
```

79

```java
public abstract class Dateisystemelement {
    private String name;

    public Dateisystemelement(String name) {
        this.name = name;
    }

    public void umbenennen(String neuerName) {
        name = neuerName;
    }

    @Override
    public String toString() {
        return name + " (" + groesse() + "B)";
    }

    @Override
    public boolean equals(Object o) {
        if (o instanceof Dateisystemelement)
            return this.name.equals( ((Dateisystemelement) o).name );
        return false;  // Vergleichsobjekt ist kein Dateisystemelement
    }

    public abstract int groesse();
}
```

```java
public abstract class Datei extends Dateisystemelement {
    public Datei(String name) {
        super(name);
    }

    @Override
    public boolean equals(Object o) {
        if (o instanceof Datei)
            return super.equals((Datei) o);
        return false;  // Vergleichsobjekt ist keine Datei
    }
}
```

```java
public class Textdokument extends Datei {
    private String text;

    public Textdokument(String name, String inhalt) {
        super(name);
        text = inhalt;
    }

    @Override
    public int groesse() {
        return 2 * text.length();   // char = 16 bit = 2 Byte
    }

    public void anfuegen(String text) {
        this.text += text;
    }
}
```

```java
public class Binärdatei extends Datei {
    private byte[] inhalt;

    public Binärdatei(String name, byte[] inhalt) {
        super(name);
        this.inhalt = inhalt;
    }

    @Override
    public int groesse() {
        return inhalt.length;
    }
}
```

```java
public class Ordner extends Dateisystemelement {
    private Dateisystemelement[] elemente;

    public Ordner(String name) {
        super(name);
        elemente = new Dateisystemelement[0];
    }

    public Dateisystemelement[] getElemente() {
        return elemente;
    }
```

Fortsetzung auf der nächsten Seite...

```java
@Override
public boolean equals(Object o) {
    if (o instanceof Ordner)
        return super.equals((Ordner) o);

    return false;  // Vergleichsobjekt ist kein Ordner
}

@Override
public int groesse() {
    int groesse = 0;

    for (Dateisystemelement e : elemente)
        groesse += e.groesse();

    return groesse;
}

private int indexOf(Dateisystemelement e) {
    for (int i = 0; i < elemente.length; i++)
        if (elemente[i].equals(e))  // Element gefunden
            return i;

    // Wenn nie etwas zurückgegeben wurde (nichts gefunden) dann:
    return -1;
}

public void einfuegen(Dateisystemelement e) {
    int index = indexOf(e);

    if (index == -1) {  // → noch nicht enthalten
        // Erzeuge neues Array mit einem freien Speicherplatz für das neue Element:
        Dateisystemelement[] elementeNeu =
                        new Dateisystemelement[elemente.length + 1];

        // Kopiere alle Elemente aus altem Array in neues Array (alternativ Schleife):
        System.arraycopy(elemente, 0,
                         elementeNeu, 0, elemente.length);

        // Neue erzeugtes Array im Attribut speichern (altes Array wird verworfen):
        elemente = elementeNeu;

        // Neues Element an die letzte Stelle im Array schreiben:
        elemente[elemente.length - 1] = e;
    } else {  // → bereits enthalten
        elemente[index] = e;  // überschreiben
    }
}
```

Fortsetzung auf der nächsten Seite...

```
    // Pfad bspw. "Bilder/TUM/" oder "" (dieses Verzeichnis)
    public void einfuegen(Dateisystemelement e, String ordnerPfad) {
        if (ordnerPfad.length() == 0 || ordnerPfad.equals("/")) {
            // → Pfad ist "/" oder ""
            this.einfuegen(e);  // hier einfügen
            return;
        }

        int slashIndex = ordnerPfad.indexOf('/');  // z. B. Position 6
        String ordnerName = ordnerPfad.substring(0, slashIndex);
        String verbleibenderPfad = "";
        if (slashIndex != ordnerPfad.length()-1) // weitere Ordner folgen
            verbleibenderPfad = ordnerPfad.substring(slashIndex+1);

        Ordner ordner = new Ordner(ordnerName);  // Unterordner-Objekt
        int ordnerIndex = indexOf(new Ordner(ordnerName));
        if (ordnerIndex == -1)  // → der Unterordner existiert noch nicht
            this.einfuegen(ordner);  // erzeugten Unterordner einfügen
        else  // → der Unterordner existiert
            // speichere das bereits enthaltene Ordner-Objekt:
            ordner = (Ordner) elemente[ordnerIndex];

        ordner.einfuegen(e, verbleibenderPfad);
    }
}
```

```
public class Dateisystem {

    private Ordner wurzel;

    public Dateisystem() {
        wurzel = new Ordner(".");
    }

    public void einfuegen(Datei d, String pfad) {
        wurzel.einfuegen(d, pfad);
    }
}
```

80

1. Da sich Dateien und Ordner in der Art ihrer Formatierung unterscheiden, benötigt der Visitor eine `visit`-Methode für Ordner und eine für Dateien. Wir benötigen aber nicht eine eigene `visit`-Methode pro Dateityp, da die Formatierung von Dateien immer gleich ist (Dateiname). Das Interface gibt daher folgende Methoden an:

```
public interface Visitor {
    void visit(Ordner o);
    void visit(Datei d);
}
```

2. Wir erweitern die Element-Klassen um die nötigen `accept`-Methoden. Auch hier müssen wir nicht zwischen Binärdatei und Textdokument unterscheiden (könnten aber, sofern die Methode trotzdem in `Datei` definiert ist (könnte dann abstrakt sein)):

```
public abstract class Dateisystemelement {

    // ... (alter Dateisystemelement-Code)

    public abstract void accept(Visitor v);
}
```

```
public class Ordner extends Dateisystemelement {

    // ... (alter Ordner-Code)

    @Override
    public void accept(Visitor v) {
        v.visit(this);
    }
}
```

```
public abstract class Datei extends Dateisystemelement {

    // ... (alter Datei-Code)

    @Override
    public void accept(Visitor v) {
        v.visit(this);
    }

    // Die Implementierung von accept wäre auch in allen Unterklassen
    // möglich, ist aber hier gar nicht nötig, da wir nur eine visit-
    // Methode haben. Für visit(KlasseX) benötigen wir also nur eine
    // accept-Methode in KlasseX.
}
```

3. Im `FormatVisitor` findet die tatsächliche Implementierung statt.
4. Wir müssen uns die Einrückung bzw. Einrückungstiefe speichern:

```
public class FormatVisitor implements Visitor {
    private String result = "";
    private int indentDepth = -1;  // Einrückungstiefe

    public String getFormattedString() {
        return result;
    }

    @Override
    public void visit(Datei file) {
        result += file + "\n";  // ruft file.toString() auf
    }

    @Override
    public void visit(Ordner folder) {
        result += folder + "\n";  // ruft folder.toString() auf

        indentDepth++;
        for (Dateisystemelement e : folder.getElemente()) {
            addIndent();
            result += "|__ ";
            e.accept(this);
        }
        indentDepth--;
    }

    private void addIndent() {
        for (int i = 0; i < indentDepth; i++) {
            result += "|   ";
        }
    }
}
```

5. Jetzt können wir den Visitor benutzen, um ein Dateisystem in einen formatierten `String` umzuwandeln, d. h. wir ergänzen `Dateisystem` um die `toString`-Methode:

```
public class Dateisystem {

    // ... (alter Dateisystem-Code)

    @Override
    public String toString() {
        FormatVisitor fv = new FormatVisitor();
        wurzel.accept(fv);
        return fv.getFormattedString();
    }
}
```

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

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

    public Liste(int e, Liste naechster) {
        wert = e;
        this.naechster = naechster;
    }

    public void einfuegen(int wert) {
        if (naechster == null)
            naechster = new Liste(wert);
        else
            naechster.einfuegen(wert);
    }

    public Liste entfernen(int wert) {
        // Damit sich ein Element selbst entfernen kann, müsste es den Nachfolger seines
        // Vorgängers auf seinen Nachfolger setzen (prev.next = next). In einer doppelt
        // verketten Liste wäre das einfach; hier ist das aber schwierig, weil ein Element
        // seinen Vorgänger nicht kennt. Daher ist die Idee, dem Vorgänger (welcher die
        // entfernen-Methode auf „uns“ aufgerufen hat) „uns“ (this) zurückzugeben, falls
        // sich nichts ändert, und den Nachfolger (naechster), wenn wir uns löschen soller

        if (wert == this.wert) // aktuelles muss entfernt werden
            return naechster;

        if (naechster == null) // beim letzten angekommen (nichts entfernt…)
            return this;

        naechster = naechster.entfernen(wert); // delegiere an den Nachfolger
        return this; // aktueller bleibt Nachfolger des nächsten
    }

    @Override  @Override ist ein optionaler Hinweis auf Überschreibung
    public String toString() {
        if (naechster == null)  // Abbruchbedingung (letztes Element)
            return "" + wert;

        return wert + ", " + naechster.toString();
    }
```

Fortsetzung auf der nächsten Seite...

```
    @Override
    public boolean equals(Object o) {
        if (o.getClass() != this.getClass()) // oder !(o instanceof Liste)
            return false;

        if (this.wert != ((Liste)o).wert) // Elemente (Inhalte) verschieden
            return false;

        if (this.naechster == null || ((Liste)o).naechster == null)
            // → mindestens eine der beiden Listen ist zu Ende
            if (naechster == null && ((Liste)o).naechster == null)
                // → beide Listen sind zu Ende → Listen sind gleich
                return true;
            else
                // → eine Liste ist zu Ende, die andere nicht → Listen ungleich
                return false;

        return (this.naechster.equals(((Liste)o).naechster));
    }

    // Beispiel:
    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);  // 4, 2, 5

        Liste y = new Liste(1);
        y.einfuegen(4);
        y.einfuegen(2);
        y.einfuegen(5);
        System.out.println(y);  // 4, 2, 5

        System.out.println(x.equals(y));  // true
    }
}
```

```java
public class List<E> {

    private Entry<E> first;  // vorderster Listenknoten

    private class Entry<E> {  // E könnte anders genannt werden (neuer Parameter)
        E elem;  // Inhalt des Knotens (das tatsächliche Element)
        Entry<E> next;  // Nachfolger-Knoten (selber generischer Typ)

        public Entry(E e) {
            elem = e;
            next = null;  // optional, weil implizit
        } // weitere Konstruktoren möglich...
    }

    public void add(E e) {
        Entry<E> toBeAdded = new Entry<>(e);  // neuen Knoten erzeugen
        if (first == null) {
            first = toBeAdded;  // erstes eingefügtes Element
        } else {
            Entry<E> current = first;  // starte Suche beim ersten Knoten
            while (current.next != null)  // letzten Knoten iterativ finden
                current = current.next;
            current.next = toBeAdded;  // Nachfolger des letzten ist neuer Knoten
        }
    }

    public E removeFirst() {
        if (first == null)  // kann nichts entfernen ☹
            return null;
        E e = first.elem;  // zu löschendes Element für die Rückgabe speichern
        first = first.next;  // Knoten löschen (Garbage Collection übernimmt das)
        return e;  // gelöschtes Element zurückgeben
    }

    public int size() {
        int size = 0;
        Entry<E> current = first;  // starte beim ersten Knoten
        while (current != null) {  // gehe über alle (bis bei null angekommen)
            size++;
            current = current.next;
        }
        return size;
    }
```

Fortsetzung auf der nächsten Seite...

```
public String toString() {
    String str = "";   // hierüber wird das Ergebnis iterativ aufgebaut

    if (first != null)   // erstes Element anfügen (ohne Komma)
        str = first.elem.toString();

    // Jetzt alle folgenden Elemente kommagetrennt hinten anhängen:
    Entry<E> current = first.next;   // starte beim zweiten Knoten
    while (current != null) {   // solange noch ein Knoten existiert
        str += ", " + current.elem.toString();  // 1, 2, 3
        current = current.next;   // zum nächsten Knoten gehen
    }

    return str;
}


public boolean remove(Object o) {
    // Um das übergebene Element aus der Liste zu löschen, müssen wir über die Liste
    // iterieren. Wenn wir das zu löschende Element finden (hier Prüfung auf Referenz-
    // gleichheit), müssen wir den Nachfolger dessen Vorgängers auf dessen Nachfolger
    // setzen. Da es sich aber um eine einfach verkettete Liste handelt, kennt ein Knoten
    // seinen Vorgänger nicht. Daher definieren wir zusätzlich noch eine Variable zur
    // Speicherung des Vorgängers, welche wir aktualisieren, bevor wir mit current
    // weitergehen. So speichert prev immer das „alte“ current.

    Entry<E> current = first;
    Entry<E> prev = null;

    while(current != null) {
        if (current.elem == o) {   // → zu löschendes Element gefunden
            if (prev == null)   // → wir sind dabei, das erste zu entfernen
                first = first.next;
            else
                prev.next = current.next;

            return true;
        }

        // else: nicht gefunden, also weitergehen

        prev = current;
        current = current.next;
    }

    return false;  // Schleife verlassen, d. h. bis zum Ende nie gefunden ☹
}
```

Fortsetzung auf der nächsten Seite...

```
    public boolean equals(Object o) {

        if (this == o)   // ein und dasselbe Objekt → muss gleich sein…
            return true;

        if (!(o instanceof List))   // anderer Typ → kann nicht gleich sein…
            return false;

        // Um die beiden Listen auf Gleichheit zu prüfen, müssen wir gleichzeitig über
        // beide Listen iterieren. Das Iterieren muss gleichzeitig passieren, d. h. wir haben
        // hier keinesfalls eine geschachtelte Schleife, sondern nur eine einzelne Schleife.
        // Für jede Liste merken wir uns, wo wir stehen, und gehen immer gleichzeitig zum
        // jeweils nächsten Element. Sind die Elemente irgendwann nicht mehr gleich, so
        // sind die Listen unterschiedlich, und wir geben false zurück (*1). Erreicht eine
        // Liste das Ende, bevor die andere zu Ende ist, so geben wir false zurück (*2).
        // Stehen am Ende beide Liste am Ende (bei null), so geben wir true zurück (*3).
        // In dieser Implementierung wird bei *3 gleichzeitig der Fall abgedeckt, das seine
        // eine der Listen komplett leer ist (dann kommen wir nicht in die Schleife).

        Entry<E> currentThis = this.first;
        Entry<E> currentOther = ((List<E>) o).first;

        while (currentThis != null && currentOther != null) {
            if (currentThis.elem == null || currentOther.elem == null) {
                if (currentThis.elem != currentOther.elem)
                    return false; // *2
            }
            else if (!currentThis.elem.equals(currentOther.elem)) {
                return false; // *1
            }

            currentThis = currentThis.next;
            currentOther = currentOther.next;
        }

        return currentThis == currentOther; // *3

    }
}
```

```java
import java.util.Iterator;
import java.util.NoSuchElementException;

public class List<E> implements Iterable<E> {

    // ... (Methoden aus der vorherigen Aufgabe)

    @Override
    public Iterator<E> iterator() {
        return new MyListIterator<>(first);
    }

    // Implementierung des Iterators als innere Klasse:
    public class MyListIterator<E> implements Iterator<E> {
        // Hinweis: Das E hier ist ein eigenständiger Typ und hat nichts
        // mit dem E von List zu tun (verschattet diesen). Das E könnte
        // für diese Klasse auch beliebig anders festgelegt werden.

        private Entry<E> current; // speichert das nächste Element

        public MyListIterator(Entry<E> first) {
            current = first;
        }

        @Override
        public boolean hasNext() {
            return current != null;
        }

        @Override
        public E next() {
            if(!hasNext())
                throw new NoSuchElementException();
            E e = current.elem;
            current = current.next;
            return e;
        }
    }
```

Fortsetzung auf der nächsten Seite...

```
    // Test:
    public static void main(String[] args) {
        class Color {
            public String name = "transparent";
            public Color(String n) { name = n; }
            public String toString() { return name; }
        }


        List<Color> colorList = new List<>();  // Farbobjekt-Liste
        colorList.add(new Color("green"));
        colorList.add(new Color("red"));
        colorList.add(new Color("orange"));

        Color blue = new Color("blue");
        colorList.add(blue);

        System.out.println("Ausgabe mit blue: " + colorList);

        colorList.remove(new Color("blue"));
        // entfernt blue nicht, da ein neues Objekt übergeben wurde != blue
        System.out.println("Ausgabe mit blue: " + colorList);

        colorList.remove(blue); // entfernt blue (→ Referenzgleichheit)
        System.out.println("Ausgabe ohne blue: " + colorList);


        // Iterator-Test:

        Iterator<Color> iter = colorList.iterator();
        while (iter.hasNext()) {
            Color c = iter.next();  // next() niemals mehrfach aufrufen!
            System.out.println("Farbe: " + c);
        }

        // bzw. for-each (durch implements Iterable<Color> nun möglich)
        for (Color c : colorList) {
            System.out.println("Farbe: " + c);
        }

    }

}
```

Anstelle eines `Object`-Arrays könnte man auch tatsächlich ein generisches Array verwenden, allerdings müsste man dieses dann mit einem gecasteten `Object`-Array initialisieren, d. h.:

`private E[] elements;` *und* `elements = (E[]) new Object[maxSize];`

```
import java.util.Iterator;
import java.util.NoSuchElementException;

public class SimpleStack<E> implements Iterable<E> {

    private Object[] elements;   // wird nur E-Elemente speichern; final optional
    private int stackPointer;   // zeigt auf die erste freie Position im Array (Index)

    public SimpleStack(int maxSize) {
        if (maxSize <= 0)
            throw new InvalidSimpleStackSizeException(maxSize);

        elements = new Object[maxSize];
        stackPointer = 0;
    }

    public void push(E element) {
        if (stackPointer == elements.length)   // kein Platz mehr...
            throw new SimpleStackOverflowException(elements.length);

        elements[stackPointer] = element;   // Element im Array speichern
        stackPointer++;
    }

    public E pop() {
        if (stackPointer == 0)   // kann nichts entnehmen...
            throw new SimpleStackEmptyException();

        Object retElem = elements[stackPointer];   // Rückgabe-Element holen
        stackPointer--;
        return (E) retElem; // Cast wird funktionieren, weil wir nur E's einfügen

        // „elements[stackPointer] = 0“ ist nicht nötig (aber auch nicht falsch),
        // denn es ist egal, ob im Array nun 0, -1, 25, 16283 oder whatever gespeichert ist
    }

    public void clear() {
        stackPointer = 0;   // erste freie Position = ganz unten → alles frei
        // Sonst ist nichts zu tun. Was auf dem Stack liegen bleibt ist egal (wie bei pop())
    }

    public int size() {
        return stackPointer;   // Anzahl der Elemente im Stack
    }
```

Fortsetzung auf der nächsten Seite...

```
    public boolean isEmpty() {
        return size() == 0;  // oder if (stackPointer == 0) ...
    }

    // Iterator-Implementierung:

    @Override  // @Override-Annotation ist wie immer optional
    public Iterator<E> iterator() {
        return new SimpleStackIterator();
    }

    private class SimpleStackIterator implements Iterator<E> {
        /* SimpleStackIterator könnte auch generisch sein, allerdings würde das nur
        den Typ E verschatten. Es ist gewissermaßen überflüssig, schadet aber auch nicht.
        SimpleStackIterator benötigt keinen generischen Typ, weil Zugriff auf den
        generischen Typ von SimpleStack besteht. Daher können wir das E aus
        SimpleStack verwenden. SimpleStackIterator könnte auch den generischen
        Typ T besitzen. next() müsste dann auch ein T zurückgeben. Wichtig ist nur, dass
        der Iterator in derMethode SimpleStack.iterator() immer mit <E> oder (falls
        wir keinen generischen Typ angeben wie hier) ohne generischen Typ erzeugt wird,
        d. h. in iterator() wird für den Platzhalter hier etwas Konkretes eingesetzt. */

        private int index;  // aktuelle Position im Stack (Array)

        public SimpleStackIterator() {
            index = stackPointer - 1;  // fange beim obersten an

            /* Iterator ist eine innere (nicht-statische) Klasse und hat daher Zugriff auf
            alle Attribute des SimpleStacks. Das Array müssen wir daher ebenfalls nicht
            zwischenspeichern und schon gar nicht kopieren; wir arbeiten auf dem Array
            von SimpleStack. Während man einen Iterator benutzt darf man den Simp-
            leStack sowieso nicht verändern (mittels push oder pop). Das wäre hier zwar
            möglich, allerdings darf der Benutzer nicht davon ausgehen, dass das wie er-
            wartet funktioniert. Es könnte zu unerwarteten Ergebnissen kommen. Collec-
            tions in java.util könnten in so einem Fall eine ConcurrentModificati-
            onException werfen. Die Veränderung einer Collection während einer Itera-
            tor- Benutzung ist nur mit zusätzlichen Iterator-Methoden (z. B. add/remove
            bei Listen) möglich – falls implementiert (sonst nicht). */
        }

        @Override
        public boolean hasNext() {
            return index >= 0;
        }

        @Override
        public E next() {
            if (!hasNext())  // bzw. if (index < 0) ...
                throw new NoSuchElementException();
            return (E) stack[index--];
            // bzw. alternativ wie in pop(): Zuerst Element zwischenspeichern, dann
            // Stack-Pointer reduzieren und zwischengespeichertes Element zurückgeben
        }
    }
```

Fortsetzung auf der nächsten Seite...

```
    // Test (nicht gefordert):
    public static void main(String[] args) {
        SimpleStack<String> s = new SimpleStack<>(3);
        s.push("ich");
        s.push("bin");
        s.pop();
        s.push("war");

        for (String elem : s)   // Iterator wird implizit genutzt
            System.out.print(elem + " ");
        System.out.println();   // → „war ich“

        s.push("dumm");

        for (String elem : s)   // Iterator neu nutzen
            System.out.print(elem + " ");
        System.out.println();   // → „dumm war ich“

        s.push("unmöglich");  // → SimpleStackOverflowException
    }
}
```

```
public class SimpleStackException extends RuntimeException {
    public SimpleStackException(String message) {
        super(message);
    }
}
```

```
public class SimpleStackOverflowException extends SimpleStackException {
    public SimpleStackOverflowException(int maxSize) {
        super("cannot push to full stack (" + maxSize + " elements)");
    }
}
```

```
public class SimpleStackEmptyException extends SimpleStackException {
    public SimpleStackEmptyException() {
        super("cannot pop from stack without any element on it");
    }
}
```

```
public class InvalidSimpleStackSizeException
                                        extends SimpleStackException {
    public InvalidSimpleStackSizeException(int size) {
        super(size + " is not a valid stack size");
    }
}
```

```java
public class Mensch {
    private Mensch vater, mutter;
    private Set<Mensch> kinder;

    // endrekursiv
    public Mensch vorfahr(String folge) {
        if (folge.equals(""))   // = Ich (Rekursionsanker)
            return this;

        char naechsteEbene = folge.charAt(0);
        String restfolge = folge.substring(1);

        switch (naechsteEbene) {
            case 'm':  // mütterlicherseits
                return mutter.vorfahr(restfolge);
            case 'v':  // väterlicherseits
                return vater.vorfahr(restfolge);
            default:  // ungültiges Zeichen
                return null;
        }
    }

    // nicht endrekursiv
    public Set<Mensch> vorfahren(int generation) {
        Set<Mensch> ergebnis = new HashSet<Mensch>(); // oder TreeSet

        if (generation == 0) {   // = Ich (Rekursionsanker)
            ergebnis.add(this);
        } else if (generation < 0) {   // = Nachkommen
            for (Mensch kind : kinder)
                ergebnis.addAll(kind.vorfahren(generation+1));
        } else {   // = Vorfahren
            ergebnis.addAll(vater.vorfahren(generation-1));
            ergebnis.addAll(mutter.vorfahren(generation-1));
        }

        return ergebnis;
    }
}
```

```java
public static Set<Integer> mostFrequentElements(int[] arr) {
    HashMap<Integer,Integer> counts = new HashMap<>(); // Element→Häufigkeit
    // Elemente zählen in der HashMap aufsummieren:
    for (int i = 0; i < arr.length; i++) {
        int element = arr[i];
        if (counts.containsKey(element)) {  // Element zuvor bereits gefunden
            int oldCount = counts.get(element);  // wie oft bisher gefunden?
            counts.put(element, oldCount + 1);  // jetzt einmal mehr gefunden
        } else {
            counts.put(element, 1); // zum ersten Mal gefunden
        }
    }

    // Finde die maximale Häufigkeit:
    int max = 0;  // möglich, weil Häufigkeiten immer > 0
    for (int elem : counts.keySet()) { // über Schlüsselmenge iterieren
        int count = counts.get(elem); // Häufigkeit für dieses Element auslesen
        if (count > max) {
            max = count;
        }
    }

    // Finde die Elemente (Keys) mit der größten Häufigkeit (Value)
    HashSet<Integer> mostFrequent = new HashSet<>();
    for (int elem : counts.keySet()) {
        if (counts.get(elem) == max) {
            mostFrequent.add(elem);
        }
    }

    return mostFrequent;
}
```

Alternativlösung: An die Profis: Ja, es geht auch noch kürzer...

```java
public static Set<Integer> mostFrequentElements(int[] arr) {
    Map<Integer, Integer> counts = new HashMap<>();  // oder TreeMap
    for (int elem : arr) {
        Integer oldCount = counts.get(elem);
        counts.put(elem, oldCount == null ? 1 : oldCount+1);
    }

    int max = 0;
    for (int count : counts.values())
        if (count > max)
            max = count;

    Set<Integer> mostFrequent = new HashSet<>();  // oder TreeSet
    for (Entry<Integer, Integer> entry : counts.entrySet())
        if (entry.getValue() == max)
            mostFrequent.add(entry.getKey());
    return mostFrequent;
}
```

Lösungen und Erklärungen sind rot markiert.
Die blauen Teile sind Zusatzinformationen
(was wäre, wenn es keinen Fehler geben würde).

		Variable	stat. Typ	dyn. Typ

a)	`A a = new A();`
	`A b = new C();`
	`B c = new B();`
		a
		b
		c
			A
			A
			B
				A
				C
				B

b)	`B a = new C();` C nicht an B zuweisbar `A b = new A();`	*Compiler-Fehler* (Zeile 1) (b wäre stat. und dyn. A)		
c)	`I i = new C();` `C a = new D();`	a i	C I	D C
d)	`D d = new D();` `I c = new I();` I nicht instanziierbar	*Compiler-Fehler* (Zeile 2) (d wäre stat. und dyn. D)		

e)	`C b = new C();`
	`D a = new D();`
	`A c = b; b = a;`
		a
		b
		c
			D
			C
			A
				D
				D
				C

f)	`I c = new C(); I d = new D();` `c = d; d = c;`	c d	I I	D D
g)	`A d = new D();` `C c = d;` A nicht immer an C zuweisbar	*Compiler-Fehler* (Zeile 2) (d wäre stat. A und dyn. D)		

h)	`A a = new B();`
	`B b = (B)a;`
	`Object o = (A)b;`
		a
		b
		o
			A
			B
			Object
				B
				B
				B

i)	`C c = new D();`
	`I i = c;`
	`c = (C)i;`
		c
		i
			C
			I
				D
				D

j)	`C a = new C();` `B c = (B)a;` Cast von C zu B nie möglich	*Compiler-Fehler* (Zeile 2) (a wäre stat. und dyn. C)		
k)	`I c = new C();` `C d = c;` I nicht immer an C zuweisbar	*Compiler-Fehler* (Zeile 2) (c wäre stat. I und dyn. C)		
l)	`A a = new D();` `B b = (B)a;` Cast von D zu B nicht möglich	*Laufzeitfehler* (Zeile 2) (a wäre stat. A und dyn. D)		

Fortsetzung auf der nächsten Seite...

m)	`I c = new C();` `C b = (C)c; c = new D();`	b C C c I D
n)	`Object o; B a = new B();` `o = (C)a;` Cast von B zu C nie möglich	*Compiler-Fehler* (Zeile 2) (a wäre stat. und dyn. B)
o)	`A c = new C();` `C a = c;` A nicht immer an C zuweisbar	*Compiler-Fehler* (Zeile 2) (c wäre stat. A und dyn. C)
p)	`Object c, d; c = new D();` `d = (I)c;`	c Object D d Object D
q)	`A a = (C)(A)new D();`	a A D
r)	`B b = (A)new C();` A nicht an B zuweisbar	*Compiler-Fehler*
s)	`I c = new D();` `C d = c;` I nicht immer an C zuweisbar	*Compiler-Fehler* (Zeile 2) (c wäre stat. I und dyn. D)

t)	`A c;`
	`I b = new C();`
	`c = b;` I nicht immer an A zuweisbar
		Compiler-Fehler (Zeile 3)
		(b wäre stat. I und dyn. C)

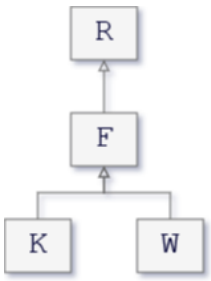
u)	`A b = (D)(A) new C();` C ist kein D	*Laufzeitfehler*

v)	`Object d = new D();`
	`I c = (C) d;`
	`d = c;`
		c I D
		d Object D

w)	`I i = new D();`
	`Object c, a;`
	`c = (A)i; a = (C)c;`
	`c = new A();`
		a Object D
		c Object A
		i I D

x)	`Object d = new D();` `A c = (C)d;`	c A D d Object D

y)	`B c = new B();`
	`A a = c;`
	`C b = (C)a;` Cast von B zu C nicht möglich
		Laufzeitfehler (Zeile 3)
		(a wäre stat. A und dyn. B,
		c wäre stat. und dyn. B)



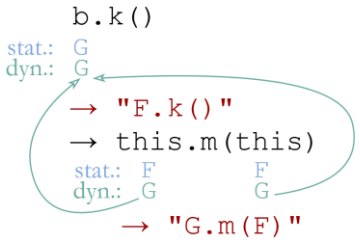
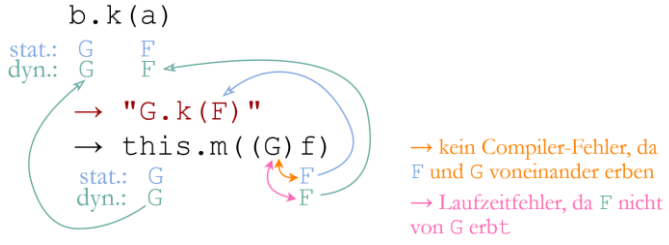
89

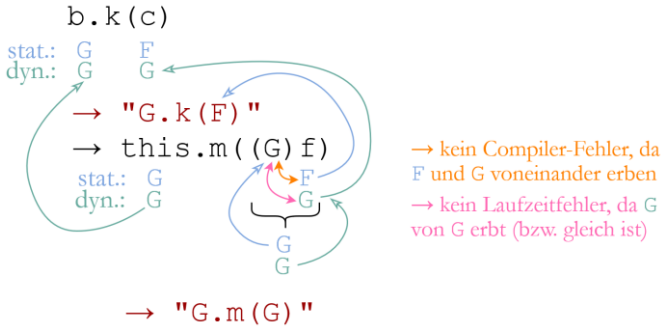
Bei Polymorphie-Aufgaben kommt es nicht selten vor, dass mehrere Aufrufe das gleiche Ergebnis produzieren – so auch bei dieser Aufgabe. Das heißt aber noch lange nicht, dass die Aufrufe identisch sind! Es gibt unterschiedliche Gründe für die Wahl einer bestimmten Methode. Die Lösungen für diese Aufgabe sind bewusst sehr ausführlich, damit du verstehst, wie man einen Aufruf schrittweise auswertet, und dir dieses grundsätzliche Lösungsverfahren in Fleisch und Blut übergeht. Die Lösungen zu den anderen Polymorphie-Aufgaben sind nicht mehr so ausführlich, da das grundlegende Vorgehen immer gleich ist wie hier.

