
Fragen

zu den einzelnen Kapiteln des Buches

"Java ist eine Sprache"

von Ulrich Grude

Seitenangaben wie (S. 17) nach einigen Fragen beziehen sich auf das Buch "Java ist eine Sprache". Die Fragen sind (unabhängig von den Kapiteln, auf die sie sich beziehen) durchnummeriert.

Antworten zu den hier gestellten Fragen sind in einer separaten Datei namens `AntwortenZuJaSp.pdf` verfügbar. Hinweise auf Fehler, Verbesserungsvorschläge und Kommentare sind jederzeit willkommen (am liebsten per e-mail an die Adresse grude@tfh-berlin.de).

Inhaltsverzeichnis

1	Einleitung.....	7
1.1	Programmieren als Rollenspiel.....	7
1.2	Das unvermeidliche Hallo-Programm.....	7
1.3	Beispielprogramme und konkrete Ausführer.....	7
1.4	Die vier wichtigsten Grundkonzepte von Programmiersprachen.....	8
1.5	Drei Arten von Befehlen.....	8
2	Ein paar kleine Beispielprogramme.....	9
3	Programme selbst ausführen.....	10
4	Anweisungen.....	10
4.1	Einfache Anweisungen (simple statements).....	11
4.2	Zusammengesetzte Anweisungen (compound statements).....	12
5	Typen.....	14
5.1	Primitive Typen, eine Übersicht.....	14
5.2	Numerische Werte umwandeln (Cast-Befehle).....	15
5.3	Mit primitiven Werten rechnen.....	16
5.4	Gleitpunktarithmetik.....	16
5.5	Ganzzahlarithmetik.....	16
5.6	Besonderheiten des Ganzzahltyps char.....	16
5.7	Variablen als Bojen darstellen.....	17
5.8	Autohüllen (auto boxing).....	17
6	Ausdrücke, einfache und zusammengesetzte.....	18
6.1	Einfache Ausdrücke (Literele und Variablennamen).....	18
6.2	Zusammengesetzte Ausdrücke und Operatoren.....	19
7	Reihungen (arrays).....	20
7.1	Reihungen vereinbaren und als Bojen darstellen.....	21
7.2	Die Erzeugung einer Reihung in 3 Schritten.....	21
7.3	Reihungen auf verschiedene Weisen kopieren.....	21
7.4	Mehrstufige Reihungen (Reihungen von Reihungen).....	22
7.5	Mehrdimensionale Reihungen.....	22
7.6	Reihungen vergleichen und sortieren.....	23
7.7	Die Klasse Arrays.....	23
7.8	Initialisierungen sind keine Zuweisungen.....	24
7.9	Das Kovarianzproblem.....	24
8	Methoden vereinbaren und aufrufen.....	25

8.1	Parameterübergabe per Wert.....	25
8.2	Prozeduren und Funktionen.....	27
8.3	Methodennamen überladen.....	28
8.4	Methoden mit variabel vielen Parametern.....	28
8.5	Kommentare für Methoden.....	29
9	Klassen.....	30
9.1	Eine kleine Beispielklasse.....	30
9.2	Eine Klasse mit Attributen eines Referenztyps.....	30
9.3	Extreme Klassen.....	31
9.4	Klassische Fachbegriffe.....	31
9.5	Objektorientierte Programmierung.....	32
10	Ein paar Standardklassen und Methoden.....	32
10.1	Die Klasse String.....	32
10.2	Objekte mit Hilfe von Bojen genauer verstehen.....	33
10.3	Die Klasse StringBuilder.....	34
10.4	Die Klasse ArrayList.....	34
10.5	Die Klasse Random und der Zufall.....	35
10.6	Die Klassen BigInteger und BigDecimal.....	35
10.7	Die Klasse Formatter und die print-Methode.....	36
11	Struktur und Ausführung eines Java-Programms.....	36
12	Klassen erweitern (beerben) und Typgraphen.....	37
12.1	Ein kleines Beispiel für Beerbung.....	37
12.2	Allgemeine Regeln zum Beerben und Vererben.....	37
12.3	Ein größeres Beispiel für Beerbung.....	38
12.4	Wozu Untertypen gut sind.....	38
12.5	Kleinere und größere Klassen.....	38
12.6	Konstruktoren und Unterklassen.....	39
12.7	Der Typ und der Zieltyp einer Referenzvariablen.....	39
12.8	Referenzen umdeuten (Cast-Befehle).....	40
12.9	Geerbte Elemente ersetzen.....	41
12.10	Überschriebene Methoden in geerbten Methoden aufrufen.....	42
12.11	Erweitern oder instanziiieren.....	42
12.12	Klassen- und Objektinitialisierer.....	43
13	Aufzählungstypen.....	43
13.1	Ein paar einfache Beispiele.....	43
13.2	Ein Aufzählungstyp ist eine Klasse.....	44
13.3	Aufzählungstypen und static import.....	44