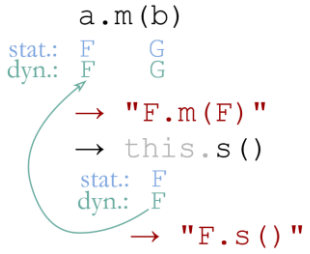
Die Statements zu Beginn erzeugen die für diese Aufgabe benötigten Objekte. So wird mit dem Statement „`G b = new G();`“ eine Variable `b` vom Typ `G` deklariert (wegen `G b` → statischer Typ). Sie speichert ein Objekt vom Typ `G` (wegen `new G()` → dynamischer Typ). Über das Statement „`F c = b;`“ wird eine weitere Variable `c` vom (statischen) Typ `F` deklariert, welche dasselbe Objekt referenziert, d. h. der Typ des Objekts (dynamischer Typ) von `c` ist `G`.

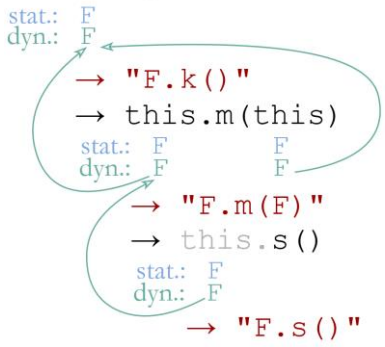
Die Konstruktoraufrufe müssen (wie Methoden) ausgeführt werden. Bspw. wird mit „`new G()`“ der parameterlose Konstruktor der Klasse `G` aufgerufen. Hier könnten bspw. Attribute gesetzt werden. In dieser Aufgabe ist das lediglich bei „`A y = new A(h);`“ der Fall. Dort wird der Konstruktor `A(F)` aufgerufen und durch „`this.m = m;`“ das Attribut `m` auf den Wert von `h` gesetzt, d. h. `y.m` ist `h` (ist `new H()`). Ein Objektdiagramm findest du bei Aufruf 65.

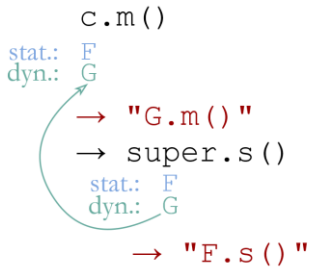
Aufruf	Ergebnis und Erklärung
1. `b.m()`	`G.m()F.s()` → Der statische Typ von `b` ist `G`, d. h. wir suchen in `G` eine Methode `m` ohne Parameter. Wir finden `m()` in `G`. Wir führen diese Methode direkt aus, weil der dynamische Typ von `b` ebenfalls `G` ist. Es folgt die Rückgabe „`G.m()`“ und der Aufruf `super.s()`, also suchen wir in der Oberklasse von `G` (das ist `F`) eine Methode `s` ohne Parameter (anders ausgedrückt: der „statische Typ“ von `super` ist `F`). Wir finden `s()` in `F` und führen diese Methode direkt aus, da wir auf dem Schlüsselwort `super` operieren (außerdem ist die Methode sowieso `static`).
2. `b.m(a)`	`G.m(F)` → Der statische Typ von `b` ist `G`, d. h. wir suchen in `G` eine Methode `m` mit Parameter `F` (stat. Typ von `a`). Wir finden `m(F)` in `G`. Wir führen diese Methode direkt aus, weil der dynamische Typ von `b` ebenfalls `G` ist.
3. `b.m(b)`	`G.m(G)` → Der statische Typ von `b` ist `G`, d. h. wir suchen in `G` eine Methode `m` mit Parameter `G` (stat. Typ von `b`). Wir finden `m(G)` in `G`. Wir führen diese Methode direkt aus, weil der dynamische Typ von `b` ebenfalls `G` ist.
4. `b.m(c)`	`G.m(F)` → *Analog zu Aufruf 2.* „[...] eine Methode `m` mit Parameter `F` (stat. Typ von **`c`**). Wir [...]“

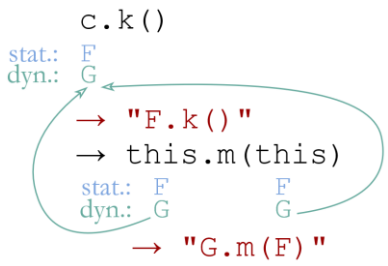
5. `b.k()`	`F.k()G.m(F)` → Der statische Typ von `b` ist `G`, d. h. wir suchen in `G` eine Methode `k` ohne Parameter. Wir finden `k()` in `F`, weil `G` von `F` erbt und die Methode `k()` in `G` nicht überschrieben wird. Wir führen diese Methode direkt aus, weil der dynamische Typ von `b` ebenfalls `G` ist. Es folgt die Rückgabe „`F.k()`“ und der Aufruf `this.m(this)`. Der statische Typ von `this` ist `F` (weil der Aufruf innerhalb der Klasse `F` stattfindet), d. h. wir suchen in `F` eine Methode `m` mit Parameter `F` (stat. Typ von `this`). Wir finden `m(F)` in `F`. Wir führen diese Methode *nicht* direkt aus (→ Schritt ❷), weil der dynamische Typ von `this` nicht `F` sondern `G` ist. *Warum?* Der dynamische Typ von `this` entspricht immer dem dynamischen Typ des Objekts, auf dem wir gerade operieren (also das, was beim vorherigen Aufruf vor dem Punkt stand, hier `b`). Weil sich statischen und dynamischer Typ unterscheiden sehen wir also nach, ob `m(F)` in `G` (dyn. Typ von `this`) überschrieben wird. Das ist der Fall, d. h. es wird nicht `m(F)` in `F` sondern `m(F)` in `G` ausgeführt. **Typenskizze:** 
6. `b.k(a)`	*Laufzeitfehler (`ClassCastException`)* → Der statische Typ von `b` ist `G`, d. h. wir suchen in `G` eine Methode `k` mit Parameter `F` (stat. Typ von `a`). Wir finden `k(F)` in `G` und führen diese Methode direkt aus, weil der dynamische Typ von `b` ebenfalls `G` ist. Es folgt die Rückgabe „`G.k(F)`“ und der Aufruf `this.m((G)f)`. Um diesen Aufruf zu bestimmen, benötigen wir den stat. und dyn. Typ von `this` und `(G)f`: Um `(G)f` zu bestimmen, sehen wir uns zunächst `f` an. Der stat. Typ von `f` ist `F` (Deklaration der Parametervariable). Der dyn. Typ von `f` entspricht dem dyn. Typ desjenigen Objekts, das im vorherigen Aufruf als Parameter übergeben wurde (weil `f` die erste und einzige Parametervariable ist). Als Parameter wurde `a` übergeben, was den dyn. Typ `F` hat, d. h. `f` hat den dyn. Typ `F`. Nun können wir überprüfen, ob der Cast funktioniert (vgl. Schritt I, Casting). Der statische Typ `F` steht in einer Vererbungsrelation mit der Cast-Klasse `G`, d. h. es gibt *keinen* Compiler-Fehler. Der dynamische Typ `F` erbt nicht von der Cast-Klasse `G` (sondern umgekehrt), d. h. es entsteht eine `ClassCastException`. Der Aufruf wird somit abgebrochen, weshalb die Rückgabe „`G.k(F)`“ *nicht* stattfindet (das wäre anders, wenn es sich bei „`G.k(F)`“ um eine *Aus*gabe (`System.out.println`) gehandelt hätte, denn dann wäre diese Ausgabe vor dem Auftreten des Laufzeitfehlers durchgeführt worden). **Typenskizze:** 

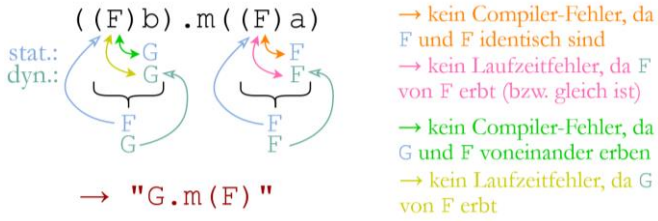
7. `b.k(b)`	`F.k(G)` → Der statische Typ von `b` ist `G`, d. h. wir suchen in `G` eine Methode `k` mit Parameter `G` (stat. Typ von `b`). Als möglichen Kandidaten finden wir zunächst `k(F)` in `G` (möglich, weil dort, wo als Parameter ein `F` erwartet wird, auch alle Unterklassen von `F` eingesetzt werden können, also insb. auch `G`). Zusätzlich ist aber auch `k(G)` aus `F` möglich, weil `G` ja von `F` erbt (d. h. alle nicht privaten Methoden sind möglich). Wir wählen die Methode mit der spezielleren Signatur, also `k(G)` in `F`, und führen sie direkt aus, weil der dynamische Typ von `b` ebenfalls `G` ist.
8. `b.k(c)`	`G.k(F)G.m(G)` → Der statische Typ von `b` ist `G`, d. h. wir suchen in `G` eine Methode `k` mit Parameter `F` (stat. Typ von `c`). Wir finden `k(F)` in `G` und führen diese Methode direkt aus, weil der dynamische Typ von `b` ebenfalls `G` ist. Es folgt die Rückgabe „`G.k(F)`“ und der Aufruf `this.m((G)f)`. Um diesen Aufruf zu bestimmen, benötigen wir den stat. und dyn. Typ von `this` und `(G)f`: • Um `(G)f` zu bestimmen, ermitteln wir zunächst den stat. und dyn. Typ von `f` (wie bereits in Aufruf 6). Der stat. Typ von `f` ist `F`. Der dyn. Typ entspricht dem dyn. Typ des Objekts, das im vorherigen Aufruf als Parameter übergeben wurde – das war `c`, weshalb der dyn. Typ `G` ist. Nun können wir überprüfen, ob der Cast funktioniert (vgl. Schritt I, Casting). Der statische Typ von `f` ist `F` und steht in einer Vererbungsrelation mit der Cast-Klasse `G`, d. h. es gibt *keinen* Compiler-Fehler. Der dyn. Typ von `f` ist `G` und somit gleich zum Cast-Typ, d. h. der Cast funktioniert. Der stat. Typ von `(G)f` ist somit `G` (= Cast-Typ), der dyn. Typ ist `G` (= dyn. Typ von `f`). • Der statische Typ von `this` ist `G` (= Klasse, in der der Aufruf stattfindet) und der dynamische Typ ist `G` (= dyn. Typ des Objekts, das zuvor vor dem Punkt stand, hier `b`). Wir suchen folglich in `G` (stat. Typ von `this`) nach einer Methode `m` mit Parameter `G` (stat. Typ von `(G)f`), welche wir finden. Wir führen diese Methode direkt aus, weil der dyn. Typ von `this` ebenfalls `G` ist und landen deshalb bei `G.m(G)`. **Typenskizze:** 
9. `b.s()`	`G.s()` → Der statische Typ von `b` ist `G`, d. h. wir suchen in `G` eine Methode `s` ohne Parameter. Wir finden `s()` in `G`. Wir führen diese Methode direkt aus, weil die Methode `static` ist.
10. `b.t()`	`F.t()` → Der statische Typ von `b` ist `G`, d. h. wir suchen in `G` eine Methode `t` ohne Parameter. Wir finden `t()` nicht in `G`, aber in `F` (weil `G` von `F` erbt). Wir führen diese Methode direkt aus, weil sie `static` ist.

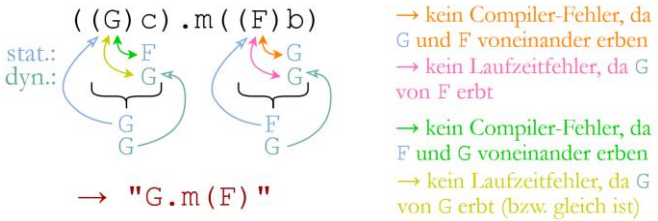
11. `a.m()`	`F.m()F.s()` → Der statische Typ von `a` ist `F`, d. h. wir suchen in `F` eine Methode `m` ohne Parameter. Wir finden `m()` in `F`. Wir führen diese Methode direkt aus, weil der dynamische Typ von `a` ebenfalls `F` ist. Es folgt die Rückgabe „`F.m()`“ und der Aufruf `this.s()`. `this` hat den statischen Typ `F` (= Klasse, in der der Aufruf stattfindet), d. h. wir suchen in `F` eine Methode `s` ohne Parameter. Wir finden `s()` in `F` und führen diese Methode direkt aus, weil sie `static` ist.
12. `a.m(a)`	`F.m(F)F.s()` → Der statische Typ von `a` ist `F`, d. h. wir suchen in `F` eine Methode `m` mit Parameter `F` (stat. Typ von `a`). Wir finden `m(F)` in `F`. Wir führen diese Methode direkt aus, weil der dynamische Typ von `a` ebenfalls `F` ist. Es folgt die Rückgabe „`F.m(F)`“ und der Aufruf `s()`. Steht vor einem Methodenaufruf in Punktnotation wie hier keine Variable (oder bei statischen Methoden alternativ der Klassenname), so ist „`this.`“ implizit (bzw. innerhalb von statischen Methoden einfach die Klasse, in der wir uns gerade befinden). Wie in Aufruf 11 suchen wir also in `F` eine Methode `s` ohne Parameter, welche wir finden und direkt ausführen, weil sie `static` ist.
13. `a.m(b)`	`F.m(F)F.s()` → Der statische Typ von `a` ist `F`, d. h. wir suchen in `F` eine Methode `m` mit Parameter `G` (stat. Typ von `b`). In `F` finden wir als passende Methode lediglich `m(F)`. Die speziellere Methode `m(G)` kommt nicht in Frage, da sie `private` ist und daher nur bei Aufrufen von innerhalb der Klasse `F` aus sichtbar ist (der Aufruf `a.m(b)` steht aber außerhalb). `m(F)` kann gewählt werden, weil dort, wo als Parameter ein `F` erwartet wird, auch alle Unterklassen von `F` eingesetzt werden können, also insb. auch `G` (stat. Typ von `b`). Wir führen diese Methode direkt aus, weil der dynamische Typ von `a` ebenfalls `F` ist. Es folgt die Rückgabe „`F.m(F)`“ und der Aufruf `s()` (wie bei Aufruf 12). **Typenskizze:** 
14. `a.m(c)`	`F.m(F)F.s()` → *Analog zu Aufruf 12.* „[...] eine Methode `m` mit Parameter `F` (stat. Typ von `c`). Wir [...]“
15. `a.k()`	`F.k()F.m(F)F.s()` → Der statische Typ von `a` ist `F`, d. h. wir suchen in `F` eine Methode `k` ohne Parameter. Wir finden `k()` in `F`. Wir führen diese Methode direkt aus, weil der dynamische Typ von `a` ebenfalls `F` ist. Es folgt die Rückgabe „`F.k()`“ und der Aufruf `this.m(this)`: Der statische Typ von `this` ist `F` (weil der Aufruf innerhalb der Klasse `F` stattfindet), d. h. wir suchen in `F` eine Methode `m` mit Parameter `F` (stat. Typ von `this`). Wir finden `m(F)` in `F`. Wir führen diese Methode direkt aus, weil der dynamische Typ von `this` ebenfalls `F` ist (*Warum?* Der dynamische Typ von `this` entspricht immer dem dynamischen Typ des Objekts, auf dem wir gerade operieren (also das, was beim vorherigen Aufruf

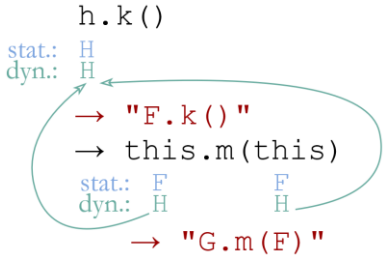
	vor dem Punkt stand, hier `a`)). Es folgt die Rückgabe „`F.m(F)`“ und der Aufruf `s()`, welcher (wie bei Aufruf 12) zur Rückgabe „`F.s()`“ führt. **Typenskizze:** `a.k()` 
16. `a.k(a)`	*Compiler-Fehler* → Der statische Typ von `a` ist `F`, d. h. wir suchen in `F` eine Methode `k` mit Parameter `F` (stat. Typ von `a`). Wir finden keine passende Methode in `F` (oder einer Oberklasse von `F`, hier nur `Object`), d. h. der Aufruf kompiliert nicht. ***Warum passt `k(G)` nicht?*** Weil wir aus statischer Sicht (Schritt ❶) ein `k(F)` suchen, d. h. der Parametertyp muss `F` *oder allgemeiner* sein, `G` ist jedoch spezieller als `F`. Möglich wäre neben `k(F)` also lediglich `k(Object)`, was hier aber ebenfalls nicht existiert. ***Warum können wir nicht `k(F)` aus `G` benutzen?*** Weil wir in Schritt ❶ nur in der Klasse des *statischen* Typs (hier `F`) oder einer Oberklasse (also hier lediglich `Object`) suchen, `G` ist jedoch eine Unterklasse von `F`. Die Methodensuche mit statischen Typen ist vorerst immer nötig, also auch wenn der dynamische Typ `G` wäre.
17. `a.k(b)`	`F.k(G)` → Der statische Typ von `a` ist `F`, d. h. wir suchen in `F` eine Methode `k` mit Parameter `G` (stat. Typ von `b`). Wir finden `k(G)` in `F`. Wir führen diese Methode direkt aus, weil der dyn. Typ von `a` ebenfalls `F` ist.
18. `a.k(c)`	*Compiler-Fehler* → *Analog zu Aufruf 16.* „[...] eine Methode `k` mit Parameter `F` (stat. Typ von `c`). Wir [...]“ *Warum passt `k(G)` nicht?* → *siehe Aufruf 16.* *Warum geht `k(F)` aus `G` nicht?* → *siehe Aufruf 16.*
19. `a.s()`	`F.s()` → Der statische Typ von `a` ist `F`, d. h. wir suchen in `F` eine Methode `s` ohne Parameter, welche wir finden und direkt ausführen, weil sie `static` ist.
20. `a.t()`	`F.t()` → Der statische Typ von `a` ist `F`, d. h. wir suchen in `F` eine Methode `t` ohne Parameter, welche wir finden und direkt ausführen, weil sie `static` ist.
21. `c.m()`	`G.m()F.s()` → Der statische Typ von `c` ist `F`, d. h. wir suchen in `F` eine Methode `m` ohne Parameter. Wir finden `m()` in `F`. Wir führen diese Methode *nicht* direkt aus (→ Schritt ❷), denn der dynamische Typ von

	`c` ist `G`. Wir sehen also nach, ob die gefundene Methode `m()` in `G` überschrieben wird. Dies ist der Fall, weshalb `m()` in `G` ausgeführt wird. *Analog zu Aufruf 1*: Es folgt die Rückgabe „`G.m()`“ und der Aufruf `super.s()`, also suchen wir in der Oberklasse von `G` (das ist `F`) eine Methode `s` ohne Parameter (anders ausgedrückt: der „statische Typ“ von `super` ist `F`). Wir finden `s()` in `F` und führen diese direkt Methode aus, da wir auf dem Schlüsselwort `super` operieren (außerdem ist die Methode sowieso `static`). **Typenskizze:** 
22. `c.m(a)`	`G.m(F)` → Der statische Typ von `c` ist `F`, d. h. wir suchen in `F` eine Methode `m` mit Parameter `F` (stat. Typ von `a`). Wir finden `m(F)` in `F`. Wir führen diese Methode *nicht* direkt aus (→ Schritt ❷), denn der dyn. Typ von `c` ist `G`. Wir sehen also nach, ob die gefundene Methode `m(F)` in `G` überschrieben wird. Dies ist der Fall, weshalb `m(F)` in `G` ausgeführt wird.
23. `c.m(b)`	`G.m(F)` → Der statische Typ von `c` ist `F`, d. h. wir suchen in `F` eine Methode `m` mit Parameter `G` (stat. Typ von `b`). Wir finden als möglichen Kandidaten lediglich `m(F)` in `F`, weil die speziellere Methode `m(G)` von außerhalb nicht sichtbar ist, da sie `private` ist. Eine private Methode ist nur innerhalb der Klasse `F` sichtbar, also wenn der Aufruf innerhalb der Klasse `F` stattfindet. Der Aufruf `c.m(b);` steht aber nicht *in* der Klasse `F`. (Das wäre der Fall, wenn `c.m(b);` bspw. in der `main`-Methode von `F` stehen würde). Wir führen die Methode `m(F)` in `F` *nicht* direkt aus (→ Schritt ❷), denn der dynamische Typ von `c` ist `G`. Wir sehen also nach, ob die gefundene Methode `m(F)` in `G` überschrieben wird. Dies ist der Fall, weshalb `m(F)` in `G` ausgeführt wird.
24. `c.m(c)`	`G.m(F)` → *Analog zu Aufruf 22.* „[...] eine Methode `m` mit Parameter `F` (stat. Typ von `c`). Wir [...]“
25. `c.k()`	`F.k()G.m(F)` → Der statische Typ von `c` ist `F`, d. h. wir suchen in `F` eine Methode `k` ohne Parameter. Wir finden `k()` in `F`. Wir führen diese Methode *nicht* direkt aus (→ Schritt ❷), denn der dynamische Typ von `c` ist `G`. Wir sehen also nach, ob die gefundene Methode `k()` in `G` überschrieben wird. Dies ist nicht der Fall, weshalb wir bei `m(F)` in `F` bleiben und diese Methode ausführen. Es folgt die Rückgabe „`F.k()`“ und der Aufruf `this.m(this)`: Der statische Typ von `this` ist `F` (= Klasse, in der der Aufruf stattfindet), d. h. wir suchen in `F` eine Methode `m` mit Parameter `F` (stat. Typ von `this`). Wir finden `m(F)` in `F`. Wir führen diese Methode *nicht* direkt aus (→ Schritt ❷), da der dynamische Typ von `this` nicht `F` sondern `G` ist (=

	dyn. Typ des Objekts, das zuvor vor dem Punkt stand, hier `c`). Wir sehen also nach, ob die gefundene Methode `m(F)` in `G` überschrieben wird. Dies ist der Fall, weshalb `m(F)` in `G` ausgeführt wird. **Typenskizze:** 
26. `c.k(a)`	*Compiler-Fehler* → *Analog zu Aufruf 16.* „Der statische Typ von **c** ist `F`, d. h. wir suchen in `F` [...]“ *Warum passt `k(G)` nicht?* → *siehe Aufruf 16.* *Warum geht `k(F)` aus `G` nicht?* → *siehe Aufruf 16.*
27. `c.k(b)`	`F.k(G)` → Der statische Typ von `c` ist `F`, d. h. wir suchen in `F` eine Methode `k` mit Parameter `G` (stat. Typ von `b`). Wir finden `k(G)` in `F`. Wir führen diese Methode *nicht* direkt aus, weil der dynamische Typ von `c` nicht `F` sondern `G` ist (→ Schritt ❷). Wir sehen also nach, ob die gefundene Methode `k(G)` in `G` überschrieben wird. Dies ist nicht der Fall, weshalb wir bei `k(G)` in `F` bleiben und diese Methode ausführen.
28. `c.k(c)`	*Compiler-Fehler* → *Analog zu Aufruf 18 (bzw. 26).* „Der statische Typ von **c** ist `F`, [...] `F` (stat. Typ von **c**) [...]“ *Warum passt `k(G)` nicht?* → *siehe Aufruf 16.* *Warum geht `k(F)` aus `G` nicht?* → *siehe Aufruf 16.*
29. `c.s()`	`F.s()` → *Analog zu Aufruf 19.* Der statische Typ von `c` ist `F`, d. h. wir suchen in `F` eine Methode `s` ohne Parameter, welche wir finden und direkt ausführen, weil sie `static` ist.
30. `c.t()`	`F.t()` → *Analog zu Aufruf 20.* Der statische Typ von `c` ist `F`, d. h. wir suchen in `F` eine Methode `t` ohne Parameter, welche wir finden und direkt ausführen, weil sie `static` ist.
31. `a.m`	`F.m` → Der statische Typ von `a` ist `F`, d. h. wir suchen in `F` ein Attribut mit dem Namen `m`. Wir finden `m` in `F`. und geben dessen Wert `"F.m"` direkt zurück, da keine Überschreibung möglich ist (weil es sich um ein Attribut handelt, d. h. kein Schritt ❷). Dass es sich nicht um einen Methodenaufruf sondern eine Variable handelt, erkennt man an den fehlenden Klammern.
32. `b.m`	`G.m` → Der statische Typ von `b` ist `G`, d. h. wir suchen in `G` (oder einer Oberklasse, falls in `G` nicht gefunden) ein Attribut mit dem Namen `m`. Wir finden `m` in `G`. und geben dessen Wert `"G.m"` direkt zurück.
33. `c.m`	`F.m` → *Analog zu Aufruf 31.* „Der statische Typ von **c** ist `F`, d. h. wir suchen in `F` [...]“

34. `((F)b).m((F)a)`	`G.m(F)` → Um die aufgerufene Methode zu bestimmen, benötigen wir den stat. und dyn. Typ der Ausdrücke `(F)b` und `(F)a`. Sofern die Casts ausgeführt werden können, geben sie den stat. Typ des Ausdrucks vor, während der dyn. Typ dem dyn. Typ der Variable entspricht: • `(F)b`: Der stat. Typ von `b` (also `G`) steht in einer Vererbungsrelation mit dem Cast-Typ `F`, d. h. es gibt *keinen* Compiler-Fehler. Der dyn. Typ von `b` (also `G`) erbt vom Cast-Typ `F`, d. h. es gibt keinen Laufzeitfehler und der Cast funktioniert. Der stat. Typ von `(F)b` ist `F` (Cast-Typ), der dyn. Typ ist `G` (dyn. Typ von `b`). • `(F)a`: Der stat. Typ von `a` (also `F`) steht in einer Vererbungsrelation mit dem Cast-Typ `F` (bzw. entspricht ihm sogar genau), d. h. es gibt *keinen* Compiler-Fehler. Der dyn. Typ von `a` (also `F`) „erbt“ vom Cast-Typ `F` (bzw. entspricht ihm sogar genau), d. h. es gibt keinen Laufzeitfehler und der Cast funktioniert. Der stat. Typ von `(F)a` ist `F` (Cast-Typ), der dyn. Typ ist `F` (dyn. Typ von `a`). Wir suchen somit in `F` (stat. Typ von `((F)b)`) eine Methode `m` mit Parameter `F` (stat. Typ von `(F)a`). Wir finden `m(F)` in `F`. Wir führen diese Methode *nicht* direkt aus (→ Schritt ❷), weil der dynamische Typ von `((F)b)` nicht `F` sondern `G` ist. Wir sehen also nach, ob die gefundene Methode `m(F)` in `G` überschrieben wird. Dies ist der Fall, weshalb `m(F)` in `G` ausgeführt wird. **Typenskizze:**  *Hinweis: Der Aufruf ist analog zu `c.m(a)`*
35. `((G)c).m((F)b)`	`G.m(F)` → Um die aufgerufene Methode zu bestimmen, benötigen wir den stat. und dyn. Typ der Ausdrücke `(G)c` und `(F)b`: • `(G)c`: Der stat. Typ von `c` (also `F`) steht in einer Vererbungsrelation mit dem Cast-Typ `G`, d. h. es gibt *keinen* Compiler-Fehler. Der dyn. Typ von `c` (also `G`) „erbt“ vom Cast-Typ `G` (bzw. entspricht ihm sogar genau), d. h. es gibt keinen Laufzeitfehler und der Cast funktioniert. Der stat. Typ von `(G)c` ist `G` (Cast-Typ), der dyn. Typ ist `G` (dyn. Typ von `c`). • `(F)b`: *Wie bei Aufruf 34:* Der stat. Typ von `b` (also `G`) steht in einer Vererbungsrelation mit dem Cast-Typ `F`, d. h. es gibt *keinen* Compiler-Fehler. Der dyn. Typ von `b` (also `G`) erbt vom Cast-Typ `F`, d. h. es gibt keinen Laufzeitfehler und der Cast funktioniert. Der stat. Typ von `(F)b` ist `F` (Cast-Typ), der dyn. Typ ist `G` (dyn. Typ von `b`).

	Wir suchen somit in `G` (stat. Typ von `((G)c)`) eine Methode `m` mit Parameter `F` (stat. Typ von `(F)b`). Wir finden `m(F)` in `G`. Wir führen diese Methode direkt aus, weil der dynamische Typ von `((G)c)` ebenfalls `G` ist. **Typenskizze:**  *Hinweis: Der Aufruf ist analog zu `b.m(c)`*
36. `((F)c).m(c)`	`G.m(F)` → Um die aufgerufene Methode zu bestimmen, benötigen wir den stat. und dyn. Typ des Ausdrucks `(F)c`: Der stat. Typ von `c` (also `F`) steht in einer Vererbungsrelation mit dem Cast-Typ `F` (bzw. entspricht diesem sogar genau), d. h. es gibt *keinen* Compiler-Fehler. Der dyn. Typ von `c` (also `G`) „erbt" vom Cast-Typ `F`, d. h. es gibt keinen Laufzeitfehler und der Cast funktioniert. Der stat. Typ von `(F)c` ist `F` (Cast-Typ), der dyn. Typ ist `G` (dyn. Typ von `c`). Wir suchen somit in `F` (stat. Typ von `((F)c)`) eine Methode `m` mit Parameter `F` (stat. Typ von `c`). Wir finden `m(F)` in `F`. Wir führen diese Methode *nicht* direkt aus (→ Schritt ❷), weil der dynamische Typ von `((F)c)` nicht `F` sondern `G` ist. Wir sehen also nach, ob die gefundene Methode `m(F)` in `G` überschrieben wird. Dies ist der Fall, weshalb `m(F)` in `G` ausgeführt wird. *Hinweis: Der Cast bewirkt nichts. Der Aufruf ist analog zu `c.m(c)`*
37. `((F)b)` `.k((F)b)`	*Compiler-Fehler* → Um die aufgerufene Methode zu bestimmen, benötigen wir den stat. und dyn. Typ des Ausdrucks `(F)b` (zweimal benutzt): Der stat. Typ von `b` (also `G`) steht in einer Vererbungsrelation mit dem Cast-Typ `F`, d. h. es gibt *keinen* Compiler-Fehler beim Cast. Der dyn. Typ von `b` (also `G`) erbt vom Cast-Typ `F`, d. h. es gibt keinen Laufzeitfehler und der Cast funktioniert. Der stat. Typ von `(F)b` ist `F` (Cast-Typ), der dyn. Typ ist `G` (dyn. Typ von `b`). Wir suchen somit in `F` (stat. Typ von `((F)b)`) eine Methode `k` mit Parameter `F` (stat. Typ von `((F)b)`). Es existiert keine passende Methode in `F` (oder einer Oberklasse von `F`), d. h. der Aufruf kompiliert nicht. *Warum passt `k(G)` nicht? → siehe Aufruf 16.* *Warum geht `k(F)` aus `G` nicht? → siehe Aufruf 16.* *Hinweis: Der Aufruf ist analog zu `c.k(c)`*
38. `((G)a).k(c)`	*Laufzeitfehler (`ClassCastException`)* → Um die aufgerufene Methode zu bestimmen, benötigen wir den stat. und dyn. Typ des Ausdrucks `(G)a`: Der stat. Typ von `a` (also `F`) steht in einer Vererbungsrelation mit dem Cast-Typ `G`, d. h. es gibt *keinen* Compiler-Fehler. Der dyn. Typ von `a` (also `F`) erbt *nicht* vom Cast-Typ `G` (sondern umgekehrt!!), d. h. bei diesem Cast kommt es zu einem Fehler (zur Laufzeit, weil *dyn.* Typ betrachtet).

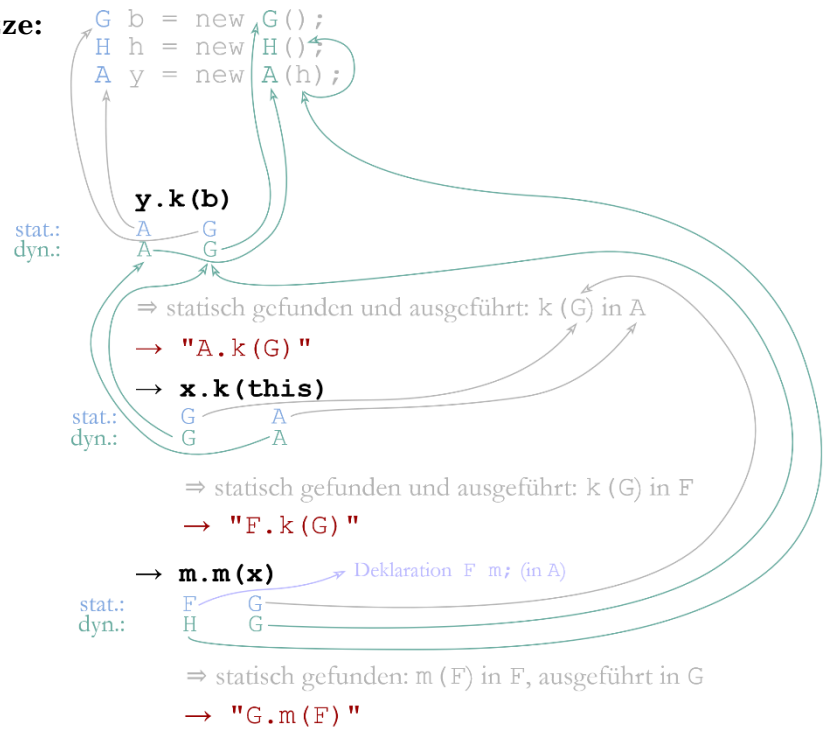
39. `((G)c).k((G)c)`	`F.k(G)` → Um die aufgerufene Methode zu bestimmen, benötigen wir den stat. und dyn. Typ des Ausdrucks `(G)c` (zweimal benutzt): Der stat. Typ von `c` (also `F`) steht in einer Vererbungsrelation mit dem Cast-Typ `G`, d. h. es gibt *keinen* Compiler-Fehler. Der dyn. Typ von `c` (also `G`) „erbt" vom Cast-Typ `G` (bzw. entspricht diesem genau), d. h. es gibt keinen Laufzeitfehler und der Cast funktioniert. Der stat. Typ von `(G)c` ist `G` (Cast-Typ), der dyn. Typ ist `G` (dyn. Typ von `c`). Wir suchen somit in `G` (stat. Typ von `((G)c)`) eine Methode `k` mit Parameter `G` (stat. Typ von `((G)c)`). *Wie bei Aufruf 7:* Als möglichen Kandidaten finden wir zunächst `k(F)` in `G` (möglich, weil dort, wo als Parameter ein `F` erwartet wird, auch alle Unterklassen von `F` eingesetzt werden können, also insb. auch `G`). Zusätzlich ist aber auch `k(G)` aus `F` möglich, weil `G` ja von `F` erbt (d. h. alle nicht privaten Methoden sind möglich). Wir wählen die Methode mit der spezielleren Signatur, also `k(G)` in `F`, und führen sie direkt aus, weil der dynamische Typ von `b` ebenfalls `G` ist.
40. `h.k()`	`F.k()G.m(F)` → Der statische Typ von `h` ist `H`, d. h. wir suchen in `H` eine Methode `k` ohne Parameter. In `H` finden wir nicht direkt eine Methode `k()`. Da `H` aber indirekt (über `G`) von `F` erbt und deshalb alle nicht-privaten Methoden aus `F` auch in `H` existieren, finden wir `k()` in `F`. Hinweis: Gäbe es `k()` zusätzlich auch in `G`, so würden wir stattdessen `G.k()` wählen, weil `G` näher an `H` ist als `F`. Wir führen die Methode `k()` in `F` direkt aus, weil der dynamische Typ von `h` ebenfalls `H` ist. Es folgt die Rückgabe „`F.k()`" und der Aufruf `this.m(this)`: Der statische Typ von `this` ist `F` (= Klasse, in der der Aufruf stattfindet), d. h. wir suchen in `F` eine Methode `m` mit Parameter `F` (stat. Typ von `this`). Wir finden `m(F)` in `F`. Wir führen diese Methode *nicht* direkt aus (→ Schritt ❷), da der dynamische Typ von `this` nicht `F` sondern `H` ist (= dyn. Typ des Objekts, das zuvor vor dem Punkt stand, hier `h`). Wir sehen also nach, ob die gefundene Methode `m(F)` in `H` überschrieben wird. Dies ist zwar nicht direkt der Fall, allerdings erbt `H` von `G` und dort wird `m(F)` überschrieben, weshalb wir `m(F)` in `G` ausführen. **Hinweis:** Dieser Fall ist im Skript als „Zwischenklassen" beschrieben. Für Schritt ❷ kannst du dir auch merken, einfach immer bei der Klasse des dyn. Typs (hier `H`) anzufangen, nach Überschreibung zu kucken, und zur Oberklasse zu gehen, falls du die Methode nicht findest. Du findest die Methode spätestens in der Klasse des stat. Typs (was immer eine Oberklasse sein muss). **Typenskizze:** 

41. `h.k(c)`	`H.k(F)` → Der statische Typ von `h` ist `H`, d. h. wir suchen in `H` eine Methode `k` mit Parameter `F` (stat. Typ von `c`). Wir finden `k(F)` in `H`. Wir führen diese Methode direkt aus, weil der dyn. Typ von `h` ebenfalls `H` ist. **Tipp:** Da es keine Unterklassen von `H` gibt, kann es nie sein, dass die gefundene Methode überschrieben wird (weil der dyn. Typ immer mindestens so speziell wie der stat. Typ ist). Immer wenn wir aus statischer Sicht eine Methode in `H` finden, ist Überschreibung sowieso unmöglich.
42. `h.k(b)`	`H.k(G)G.m(F)` → Der statische Typ von `h` ist `H`, d. h. wir suchen in `H` eine Methode `k` mit Parameter `G` (stat. Typ von `b`). Wir finden `k(G)` in `H`. Wir führen diese Methode direkt aus, weil der dynamische Typ von `h` ebenfalls `H` ist. Es folgt die Rückgabe „`H.k(G)`“ und der Aufruf `super.m((F)g)`. Um diesen Aufruf zu bestimmen, benötigen wir den stat. und dyn. Typ von `(F)g`: Der stat. Typ von `g` ist `G` (wegen der Deklaration `String k(G g)`) und steht in einer Vererbungsrelation mit dem Cast-Typ `F`, d. h. es gibt *keinen* Compiler-Fehler. Der dyn. Typ von `g` ist `G` (weil `b` übergeben wurde, was dyn. ein `G` ist) und „erbt“ somit vom Cast-Typ `G` (bzw. entspricht diesem genau), d. h. es gibt keinen Laufzeitfehler und der Cast funktioniert. Der stat. Typ von `(F)g` ist `F` (Cast-Typ), der dyn. Typ ist `G` (dyn. Typ von `g`). Das Schlüsselwort `super` indiziert, dass wir in der Oberklasse der aktuellen Klasse (also in `G`) eine Methode `m` mit Parameter `F` suchen, anschließend aber nicht nach Überschreibung kucken! Wir finden `m(F)` in `G` und führen diese Methode direkt aus.
43. `h.k(g)`	*Compiler-Fehler* → An dieser Stelle existiert keine Variable `g`.
44. `((F)h).k(b)`	`H.k(G)G.m(F)` → Um die aufgerufene Methode zu bestimmen, benötigen wir den stat. und dyn. Typ des Ausdrucks `(F)h`: Der stat. Typ von `h` (also `H`) steht in einer Vererbungsrelation mit dem Cast-Typ `F`, d. h. es gibt *keinen* Compiler-Fehler. Der dyn. Typ von `h` (also `H`) erbt (indirekt über `G`) vom Cast-Typ `F`, d. h. es gibt keinen Laufzeitfehler und der Cast funktioniert. Der stat. Typ von `(F)h` ist `F` (Cast-Typ), der dyn. Typ ist `H` (dyn. Typ von `h`). Wir suchen somit in `F` (stat. Typ von `((F)h)`) eine Methode `k` mit Parameter `G` (stat. Typ von `b`). Wir finden `k(G)` in `F`. Wir führen diese Methode *nicht* direkt aus (→ Schritt ❷), weil der dynamische Typ von `((F)h)` nicht `F` sondern `H` ist. Wir sehen also nach, ob die gefundene Methode `k(G)` in `H` (oder einer Zwischenklasse) überschrieben wird. Dies ist der Fall, weshalb wir `k(G)` in `H` ausführen. Es folgt die Rückgabe „`H.k(G)`“ und der Aufruf `super.m((F)g)` *wie in Aufruf 42.*
45. `((F)h).k(c)`	*Compiler-Fehler* → Um die aufgerufene Methode zu bestimmen, benötigen wir den stat. und dyn. Typ des Ausdrucks `(F)h`: *Siehe Aufruf 44.* Der statische Typ von `((F)h)` ist `F`, d. h. wir suchen in `F` eine Methode `k` mit Parameter `F` (stat. Typ von `c`). Es existiert keine passende Methode in `F` (oder einer Oberklasse von `F`), d. h. der Aufruf kompiliert nicht. *Warum passt `k(G)` nicht?* → *siehe Aufruf 16.* *Warum geht `k(F)` aus `H` (oder `G`) nicht?* → *siehe Aufruf 16.*

46.	`((G)h).k(a)`	`H.k(F)` → Um die aufgerufene Methode zu bestimmen, benötigen wir den stat. und dyn. Typ des Ausdrucks `(G)h`: Der stat. Typ von `h` (also `H`) steht in einer Vererbungsrelation mit dem Cast-Typ `G`, d. h. es gibt *keinen* Compiler-Fehler. Der dyn. Typ von `h` (also `H`) erbt vom Cast-Typ `G`, d. h. es gibt keinen Laufzeitfehler und der Cast funktioniert. Der stat. Typ von `(G)h` ist `G` (Cast-Typ), der dyn. Typ ist `H` (dyn. Typ von `h`). Wir suchen in `G` (stat. Typ von `((G)h)`) eine Methode `k` mit Parameter `F` (stat. Typ von `a`). Wir finden `k(F)` in `G`. Wir führen diese Methode *nicht* direkt aus (→ Schritt ❷), weil der dynamische Typ von `((G)h)` nicht `G` sondern `H` ist. Wir sehen also nach, ob die gefundene Methode `k(F)` in `H` überschrieben wird. Das ist der Fall, d. h. es wird `k(F)` in `H` ausgeführt.
47.	`((G)h).k(b)`	`H.k(G)G.m(F)` → Um die aufgerufene Methode zu bestimmen, benötigen wir den stat. und dyn. Typ des Ausdrucks `(G)h`: *Siehe Aufruf 46.* Wir suchen in `G` (stat. Typ von `((G)h)`) eine Methode `k` mit Parameter `G` (stat. Typ von `b`). Als möglichen Kandidaten finden wir zunächst `k(F)` in `G` (möglich, weil dort, wo als Parameter ein `F` erwartet wird, auch alle Unterklassen von `F` eingesetzt werden können, also insb. auch `G`). Zusätzlich ist aber auch `k(G)` aus `F` möglich, weil `G` ja von `F` erbt (d. h. alle nicht privaten Methoden der Oberklassen sind möglich). Wir wählen die Methode mit der spezielleren Signatur, also `k(G)` in `F`. Wir führen diese Methode *nicht* direkt aus (→ Schritt ❷), weil der dynamische Typ von `((G)h)` nicht `G` sondern `H` ist. Wir sehen also nach, ob die gefundene Methode `k(G)` in `H` überschrieben wird. Dies ist der Fall, weshalb wir `k(G)` in `H` ausführen. Es folgt die Rückgabe „`H.k(G)`“ und der Aufruf `super.m((F)g)` *wie in Aufruf 42.*
48.	`((G)h).k(c)`	`H.k(F)` → *Analog zu Aufruf 46.* „[...] eine Methode `k` mit Parameter `F` (stat. Typ von **`c`**). Wir [...]“
49.	`((G)h)` `.k((G)c)`	`H.k(G)G.m(F)` → Um die aufgerufene Methode zu bestimmen, benötigen wir den stat. und dyn. Typ der Ausdrücke `(G)h` und `(G)c`: • `(G)h`: Der stat. Typ ist `G` (Cast-Typ), der dyn. Typ ist `H` (dyn. Typ von `h`). *Der Cast funktioniert wie in Aufruf 46 beschrieben.* • `(G)c`: Der stat. Typ ist `G` (Cast-Typ), der dyn. Typ ist `G` (dyn. Typ von `c`). *Der Cast funktioniert wie in Aufruf 39 beschrieben.* Wir suchen in `G` (stat. Typ von `((G)h)`) eine Methode `k` mit Parameter `G` (stat. Typ von `((G)c)`). Wir finden `k(G)` in `F` (*wie in Aufruf 47*). Wir führen diese Methode *nicht* direkt aus (→ Schritt ❷), weil der dynamische Typ von `((G)h)` nicht `G` sondern `H` ist. Wir sehen also nach, ob die gefundene Methode `k(G)` in `H` überschrieben wird. Dies ist der Fall, weshalb wir `k(G)` in `H` ausführen. Es folgt die Rückgabe „`H.k(G)`“ und der Aufruf `super.m((F)g)` *wie in Aufruf 42.*

50. `h.k(h)`	`H.k(G)G.m(F)` → Der statische Typ von `h` ist `H`, d. h. wir suchen in `H` eine Methode `k` mit Parameter `H` (stat. Typ von `h`). Als mögliche Kandidaten finden wir `k(G)` und `k(F)` in `H` (sowie theoretisch auch noch `k(G)` in `F` und `k(F)` in `G`). Wir wählen die Methode mit der speziellsten Signatur aus der speziellsten Klasse, also `k(G)` aus `H`. Wir führen diese Methode direkt aus, weil der dyn. Typ von `h` ebenfalls `H` ist. Es folgt die Rückgabe „`H.k(G)`“ und der Aufruf `super.m((F)g)` *wie in Aufruf 42.*
51. `h.s()`	`G.s()` → Der statische Typ von `h` ist `H`, d. h. wir suchen in `H` eine Methode `s` ohne Parameter. Wir finden `s()` nicht in `H`, aber in `G` (weil `H` von `G` erbt). Wir führen diese Methode direkt aus, weil sie `static` ist.
`a = h;`	Der dynamische Typ der Variable `a` ist ab jetzt (also für die nachfolgenden Aufrufe) `H`, der statische Typ von `a` bleibt `F`.
52. `((H)a).k(b)`	`H.k(G)G.m(F)` → Um die aufgerufene Methode zu bestimmen, benötigen wir den stat. und dyn. Typ des Ausdrucks `(H)a`: Der stat. Typ von `a` (also `F`) steht in einer Vererbungsrelation mit dem Cast-Typ `H`, d. h. es gibt *keinen* Compiler-Fehler. Der dyn. Typ von `a` (also wegen der Zuweisung jetzt `H`) „erbt“ vom Cast-Typ `H` (bzw. entspricht diesem genau), d. h. es gibt keinen Laufzeitfehler und der Cast funktioniert. Der stat. Typ von `(H)a` ist `H` (Cast-Typ), der dyn. Typ ist `H` (dyn. Typ von `a`). Wir suchen in `H` (stat. Typ von `((H)a)`) eine Methode `k` mit Parameter `G` (stat. Typ von `b`). Wir finden `k(G)` in `H`. Es folgt die Rückgabe „`H.k(G)`“ und der Aufruf `super.m((F)g)` *wie in Aufruf 42.* *Hinweis: Der Aufruf ist analog zu `h.k(b)`*
53. `((H)a).m(h)`	`G.m(G)` → Um die aufgerufene Methode zu bestimmen, benötigen wir den stat. und dyn. Typ des Ausdrucks `(H)a`. Der Cast funktioniert und ist stat. und dyn. Typ `H` (Begründung: *siehe Aufruf 52*). Wir suchen somit in `H` (stat. Typ von `((H)a)`) eine Methode `m` mit Parameter `H` (stat. Typ von `h`). Als mögliche Kandidaten finden wir `m(G)` und `m(F)` in `G`. Diese Methoden passen, weil wo als Parameter ein `G` bzw. `F` erwartet wird auch jede Unterklasse von `G` bzw. `F` eingesetzt werden kann (und `H` ist eine Unterklasse von `G` und `F`). Diese Methoden sind wählbar, weil `H` von `G` erbt (also insb. auch die beiden besagten Methoden). Wir wählen die Methode mit dem spezielleren Parameter, also `m(G)` in `G`
54. `((G)a).k(h)`	`H.k(G)G.m(F)` → Um die aufgerufene Methode zu bestimmen, benötigen wir den stat. und dyn. Typ des Ausdrucks `(G)a`: Der stat. Typ von `a` (also `F`) steht in einer Vererbungsrelation mit dem Cast-Typ `G`, d. h. es gibt *keinen* Compiler-Fehler. Der dyn. Typ von `a` (also wegen der Zuweisung jetzt `H`) erbt vom Cast-Typ `G`, d. h. es gibt keinen Laufzeitfehler und der Cast funktioniert. Der stat. Typ von `(G)a` ist `G` (Cast-Typ), der dyn. Typ ist `H` (dyn. Typ von `a`). Wir suchen in `G` (stat. Typ von `((G)a)`) eine Methode `k` mit Parameter `H` (stat. Typ von `h`). Wir finden `k(G)` in `G`. Die Methode wird in `H` (dyn. Typ von `((G)a)`) überschrieben, d. h. es wird `k(G)` aus `H` ausgeführt. Es folgt die Rückgabe „`H.k(G)`“ und `super.m((F)g)` *wie in Aufruf 42.*

55. `a.k(a)`	*Compiler-Fehler* → Wie schon bei Aufruf 16 gibt es keine Methode `k(F)` in `F`. Der dyn. Typ ist für die Methodensuche zunächst irrelevant.
56. `a.m(h)`	`G.m(F)` → Wir suchen in `F` (stat. Typ von `a`) eine Methode `m` mit Parameter `h`. Wir finden `m(F)` in `F`, weil dort, wo als Parameter `F` erwartet wird, auch Unterklassen von `F` übergeben werden können (und `H` erbt von `G`, also indirekt auch von `F`). `m(G)` in `F` würde zwar besser passen, können wir aber nicht auswählen, da sie `private` und damit von außerhalb der Klasse unsichtbar ist. Der dyn. Typ von `a` ist seit der Zuweisung `H`, also prüfen wir, ob die gefundene Methode `m(F)` in `H` (dyn. Typ von `a`) überschrieben wird. Das ist in `H` selbst nicht der Fall, allerdings erbt `H` von `G` und dort wird die Methode überschrieben.
57. `A.t()`	`F.t()` → Hier findet der Zugriff nicht über eine Variable (bzw. ein Objekt) sondern über einen Klassennamen statt. Das ist möglich, erlaubt aber lediglich den Zugriff auf statische Variablen und Methoden. Wir suchen in `A` eine statische Methode `t` ohne Parameter. Wir finden die Methode `t()` zwar nicht direkt in `A`, aber in `F` (denn `A` erbt indirekt von `F`). Die Methode ist `static` und kann daher gewählt werden.
58. `H.s`	`G.s` → Zugriff über einen Klassennamen. Wir suchen in `H` eine statische Variable `s`. Wir finden die Variable `s` zwar nicht direkt in `H`, aber in `G` (und `H` erbt ja von `G`). Die Variable ist `static` und kann daher gewählt werden.
59. `A.k(h)`	*Compiler-Fehler* → Zugriff über einen Klassennamen. Wir suchen in `A` eine statische Methode `k` mit Parameter `H` (stat. Typ von `h`). Es existiert zwar eine Methode `k(H)` in `A`, allerdings ist diese nicht `static` (sondern immer an ein Objekt gebunden). Der Zugriff über den Klassennamen ist daher nicht möglich.
60. `y.k(h)`	`A.k(H)F.t()` → Der statische Typ von `y` ist `A`, d. h. wir suchen in der Klasse `A` nach einer Methode `k`, die den Parameter `H` (stat. Typ von `h`) akzeptiert. Wir finden `k(H)` in `A`. Wir führen die gefundene Methode direkt aus, weil der dynamische Typ von `y` ebenfalls `A` ist. Es folgt die Rückgabe „`A.k(H)`“ und der Aufruf `t()`: Keine Angabe vor dem Methoden-/Variablennamen impliziert innerhalb von Objektmethoden „`this.`“, d. h. `t()` ist äquivalent zu `this.t()`. Der statische Typ von `this` ist `A` (da wir uns in der Klasse `A` befinden), d. h. wir suchen in `A` eine Methode `t` ohne Parameter. Wir finden `t()` nicht direkt in der Klasse `A`, aber in `F` (möglich, weil `A` von `G` und `G` von `F` erbt). Wir führen `t()` in `F` direkt aus, weil die Methode `static` ist.
61. `new H()` `.k(y)`	`H.k(A)` → `new H()` ist statisch und dynamisch `H`, d. h. wir suchen in `H` eine Methode `k` mit Parameter `A` (stat. Typ von `y`). Wir finden `k(A)` in H und führen sie direkt aus.
62. `((H)y).k(c)`	*Compiler-Fehler* → Um die aufgerufene Methode zu bestimmen, benötigen wir den stat. und dyn. Typ des Ausdrucks `(H)y`: Der stat. Typ von `y` (also `A`) steht *nicht* in einer Vererbungsrelation mit dem Cast-Typ `H`, weil weder `A` von `H` noch `H` von `A` erbt. Dieser Cast kann daher unter