14	Abstrakte Klassen und Schnittstellen.....	45
14.1	Abstrakte Klassen.....	45
14.2	Schnittstellen (interfaces).....	46
14.3	Schnittstellen dürfen auch Konstanten enthalten.....	46
14.4	Schnittstellen als Parametertypen.....	46
14.5	Leere Schnittstellen als Markierungen.....	47
14.6	Anmerkungen (annotations) statt leere Schnittstellen.....	47
15	Ausnahmen fangen und werfen.....	48
15.1	Ausnahmen fangen (der try-catch-Befehl).....	48
15.2	Flugregeln für Ausnahmen.....	48
15.3	Der try-catch-finally-Befehl.....	49
15.4	Ausnahmen beim Behandeln von Ausnahmen.....	49
15.5	Geprüfte und ungeprüfte Ausnahmen.....	49
15.6	Eigene Ausnahmeklassen vereinbaren.....	50
15.7	Übersicht über ein paar wichtige Ausnahmeklassen.....	50
15.8	Zusicherungen (assertions).....	50
15.9	Kritik am Konzept der geprüften Ausnahmen.....	50
16	Generische Einheiten.....	51
16.1	Warum generische Einheiten?.....	51
16.2	Generische Klassen selbst vereinbaren.....	51
16.3	Generische Schnittstellen implementieren.....	52
16.4	Typparameter mit extends beschränken.....	53
16.5	Generische Methoden.....	53
16.6	Die Jokervariable ? im Vergleich mit anderen Typvariablen.....	54
16.7	Die Jokervariable ? und schreibende Zugriffe auf Parameter.....	54
16.8	Joker fangen (wildcard capture).....	55
16.9	Rohe Typen.....	55
16.10	Typgraphen roher und parametrisierter Typen.....	55
16.11	Falsche und fragwürdige Befehle.....	55
17	Pakete.....	56
17.1	Pakete und Klassen.....	56
17.2	Öffentliche und nur paketweit erreichbare Klassen.....	56
17.3	Mit import Abkürzungen vereinbaren.....	56
17.4	Der Klassenbaum und der Paketwald.....	57
17.5	Programme in Paketen compilieren und ausführen.....	58
18	Sammlungen (collections) und Abbildungen (maps).....	58
18.1	Schnittstelle sind Verträge.....	59