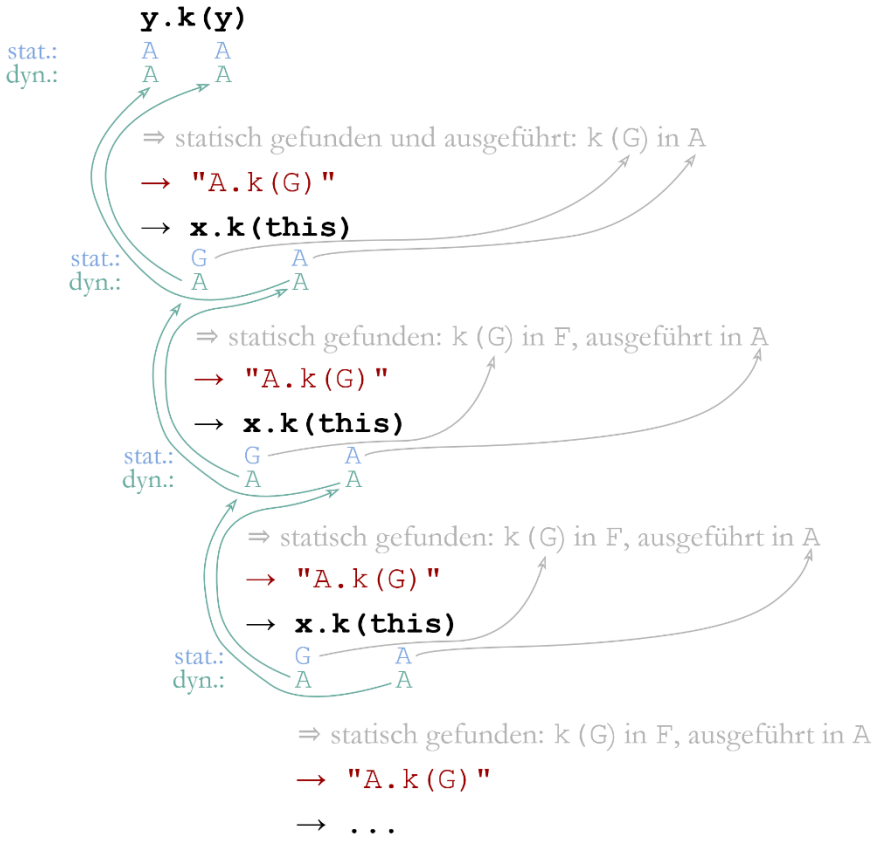
	keinen Umständen funktionieren, weshalb der Compiler den Cast gar nicht erst übersetzt.
63. `y.k(b)`	`A.k(G)F.k(G)G.m(F)` → Der stat. Typ von `y` ist `A`, d. h. wir suchen in `A` eine Methode `k` mit Parameter `G` (stat. Typ von `b`). Wir finden `k(G)` in `A` und führen diese Methode direkt aus, da der dyn. Typ von `y` ebenfalls `A` ist. Es folgt die Rückgabe „`A.k(G)`“ gefolgt von den beiden Aufrufen: • `x.k(this)`: Der stat. Typ von `x` ist `G` (wegen der Deklaration `String k(G x)`), d. h. wir suchen in `G` eine Methode `k` mit Parameter `A` (stat. Typ von `this`). Wir finden `k(F)` in `G` und die von `F` geerbte `k(G)`, wobei wir die speziellere Methode `k(G)` in `F` wählen. Wir führen diese Methode direkt aus, weil der dyn. Typ von `x` ebenfalls `G` ist (*Wieso?* Weil für den Parameter `x` zuvor `b` übergeben wurde, was den dyn. Typ `G` hatte, siehe Typenskizze). Es folgt die Rückgabe „`F.k(G)`“. • `m.m(x)`: Hier wird das Attribut `m` des aktuellen Objekts verwendet (`this.m`). Der stat. Typ von `m` ist `F` (ablesbar an der Deklaration `F m;`), d. h. wir suchen in `F` eine Methode `m` mit dem Parameter `G` (stat. Typ von `x`). Wir finden `m(F)` in `F`, da `m(G)` in `F` von außen nicht sichtbar ist (`private`). Um auf Überschreibung (Schritt ❷) zu prüfen, benötigen wir nun den dyn. Typ von `m`. Um zu bestimmen, was `m` speichert, müssen wir wissen, auf welchem Objekt wir gerade operieren. Wir kommen vom Aufruf `((G)y).k(h)`, d. h. wir operieren auf dem von `y` referenzierten `A`-Objekt. Dieses wurde zu Beginn mit `new A(h)` initialisiert, d. h. es wurde der Konstruktor `A(F m)` aufgerufen und das Attribut `m` auf den übergebenen Wert `h` (dyn. Typ `H`) gesetzt (`this.m = m;`). Der dyn. Typ von `m` ist also `H` und unterscheidet sich daher vom stat. Typ `F`. Wir sehen also nach, ob die gefundene Methode `m(F)` in `H` überschrieben wird. Dies ist zwar nicht direkt der Fall, allerdings erbt `H` von `G` und dort wird `m(F)` überschrieben, weshalb wir `m(F)` in `G` ausführen, was zu „`G.m(F)`“ auswertet. **Typenskizze:** 

64. `y.k(y)`

Laufzeitfehler (`StackOverflowError`) → Der stat. Typ von `y` ist `A`, d. h. wir suchen in `A` eine Methode `k` mit Parameter `A` (stat. Typ von `y`). Wir wählen `k(G)` in `A`, da es (auch in Oberklassen) keine speziellere Methode gibt (`k(H)` in `A` ist nicht möglich, weil `A` nicht von `H` erbt). Wir führen diese Methode direkt aus, weil der dyn. Typ von `y` ebenfalls `A` ist. Es folgt die Rückgabe „`A.k(G)`“ gefolgt von den Aufrufen `x.k(this)` und `m.m(x)`:

- `x.k(this)`: Der stat. Typ von `x` ist `G` (wegen der Deklaration `String k(G x)`), d. h. wir suchen in `G` eine Methode `k` mit Parameter `A` (stat. Typ von `this`). Wir finden `k(F)` in `G` und die von `F` geerbte `k(G)`, wobei wir die speziellere Methode `k(G)` in `F` wählen. Wir führen diese Methode *nicht* direkt aus (→ Schritt ❷), weil der dyn. Typ von `x` nicht `G` sondern `A` ist (*Wieso?* Weil für den Parameter `x` zuvor `y` übergeben wurde, was den dyn. Typ `A` hatte, siehe Typenskizze). Wir sehen also nach, ob die gefundene Methode `k(F)` in `H` überschrieben wird. Das ist der Fall, d. h. wir führen `k(G)` in `A` aus. Es folgt die Rückgabe „`A.k(G)`“ gefolgt von den Aufrufen `x.k(this)` und `m.m(x)`:
  - `x.k(this)`: Sowohl `x` und `this` ist wieder das von `y` referenzierte `A`-Objekt, d. h. dieser Aufruf wird sich unendlich oft wiederholen. Endlosrekursion endet in einem Stack-Überlauf.

Typenskizze:



65. `((G)y).k(h)`

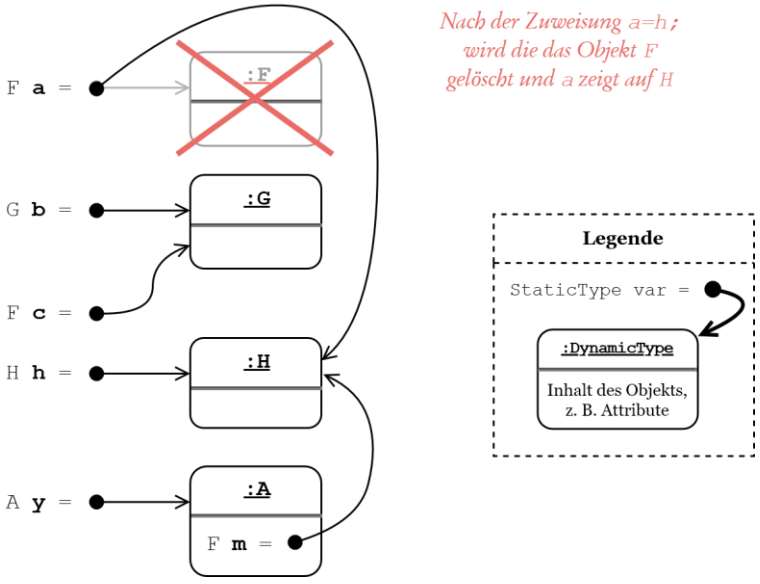
`A.k(G)H.k(G)G.m(F)G.m(F)` → Um die aufgerufene Methode zu bestimmen, benötigen wir den stat. und dyn. Typ des Ausdrucks `(G)y`: Der stat. Typ von `y` (also `A`) steht in einer Vererbungsrelation mit dem Cast-Typ `G`, d. h. es gibt *keinen* Compiler-Fehler beim Cast. Der dyn. Typ von `y` (also `A`) erbt vom Cast-Typ `G`, d. h. es gibt keinen Laufzeitfehler und der Cast funktioniert. Der stat. Typ von `(G)y` ist `G` (Cast-Typ), der dyn. Typ ist `A` (dyn. Typ von `y`).

Folglich suchen wir in `G` (stat. Typ von `((G)y)`) eine Methode `k` mit Parameter `H` (stat. Typ von `h`). Die Methode `k(F)` in `G` würde passen (weil dort, wo ein `F` erwartet wird, auch alle Unterklassen von `F` akzeptiert werden, insb. also auch `G`), allerdings passt die von `F` geerbte Methode `k(G)` besser (speziellerer Parametertyp), weshalb wir `k(G)` in `F` wählen.

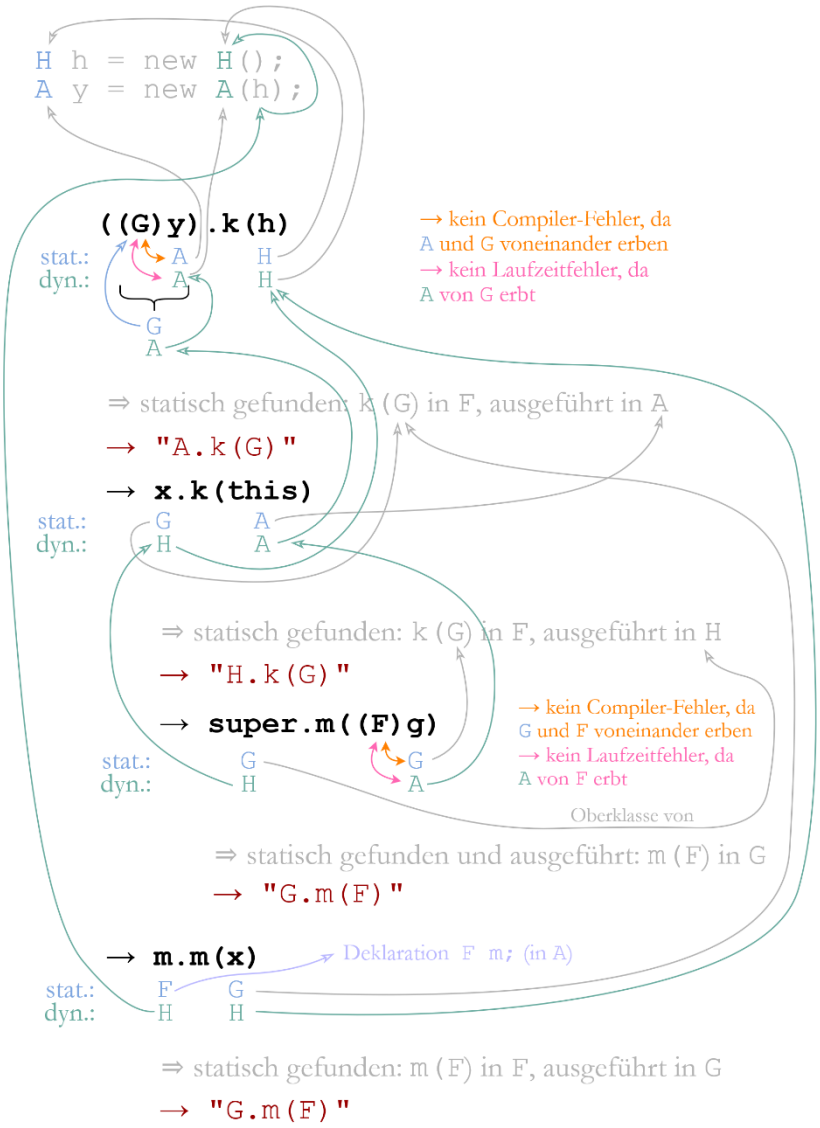
Wir führen diese Methode *nicht* direkt aus (→ Schritt ❷), weil der dynamische Typ von `((G)y)` nicht `G` sondern `A` ist. Wir sehen also nach, ob die gefundene Methode `k(G)` in `A` überschrieben wird. Dies ist der Fall, weshalb wir `k(G)` in `A` ausführen. Es folgt die Rückgabe „`A.k(G)`“ gefolgt von den beiden Aufrufen:

- `x.k(this)`: Der stat. Typ von `x` ist `G` (wegen der Deklaration `String k(G x)`), d. h. wir suchen in `G` eine Methode `k` mit Parameter `A` (stat. Typ von `this`). Wir finden `k(F)` in `G` und die von `F` geerbte `k(G)`, wobei wir die speziellere Methode `k(G)` in `F` wählen. Wir führen diese Methode *nicht* direkt aus (→ Schritt ❷), weil der dyn. Typ von `x` nicht `G` sondern `H` ist (*Wieso?* Weil für den Parameter `x` zuvor `h` übergeben wurde, was den dyn. Typ `H` hatte, siehe Typenskizze). Wir sehen also nach, ob die gefundene Methode `k(F)` in `H` überschrieben wird. Das ist der Fall, d. h. wir führen `k(G)` in `H` aus. Es folgt die Rückgabe „`H.k(G)`“ gefolgt vom Aufruf `super.m((F)g)`, welcher wiederum zu „`G.m(F)`“ auswertet.
- `m.m(x)`: *Siehe `m.m(x)` in Aufruf 63.*

  Man kann sich für Polymorphie-Aufgaben, bei denen Attribute im Spiel sind, auch angewöhnen, zu Beginn ein Objektdiagramm zu zeichnen. Hier ein Beispiel für diese Aufgabe:



Typenskizze:



90

1. `19` (= 9 ∗ 2 + 1)
2. `14` (= 7 ∗ 2)
3. `3` (= 7 ∗ 2)
4. `10` (= 5 ∗ 2)
5. `10` (= 5 ∗ 2)
6. *Compiler-Fehler*: `Y` verfügt über keine Methode `write(X)`.
7. `14` (= 7 ∗ 2)
8. `19` (= 9 ∗ 2 + 1)
9. *Laufzeitfehler* (`ClassCastException`)
10. `14` (= 7 ∗ 2)
11. `10` (= 5 ∗ 2)
12. `14` (= 7 ∗ 2)
13. *Laufzeitfehler* (`ClassCastException`)
14. *Compiler-Fehler*: `X` verfügt über keine Methode `write(X)`.
15. *Laufzeitfehler* (`ClassCastException`)

91

1. `1`
2. `7` (= 2 + 4 + 1)
3. `7` (= 2 + 4 + 1)
4. `15` (= 8 + 2 + 4 + 1)
5. `5` (= 4 + 1)
6. `21` (= 16 + 4 + 1)
7. `1`
8. `7` (= 2 + 4 + 1)
9. `5` (= 4 + 1)
10. `5` (= 4 + 1)
11. *Laufzeitfehler* (`ClassCastException`):
    `a` kann nicht zu `B` gecastet werden, weil der dyn. Typ von `a` `A` ist und deshalb nicht mindestens so speziell ist wie `B` (d. h. `A` erbt nicht von `B`).
12. `5` (= 4 + 1)
13. `7` (= 2 + 4 + 1)
14. `21` (= 16 + 4 + 1)
15. `5` (= 4 + 1)
16. `5` (= 4 + 1)
17. *Laufzeitfehler* (`ClassCastException`):
    Der erste Cast (`(C)c`) bewirkt nichts, ist aber auch nicht falsch. Zweiter Cast: `b` kann nicht zu `C` gecastet werden, weil der dyn. Typ von `b` `B` ist und deshalb nicht mindestens so speziell ist wie `C` (d. h. `B` erbt nicht von `C`).
18. *Laufzeitfehler* (`ClassCastException`):
    Ein neues `Object` kann nicht zu `A` gecastet werden, weil der dyn. Typ `Object` ist und deshalb nicht mindestens so speziell wie `A` (d. h. `Object` erbt nicht von `A`).
19. *Compiler-Fehler*:
    Der Cast ist möglich (weil `B` von `A` von `Object` erbt), aber `C` verfügt über keine Methode `method(Object)`.

92

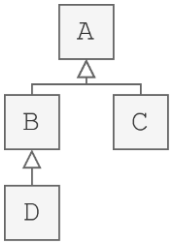
1. `D`
2. `G`
3. `D`
4. `G`
5. `D`
6. `E`
7. `ClassCastException`
8. *Compiler-Fehler*
9. `B`
10. `C`
11. `B`
12. `C`
13. `F`
    `A`
14. `C`
15. *Compiler-Fehler*
16. `F`
    `H`
17. `N`
18. `L`
19. `N`

(93)

	Statement	Ausgabe
1.	`System.out.println(z.a);` greift auf `Y.a` zu	7
2.	`System.out.println(z.get(z));` ruft `Z.get(X)` auf	4
3.	`System.out.println(((X) z).get());` ruft `get()` in `Z` auf	10
	`z.set('c' - 'a' - 1);` setzt `Y.a` auf 3 (`'c'-'a'-1` = 2-1 = 1)	
4.	`System.out.println(z.get(z));` ruft `Z.get(X)` auf	4
5.	`System.out.println(z.get());` ruft `get()` in `Z` auf	6
	`y.set(2);` ruft `Y.set(int)` auf (da statisch) und setzt `Y.a` auf 2	
6.	`System.out.println(z.get());` ruft `get()` in `Z` auf	5
	`z.set(y, 0);` ruft `Z.set(X, int)` auf und setzt `Y.a` auf 0	
7.	`System.out.println(y.get());` ruft `get()` in `Z` (dyn. Typ) auf	3

Nein, die Klasse `X` kann nicht `final` markiert werden, da sie Unterklassen besitzt.

94



In der Klausur wäre lediglich die Lösung (roter Text) anzugeben.

	Aufruf	Lösung	Erklärung oder Typenskizze
1.	`z.m(c)`	`D.m(A)`	Der stat. Typ von `z` ist `A`, d. h. wir suchen in `A` eine Methode `m` mit Parameter `C` (stat. Typ von `c`). Wir finden `m(C)` weder in `A` noch in einer Oberklasse von `A` (das wäre hier lediglich `Object`), also müssen wir den Parametertyp generalisieren (d. h. allgemeiner machen) und suchen somit als nächstes `m(A)` (weil `A` die Oberklasse von `C` ist) in `A`, welche wir finden. Wir führen diese – statisch gefundene – Methode *nicht* direkt aus, weil der dyn. Typ von `z` nicht `A` sondern `D` ist (→ dynamischer Dispatch nötig = Schritt ❷). Wir sehen also nach, ob die gefundene Methode `m(A)` in der Klasse `D` überschrieben wird. Das ist der Fall, weshalb wir `m(A)` in `D` ausführen.
2.	`x.m(d)`	`B.m(B)`	Der stat. Typ von `x` ist `A`, d. h. wir suchen in `A` eine Methode `m` mit Parameter `D` (stat. Typ von `d`). Wir finden `m(D)` weder in `A` noch in einer Oberklasse von `A`, also müssen wir den Parametertyp generalisieren (d. h. allgemeiner machen) und suchen somit als nächstes `m(B)` (weil `B` die Oberklasse von `D` ist) in `A`, welche wir finden. Wir führen diese – statisch gefundene – Methode *nicht* direkt aus, weil der dyn. Typ von `x` nicht `A` sondern `B` ist (→ dynamischer Dispatch nötig = Schritt ❷). Wir sehen also nach, ob die gefundene Methode `m(B)` in der Klasse `B` überschrieben wird. Das ist der Fall, weshalb wir `m(B)` in `B` ausführen.
3.	`((A)c).m(x)`	`C.m(A)`	Um den Aufruf durchzuführen, müssen wir zunächst den stat. und dyn. Typ des Ausdrucks `((A)c)` bestimmen. Es handelt sich hierbei um einen Cast der Variable `c` zum (stat.) Typ `A`. Der Cast führt *nicht* zu einem Compiler-Fehler, da der stat. Typ der Variable `c` (also `C`) in einer Vererbungsrelation mit der Cast-Klasse `A` steht. Der Cast führt außerdem nicht zu einem Laufzeitfehler, da der dyn. Typ der Variable `c` (also `C`) von der Cast-Klasse `A` erbt. Der Cast `((A)c)` funktioniert somit und hat den stat. Typ `A` (= Cast-Typ) und den dyn. Typ `C` (= dyn. Typ von `c`). Wir suchen somit in `A` (stat. Typ von `((A)c)`) eine Methode `m` mit Parameter `A` (stat. Typ von `x`). Wir finden `m(A)` in `A`. Wir führen diese – statisch gefundene – Methode *nicht* direkt aus, weil der dyn. Typ von `((A)c)` nicht `A` sondern `C` ist (→ dynamischer Dispatch nötig = Schritt ❷). Wir sehen also nach, ob die gefundene Methode `m(A)` in `C` überschrieben wird. Dies ist der Fall, weshalb wir `m(A)` in `C` ausführen.

4.	`c.m(b)`	`A.m(B)` `C.m(A)`	Der stat. Typ von `c` ist `C`, d. h. wir suchen in `C` eine Methode `m` mit Parameter `B` (stat. Typ von `b`). Wir finden `m(B)` zwar nicht direkt in `C` aber in der Oberklasse `A` (welche `C` somit erbt), d. h. wir wählen aus stat. Sicht die Methode `m(B)` in `A`. Diese Methode wird ausgeführt, da der dyn. Typ von `c` ebenfalls `C` ist (→ kein dyn. Dispatch) ⇒ `A.m(B)` Es folgt der Aufruf **`this.m((A)b)`**, weshalb wir zunächst den stat. und dyn. Typ von `this` und `(A)b` bestimmen: • Der stat. Typ von `this` ist `A` (= Klasse, in der wir uns befinden) und der dyn. Typ ist `C` (= dyn. Typ des Objekts, das beim vorherigen Aufruf (hier: `c.m(b)`) vor dem Punkt stand, hier also dyn. Typ vom vorherigen `c`). • Um `(A)b` zu bestimmen, ermitteln wir zunächst den stat. und dyn. Typ von `b`. Der stat. Typ von `b` ist `B` (weil als Parameter so deklariert). Der dyn. Typ von `b` entspricht dem dyn. Typ des Objekts, das im vorherigen Aufruf als Parameter übergeben wurde (→ `c.m(b)`), also `B` (s. o.). Nun zum Cast: Der stat. Typ von `b` ist `B` und steht in einer Vererbungsrelation mit der Cast-Klasse `A`, d. h. es gibt *keinen* Compiler-Fehler. Der dyn. Typ von `b` ist `B` und erbt vom Cast-Typ `A`, d. h. es kommt zu *keinem* Laufzeitfehler. Der stat. Typ `(A)b` von ist somit `A` (= Cast-Typ), der dyn. Typ ist `B` (dyn. Typ von `b`). Wir suchen folglich in `A` (stat. Typ von `this`) eine Methode `m` mit Parameter `A` (stat. Typ von `(A)b`), welche wir finden. Wir führen `A.m(A)` jedoch *nicht* direkt aus, weil der dyn. Typ von `this` nicht `A` sondern `C` ist (→ dynamischer Dispatch nötig = Schritt ❷). Wir sehen also nach, ob die gefundene Methode `m(A)` in der Klasse `C` überschrieben wird. Das ist der Fall, weshalb wir `m(A)` in `C` ausführen ⇒ `C.m(A)`
5.	`x.m(x)`	`A.m(A)` `B.m(B)`	Erklärung folgt...
6.	`z.m(b)`	`D.m(B)` `B.m(B)`	Erklärung folgt...

7.	`new A().m(b)`	`A.m(B)`
		`A.m(A)`
		`B.m(B)`
			Erklärung folgt...

8.	`x.m(c)`	`A.m(A)` *dann Laufzeitfehler beim Cast*	Erklärung folgt...

9.	`z.m(d)`	`D.m(B)`
		`D.m(B)`
		`B.m(B)`
			Erklärung folgt...

10. `b.m(c)`	`B.m(C)`
	`B.m(B)`
	`A.m(B)`
	`C.m(A)`
		Erklärung folgt...

11. `((B)c).m(x)`	*Compiler-Fehler beim Cast*	Erklärung folgt...

12. `x.m(z)`	`A.m(A)`
	`D.m(B)`
	`D.m(B)`
	`B.m(B)`
		Erklärung folgt...

13. `b.m(a)`	*Compiler-Fehler, weil `a` nicht definiert ist*	Erklärung folgt...

14. `d.m(c)`	`D.m(C)`
	`C.m(C)`
	`B.m(C)`
	`D.m(B)`
	`D.m(B)`
	`B.m(B)`
	`A.m(B)`
	`C.m(A)`
		Erklärung folgt...

15. `c.m(c.m(d))`	`C.m(D)` `C.m(A)`	Erklärung folgt...

16. `c.m(c).m(d)`	`C.m(C)`
	`A.m(B)`
	`C.m(A)`
		Erklärung folgt... (geändert, alte Version `c.m(c).m(c)` gleich aber ohne Ausgabe `A.m(B)`)

17. `c.m(d)`
`.m(c.m(c))`
	`C.m(D)`
	`C.m(C)`
	`D.m(A)`
		Erklärung folgt...

	Statement	Ergebnis
a)	`K k = new L();` `k.c(k);`	`K.c`
b)	`(new L()).a();`	`K.c`
c)	`((K) (new L())).c(new L());`	`K.c`
d)	`L l = (L) (new K());` `l.c();`	*Laufzeitfehler* (beim Cast)
e)	`((K) new L()).b();`	`K.b`
f)	`(((new L()))).b(new L());`	`L.b`
g)	`L l = new L();` `((K) l).c(new L());`	`K.c`
h)	`new K().c(new L());`	`K.c`
i)	`K l = new L();` `((K) new L()).b(l);`	*Compiler-Fehler* (`b(K)` ist privat)
j)	`new L().a();`	`K.c`
k)	`((K) (L) new K()).c(new L());`	*Laufzeitfehler* (beim Cast)
l)	`(K) new L().c();`	*Compiler-Fehler* (Syntaxfehler: Punkt bindet stärker als Cast)
m)	`K l = new L();` `l.c(l);`	`K.c`
n)	`L l = (L) (K) (new L());` `l.b();`	`K.b`
o)	`((K)(K)(K) new K()).c(new L());`	`K.c`
p)	`L l; K k = l = new L();` `k.c(l);`	`K.c`
q)	`M<K> m = new M<K>(); m.c(m);`	`M<N>.c`
r)	`M<K> m = new M<L>(); m.c(m);`	*Compiler-Fehler* (muss `<K>` sein)
s)	`M<L> m = new M<L>(); m.c(m);`	`M<N>.c`
t)	`(new M<K>()).c(new L());`	`K.c`
u)	`(new M<new K()>()).c();`	*Compiler-Fehler* (Objekt als Typ)
v)	`new M<M<K>>().b();`	`K.b`
w)	`M<K> m = new M<K>();` `((K)m).c(m);`	`K.c`
x)	`new M<K>().c(new M<L>());`	`K.c`

y)	`class Y extends L {`
	`  private void b(K k) {}`
	`}`
		Compiler-Fehler (Reduzierung der Sichtbarkeit der geerbten Methode)

z)	`class Z<R> extends M<R> { }`	*Keine Ausgabe* (kompiliert)
A)	`class A extends M<N> { }`	*Compiler-Fehler* (`N` undefiniert)
B)	`class B<R,S> extends M<R> { }`	*Keine Ausgabe* (kompiliert)
C)	`class C extends M<K> { }`	*Keine Ausgabe* (kompiliert)
D)	`abstract class D extends M<L> { }`	*Keine Ausgabe* (kompiliert)
E)	`new L().c((L) ((K) new L()));`	`L.c`
F)	`class F<P> extends M<M<P>> { }`	*Keine Ausgabe* (kompiliert)
G)	`class G extends M { }`	*Keine Ausgabe* (kompiliert, leider)

96

Es wird die Ausgabe `0 1 1 3` produziert. Hieran soll man sehen, dass die `accept`-Methode tatsächlich in allen Klassen implementiert werden muss, selbst wenn eine Klasse sie scheinbar von der Oberklasse erbt. Der Grund dafür ist, dass Parameter immer nur nach dem statischen Typ gebunden werden. Der Aufruf `v.visit(this)` hat `this` als Parameter, d. h. es wird die Klasse benutzt, in der der Aufruf stattfindet.

Aus genau diesem Grund rufen wir die `visit`-Methode auch niemals direkt auf, sondern gehen immer den Umweg über die `accept`-Methode! Nur in den `accept`-Methoden wird `visit` aufgerufen. Betrachte dazu auch folgendes Beispiel:

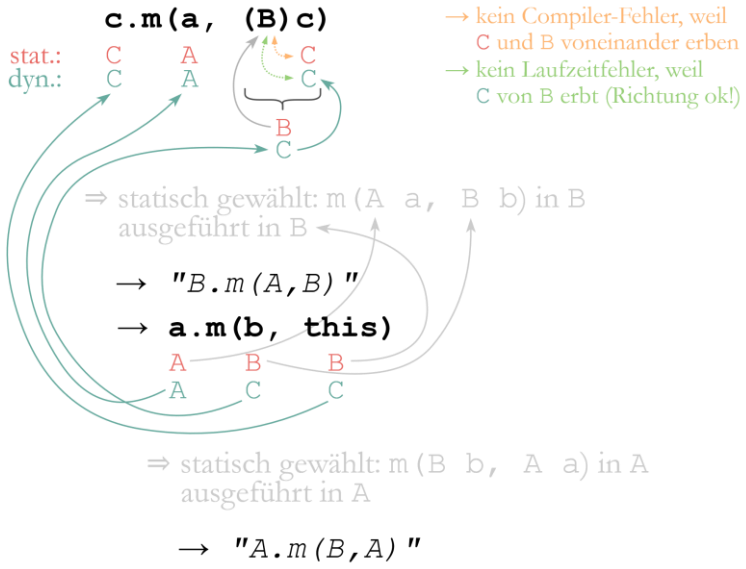
```
Visitor v = new Visitor();
Parent obj = new Child2();
obj.accept(v); // ruft v.visit(Child2) auf (richtig)
v.visit(obj); // ruft v.visit(Parent) auf (falsch)
```

(97)

Da das Lösungsverfahren immer gleich ist, werden die Erklärungen nach und nach durch Typenskizzen ersetzt, aus welchen hervorgeht, woher der statische und dynamische Typ eines jeden Ausdrucks kommt. Falls du Probleme mit einem Aufruf hast, lies dir daher bitte auch die Erklärung zu den vorherigen Aufrufen durch. Dort könnten gewisse Stellen genauer erklärt worden sein. Es ist wichtig, dass du richtig vorgehst und nicht nur durch Zufall auf die richtige Lösung kommst. Ich empfehle, in der Klausur mit einer Typenskizze zu arbeiten, um die dyn. Typen richtig zu bestimmen. Hellgraue Pfeile sind hier nur der Vollständigkeit wegen eingezeichnet, damit du siehst, wo du die statischen Typen ablesen musst. Diese Pfeile würde ich nicht einzeichnen, sondern eben einfach direkt ablesen und dazuschreiben.

Zunächst sehen wir anhand der Deklarationen / Zuweisungen zu Beginn der `main`-Methode: Der stat. / dyn. Typ von `a` ist `A` / `A` (wegen `A a = new A();`) und `B` / `B` für `b` bzw. `C` / `C` für `c`.

Aufruf	Lösung	Erklärung
1.	`A.m(B,A)`	Wir suchen in der Klasse `A` (stat. Typ von `a`) eine Methode `m` mit den Parametern `C` (stat. Typ von `c`) und `B` (stat. Typ von `b`). `m(A,A)` in `A` wäre möglich, weil wo ein `A` erwartet wird (1. und 2. Parameter) auch jede speziellere Klasse eingesetzt werden kann, insb. also `B` und `C`. Wir wählen jedoch `m(B,A)` in `A`, weil die Methode im ersten Parameter spezieller ist als `m(A,A)`. Sie kann gewählt werden kann, weil wo ein `B` erwartet wird (1. Parameter) auch ein `C` eingesetzt werden kann und wo ein `A` erwartet wird (2. Parameter) auch ein `B` eingesetzt werden kann. `m(A,C)` in `A` können wir hingegen nicht wählen, weil der erwartete zweite Parameter `C` ist, wir aber nur ein `B` haben (was nicht mind. so speziell ist wie `C`). Wir führen die gefundene Methode `m(B,A)` aus `A` direkt aus, weil der dyn. Typ von `a` ebenfalls `A` ist (dyn. Typ = stat. Typ, daher keine Überschreibung möglich).
2.	`B.m(A,B)` `A.m(B,A)`	Zunächst bestimmen wir stat./dyn. Typ des Ausdrucks `(B)c`: Der stat. Typ von `c` ist `C` und steht in einer Relation mit der Cast-Klasse `B`, daher gibt es keinen Compiler-Fehler beim Cast. Der dyn. Typ von `c` ist `C` und erbt von der Cast-Klasse `B`, daher gibt es keinen Laufzeitfehler. Der Cast funktioniert und hat den stat. Typ `B` (= Cast-Klasse) und dyn. Typ `C` (= dyn. Typ von `c`). Jetzt zum Aufruf: Wir suchen in der Klasse `C` (stat. Typ von `c`) eine Methode `m` mit den Parametern `A` (stat. Typ von `a`) und `B` (stat. Typ von `(B)c`). Die beiden Methoden aus Klasse `C` sind nicht wählbar, weil beide als ersten Parameter ein `B` erwarten, wir aber nur ein `A` übergeben. Weil `C` von `B` erbt, können wir in der Vererbungshierarchie nach oben zu `B` gehen. Dort finden wir die passende Methode `m(A,B)`. Weil die Methode perfekt passt, müssen wir *nicht* noch in die Klasse `A` gehen, um nach einer spezielleren Methode zu suchen.

		Wir führen die gefundene Methode `m(A,B)` aus `B` direkt aus, weil der dyn. Typ von `c` ebenfalls `C` ist (dyn. Typ = stat. Typ, daher keine Überschreibung möglich). Es folgt die Ausgabe „*B.m(A,B)*“ und der Aufruf `a.m(b,this)`: Wir bestimmen den stat. Typ von `a`, `b` und `this`. Der stat. Typ von `a` und `b` ist an der Methodendefinition `void m(A a, B b))` ablesbar. Hier werden die Variablen definiert. Das hat also nichts mit dem `a` bzw. `b` aus der `main`-Methode zu tun! Der stat. Typ von `this` ist `B`, weil wir uns gerade in der Klasse `B` befinden. *Wie sieht es mit den dyn. Typen aus?* Der dyn. Typ von `a` ist `A`, weil `a` der 1. Parameter ist, für welchen zuvor `a` übergeben wurde (`c.m(a,(B)c)`), was den dyn. Typ `A` hatte. Der dyn. Typ von `b` ist `C`, weil `C` der 2. Parameter ist, für den zuvor `(B)c` übergeben wurde, was den dyn. Typ `C` hatte. Der dyn. Typ von `this` ist `C`, weil das Objekt, auf dem wir operieren `c` ist, was den dyn. Typ `C` hatte (`this` kommt von der Var. vor dem Punkt: `c.m(a,(B)c)`). Wir suchen folglich in `A` (stat. Typ von `a`) eine Methode `m` mit den Parametern `B` (stat. Typ von `b`) und `B` (stat. Typ von `this`). Wie bei Aufruf 1 finden wir `m(B,A)` in `A`, weil die Methode spezieller ist als `m(A,A)` und `m(A,C)` nicht passt. Wir führen die gefundene Methode `m(B,A)` aus `A` direkt aus, weil der dyn. Typ von `a` ebenfalls `A` ist (dyn. Typ = stat. Typ, keine Überschreibung möglich). **Typenskizze:** 
3.	*Compiler-Fehler*	Wir suchen in `B` (stat. Typ von `b`) eine Methode `m` mit den Parametern `C` (stat. Typ von `c`) und `B` (stat. Typ von `b`). `m(A,B)` in `B` ist möglich, da wo ein `A` verlangt wird (1. Parameter) auch ein `C` eingesetzt werden kann. Anschließend suchen wir in der Ober-

		klasse von `B` – das ist `A` – nach einer spezielleren Methode (nachdem `B` die Methoden von `A` erbt). `m(A,A)` würde passen, ist aber im 2. Parameter allgemeiner als `m(A,B)`. `m(A,C)` passt nicht, weil wo ein `C` erwartet wird (2. Parameter) kein `B` eingesetzt werden kann (sondern nur `C` und Unterklassen von `C`). `m(B,A)` ist möglich, weil wo ein `B` verlangt wird (1. Parameter) auch ein `C` eingesetzt werden kann und wo ein `A` verlangt (2. Parameter) wird auch ein `B` eingesetzt werden kann. Wir wählen immer diejenigen Methode, die in *allen* Parametern am besten passt. Suchen wir bspw. nach `g(C,C)` und finden nur `g(A,B)` und `g(C,B)` als passende Kandidaten, dann könnten wir uns eindeutig für `g(C,B)` entscheiden, da der 1. Parameter spezieller und der 2. genauso gut passt wie bei der `g(A,B)`. Hier suchen wir nach `m(C,B)` (in `B`) und finden die beiden am besten passenden Kandidaten `m(A,B)` in `B` und `m(B,A)` in `A`. Keine dieser Methoden ist *in allen Parametern* spezieller als die andere, d. h. der Aufruf ist mehrdeutig (*ambiguous*) und kann daher nicht übersetzt werden.
4.	`A.m(B,A)`	Zunächst bestimmen wir stat./dyn. Typ des Ausdrucks `(A)b`: Der stat. Typ von `b` ist `B` und steht in einer Relation mit der Cast-Klasse `A`, daher gibt es keinen Compiler-Fehler beim Cast. Der dyn. Typ von `b` ist `B` und erbt von der Cast-Klasse `A`, daher gibt es auch keinen Laufzeitfehler. Der Cast funktioniert und hat den stat. Typ `A` (= Cast-Klasse) und dyn. Typ `B` (= dyn. Typ von `b`). Jetzt zum Aufruf: Wir suchen in `A` (stat. Typ von `((A)b)`) eine Methode `m` mit den Parametern `B` (stat. Typ von `b`) und `B` (stat. Typ von `b`). Wie in Aufruf 1 finden wir die beiden Kandidaten `m(A,A)` und `m(B,A)`. Wir wählen die speziellere Methode, also `m(B,A)`. Wir führen diese Methode *nicht* direkt aus, weil der dyn. Typ von `((A)b)` nicht `A` sondern `B` ist (→ Schritt ❷). Wir sehen also nach, ob die Methode `m(B,A)` in `B` überschrieben wird. Dies ist nicht der Fall, also bleiben wir in `A` und führen `m(B,A)` aus.
5.	*Compiler-Fehler*	Wir suchen in `C` (stat. Typ von `c`) eine Methode `m` mit den Parametern `B` (stat. Typ von `b`) und `B` (stat. Typ von `b`). In `C` selbst käme nur `m(B,A)` in Frage, weil wo ein `A` verlangt wird (2. Parameter) auch ein `B` eingesetzt werden kann. `m(B,C)` passt nicht, weil wo ein `C` erwartet wird (2. Parameter) kein `B` eingesetzt werden kann (sondern nur `C` und Unterklassen von `C`). Anschließend suchen wir in der Oberklasse von `C` – das ist `B` – und finden `m(A,B)`. Die Methode ist ebenfalls möglich, weil wo ein `A` verlangt wird (1. Parameter) auch ein `B` eingesetzt werden kann. In `A` (Oberklasse von `B`) finden wir keine spezielleren Methoden, d. h. dass der Aufruf aufgrund von Mehrdeutigkeit zwischen `m(A,B)` aus `B` und `m(B,A)` aus `C` nicht übersetzt werden (vgl. 3.).

6.

```
C.m(B,A)
A.m(A,A)
A.m(A,C)
```

(1.) Der Cast `(A)c` funktioniert, weil der stat. Typ von `c` in Relation zum Cast-Typ steht und die Richtung stimmt, d. h. der dyn. Typ von `c` erbt vom Cast-Typ. Der stat. Typ von `((A)c)` ist `A` (= Cast-Klasse) und der dyn. Typ ist `C` (dyn. Typ von `c`).

Wir suchen somit in `A` (stat. Typ von `((A)c)`) eine Methode `m` mit Parametern `C` (stat. Typ von `c`) und `B` (stat. Typ von `b`). Wir finden `m(A,A)` und `m(B,A)` und wählen die speziellere, also `m(B,A)`. Weil der dyn. Typ von `c` nicht `A` sondern `C` ist, sehen wir nach, ob `m(B,A)` in `C` überschrieben wird) (→ Schritt ❷ = dyn. Dispatch). Dies ist der Fall, weshalb `m(B,A)` aus `C` ausgeführt wird. Es folgt die Ausgabe „*`C.m(B,A)`*“ und der Aufruf `m((A)this, a)`:

(2.) Da wir uns in einer nicht-statischen Methode befinden und vor dem Methodennamen dieses Mal kein Objekt steht, können wir uns „`this.`“ davor denken, d. h. der nächste Aufruf erfolgt auf demselben Objekt, auf dem wir gerade operieren (stand beim vorherigen Aufruf vor dem Punkt, hier also `((A)c)` und somit `c`).

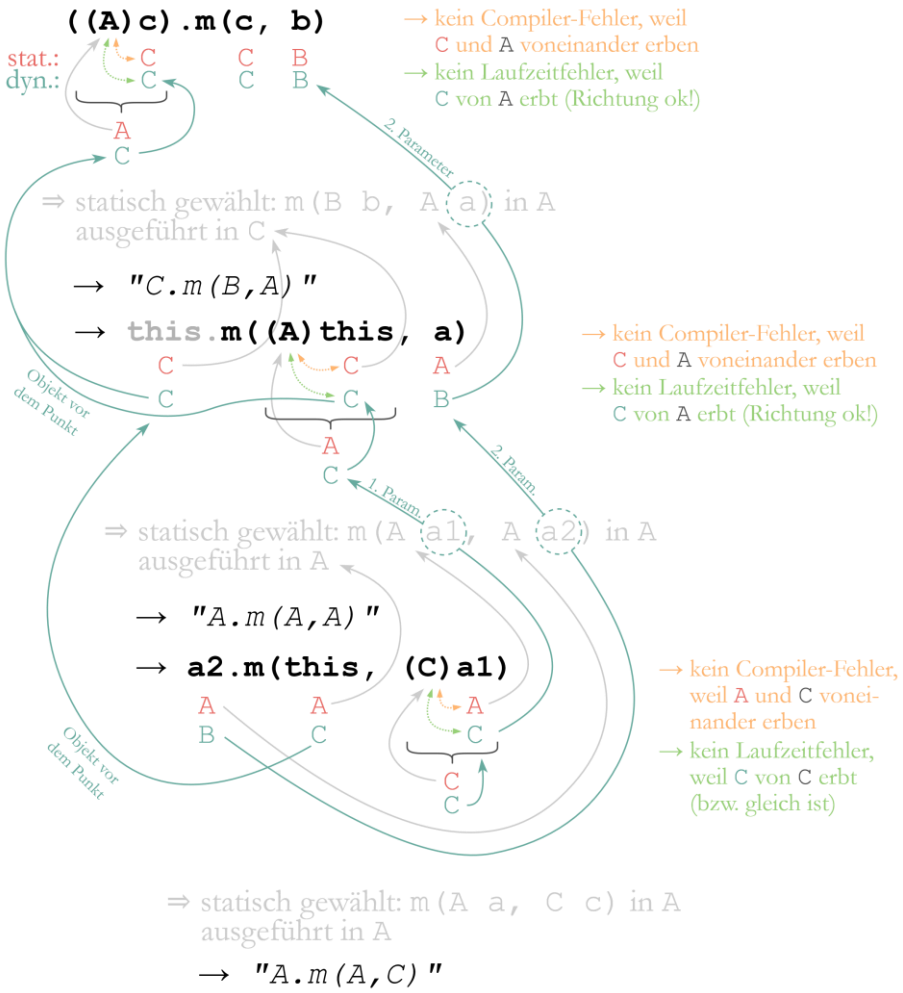
Außerdem überprüfen wir den Cast `(A)this`. Der stat. Typ von `this` ist `C` (weil wir in der Klasse `C` sind) und steht in einer Vererbungsrelation zum Cast-Typ `A`, d. h. der Cast kompiliert. Der dyn. Typ von `this` ist `C` (= dyn. Typ des Objekts, was beim vorherigen Aufruf vor dem Punkt stand, siehe Typenskizze) und `C` erbt vom Cast-Typ `A`, d. h. es gibt keinen Laufzeitfehler.

Wir suchen somit in `C` (stat. Typ des „gedachten“ `this`) eine Methode `m` mit Parametern `A` (stat. Typ von `(A)this`) und `A` (stat. Typ des Parameters `a`). In `C` selbst finden wir keine passende Methode, allerdings erbt `C` (von `B`) von `A` und verfügt somit über die Methode `m(A,A)` aus `A`, welche wir auswählen. Wir führen diese Methode direkt aus, weil der dyn. Typ vom „gedachten“ `this` ebenfalls `C` ist (dyn. Typ = stat. Typ, keine Überschreibung möglich). Es folgt die Ausgabe „*`A.m(A,A)`*“ und der Aufruf `a2.m(this, (C)a1)`:

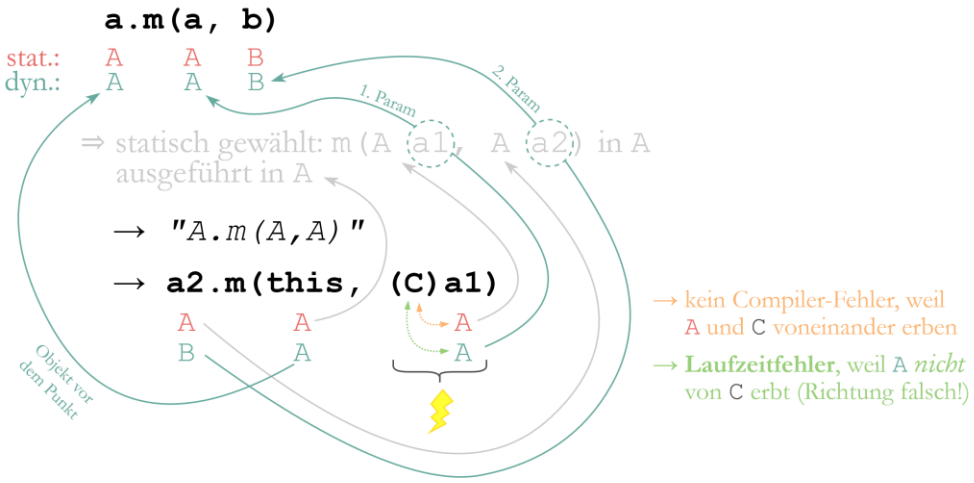
(3.) Wir werten zunächst den Cast aus: Der stat. Typ von `a1` ist `A` und steht in Relation zum Cast-Typ `C`, d. h. der Cast kompiliert (wird er innerhalb von Klasse immer). Der dyn. Typ von `a1` ist `C` (weil `a1` der erste Parameter war, für den beim vorherigen Aufruf `a` übergeben wurde, was den dyn. Typ `C` hatte, siehe Typenskizze!), d. h. der Cast funktioniert. Der stat. Typ von `(C)a1` ist `C` (= Cast-Klasse), der dyn. Typ ist `C` (dyn. Typ von `a1`).

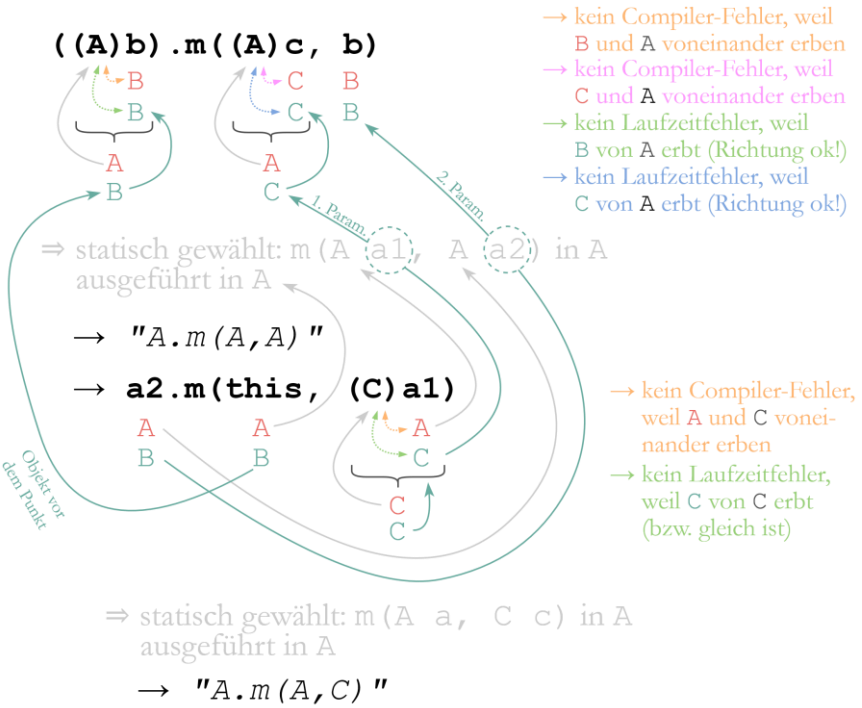
Wir suchen somit in `A` (stat. Typ von `a2`) eine Methode `m` mit Parametern `A` (stat. Typ von `this`) und `C` (stat. Typ von `(C)a1`). Wir finden `m(A,C)` in `A`. Wir führen die gefundene Methode *nicht* direkt aus, weil der dyn. Typ von `a2` nicht `A` sondern `B` ist (weil `a2` der 2. Parameter war, für welchen zuvor `a` übergeben wurde, wofür `b` übergeben wurde, siehe Typenskizze) (→ Schritt ❷).

Typenskizze:



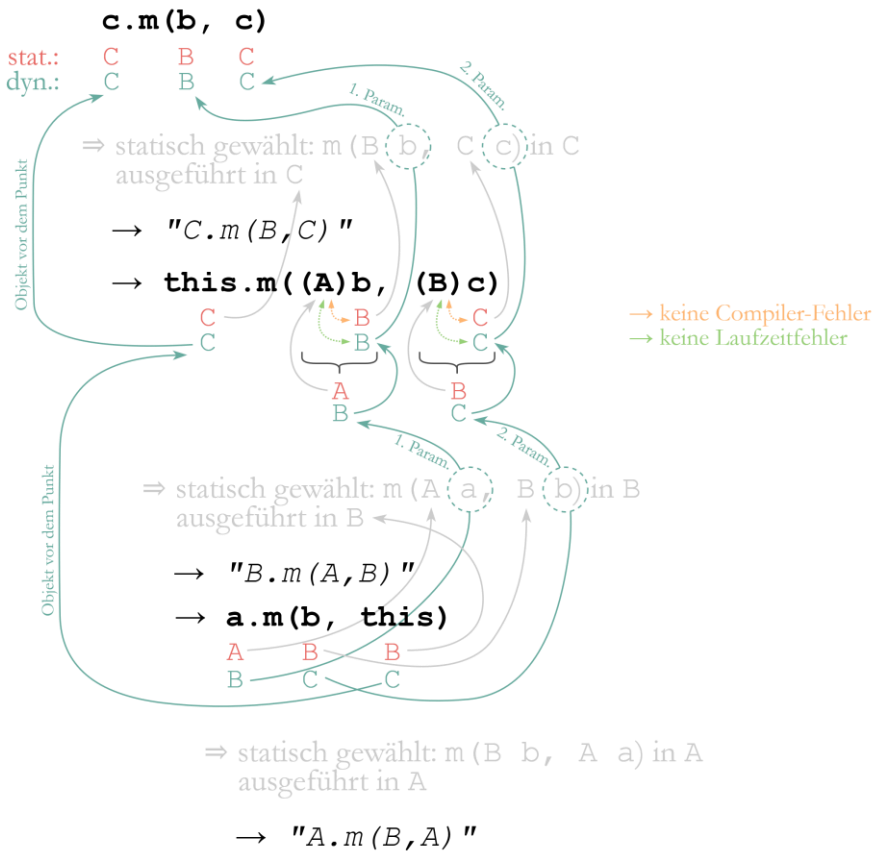
7.	`A.m(A,A)` *dann* *Laufzeitfehler beim Cast*	Wir suchen in `A` (stat. Typ von `a`) eine Methode `m` mit Parametern `A` (stat. Typ von `a`) und `B` (stat. Typ von `b`). `m(A,A)` ist die einzige passende Methode. Wir führen `m(A,A)` in `A` direkt aus, weil der dyn. Typ von `a` ebenfalls `A` ist. Es folgt die Ausgabe „*A.m(A,A)*“ und der Aufruf `a2.m(this, (C)a1)`: Wir werten zunächst den Cast aus: Der stat. Typ von `a1` ist `A` und steht in Relation zum Cast-Typ `C`, d. h. der Cast kompiliert. Der dyn. Typ von `a1` ist `A` (weil `a1` der erste Parameter war, für den beim vorherigen Aufruf `a` übergeben wurde, was den dyn. Typ `A` hatte, siehe Typenskizze). Da `A` nicht mindestens so speziell ist wie der Cast-Typ `C`, führt dieser Cast zur Laufzeit zu einer `ClassCastException` (denn `A` erbt nicht von `C`, sondern andersrum). Die vorherige Ausgabe muss trotzdem angegeben werden, weil sie zuvor ausgeführt wurde (anders bei Rückgaben). **Typenskizze:** *Siehe nächste Seite.*



8.	`A.m(A,A)` `A.m(A,C)`	Zunächst überprüfen wir die beiden Casts auf Compiler-Fehler, anschließend beide auf Laufzeitfehler - beide funktionieren. Wir suchen in `A` (stat. Typ von `((A)b)`) eine Methode `m` mit Parametern `A` (stat. Typ von `(A)c`)) und `B` (stat. Typ von `b`). Wir finden `m(A,A)` als einzige passende Methode. Weil `((A)b)` dynamisch jedoch `B` ist sehen wir nach, ob `m(A,A)` in `B` überschrieben wird. Das ist nicht der Fall, also führen wir `m(A,A)` in `A` aus. Es folgt die Ausgabe „`A.m(A,A)`“ und der Aufruf `a2.m(this, (C)a1)`: Der Cast `(C)a1` ist möglich, weil `a1` das im ersten Parameter übergebene Objekt speichert, was im vorherigen Aufruf `(A)c` war und deshalb dynamisch `C` ist. Wir suchen in `A` (stat. Typ von `a2`) eine Methode `m` mit Parametern `A` (stat. Typ von `this`) und `C` (stat. Typ von `(C)a1`). Wir finden `m(A,C)` in `A`. Da `a2` dyn. jedoch `B` ist (war der zweite Parameter, also `b`), sehen wir nach, ob `m(A,C)` in `B` überschrieben wird. Das ist nicht der Fall, also bleiben wir bei `m(A,C)` in `A`. 

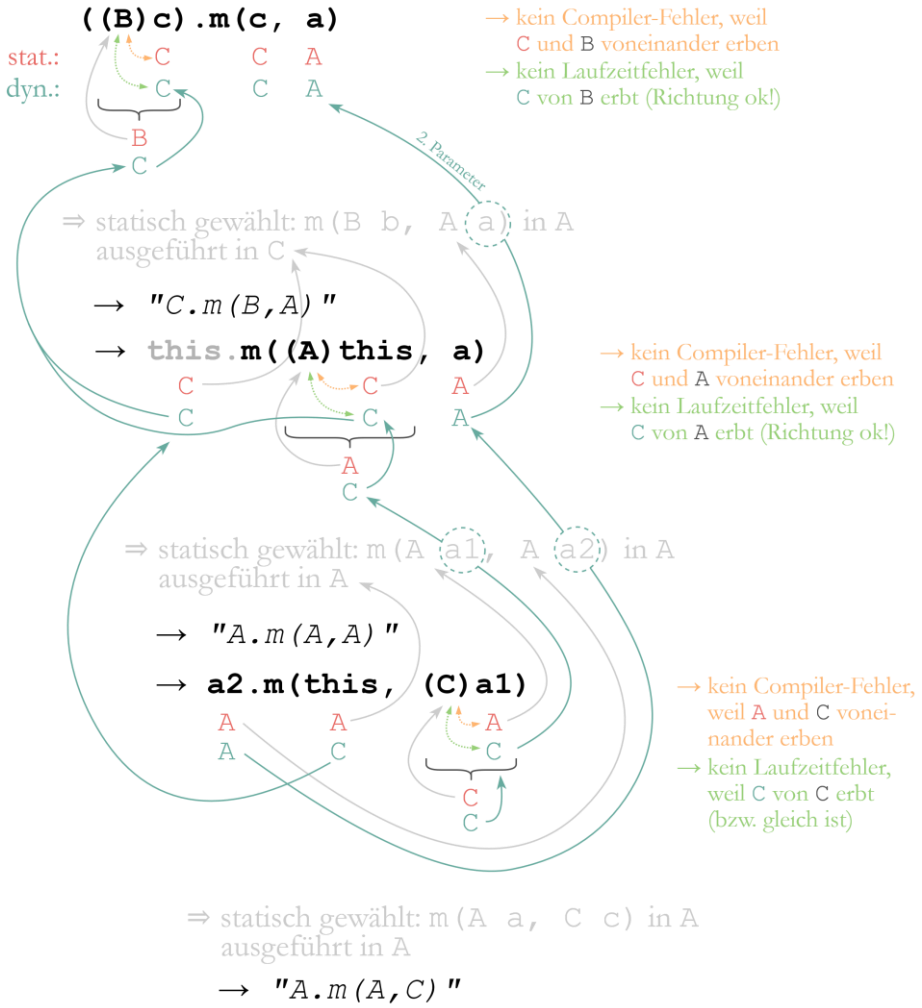
9.

```
C.m(B,C)
B.m(A,B)
A.m(B,A)
```

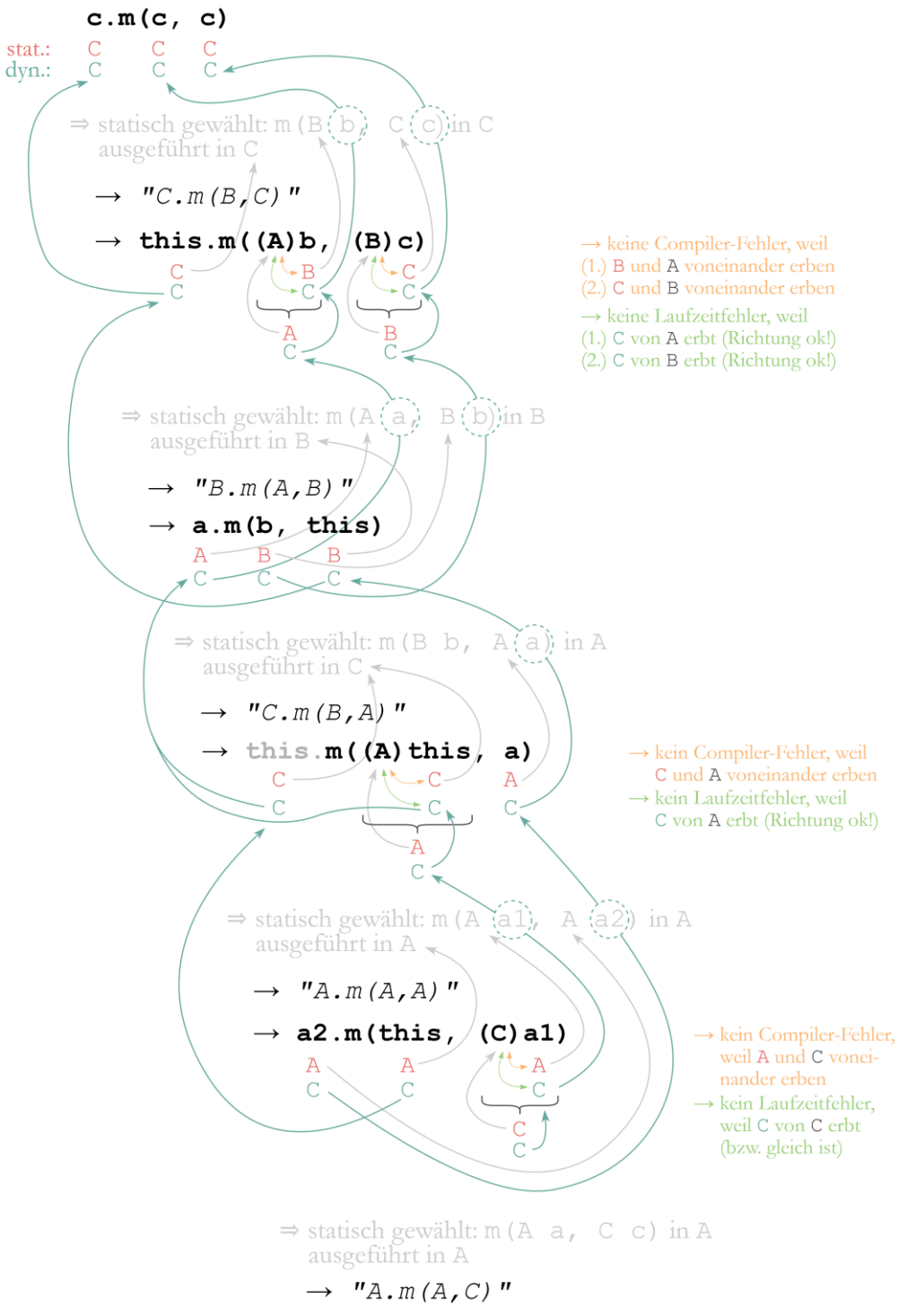


10.

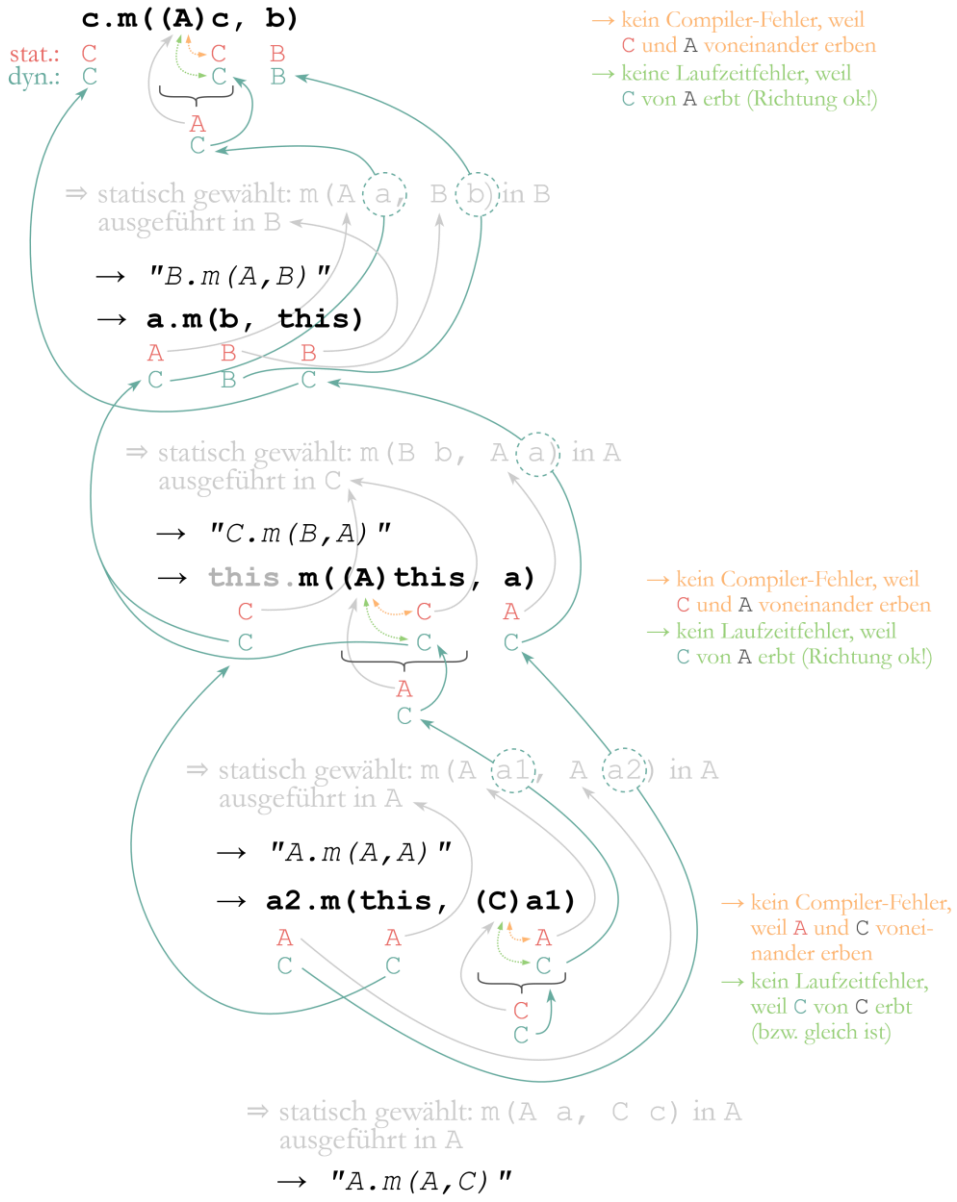
```
C.m(B,A)
A.m(A,A)
A.m(A,C)
```



11.	*Compiler-Fehler*	Wir suchen in `B` eine Methode `m` mit Parametern `C` und `C`. Kompatibel sind die Methoden `m(A,B)` aus `B` sowie die aus `A` geerbten Methoden `m(A,A)`, `m(A,C)` und `m(B,A)`. Die Kandidaten `m(A,A)` und `m(A,B)` können wir ausschließen, weil `m(A,C)` spezieller ist. Zwischen `m(B,A)` und `m(A,C)` herrscht hingegen keine Ordnung, weil keine der beiden Methoden *in allen Parametern* mindestens genauso speziell ist wie die andere. Der Aufruf ist somit *nicht eindeutig* (*ambiguous*).

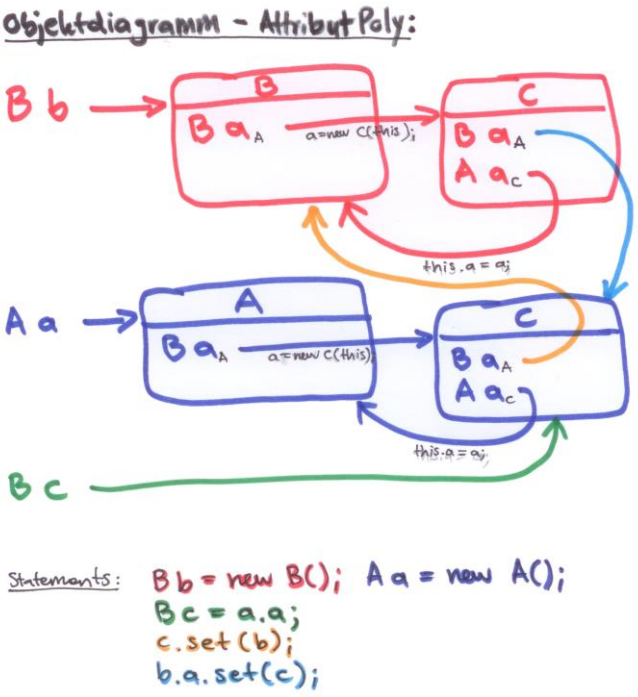
12.	`C.m(B,C)`
	`B.m(A,B)`
	`C.m(B,A)`
	`A.m(A,A)`
	`A.m(A,C)`
		

13.	*Compiler-Fehler*	Wir suchen in `A` eine Methode `m` mit Parametern `B` und `C`. Kompatibel sind die Methoden `m(A,A)`, `m(A,B)` und `m(A,C)`, wobei wir `m(A,A)` ausschließen können, da die anderen beiden in jeweils einem Parameter spezieller sind (und im anderen Parameter gleich speziell). Zwischen `m(B,A)` und `m(A,C)` herrscht jedoch keine Ordnung, weil keine der beiden *in allen Parametern* mindestens genauso speziell ist wie die andere. Der Aufruf ist somit *nicht eindeutig* (*ambiguous*) und kompiliert deshalb nicht.

14.	`B.m(A,B)`
	`C.m(B,A)`
	`A.m(A,A)`
	`A.m(A,C)`
		

15.	*Laufzeitfehler beim Cast*	Zunächst bestimmen wir stat. und dyn. Typ des Ausdrucks `(C)a`: Der stat. Typ von `a` ist `A` und steht in einer Relation mit der Cast-Klasse `C`, daher gibt es keinen Compiler-Fehler. Der dyn. Typ von `a` ist `A` und erbt *nicht* von der Cast-Klasse `C`, deshalb kommt es beim Ausführen zu einer `ClassCastException`.

Polymorphie mit Attributen (`AttributPoly`)

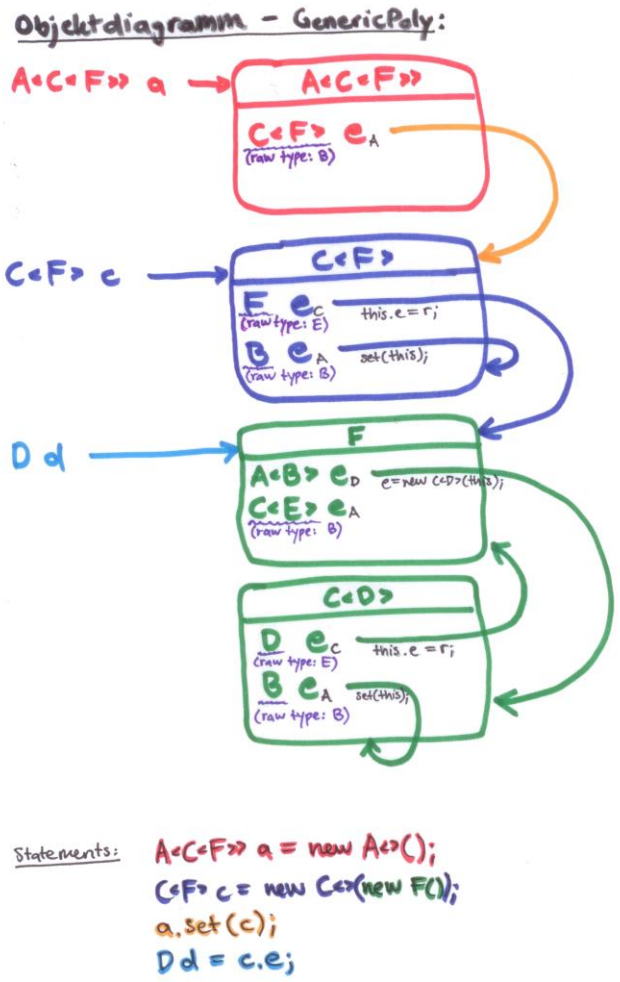


	Aufruf	Lösung	Erklärung oder Typenskizze
1.	`b.f(a)`	`B.f(A)` `C.f(A)`	Erklärung folgt...

2.	`c.f((C)c)`	`C.f(B)`
		`A.f(B)`
		`C.f(A)`
			Erklärung folgt...

3.	`a.f(a)`	*Compiler-Fehler* *(weil keine passende Methode)*	Erklärung folgt...
4.	`b.a.f(c)`	`C.f(B)` `B.f(B)`	Erklärung folgt...
5.	`a.f(b)`	`A.f(B)` `C.f(A)`	Erklärung folgt...
6.	`((C)c).a.f(c)`	`A.f(B)` `C.f(A)`	Erklärung folgt...
7.	`((C) a.a.a.a).a.a.a.f(a)`	`C.f(A)`	Erklärung folgt...

Polymorphie mit Generics (`GenericsPoly`)



	Aufruf	Lösung	Erklärung oder Typenskizze
1.	`d.f(c)`	`A.f(B)`	Erklärung folgt...
2.	`c.f(a.e)`	`C.f(B)`	Erklärung folgt...
3.	`d.f(a)`	*Compiler-Fehler* *(weil keine passende Methode)*	Erklärung folgt...

4.	`d.f(d)`	`F.f(D)`
		`D.f(D)`
		`B.f(C)`
			Erklärung folgt...

5.	`new B().f(c.e)`	`B.f(F)` `A.g(F)`	Erklärung folgt...

6.	`c.f(d)`	`C.f(A)`
		`A.f(B)`
		`D.g(E)`
			Erklärung folgt...

7.	`d.e.f(c)`	`C.f(B)`	Erklärung folgt...
8.	`a.f((C<E>)c)`	*Compiler-Fehler beim Cast*	Erklärung folgt...

9.	`((F)d).f(a)`	`F.f(A)`
		`A.f(B)`
		`C.f(B)`
			Erklärung folgt...

10.	`c.f((F)d)`	`B.f(F)`
		`C.g(F)`
		`D.g(E)`
			Erklärung folgt...

11.	`a.f(`
	`((A<C<E>>)d).e`
	`)`
		`A.f(C)`
		`C.f(B)`
		dann ***Null-Pointer-Exception*** *bei* `c.f(e)`
			Erklärung folgt...

100

Teilaufgabe (1)

1. `T.m(65.0)`
2. `T.m((short)1), T.m(-1.0), S.m(1.0,1), T.m(1.0)`
3. `T.m(2.0)`
4. `S.m(2.0,4), T.m(4.0)`
5. `T.m(1.0)`
6. `ClassCastException`: `t1` speichert ein `S<Integer>` und ein `S<Integer>` ist kein `X<Integer>`.
7. `T.m(5.0)`
8. `T.m((short)2), X.m(0.0), S.m(2.0,2), X.m(2.0)`
9. `X.m(5.0)`
10. *Compiler-Fehler*: `in == 1` ist ein Vergleich zweier Zahlen (`in` automatisch zu `int` entpackt), der zu einem `boolean` auswertet. Es gibt keine passende Methode für den Aufruf `m(boolean)`; `boolean` kann *nicht* implizit/explizit zu `int` gecastet werden.
11. `S.m(5.0,5), X.m(5.0)`
12. `S.m(4.0,1), X.m(1.0)`
13. `X.m(2), S.m(2.0,2), X.m(2.0)`
14. `T.m((short)1), X.m(-1.0), S.m(1.0,1), X.m(1.0)`
15. `S.m(1.0,7), X.m(7.0)`
16. `X.m(3,5)`
17. `X.m(9.0)`
18. *Compiler-Fehler*: Der Cast scheitert schon aus statischer Sicht, weil `T<Integer>` in keiner Vererbungsrelation zu `S<Object>` steht.
19. `S.m(2.0,4), X.m(4.0)`

20. `X.m(65), S.m(65.0,65), X.m(65.0)`

21. `T.m((short)6), X.m(4.0), S.m(6.0,6), X.m(6.0)`

22. *Compiler-Fehler*: Der Cast scheitert schon aus statischer Sicht, weil `T<Integer>` in keiner Vererbungsrelation zu `U` steht. `U` erbt von `T<Character>`. Der generische Typ ist wesentlicher Bestandteil der Vererbung.

23. *Compiler-Fehler*: Tatsächlich speichert `t2` ein `X<Integer>`. `t3` hat jedoch den statischen Typ `T<Character>`, d. h. es können nur Variablen zugewiesen werden, deren statischer Typ mindestens `T<Character>` ist. `t2` hat aber den statischen Typ `T<Integer>` und `T<Integer>` ist nicht mindestens so speziell wie `T<Character>` (auch andersrum gilt das nicht, da die Klassen in keiner Vererbungsrelation stehen).

24. `S.m(5.0,5), X.m(5.0)`

25. *Compiler-Fehler*: Aus statischer Sicht soll ein `X<Integer>` an eine Variable vom Typ `T<Character>` zugewiesen werden. Das geht nicht, da die Klassen in keiner Vererbungsrelation stehen (`X<Integer>` erbt von `T<Integer>`).

26. `S.m(5.0,9), X.m(9.0)`

27. `U.m(4), T.m(0.4,4)`

28. `T.m(12.0)`

29. *Compiler-Fehler*: In der Klasse `U` wird nach einer Methode `m(short,int)` gesucht. Wir finden `m(int,double)` und `m(double,int)` (von `T<Character>` geerbt). Beide Methoden passen, da `short` implizit zu `int`/`double` und `int` implizit zu `double` gecastet werden kann. In so einem Fall wählen wir immer die am besten passende Methode! Problem hier ist nur, dass beide gleich gut/schlecht passen. Man spricht von einem *ambiguous method call* (dt. mehrdeutiger Methodenaufruf).

30. `U.m(1,1.0)`

31. *Compiler-Fehler*: In der Klasse `U` wird nach einer Methode `m(float,float)` gesucht. Keine Methode genügt diesen Anforderungen (`float` kann nicht implizit zu `int` gecastet werden).

32. `T.m(6.0,1)`

33. *Compiler-Fehler*: Der statische Typ von `t4` ist `T<Character>` und in dieser gibt es keine passende Methode für `m(int,float)`. `m(double,int)` passt nicht, weil `float` nicht implizt zu `int` gecastet werden kann. `m(int,double)` gibt es hier nicht.

34. `T.m(5.0,5)`

35. *Compiler-Fehler*: `U` kann nicht zu `S<Character>` gecastet werden, da die Klassen `U` und `S` in keiner Vererbungsrelation stehen.

36. `ClassCastException`: Der statische Typ von `t4` ist `T<Character>`. Hier tritt kein Compiler-Fehler auf, weil der Cast zu `S<Character>` theoretisch möglich sein könnte, da die beiden Klassen in einer Vererbungsrelation stehen (`S<Character>` erbt von `T<Character>`). Der Cast schlägt dennoch fehl, weil `t4` ein `U` (dyn. Typ) speichert und `U` ist nicht mindestens so speziell wie `S<Character>`. Der Cast wäre nur möglich, wenn `t4` ein `S<Character>` oder `X<Character>` speichern würde.

37. `T.m(5.0)`

38. `U.m(3), T.m(0.300000...0004,3)`

    Man kann nicht erwarten, dass du weißt, dass die Operation `3 * 0.1` aufgrund eingeschränkter Maschinengenauigkeit nicht exakt `0.3` ergibt.

39. *Compiler-Fehler*: Die Zuweisung ist soweit ok. Das Problem ist, dass wir dann in `T<Character>` nach einer Methode `m(int,float)` suchen, weil `f` den statischen Typ `float` hat, wenngleich ihm ein `int` zugewiesen wird. Dieser `int` wird implizit zu `float` gecastet. Zu `m(int,float)` existiert keine passende Methode.

Teilaufgabe (2) - (a)

1. `T(Gen)`
2. `T(Integer)`
3. `T(Gen)`
4. `T(Integer)`
5. `T(Integer), S(int)`
   `T(Gen), S(Object), X(Object,T)`
6. `T(Gen), S(Object), S()`
7. `T(Integer), S(int)`
   `T(Gen), S(Object), X(Object,T)`
8. `T(Gen), S(Object)`
9. *Compiler-Fehler*: Es existiert kein passender Konstruktor in `X`, der den Parameter `float` akzeptiert.
10. `T(Integer), U()`
11. `T(Gen), U(char)`
12. `T(Gen), S(Object), X(Integer)`
    `T(Integer), U()`
13. `T(Gen), S(Object), X(Object,T)`
14. `A`
15. `null`
16. `5`
17. `NullPointerException`: `t4.t` speichert den Wert `null`, hat also keine Referenz auf ein `Integer`-Objekt. `null` kann nicht auf etwas addiert werden.
18. `null`
19. `null`
20. `null`
21. `null`
22. `13`
23. `7`
24. `1.5`

Teilaufgabe (2) **- (b)**

1. `T.m(S), T.m(T)`
2. `ClassCastException`, denn `t2` speichert keinen `S<Float>`.
3. `T.m(T)`
4. *Compiler-Fehler*, denn `S<Integer>` (bzw. `T<Integer>`) stellt keine Methode `m` mit Parameter `X<Character>` oder `S<Character>` oder `T<Character>` bereit.
5. `T.m(T)`
6. `X.m(S), T.m(T), T.m(S), T.m(T), T.m(T)`
7. `X.m(T)`
8. `X.m(U), X.m(67), S.m(67.0,67), X.m(67.0), T.m(), T.m(67.0)`
9. `X.m(U)`, dann `NullPointerException` bei `t.m().m(u.t)`, denn `x1.t` ist `null`.
10. `X.m(T)`
11. *Compiler-Fehler*, denn der Cast von `U` zu `S` kann nie möglich sein (statisch).
12. `X.m(U), X.m(67), S.m(67.0,67), X.m(67.0), S.m(),`
    `T.m(T), T.m(), X.m(67.0)`
13. `T.m(S), T.m(T)`
14. `U.m(S), T.m(S), T.m(T)`

    Dann tritt eine „ungewöhnliche Art“ `NullPointerException` bei `s.m(t)` auf: Das Problem liegt bei `t`, denn `u1.t` ist `null`, hat aber den statischen Typ `Character`. Durch automatisches Unboxing wird hier aus statischer Sicht die Methode `m(double)` gewählt, wenn man nach `m(Character)` sucht. `null` ist jedoch kein gültiger Wert für `char` bzw. `double` sondern nur für `Character`, daher ist der Aufruf nicht möglich.
15. `U.m(S), T.m(S), T.m(T), T.m(67.0)`
16. `T.m(T)`
17. *Compiler-Fehler*: `S<Long>` stellt keine Methode `m(T<Character>)` bereit.
18. `S.m(U), T.m(67.0), T.m(S), X.m(T)`
19. `S.m(U), T.m(67.0)`

    Dann `NullPointerException` an der Stelle `t.m(this)`, denn `s2.t` ist `null`.

Teilaufgabe (2) - **(c)**

1. `T.m(67.0)`, denn `u2.t` speichert `'C'`.
2. `NullPointerException`: Aufgrund des generischen Typs von `u1` (`Character`) wird aus statischer Sicht durch automatisches Unboxing von `u1.t` zu `char` die Methode `m(double)` als passendste ausgewählt (impliziter Upcast von `char` zu `double` möglich, zu `short` unmöglich). `u1.t` ist jedoch tatsächlich `null` und `null` kann nicht zu `char` entpackt werden, daher ist der Aufruf nicht möglich.
3. `S.m(), T.m(T), T.m(), X.m(65.0)`
4. `X.m(4), S.m(4.0,4), X.m(4.0)`
5. *Compiler-Fehler*: `s.t.t` speichert den `Integer 4` und `Integer` hat kein Attribut `t`.
6. `NullPointerException`: `T<Float> s4.t` wurde nicht zugewiesen, ist also `null`.
7. `S.m(), T.m(T), T.m(), 4`
8. `S.m(), T.m(T), T.m(), X.m(T)`
9. `S.m(), T.m(T), T.m(), U.m(S), T.m(S), T.m(T),`
   dann `NullPointerException` wie bei Aufruf (2) (a) 14.
10. `S.m(), X.m(T), T.m(), T.m(S), T.m(T)`
11. `S.m(), X.m(T), T.m(), S.m(), T.m(T), T.m(), T.m(4.0)`
12. `ClassCastException`: Wie aus den vorherigen Aufrufen bekannt ist, speichert `((T<Integer>)x2).t` aus statischer Sicht einen `Integer` (wegen des generischen Typs `Integer`). Konkret wurde hier bei der Objekterzeugung aber `1.5` (als `Object`) eingesetzt. Die Variable `t` hat aus statischer Sicht also den Typ `Integer`, weshalb der Aufruf kompiliert, da `Integer` automatisch zu `int` entpackt werden kann und `t1.m(t)` somit die Methode `m(double)` ausführen würde, da jeder `int` implizit zu `double` gecastet werden kann. Erst zur Laufzeit wird festgestellt, dass in `t` gar kein `Integer` gespeichert wurde, sondern ein `Double`. Normalerweise ist das nicht möglich, allerdings haben wir bei der Instanziierung eben durch einen Cast zugesichert, dass der `Double` eigentlich ein `Integer` ist (durch den generischen Cast in `S`). Normalerweise schlägt dieser Cast dann fehl. Das ist hier nicht der Fall, da tatsächlich zu `Object` gecastet wird, denn der generische Typ wird immer durch `Object` ersetzt (*type erasure*) und dieser Cast funktioniert immer. Generische Casts werden in der Entwicklungsumgebung daher auch als „*unchecked cast*“ gekennzeichnet. Die durch den generischen Cast vermutete Zusicherung ist also gar nicht wahr, da wir ja einen `Double` übergeben haben. Dieser in `t` gespeicherte `Double 1.5` soll nun beim Zugriff als `Integer` zurückgegeben werden. Das ist nicht möglich, da nicht von `Double` zu `Integer` gecastet werden kann (`double` zu `int` casten ginge, `Double` und `Integer` haben aber keinen Zusammenhang). Eine generische Variable wird also immer zum statischen Typ `Object` übersetzt und kann daher leider erstmal alles aufnehmen.

13. `X.m(2.0)` – Erklärung: Man könnte hier das gleiche vermuten wie in der vorherigen Aufgabe. Den entscheidenden Unterschied macht der Cast zum *raw type* `T`. *Raw Type* bedeutet, dass hier kein Typparameter angegeben wurde, d. h. man bezieht sich auf den allgemeinen Fall mit dem generischen Typ `Object`. Greift man nun auf `t` zu, so wird nicht versucht, den `Double` (der statisch sowieso als `Object` gespeichert ist) zu `Integer` zu casten. Der Cast zu `double` oder `Double` ist nötig, weil statisch jetzt nur bekannt ist, dass `t` den Typ `Object` hat. Der Cast funktioniert, da `t` wirklich einen `Double` speichert. Anschließend wird der `double` `1.5` zu `int` gecastet und `1` addiert. Beachte, dass ein direkter Cast zu etwas anderem als `Double` nicht möglich wäre, d. h. hier ist tatsächlich das Casten in zwei Schritten notwendig, um den `Double` (statisch `Object`) zu einem `int` entpacken. Der entstandene `int` `2` (`1 + (int)1.5`) wird nun der Methode `m` übergeben. `s1.t` ist statisch ein `T`, daher wählen wir `m(double)` für `m(int)`. Dynamisch wird diese Methode jedoch überschrieben, denn `s1.t` ist ein `X`.

14. *Compiler-Fehler*: `t3.t` speichert tatsächlich den Wert `5` (`short`), allerdings hat `t3.t` statisch den Typ `Object` und es existiert keine Methode `m(Object)`.

15. `S.m(), X.m(T), T.m(), T.m(), T.m(), T.m(1.0,2)`

16. `ClassCastException`: `s3.t` ist ein Objekt der Klasse `X<Long>` (statisch `T<Long>`), daher ist `s3.t.t` das `Gen t`, wobei `Gen` hier durch `Long` ersetzt wurde. Bei der Objekterzeugung wird `'B'` übergeben, welches implizit zum `int` `66` gecastet wird (`char` → `int`). Im Konstruktor erfolgt dann ein automatisches Boxing in ein Objekt vom Typ `Integer`. Die Speicherung in der Variable `t` ist zunächst erfolgreich, obwohl `Integer` nicht von `Long` erbt. Grund dafür ist, dass jeder generische Typ eigentlich in den Typ `Object` übersetzt wird (→ *generic type erasure*). Beim Zugriff auf `s3.t.t` erwarten wir einen `Long` (schließlich ist `Long` der generische Typ). Die JVM muss uns einen `Long` zurückgeben, stellt aber fest, dass ursprünglich ein `Integer` eingesetzt wurde, welcher nicht zu `Long` gecastet werden kann (weder implizit noch explizit), da die Klassen in keiner Vererbungsrelation stehen.

17. `X.m(S), T.m(T), T.m(S), T.m(T), X.m(T)`

18. *Compiler-Fehler*: Die Zuweisung von `S<Character>` an `S<Integer>` ist nicht möglich, da die Klassen in keiner Vererbungsrelation stehen (wegen gen. Typ).

19. Zunächst müssen alle Objekte erzeugt werden. Beachte die Reihenfolge, denn bevor das vorderste `X` erzeugt werden kann, muss das innere `X` erzeugt werden (`X<Byte>(7)` zuerst). Bei der Erzeugung des inneren `X` wird außerdem ein `U` erzeugt. Erst zum Schluss wird das `U`-Objekt, welches als Parameter in `m` übergeben wird, erzeugt.

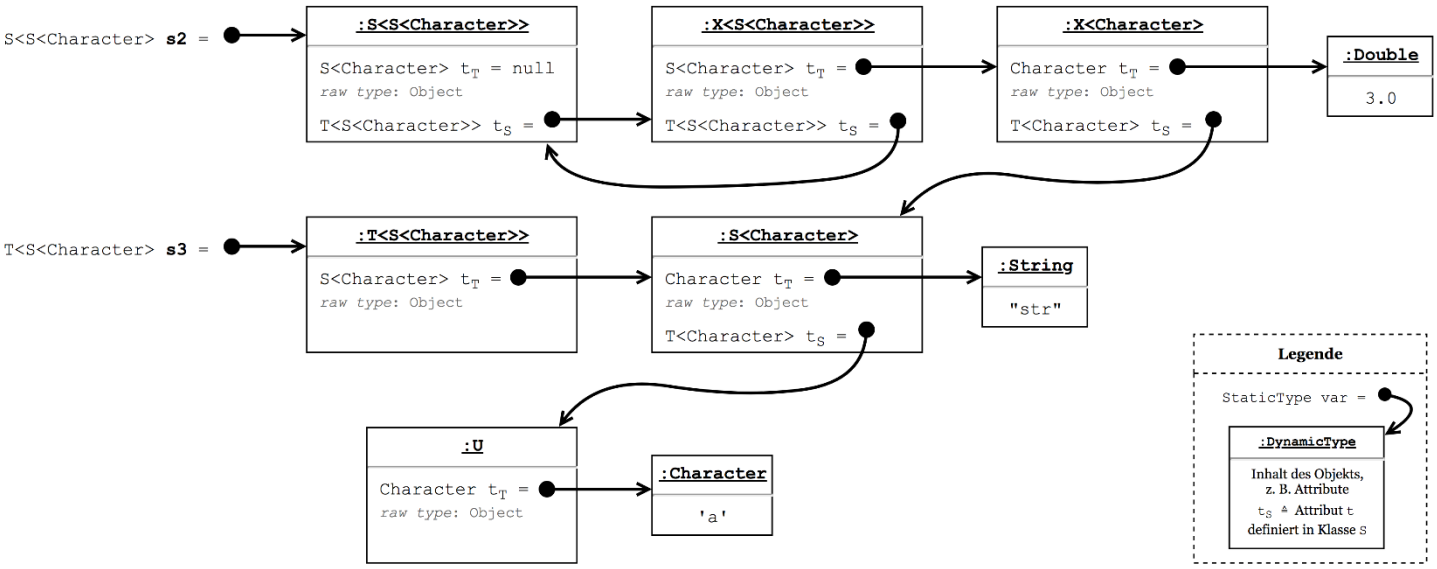
    ```
    T(Gen), S(Object), X(Integer)
    T(Integer), U()
    T(Gen), S(Object), X(Object,T)
    T(Gen), U(char)
    ```

    Erst jetzt folgt der Methodenaufruf mit der Ausgabe:

    ```
    X.m(U), X.m(6), S.m(6.0,6), X.m(6.0), S.m(), X.m(T), T.m(), T.m(6.0)
    ```

Teilaufgabe (3)

Die folgende Skizze ist an ein Objektdiagramm angelehnt und zeigt die Situation der Referenzen:



Die von `T<Gen>` geerbte Variable `t` (im Diagramm t_T geschrieben) hat zur *compile time* den „statische Typ“ `Gen` (je nach Typparameter anders!). Dieser Typ wird z. B. von der Entwicklungsumgebung benutzt und ist auch im Diagramm jeweils eingetragen. Wir können an `t` also nur Objekte zuweisen, die mindestens den Typ `Gen` haben, sonst kompiliert das Programm nicht. Zur Laufzeit existiert der Code der Klasse `T` jedoch nur ein einziges Mal, daher bekommt `t` dort den „echten statischen Typ“ `Object`. Wir können zur Laufzeit also alles in `t` speichern. Dafür muss der Code aber überhaupt kompilieren! Durch einen generischen Cast der Form „`(Gen)`“ (wie im Konstruktor `S(Object)` oder `X(Integer)` der Fall) versichern wir dem Compiler, dass das, was wir da gerade zuweisen, schon den richtigen Typ (mind. `Gen`) hat. Das wird aber nicht überprüft, wie es sonst bei Casts der Fall ist (vgl. erste Seite des Polymorphie-Kapitels)...

1. `X.m(T)`
2. `str`
3. `X.m(65.0)`
4. `S.m(), X.m(T), T.m(), S.m(1.0,97), T.m(97.0)`
5. `S.m(), T.m(T), T.m(), X.m(U), X.m(97), S.m(97.0,97), X.m(97.0), S.m(), T.m(T), T.m(), T.m(97.0)`
6. *Compiler-Fehler*: `s3.t.t` ist statisch `T<Character>` (dynamisch `U`) und hat somit keine Methode `m(S<S<Character>>)`.
7. `T.m(), S.m(), T.m(T), T.m(), S.m(U), T.m(97.0), T.m(S), X.m(T)`
8. `3.0`

9. `ClassCastException`: Wie aus dem vorherigen Aufruf hervorgeht, hat der Parameter (`((T<Character>)s2.t.t).t`) eigentlich den Wert `3.0`, d. h. man könnte vermuten, dass `T.m(double)` aufgerufen wird. Das Problem ist hier wieder, dass eigentlich gar kein `Double` gespeichert sein dürfte, da wir auf das `t` eines `X<Character>` zugreifen. Das müsste eigentlich ein `Character` sein, wurde aber im Konstruktor auf den `Double` `3.0` gesetzt. Wieder macht *Type Erasure* einen Strich durch die Rechnung. `t` hat nämlich nur bei der *compile time* den Typ `Character`, wird zur Laufzeit aber durch den Typ `Object` ersetzt (so werden Generics übersetzt). Erst beim Zugriff wird Java das gespeicherte `Object` als `Character` (gen. Typ) zurückgeben, indem es dieses castet. Java geht dann davon aus, dass der Cast funktioniert und wir nicht eine unzulässige Typzusicherung gemacht haben, was wir hier aber haben, da wir im Konstruktor explizit zum generischen Typ `Character` casten, ohne dass es sich tatsächlich um einen `Character` handelt. Die Umwandlung von `Object` zu `Character` schlägt fehl, da ein `Double` gespeichert wurde (d. h. statisch `Object`, dynamisch `Character`, Cast zu `Double`). Im vorherigen Aufruf wurde als Parameter nur ein `Object` erwartet (`println(Object)`) erwartet, d. h. es war dort nicht nötig, zu `Character` zu casten.

10. `S.m(), T.m(T), T.m(), a`

101

```java
private static void swap(long[] numbers, int a, int b) {

    long tmp = numbers[a];
    numbers[a] = numbers[b];
    numbers[b] = tmp;

}

public static void insertionSort(long[] numbers) {

    for (int i = 1; i < numbers.length; i++) { // i = 0 auch möglich
        for (int j = i; j >= 1; j--) { // nach vorne/links schieben
            if (numbers[j] < numbers[j-1])
                swap(numbers, j, j-1);
            else  // kann nicht weiter nach vorne
                break;  // nicht nötig, beschleunigt aber bisschen
        }
    }

}
```

102

```
public static void selectionSort(int[] array) {
    for (int i = 0; i < array.length-1; i++) {
        int maxIndex = [ i ];
        for (int j = [ i ]; j < array.length; j++)
            if ( [ array[j] > array[maxIndex] ] )
                maxIndex = j;
        swap(array, [ i ], [ maxIndex ]);
    }
}
```

103

```
public static int bubbleSort(int[] arr) {
    int swaps = 0;  // Zähler für Vertauschungen

    for (int j = 0; j < arr.length-1; j++) {  // Array mehrmals ...

        boolean atLeastOneSwap = false;
        for (int i = 0; i < arr.length-1-j; i++) // ... durchlaufen
            if (arr[i] > arr[i+1]) {    // ... und, falls nötig,
                swap(arr, i, i+1);      // ... benachbarte tauschen
                swaps++;
                atLeastOneSwap = true;
            }

        if (!atLeastOneSwap)  // kein Tausch durchgeführt?
            break;  // kann jetzt schon aufhören (fertig)
    }

    return swaps;
}
```

```java
public static void mergeSort(int[] array) {
    // (1.) Abbruchbedingung: nichts zu sortieren
    if (array.length <= 1)
        return;

    // (2.) Array in linke und rechte Hälfte teilen:
    //      Statt der for-Schleifen geht alternativ auch System.arraycopy
    int middle = array.length / 2;
    int[] leftHalf = new int[middle];
    for (int i = 0; i < leftHalf.length; i++)
        leftHalf[i] = array[i];
    int[] rightHalf = new int[array.length - middle];
    for (int i = 0; i < rightHalf.length; i++)
        rightHalf[i] = array[middle + i];

    // (3.) Beide Hälften sortieren:
    mergeSort(leftHalf);
    mergeSort(rightHalf);

    // (4.) Die sortierten Hälften in das Ursprungsarray mergen:
    int left = 0, right = 0;
    for (int merge = 0; merge < array.length; merge++) {
        // eine der beiden bereits komplett übernommen?
        if (right >= rightHalf.length) {
            array[merge] = leftHalf[left];
            left++;
        } else if (left >= leftHalf.length) {
            array[merge] = rightHalf[right];
            right++;
        }
        // ansonsten Hälfte mit kleinerem Element verwenden
        else if (leftHalf[left] < rightHalf[right]) {
            array[merge] = leftHalf[left];
            left++;
        } else {
            array[merge] = rightHalf[right];
            right++;
        }
    }
}
```

```
private static void quickSort(double[] array, int from, int to) {
    int length = to - from + 1;
    if (length >= 2) {
        int left = from;
        int right = to;

        while (left != right) {
            while (left != right && array[left] < array[to])
                left++;
            while (left != right && array[right] >= array[to])
                right--;
            swap(array, left, right);
        }

        swap(array, right, to);

        int pivotIndex = right;

        quickSort(array, from, pivotIndex-1);
        quickSort(array, pivotIndex+1, to);
    }
}

public static void quickSort(double[] array) {
    quickSort(array, 0, array.length - 1);
}
```

```java
public class Binaerbaum {
    protected int element;
    protected Binaerbaum links, rechts;  // linkes/rechtes Kind

    public Binaerbaum(int element) {
        this.element = element;
    }

    public boolean binaereSuche(int wert) {
        if (wert == element)  // Wert wurde gefunden
            return true;
        else if (wert < element)  // im linken Teilbaum weitersuchen
            return (links == null) ?
                            false : links.binaereSuche(wert);
        else  // im rechten Teilbaum weitersuchen (falls rechtes Kind existiert)
            return (rechts == null) ?
                             false : rechts.binaereSuche(wert);
    }

    public int laenge() {
        int laenge = 1;
        if (links != null)
            laenge += links.laenge();
        if (rechts != null)
            laenge += rechts.laenge();
        return laenge;
    }
}
```

```java
import java.util.*;

public class BinaerbaumIterator implements Iterator<Integer> {

    private Stack<Binaerbaum> stack;

    public BinaerbaumIterator(Binaerbaum wurzel) {
        stack = new Stack<>();

        // Lege alle Knoten des rechtesten Pfades auf den Stack:
        pushBranch(wurzel);
    }

    @Override
    public boolean hasNext() {
        return !stack.isEmpty();
    }

    @Override
    public Integer next() {
        if (!hasNext())
            throw new NoSuchElementException();

        // Hole den nächsten zu bearbeitenden Knoten (Dieser hat definitiv keine
        // unbearbeiteten rechten - also größeren – Nachfolger mehr):
        Binaerbaum naechster = stack.pop();

        // Beim nächsten next()-Aufruf würde der Elternknoten von naechster
        // bearbeitet werden. Davor müssen aber noch alle linken Nachfolder von
        // naechster behandelt werden (stehen rechts vom Elternknoten), daher
        // lege den gesamten rechtesten Pfad des linken Nachfolgers von naechster
        // (sowie den linken Nachfolger selbst) auf den Stack:
        pushBranch(naechster.links);

        return naechster.element;   // Autoboxing von int zu Integer
    }

    /** Legt Knoten b und die Knoten seines gesamten rechten Teilzweigs auf den Stack. */
    private void pushBranch(Binaerbaum b) {
        while (b != null) {
            stack.push(b);
            b = b.rechts;
        }
    }
```

```
    /** Test für das Beispiel aus der Angabe. */
    public static void main(String[] args) {
        Binaerbaum wurzel = new Binaerbaum(6);
        wurzel.links = new Binaerbaum(3);
        wurzel.links.links = new Binaerbaum(1);
        wurzel.links.links.rechts = new Binaerbaum(2);
        wurzel.links.rechts = new Binaerbaum(4);
        wurzel.links.rechts.rechts = new Binaerbaum(5);
        wurzel.rechts = new Binaerbaum(7);
        wurzel.rechts.rechts = new Binaerbaum(11);
        wurzel.rechts.rechts.links = new Binaerbaum(9);
        wurzel.rechts.rechts.links.links = new Binaerbaum(8);
        wurzel.rechts.rechts.links.rechts = new Binaerbaum(10);

        for (Integer i : wurzel) {
            System.out.print(i + ", ");
        }
    }
}
```

Ergänzungen in der `Binaerbaum`-Klasse:

```
import java.util.Iterator;

public class Binaerbaum implements Iterable<Integer> {

    // ... (alter Binaerbaum-Code)

    public Iterator<Integer> iterator() {
        return new BinaerbaumIterator(this);
    }
}
```

```java
public abstract class Node<E extends Comparable<E>> {
    protected final E elem;

    protected Node(E content) {
        super(); // überflüssig, da implizit (Konstruktorauf der Oberklasse)
        if (content == null) // Versuch, Knoten mit Inhalt null zu erzeugen
            throw new IllegalArgumentException("must not be null");
        elem = content;
    }

    public boolean contains(E e) { // alternativ auch abstract möglich *1
        return elem.compareTo(e) == 0; // alternativ: elem.equals(e)
    }

    // Abstrakte Methoden (müssen von den Unterklassen implementiert werden):
    public abstract Node<E> append(E e);
    public abstract boolean equals(Object o);

    public String toString() { // alternativ abstract mögl. (oder weglassen)
        return elem.toString() + ", ";
    }
}
```

```java
public class Leaf<E extends Comparable<E>> extends Node<E> {
    public Leaf(E elem) {
        super(elem);  // explizit notwendig, da super() (ohne Param.) unmöglich
    }

    @Override  // @Override-Annotation ist bei Überschreibung immer optional
    public Node<E> append(E e) {
        Leaf<E> newNode = new Leaf<>(e); // neu einzufügender Blattknoten
        if (elem.compareTo(e) > 0)  // *2 (siehe Ende der Aufgabe)
            return new InnerNode<>(elem, newNode, null);
        else
            return new InnerNode<>(elem, null, newNode);
    }

    @Override
    public boolean equals(Object o) {  // *3
        if (o == null || !(o instanceof Leaf))
            return false;
        Leaf<E> oCasted = (Leaf<E>) o;
        return this.elem.compareTo(oCasted.elem) == 0;
    }
}
```

```java
public class InnerNode<E extends Comparable<E>> extends Node<E> {

    private Node<E> left, right;

    public InnerNode(E elem, Node<E> left, Node<E> right) {
        super(elem);
        this.left = left;
        this.right = right;
    }

    public boolean contains(E e) {
        if (super.contains(e)) // alternativ elem.compareTo(e) == 0
            return true;
        else if (elem.compareTo(e) > 0) // e < elem  *1
            return (left != null) ? left.contains(e) : false;
        else                                // e > elem
            return (right != null) ? right.contains(e) : false;
    }

    public Node<E> append(E e) {   // *2 (siehe Ende der Aufgabe)
        if (elem.compareTo(e) > 0) // e < elem
            left = (left == null) ? new Leaf<>(e) : left.append(e);
        else                               // e >= elem
            right = (right == null) ? new Leaf<>(e) : right.append(e);

        return this;   // „ich“ bleibe Nachfolger des Vorgängers
    }

    public boolean equals(Object o) {   // *3
        if (o == null || !(o instanceof InnerNode))
            return false;

        InnerNode<E> casted = (InnerNode<E>) o;

        if (this.elem.compareTo(casted.elem) != 0) // Inhalt gleich?
            return false;

        // Linke Nachfolger auf Gleichheit prüfen (könnten auch beide null sein!):
        if (this.left != null)
            if (!this.left.equals(casted.left))
                return false;  // this.left und casted.left ungleich
        else if (casted.left != null)
            return false;  // entweder left oder casted.left ist null

        // Rechte Nachfolger auf Gleichheit prüfen (gleich wie bei den linken, s. o.):
        if (this.right != null)
            if (!this.right.equals(casted.right))
                return false;
        else if (casted.right != null)
            return false;

        return true;  // nie false returnt → alles hat gepasst → Knoten gleich
    }
```

```
    public String toString() {
        String result = "";
        if (left != null)
            result += left; // identisch zu result += left.toString() ...
        result += super.toString(); // result += elem.toString() + ", "
        if (right != null)
            result += right; // ... wegen String-Konkatenation
        return result;
    }
}
```

```
public class BinaryTree<T extends Comparable<T>> {
    private Node<T> root;

    public boolean contains(T element) {
        if (root == null)
            return false;
        return root.contains(element);
    }

    public void append(T element) {
        if (root == null)
            root = new Leaf<>(element);
        else
            root = root.append(element); // *2
    }

    @Override
    public boolean equals(Object o) {
        if (o == null)  // nicht unbedingt nötig (instanceof tut das auch)
            return false;
        if (!(o instanceof BinaryTree))
            return false;

        BinaryTree<T> casted = (BinaryTree<T>) o;
        if (this.root == null && casted.root == null) // beide sind leer
            return true;
        return root.equals(casted);  // delegiere an equals in Node
    }

    @Override
    public String toString() {
        if (root == null)
            return "[]";
        else
            return "[" + root.toString() + "]";
    }
}
```

Anmerkungen:

*1 Da Blätter keine Nachfolger haben und somit nur ein einziges Element beinhalten, muss die `contains`-Methode in `Leaf` lediglich prüfen, ob das als Parameter übergebende Element `e` gleich zu dem Element ist, das das Blatt in seinem Attribut `elem` speichert. Wir prüfen dabei nicht auf Referenzgleichheit (`elem == e`) sondern auf Objektgleichheit (`elem.equals(e)`), wobei in der Angabe keine näheren Angaben dazu gemacht werden, weshalb auch eine Lösung mit Referenzgleichheitsprüfung korrekt wäre. Da der Typ `E` das Interface `Comparable<E>` implementiert, wissen wir, dass wir mittels „`elem.compareTo(e) == 0`“ prüfen können, ob Elemente gleich sind (Objektgleichheit). Die Verwendung der `equals`-Methode ist hier riskant, da nicht sichergestellt ist, dass der Typ `E` diese (aus `Object` geerbte) Methode wirklich selbst implementiert. Die Standardimplementierung in `Object` prüft lediglich auf Referenzgleichheit. In dieser Aufgabe wäre sowohl „`elem == e`“ als auch „`elem.compareTo(e) == 0`“ als auch „`elem.equals(e)`“ korrekt.

Wenn wir diesen Vergleich in der Oberklasse `Node` statt in `Leaf` implementieren, können wir in `InnerNode` die Oberklassenmethode über `super` aufrufen. Alternativ können wir den Vergleich auch zweimal machen. In `InnerNode` kann es zudem sein, dass das gesuchte Elemente `e` über den rechten oder linken Nachfolger gefunden wird, daher müssen wir - je nachdem ob das Element links oder rechts stehen müsste (nach Invariante des Baums) - entweder links oder rechts weitersuchen, falls der innere Knoten selbst das Element nicht speichert. Dabei ist zu beachten, dass der jeweilige Nachfolger auch `null` sein könnte, d. h. bspw. dass wir links suchen müssten, aber kein linker Nachfolger existiert (d. h. dieser ist `null`). In diesem Fall geben wir `false` zurück.

*2 Wird `append(e)` auf einem inneren Knoten aufgerufen, so muss überprüft werden, ob das neu einzufügende Element `e` links oder rechts eingefügt werden muss. Ist `e` kleiner als `elem`, so muss `e` links, anderenfalls rechts eingefügt werden. Dabei ist noch zu beachten, dass es sein könnte, dass der linke/rechte Nachfolger noch nicht existiert, also `null` ist. In diesem Fall wird ein `new Leaf<>(e)` zugewiesen, d. h. der linke oder rechte Nachfolger ist anschließend nicht mehr `null`, sondern ein neuer Blattknoten mit dem Inhalt `e`. Ist der rechte/linke Nachfolger hingegen bereits vorhanden (also nicht `null`), so delegieren wir die Einfügen-Aufgabe einfach an diesen (z. B. `left.append(e)`).

Spannender wird es bei `append` in `Leaf`. Soll auf einem Blattknoten etwas eingefügt werden, so ist das Blatt anschließend kein Blatt mehr. Das Blatt muss sich zum inneren Knoten umwandeln, da es das einzufügende Element `e` als linken oder rechten Nachfolger (je nachdem ob `e` kleiner oder größer/gleich `elem` ist) erhält. Nun können Objekte ihren Typ bekanntlich nicht ändern, d. h. das `Leaf` kann sich nicht einfach zum `InnerNode` machen. Dieses Problem kann nur gelöst werden, indem das `Leaf`

einen neuen inneren Knoten erzeugt, der ein Klon des Blatts ist, aber den neuen Blattknoten mit Inhalt `e` als linken/rechten Nachfolger erhält. Das Blatt, welches nun ein innerer Knoten ist, muss seinem Vorgänger aber mitteilen, dass es sich nun um ein neues Objekt handelt. Nachdem ein Blatt seinen Vorgänger nicht kennt, müssen wir uns entweder den Vorgänger speichern, oder alternativ den neuen inneren Knoten an den Vorgänger per Rückgabewert übergeben. Wir haben uns hier für die zweite Variante entschieden. Der Vorgänger (welcher ein innerer Knoten sein muss) muss sich wiederum darum kümmern, dass er den Rückgabewert des (alten) Blatts übernimmt. Deshalb steht in `InnerNode` nicht nur „`left.append(e)`“ sondern „`left = left.append(e)`“ (siehe ternärer Operator). Ein innerer Knoten hingegen muss seinem Vorgänger wiederum sagen, dass er sich nicht geändert hat, weshalb er sich selbst (`this`) zurückgibt, sodass dort alles beim Alten bleibt.

*3 Die `equals`-Methode soll prüfen, ob die beiden Knoten inhaltsgleich sind (Objektgleichheit). Dazu starten wir sowohl in `Leaf` als auch in `InnerNode` mit der Prüfung, ob es sich beim als Parameter übergebenen Vergleichsobjekt (welches per Definition erstmal nur den Typ `Object` hat) überhaupt um ein Objekt des gleichen Typs handelt. Dazu nutzen wir hier einen `instanceof`-Vergleich. Hat der Parameter nicht den gleichen Typ oder ist dieser `null`, so können die Objekte `this` und `o` nicht gleich sein und wir geben `false` zurück. Anderenfalls können wir das als Parameter übergebene Vergleichsobjekt `o` casten (zu `Leaf` bzw. `InnerNode`). Dieser Cast ist eine Zusicherung an den Compiler, dass der Typ des Objekts wirklich `Leaf` bzw. `InnerNode` ist. Dadurch haben wir anschließend Zugriff auf dessen Attribut `elem` (bzw. `left` und `right`). In `Leaf` muss lediglich geprüft werden, ob die beiden Blätter `this` und `oCasted` den gleichen Inhalt speichern. Dazu können wir bspw. „`elem.equals(oCasted.elem)`“ nutzen; oder noch besser (nachdem `equals` in `E` nicht sicher implementiert ist, `E` aber sicher `Comparable<E>` implementiert): `compareTo` wie im Code gezeigt.

In `InnerNode` müssen wir zudem überprüfen, ob der linke Nachfolger von this gleich ist zum linken Nachfolger von `casted` (und analog auch der rechte Nachfolger). Im Prinzip ist das der rekursive Aufruf „`this.left.equals(casted.left)`“ (und analog für `right`). Dabei müssen wir noch beachten, dass auch beide Knoten keinen linken Nachfolger haben könnten (`this.left == null` und `casted.left == null`, dann wären diese wiederum gleich) oder dass lediglich einer der beiden Knoten keinen linken Nachfolger haben könnte (`this.left == null` oder `casted.left == null` aber nicht beides, dann `return false`). Dies lässt sich auch mit mehreren `if`-Abfragen lösen.

1. ```
   IntStream.of(3, 4, 5, 6, 7)
   IntStream.range(3, 8)
   Arrays.stream(new int[] {3, 4, 5, 6, 7})
   IntStream.iterate(3, i -> i + 1).limit(5)
   Stream.iterate(3, i -> i+1) .limit(5).mapToInt(i -> i)
   LongStream.range(3, 8).mapToInt(x -> (int)x)
   Stream.iterate(0, i -> 1 + i).mapToInt(i -> i).limit(8).skip(3)
   Stream.of("3", "4", "5", "6", "7").mapToInt(s -> Integer.parseInt(s));
   ```

   uvm.

2. ```
   Stream.of("a", "aa", "aaa", "aaaa", "aaaaa")
   Stream.iterate("a", str -> str + "a").limit(5)
   Stream.iterate("", str -> str + "a").skip(1).limit(5)
   IntStream.range(1, 6).mapToObj(i -> new String("a").repeat(i))
   ```

   uvm.

3. Es gibt keinen *CharStream*, daher müssen wir einen `Stream<Character>` erzeugen.

   ```
   Stream.iterate('A', c -> (char) (c + 1)).limit(26)
   ```

   Seit Java 9 kann man der `iterate`-Funktion einen zusätzlichen zweiten Parameter übergeben, um eine Abbruchbedingung zu spezifizieren. In folgendem Beispiel wird der Stream nur solange generiert, bis `c <= 'Z'` erstmals nicht mehr gilt.

   ```
   Stream.iterate('A', c -> c <= 'Z', c -> (char) (c + 1))
   ```

   Falls du in einer Situation einen `Stream<Integer>` zu einem `Stream<Character>` machen musst, beachte, dass ein direkter Cast von `Integer` zu `char` nicht möglich ist. Ein `Integer` muss zu `int` entpackt werden, bevor er zu `char` gecastet werden kann:

   ```
   Stream.iterate(0, i -> i + 1).limit(26).map(i -> (char)(int)i)
   ```

4. ```
   double[] noten = {1.7, 1.3, 2.0, 1.3, 4.0, 2.7};
   double avg = Arrays.stream(noten).average().getAsDouble();
   ```

   Die `average`-Methode gibt für alle primitiven Streams ein `OptionalDouble`-Objekt zurück. Wenn der Stream nicht leer war, kann daraus der echte `double`-Wert mittels `getAsDouble()` entpackt werden.
   ```

1.
```java
public long sum(long n) {
    return LongStream.range(1, n+1).sum();
}
```

2. (a)
```java
public static double avgTemp(double[] t) {
    return Arrays.stream(t)          // DoubleStream über alle doubles
                 .average()          // Durchschnitt als OptionalDouble
                 .getAsDouble();     // OptionalDouble zu double
}      // (Das geht ohne isPresent()-Prüfung, da das Array mind. 2 Elem. hat)
```

(b)
```java
public static double minTempNoon(double[] t) {
    return IntStream.range(0, t.length)   // IntStream über alle Indizes
                    .filter(i -> i % 2 == 1)     // beschränkt auf ungerade
                    .mapToDouble(i -> t[i])      // zu Stream der Elemente
                    .min().getAsDouble();        // davon das Minimum
}             // (isPresent()-Prüfung wieder nicht nötig, da Array-Länge ≥ 2)
```

Natürlich gibt es mit Streams unzählige Lösungen, z. B. alternativ:

```java
public static double minTempNoon(double[] t) {
    return IntStream.iterate(1, i -> i + 2) // IntStream über 1,3,5,...
                    .limit(t.length / 2)    // beschränkt auf gültige Indizes
                    .boxed()                // Als Stream<Integer>, damit map
                    .map(i -> t[i])         // Stream<Double> erzeugen kann.
                    .min(Double::compare)   // findet minimalen Double
                    .get();                 // Optional<Double> zu Double
}
```

3.
```java
public static int[] removeDuplicates(int[] arr) {
    return Arrays.stream(arr)      // int-Array zu IntStream
                 .distinct()       // Duplikate entfernen
                 .toArray();       // IntStream zu int-Array
}
```

4.
```java
public static int[] extractMultiplesOf(int[] array, int num) {
    return Arrays.stream(array)
                 .filter(x -> x % num == 0) // beschränkt den Stream auf
                 .distinct()                // Elemente, für die x%num = 0
                 .toArray();
}
```

5.
```java
static void reverse(Object[] a) {
    // Folgendes entspricht genau der for-Schleife aus der urspr. Lösung:
    IntStream.range(0, a.length/2).forEach(i -> {
        Object tmp = a[i];
        a[i] = a[a.length-1-i];
        a[a.length-1-i] = tmp;
    });
}
```

6. Hier kann man sehr schön sehen, wann `iterate()` Sinn macht. Es wird zunächst ein unendlicher Stream erzeugt, der den Anforderungen entspricht (1, x^1, x^2, x^3, ...). Anschließend wird der Stream auf die geforderte Anzahl an Elementen beschränkt (hier `0` bis `n`, also `n+1` Elemente) und als Array zurückgegeben. Bestenfalls stellt man zuvor noch sicher, dass der `limit`-Methode keine negativen Werte übergeben werden (z. B. mit einem *if* wie in der ursprünglichen Lösung).

```java
public static long[] powerOfX(int x, byte n) {
    return LongStream.iterate(1, val -> val * x)
                     .limit(n+1).toArray();
}
```

Es geht aber auch ineffizienter:

```java
public static long[] powerOfX(int x, byte n) {
    return IntStream.range(0, n+1)
                    .mapToLong(i -> (long) Math.pow(x, i))
                    .toArray();
}
```

7.
```
public static int countVowels(String text) {
    return (int) text.chars().filter(
            c -> "aeiouy".indexOf(Character.toLowerCase(c)) >= 0
        ).count();
}
```

Der Text wird in einen Stream von Zeichen umgewandelt (jeder `char` ist ein `int`). Mit `filter()` werden nur diejenigen Zeichen im Stream „behalten“, die Vokale sind. `count()` gibt uns anschließend die Anzahl der verbleibenden Elemente, welche wir jedoch noch explizit von `long` zu `int` casten müssen.

Für die Vokalprüfung mit `filter` gibt es viele Möglichkeiten. Es ginge bspw. auch:

```
(...).filter(x -> x == 'A' || x == 'E' || x == 'I' || x == 'O' ||
                  x == 'U' || x == 'Y' || x == 'a' || x == 'e' ||
                  x == 'i' || x == 'o' || x == 'u' || x == 'y')). (...)
```

8. Einen String würde man normalerweise nicht so invertieren (sondern über `reverse` von `StringBuilder`). Sinn der Aufgabe ist es, den Umgang mit Streams zu üben.

```
static String invert(String s) {
    return s.chars()                          // IntStream von Zeichen
        .mapToObj(i -> (char)i + "")          // jedes Zeichen als String
        .reduce("", (acc, c) -> c + acc);     // Aneinanderreihen
}
```

Der String `s` wird durch `chars()` in einen `IntStream` von Zeichen umgewandelt. Wir müssen nun die einzelnen Elemente des Streams rückwärts miteinander verketten. Hierfür eignet sich `reduce("",(acc,c)->c+acc)` perfekt: Der Akkumulator `acc` ist anfangs der leere String `""`. Dann wird für jedes Zeichen `c` die Funktion `c+acc` ausgeführt, d. h. ein Zeichen wird jeweils vorne angefügt ($acc_{neu} = c + acc_{alt}$). Zuletzt gibt `reduce()` den `acc`, also genau unser gewünschtes Ergebnis zurück.

Wofür `mapToObj()`? Betrachte die Signatur der `reduce`-Methode! Die Akkumulatorfunktion (2. Parameter von `reduce()`) erwartet zwei `T`-Objekte; im Fall eines *IntStream*s sind das zwei `int`s. Der Akkumulator soll aber den Typ `String` haben. Aus diesem Grund wandeln wir einfach alle Zeichen des Streams in einzelne Strings um. Normalerweise nutzen wir dafür `map()`. *IntStream*s haben jedoch die Besonderheit, dass die „normale“ `map`-Methode wieder einen *IntStream* zurückgibt, was uns nicht weiterhilft, da wird einen `Stream<String>` benötigen. `mapToObj` schafft Abhilfe (verhält sich genau wie `map`).

Alternative Lösungsmöglichkeiten sind bspw.:

```
s.chars().boxed().reduce("",                  // reduce mit 3. Parameter!
     (acc, i) -> (char)(int)i + acc,
     (a, b) -> "NOTUSED");

s.chars().boxed().map(i -> (char)(int)i + "")
          .reduce("", (acc, x) -> x + acc);
```

9. (a) Hier gibt es unzählige Möglichkeiten. Am naheliegendsten ist es, `iterate` zu nutzen, um die einzelnen Zeilen zu generieren. Man kann `collect` nutzen, um die Zeilen zu verknüpfen (getrennt durch je einen Zeilenumbruch), und schließlich auszugeben:

```
public static void printTriangle(int size) {
    System.out.println(
        Stream.iterate("*", str -> str + "*").limit(size)
            .collect(Collectors.joining("\n"))
    );
}
```

Alternativ können die Zeichen und Zeilenumbrüche z. B. einzeln ausgegeben werden:

```
public static void printTriangle(int size) {
    IntStream.range(0, size).forEach(x -> {
            IntStream.range(0, x+1).forEach(y ->
                    System.out.print("*")
            );
            System.out.println();
    });
}
```

(b) Unterscheidet sich nicht sehr stark von (a). Eine weitere Möglichkeit ist:

```
public static void printTriangleRight(int size) {
    System.out.println(
        IntStream.range(1, size+1)
            .mapToObj(i -> repeat(' ', size-i) + repeat('*', i))
            .collect(Collectors.joining("\n"))
    );
}

// Gibt einen String der Länge n zurück, welcher nur das Zeichen c enthält.
public static String repeat(char c, int n) {
    return Stream.iterate("", str -> str + c)
            .skip(n)
            .findFirst().get();
}
```

10.
```
public boolean isPalindrome(char[] c) {
    return IntStream.range(0, c.length)
              .allMatch(i -> c[i] == c[c.length-i-1]);
}
```

„Iteriere“ mit einem Stream über alle Indizes des Arrays (bzw. `c.length/2` möglich) und prüfe, ob die Bedingung `c[i] == c[c.length-i-1]` für jede Position wahr ist.

11. Dies ist nicht die kürzeste, schönste oder effizienteste Lösung, aber die m. M. n. am besten verständliche, da sie sich relativ nah an der Lösung mit Schleifen orientiert:

```
public static Set<Integer> mostFrequentElements(int[] arr) {
    // Wie bei der Lösung mit Schleifen wird zunächst die Map erstellt,
    // welche die Elemente auf ihre Häufigkeiten abbildet.
    Map<Integer, Integer> counts = Arrays.stream(arr)
            .distinct()     // jedes Element nur einmal zählen
            .boxed()        // damit collect möglich ist
            .collect(Collectors.toMap(
                elem -> elem,   // Array-Element als Schlüssel übernehmen
                elem -> (int) Arrays.stream(arr)   // neuer Stream
                        .filter(x -> x == elem)    // nur elem behalten
                        .count()   // Stream-Länge als Value übernehmen
            ));            // Cast zu int nötig, weil count() long zurückgibt

    // Dann bestimmen wir das Maximum:
    int max = counts.values()   // Häufigkeiten
            .stream()   // Stream<Integer>
            .mapToInt(x -> x)   // IntStream (damit max() möglich ist)
            .max()   // OptionalInt
            .orElse(0); // max = Maximum, falls Stream nicht leer, sonst 0

    // Jetzt suchen wir alle Keys, deren Value gleich max ist.
    return counts.keySet()   // alle Elemente
            .stream()   // Stream<Integer>
            .filter(x -> counts.get(x) == max) // Häufigkeit ist maximal
            .collect(Collectors.toSet()); // verbleibende zurückgeben
}
```

```
public static void main(String[] args) {
    int zahlen[] = {5, 8, 9, 2, -10, 3, 4, 7, 6, 1};
    System.out.println(
        Arrays.stream(zahlen)      // => [5,8,9,2,-10,3,4,7,6,1]
        .filter(i -> i % 2 == 0)   // => [8,2,-10,4,6]
        .map(i -> -i)              // => [-8,-2,10,-4,-6]
        .sorted()                  // => [-8,-6,-4,-2,10]
        .map(i -> -i)              // => [8,6,4,2,-10]
        .limit(3)                  // => [8,6,4]
        .sum());                   // => 18
}
```

Die Reihenfolge kann teilweise variiert werden, d. h. bspw. könnte `filter` nach `map` und `sorted` stehen, allerdings muss `limit` definitiv nach `filter` angewendet werden. `sum` muss am Ende stehen. Auf das zweite `map` kann verzichtet werden, wenn man das Gesamtergebnis von `sum` stattdessen negiert, d. h. ein Minus (–) zu Beginn vor `Arrays.stream` schreibt.

Es ist nicht möglich, der `sorted`-Methode eine Vergleichsfunktion zu übergeben und somit auf das Negieren zu verzichten. Dafür wäre ein `Stream<Integer>` nötig, was jedoch die Verwendung von `boxed` oder `mapToObj` voraussetzt, da `map` bei *IntStreams* immer von `int` auf `int` abbildet. Außerdem müsste man den `Stream<Integer>` dann zurück zu `IntStream` umwandeln (`mapToInt`), da `sum` nicht auf `Stream<T>` definiert ist. Wären diese Methoden erlaubt gewesen, dann:

```
Arrays.stream(zahlen)
    .filter(i -> i % 2 == 0)
    .boxed()
    .sorted((a, b) -> b - a)
    .limit(3)
    .mapToInt(i -> i)
    .sum()
```

1.
```
int[] primes(int max) {
    return IntStream.range(2, max+1)
                    .filter(num ->
                        IntStream.range(2, num)
                                 .allMatch(div -> num % div != 0)
                    )
                    .toArray();
}
```

Diese Implementierung ist eine ziemlich genaue Nachbildung einer geschachtelten *for*-Schleife. Es wird ein Stream über die Zahlen 2, 3, 4, ..., `max` erzeugt, wobei nur diejenigen Zahlen behalten werden, für die die an `filter` übergebene Funktion wahr ist. Diese Funktion wertet für eine Zahl `num` genau dann zu `true` aus, wenn sie durch *alle* (`allMatch`) Zahlen im Bereich von 2 bis `num-1` *nicht* teilbar ist (`num % div != 0`).

2.
```
long[] morePrimes(int count) {
    return LongStream.iterate(2, x -> x + 1)
                     .filter(num -> LongStream.range(2, num/2+1)
                                              .noneMatch(div -> num % div == 0))
                     .limit(count)
                     .toArray();
}
```

Erzeugt einen unendlichen Stream von `long`s (beginnend bei 2). Durch `filter()` werden wie bei der vorherigen Teilaufgabe nur diejenigen Zahlen `num` behalten, welche durch *keine* (`noneMatch`) Zahl im Bereich von 2 bis $\frac{\text{num}}{2} + 1$ (das ist eine Optimierung; man könnte auch wieder `num` als obere Grenze wählen) teilbar ist (`num % div == 0`). Man könnte genauso wieder mit `allMatch` und `!=` arbeiten.

3.
```
static void fibonacci(int n) {
    System.out.println(Stream.iterate(new long[] {0, 1},
                              arr -> new long[] {arr[1], arr[0] + arr[1]})
                             .limit(n).map(arr -> arr[1]).map(String::valueOf)
                             .collect(Collectors.joining(", ")));
}
```

Statt den beiden `map`s kann man bspw. auch `map(arr -> ""+arr[1])` verwenden.
Es geht außerdem deutlich effizienter, indem man auf Arrays verzichtet (vgl. iterative Fibonacci-Implementierung aus dem Rekursionskapitel).

```java
public static long hexToDez(String hex) {
    if (hex == null || !hex.startsWith("0x"))
        throw new IllegalArgumentException();

    return hex.chars()     // IntStream über alle Zeichen von hex
        .skip(2)           // Überspringe das Präfix (0 und x)
        .map(digit -> {    // Bilde jedes Zeichen digit auf ...
                           // ... den zugehörigen int-Wert ab:
            if (digit >= '0' && digit <= '9') // '0'-'9'
                return digit - '0';           // wird zu 0-9
            if (digit >= 'a' && digit <= 'f') // 'a'-'f'
                return digit - 'a' + 10;      // wird zu 10-15
            // Alle anderen Zeichen sind ungültig:
            throw new IllegalArgumentException();
        })
        .reduce(0, (acc, i) -> acc * 16 + i); // siehe unten
}
```

Veranschaulichung von `reduce` am Beispiel `0x13b`:

Nach `map` ergibt sich der IntStream 1, 3, 11. An erster Stelle steht dabei der Wert der Ziffer mit der höchsten Wertigkeit („256er“, „16er“, Einer). Die Berechnung erfolgt daher wie in der Aufgabe zum Zahlenbasenumwandler.

Der Startwert von `acc` ist `0`. `reduce` baut das Ergebnis nun iterativ auf:

Schritt	**acc**	**i** (Element)	**Ergebnis** (acc_{neu})
1	0	1	1 (= 0*16+1)
2	1	3	19 (= 1*16+3)
3	19	11	**315** (= 19*16+11)

Dieses Verfahren ist in der Mathematik als „Horner-Schema“ bekannt.

114

Link zur **Angabe**: *https://stecrz.de/files/skript/schule.zip*

Link zur **Lösung**: *https://stecrz.de/files/skript/schule_loesung.zip*

Es gibt noch viele viele weitere Lösungsmöglichkeiten! Teste deine Implementierung.

Ein paar Zeugnisbeispiele:

```
Nora Nesper (3C)     Nils Netzes (4B)    Bianka Bauder (4A)   Elli Ebsen (3B)
---------            ---------           ---------            ---------
Deutsch: 2           Deutsch: 2          Deutsch: 3           Deutsch: 1
HSU: 2               HSU: 2              HSU: 1               HSU: 2
Kunst: 2             Kunst: 1            Mathematik: 2        Kunst: 2
Mathematik: 3        Mathematik: 2                            Mathematik: 1
Sport: 1
```

Teilaufgabe (o): Beantwortung der Fragen der Schulverwaltung

- Sind Mädchen durchschnittlich signifikant besser oder schlechter als Jungs?
  Mädchen sind durchschnittlich um ca. 0,172 Notenpunkte besser als Jungs.
- Welcher Schüler hat die Note 1 am häufigsten erhalten (in einzelnen Tests)?
  Die meisten Einsen haben ***Anna Apweiler*** und ***Guenther Groell*** aus der *3A*, ***Antonia Ammer*** aus der *4A* und ***Laura Liebmann*** aus der *4C*. Sie haben jeweils 4 Einsen.
- Was ist die beste erreichte Zeugnisdurchschnittsnote und wer hat diese erreicht?
  Der beste Zeugnisschnitt ist ***1,5*** und wurde von ***Elli Ebsen*** aus der *3B* erreicht.

  Wer ist der beste Viertklässler?
  Nils Netzes aus der *4B* und ***Laura Liebmann*** aus der *4C* sind gleichauf (Schnitt: *1,75*).

  Wer ist der Klassenbeste aus der *4A*?
  Bianka Bauder und ***Mandy Meder*** mit einem Schnitt von *2,0*.
- Wie vielen Schülern wurde Gymnasium/Realschule/Mittelschule empfohlen?
  11x Gymnasium, **6**x Realschule, **3**x Mittelschule

Nein, die Ausgabereihenfolge ist nicht eindeutig. Jeder *Writer* gibt die Zahlen für sich zwar in der immergleichen Reihenfolge aus, allerdings können die *Writer* in einem „beliebigen“ Wechsel schreiben (z. B. *1,1,2,2,3* oder *1,2,1,2,3* oder *1,2,1,3,2*). Wann welcher *Writer* an der Reihe ist ist dabei zufällig und unvorhersehbar.*

* Zufall gibt es nicht. Der *Scheduler* entscheidet, wann welcher Thread an der Reihe ist (vgl. Vorlesung: Betriebssysteme / GBS).

```
public class Reader extends Thread {
    private final Shared share;

    public Reader(Shared s) {
        share = s;
    }

    public void run() {
        synchronized (share) {
            while (!share.changed) { // wird manchmal falsch geweckt
                try {
                    share.wait();
                } catch (InterruptedException e) {
                    return;
                }
            }
            System.out.println(share.value);
            share.changed = false;
            share.notifyAll();  // weckt auch andere Reader auf :/
        }
    }
}
```

```java
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() {
        for (int i = 1; i <= n; i++) {
            synchronized (share) {
                while (share.changed) { // wird manchmal falsch geweckt
                    try {
                        share.wait();
                    } catch (InterruptedException e) {
                        return;
                    }
                }
                share.value = i;
                share.changed = true;
                share.notifyAll();  // weckt auch andere Writer auf :/
            }
        }
    }
}
```

Hätte man den Code erweitern dürfen, so wäre es am schönsten gewesen, für die Synchronisierung zwei verschiedene Objekte (z. B. vom Typ `Object`) zu verwenden (eines für die *Reader* und eines für die *Writer*), sodass *Reader* und *Writer* immer gezielt einen (mittels `notify` statt `notifyAll`) Thread des jeweils anderen Typs hätten aufwecken können. In dieser Implementierung kann es bspw. vorkommen, dass ein *Writer* einen anderen *Writer* aufweckt. Zur Veranschaulichung, welches *notify* sich eigentlich auf welches *wait* bezieht, wurde der Code entsprechend eingefärbt. In der darauffolgenden Aufgabe wird explizit unterschieden. Dort sind dann entsprechend auch `synchronized`-Blöcke notwendig.

Die nachfolgende Lösung findest du auch unter *stecrz.de/files/skript/fabrik_loesung.zip*

```
public abstract class Mitarbeiter implements Runnable {
    // abstrakt, damit sichergestellt ist, dass alle Unterklassen run implementieren (weil die
    // in Fabrik gespeicherten Mitarbeiter erstmal nur den stat. Typ Mitarbeiter haben)
    private final int id;
    private final String name;
    private int guthaben; // in Cent
    protected final Fabrik fabrik;

    public Mitarbeiter(Fabrik f, int id, String name) {
        this.fabrik = f;
        this.id = id;
        this.name = name;
        this.guthaben = 0;
    }

    public double gehaltAusbezahlen() {
        double euro = guthaben / 100d;
        guthaben = 0;
        return euro;
    }

    protected void gehaltZubuchen(int wert) {
        guthaben += wert;
    }

    public String getName() {
        return name;
    }
}
```

```
import java.util.LinkedList;
import java.util.List;

public class Fabrik implements Runnable {
    public static final int FASS_VOLUMEN = 50;
    protected final Object lock = new Object();
    protected final Object lieferantSync = new Object();
    protected final Object abholerSync = new Object();
    private int fuellstand;
    private List<Mitarbeiter> mitarbeiter;

    public Fabrik() {
        fuellstand = 0;
        mitarbeiter = new LinkedList<>();
    }
```

Fortsetzung auf der nächsten Seite...

```
    public int getFassVolumen() {
        return fuellstand;
    }

    public void setFassVolumen(int volumen) {
        if (volumen < 0 || volumen > FASS_VOLUMEN)
            throw new IllegalArgumentException();
        fuellstand = volumen;
        System.out.println("Neuer Füllstand: " + fuellstand);
    }

    public void addMitarbeiter(Mitarbeiter m) {
        if (!mitarbeiter.contains(m))
            mitarbeiter.add(m);
    }

    @Override
    public void run() {
        // Abholer und Lieferanten-Threads erzeugen und starten:
        Thread[] t = new Thread[mitarbeiter.size()];
        int i = 0;
        for (Mitarbeiter m : mitarbeiter) {
            t[i] = new Thread(m);
            t[i++].start();
        }

        // 10 Sekunden laufen lassen (warten):
        try {
            Thread.sleep(10000);
        } catch (InterruptedException e) {}

        // Alle beenden und auf Beendigung warten:
        for (Thread t0 : t) {
            t0.interrupt();
            try {
                t0.join();
            } catch (InterruptedException e) {}
        }
    }

    public static void main(String[] args) {   // Test (nicht verlangt):
        Fabrik f = new Fabrik();
        Lieferant henry = new Lieferant(f, 1, "Henry", 18);
        Lieferant gerty = new Lieferant(f, 2, "Gerty", 22);
        Abholer freddy = new Abholer(f, 3, "Freddy", 42);
        f.addMitarbeiter(henry);
        f.addMitarbeiter(gerty);
        f.addMitarbeiter(freddy);
        new Thread(f).start();
    }
}
```

```
public class Abholer extends Mitarbeiter {

    public static final int ABHOLMENGE = 10;

    private int gehaltProLiter;

    public Abholer(Fabrik f, int id, String name, int gehaltProLiter) {
        super(f, id, name);
        this.gehaltProLiter = gehaltProLiter;
    }

    @Override
    public void run() {
        while (true) {   // unbegrenzt lange Abholungen durchführen
            // Zufällig 0-500 ms warten (nicht Teil der Aufgabe):
            try {
                Thread.sleep((int) (Math.random() * 500));
            } catch (InterruptedException e) {
                return;
            }

            // Abholung einmal durchführen:
            int abholungAusstehend = ABHOLMENGE;
            while (abholungAusstehend > 0) {
                synchronized (fabrik.lieferantSync) {
                    int fuellstand = fabrik.getFassVolumen();

                    int abholung = abholungAusstehend;
                    if (abholung > fuellstand)
                        abholung = fuellstand; // mehr ist nicht drin
                    System.out.println(getName() + " entnimmt " +
                                       abholung + " Liter...");

                    fabrik.setFassVolumen(fuellstand - abholung);
                    abholungAusstehend -= fuellstand;
                    gehaltZubuchen(abholung * gehaltProLiter);

                    fabrik.lieferantSync.notify();
                }

                // Warte bis wieder nachgeliefert wurde:
                synchronized (fabrik.abholerSync) {
                    while (fabrik.getFassVolumen() <= 0) {
                        try {
                            fabrik.abholerSync.wait();
                        } catch (InterruptedException e) {
                            return;
                        }
                    }
                }
            }
        }
    }
}
```

```java
public class Lieferant extends Mitarbeiter {

    public static final int LIEFERMENGE = 8;

    private int gehaltProLiter;

    public Lieferant(Fabrik f, int id, String name, int centProLiter) {
        super(f, id, name);
        this.gehaltProLiter = centProLiter;
    }

    @Override
    public void run() {
        while (true) {  // unbegrenzt lange Lieferungen durchführen
            // Zufällig 0-500 ms warten (nicht Teil der Aufgabe)
            try {
                Thread.sleep((int) (Math.random() * 500));
            } catch (InterruptedException e) {
                return;
            }

            // Lieferung einmal durchführen:
            int lieferungAusstehend = LIEFERMENGE;
            while (lieferungAusstehend > 0) {
                synchronized (fabrik.abholerSync) {
                    int fuellstand = fabrik.getFassVolumen();
                    int platzVerbleibend = 50 - fuellstand;

                    int lieferung = lieferungAusstehend;
                    if (lieferung > platzVerbleibend)
                        lieferung = platzVerbleibend; // mehr geht nicht

                    System.out.println(getName() + " liefert " +
                                       lieferung + " Liter...");

                    fabrik.setFassVolumen(fuellstand + lieferung);
                    lieferungAusstehend -= lieferung;
                    gehaltZubuchen(lieferung * gehaltProLiter);

                    fabrik.abholerSync.notify();
                }

                // Warte bis wieder Platz ist (bis Benachrichtigung):
                synchronized (fabrik.lieferantSync) {
                    while (fabrik.getFassVolumen() >= 50) {
                        try {
                            fabrik.lieferantSync.wait();
                        } catch (InterruptedException e) {
                            return;
                        }
                    }
                }
            }
        }
    }
}
```

Der Zusatz `<E extends Comparable<E>>` vor einer Methode besagt lediglich, dass es sich um eine generische Methode handelt (vergleichbar mit einer generischen Klasse). Derartige Methoden erwarten den Typ `E` dann i. d. R. als Parameter (z. B. wie hier ein E-Array). Der Zusatz `extends Comparable<E>` versichert, dass für `E` nur Klassen eingesetzt werden können, die über die `compareTo(E)`-Methode verfügen und Objekte daher untereinander vergleichbar sind.

```
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];   /* Pivot-Element beliebig wählbar */

        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;

        /* Aufgabe (b): */

        Thread threadLeft = null;
        if (nThreads > 1) {  /* *1) */
            int nThreadsLeft = splitThreads(endLeft-startLeft+1,
                                            end-start+1, nThreads);
            nThreads -= nThreadsLeft;
            threadLeft = new Thread(() -> quickSort(array, startLeft,
                                                    endLeft, nThreadsLeft));
            threadLeft.start();
        } else {  /* *2) */
            quickSort(array, startLeft, endLeft, 1);
        }

        quickSort(array, startRight, endRight, nThreads);  /* *3) */

        if (threadLeft != null)  /* *4) */
            while (true)
                try {
                    threadLeft.join();
                    break;
                } catch (InterruptedException e) { }
    }
}
```

```java
private static int splitThreads(int sizeLeft, int size, int nThreads) {
    return 1 + Math.round(1F * sizeLeft / size * (nThreads-2));
}

/* Aufgabe (a): */

private static <E extends Comparable<E>> void swap(E[] array, int i, int j) {
    E tmp = array[i];
    array[i] = array[j];
    array[j] = tmp;
}
```

Kommentare im Code:

1) Es können noch weitere Threads erzeugt werden. Erzeuge daher einen neuen Thread und beauftrage ihn mit der Sortierung des linken Teilbereichs. Teile die Anzahl an Threads gerecht auf beide Hälften auf.
2) Es ist nur noch ein Thread für die Ausführung verbleibend, d. h. dieser Thread muss sowohl den rechten als auch linken Teilbereich rekursiv sortieren.
3) Der aktuelle Thread kümmert sich um die Sortierung des linken Teilbereichs, indem er die Quicksort-Methode rekursiv aufruft. Wichtig ist, dass – falls ein neuer Thread erzeugt werden soll – dieser zuvor erzeugt werden muss, sodass beide Threads zeitgleich sortieren.
4) Da der Thread ursprünglich mit der Sortierung des gesamten Bereichs von `start` bis `end` beauftragt wurde, muss sichergestellt werden, dass er sich erst beendet, wenn auch der für den linken Teilbereich zuständige Thread fertig mit der Sortierung ist (sofern dafür denn überhaupt ein neuer Thread erzeugt wurde). Dafür joint der aktuelle Thread den erzeugten Thread. Wird der aktuelle Thread beim Warten auf den anderen Thread unterbrochen, so wollen wir nicht darauf reagieren und weiter warten, d. h. wir rufen `join` erneut auf. Das passiert solange (daher die Schleife), bis das `join` erfolgreich ist (`break`).

Diese Lösung findest du (inkl. Debug-Ausgaben, des Beispiels aus der Angabe sowie eines Laufzeitvergleichs mit der „normalen“ Lösung) unter *stecrz.de/files/skript/QuicksortParallel.java*

Komm in mein Team!

EIDI bestanden? Zeit für ein echtes Projekt.

2020 hätte ich nicht gedacht, dass dieses Skript auch fünf Jahre später noch genutzt werden würde. Und noch weniger hätte ich gedacht, dass ich hierüber mal eine Stelle ausschreiben würde. Aber ich hätte auch erst gar nicht gedacht, dass einer der TUM-BWLer aus meinen EIDI-Crashkursen später mal mein Geschäftspartner wird.

Long Story short: Seit Ende 2023 sind Paul und ich Inhaber und Geschäftsführer von **VollCorner**, einem Münchner Bio-Lebensmittelhändler mit fast 400 Mitarbeitenden in 20 Biomärkten, der seit über 35 Jahren eine klare Mission verfolgt: Nachhaltige, faire und regionale Lebensmittel für München. Daran arbeite ich heute – mit derselben Energie, die ich während meines Informatik-Bachelors in Projekte wie dieses EIDI-Skript gesteckt habe.

Jetzt suche ich einen Allrounder, der Lust hat, unsere Daten und Prozesse auf das nächste Level zu bringen. Einen Denker, der kritisch hinterfragt, und Macher, der Lösungen findet und selbst entwickelt. Jemanden, der Micromanagement genauso hasst wie ich, stattdessen lieber frei, flexibel und eigenverantwortlich arbeitet, aber dafür so lange im Tunnel bleibt bis auch der letzte Edge-Case gefixt ist.

Wir sind ein mittelständisches Unternehmen im Wandel, den du aktiv mitgestalten kannst. Du übernimmst Themen, die unmittelbaren Mehrwert stiften – und wenn du willst gibt es auch jede Menge Freiraum für eigene Ideen und Projekte.

Klingt spannend? Dann packen wir‘s noch in **2025** an! Lebensläufe sind nett, aber viel mehr interessiert mich dein Drive.

Ich freu mich auf deine Nachricht!

Stefan

Kontakt: linkedin.com/in/berktold
Mail: stefan.berktold@vollcorner.de

Paul Pfaff & Stefan Berktold

Data- & Process-Engineer

Dein Profil:

- (Laufendes) Studium der (Wirtschafts-) **Informatik** o. ä.
- Hohes Maß an analytischem und **logischem Denkvermögen**
- Interesse an **Datenintegration** und Prozessoptimierung
- **Programmiererfahrung** (bestenfalls mit **Python**)
- Grundlegendes Verständnis von **Datenbanken** und **SQL**
- Frontend-Skills (Flask, JavaScript) sind nice, aber kein Muss

Mögliche Aufgaben:

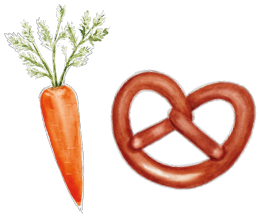
- **Data Warehouse** betreuen, **ETL-Prozesse** implementieren, Schnittstellen (z. B. Zeiterfassungssystem) per API anbinden, ggf. DWH-API entwickeln, um externe Tools auszurollen
- **Reporting** optimieren und Datenqualität verbessern
- **Digitalisieren und Automatisieren**, was das Zeug hält
- Neue Tools ausprobieren & integrieren (z.B. Web-Scraping)

Dein Team: 1 Geschäftsführer (also ich, wir arbeiten direkt zusammen), 1 IT-Systemadministrator, ca. 25 Mitarbeitende in der Zentrale (für dich insb. Controlling & Einkauf) und 20 Märkte

Dein Arbeitsplatz: Im Tunnel. Du entscheidest, wie, wo und wann du arbeitest. Kein Zeittracking, Hauptsache, du lieferst.

Umfang: Werkstudent, Teilzeit oder mehr. Es gibt genug zu tun.

Diese Beschreibung gilt für das Jahr **2025** (aber auch danach wird es spannende Projekte geben).



VollCorner Bio