18.2	Die Sammlungsschnittstelle Collection (ohne s).....	59
18.3	Die Sammlungsschnittstellen Set, Queue und List.....	60
18.3.1	Die Schnittstelle Set.....	60
18.3.2	Die Schnittstelle Queue.....	60
18.3.3	Die Schnittstelle List.....	61
18.4	Die Vergleichsschnittstellen Comparable und Comparator.....	61
18.5	Die Sammlungsschnittstelle SortedSet.....	62
18.6	Standard-Sammlungsklassen.....	62
18.7	Die Klasse Collections (mit s).....	63
18.8	Reihungen und Sammlungen.....	63
18.9	Abbildungen (map objects).....	63
19	Ein-/Ausgabe mit Strömen (streams).....	64
19.1	Erste Beispiele mit Strömen.....	64
19.2	Zeichenorientierte und byteorientierte Ströme.....	64
19.3	Ströme, Quellen und Senken miteinander verbinden.....	65
19.4	Die Standardströme System.out, -err und -in.....	65
19.5	Brücken zwischen char-Werten und byte-Werten.....	65
19.6	Objekte in Ströme schreiben/aus Strömen lesen.....	66
19.7	Programmteile mit Röhren (pipes) verbinden.....	66
19.8	Kurze Hinweise auf weitere Stromklassen.....	66
19.9	Ein-/Ausgabe ohne Ströme (mit RandomAccessFile und nio).....	67
20	Steuerfäden (threads of control).....	67
20.1	Prozesse und Fäden.....	68
20.2	Die Klasse Threads und die Schnittstelle Runnable.....	68
20.3	Der Hauptfaden (main thread) eines Programms.....	68
20.4	Reihungen und Gruppen von Fäden.....	69
20.5	Nebenläufigkeitsfehler und synchronized-Blöcke.....	69
20.6	Synchronized-Blöcke und Verklemmungen.....	69
21	Reflexion.....	70
21.1	Class-Objekte reflektieren Klassen.....	70
21.2	Methoden einer beliebigen Klasse ausführen.....	70
21.3	Oberklassen, Konstruktoren, Methoden, Attribute etc.....	71
22	Grafische Benutzeroberflächen (Grabos).....	71
22.1	Ein paar kleine Grabo-Programme.....	71
22.2	Grabo-Objekte, -Klassen, -Pakete und Behälter.....	72
22.3	Die Pakete awt und swing, schwere und leicht Komponenten.....	73
22.4	Aktionen, Ereignisse, Listener-Schnittstellen und Adapter-Klassen.....	73

22.5	Wettläufe zwischen Haupt- und Ereignisfaden.....	74
23	Java und der Unicode.....	75
23.1	Der ASCII-Code, der Unicode und der UTF-8-Code.....	75
23.2	Unicode-Editoren und Compilationskommandos.....	75
23.3	Java-Konsolen-Programme und Unicode-Zeichen.....	75

1 Einleitung

1. Was für Begriffe werden in diesem ersten Kapitel vorgestellt?

1.1 Programmieren als Rollenspiel

2. Wie heißen die fünf *Rollen* des Rollenspiels Programmieren? (S. 2)
3. Geben Sie zwei unterschiedliche *Besetzungen* für diese fünf Rollen an.
4. Was sind die *charakteristischen Tätigkeiten* des Programmierers?
5. Was sind die *charakteristischen Tätigkeiten* des Ausführers?
6. Definieren Sie, was ein *Programm* ist (und erwähnen Sie dabei zwei Rollen des Rollenspiels Programmieren). (S. 3)
7. Welche "Dinge" haben wir unter dem Begriff des Ausführers zusammengefaßt? Alle Dinge die man wozu braucht?
8. Warum wird in diesem Buch der abstrakte Begriff *Ausführer* verwendet anstelle von konkreteren Begriffen wie Computer, Betriebssystem, Compiler etc.?

1.2 Das unvermeidliche Hallo-Programm

9. Wann schreibt man typischerweise ein sogenanntes *Hallo-Programm*? (S. 4)
10. Welche *Arbeitsschritte* probiert der Programmierer typischerweise anhand eines Hallo-Programms zum ersten Mal aus?
11. Wie kann man in einem Java-Programm eine Zeile (oder einen Teil einer Zeile) als *Kommentar* kennzeichnen?
12. Für *wen* sind solche Kommentare gedacht? Wer ignoriert sie weitgehend?
13. Jedes Java-Programm muss ein Unterprogramm mit einem bestimmten Namen enthalten. Geben Sie den Namen dieses Unterprogramms an.
14. An bestimmten Stellen eines Java-Programms darf der Programmierer eine *neue Textzeile* beginnen. Beschreiben Sie drei solche Stellen.
15. Wie kann man ein langes String-Literal in einem Java-Programm "auf mehrere Zeilen verteilen"?

1.3 Beispielprogramme und konkrete Ausführer

16. Wie bezeichnet man die *Dateien*, in die der Programmierer seine Programme schreibt (obwohl aus diesen Dateien gar kein Wasser hervorsprudelt)? (S. 8)
17. Was verstehen Informatiker unter einer *Plattform*?
18. Was macht ein *Quelldateien-Interpreter*?

19. Was macht ein *nativer* Java-Compiler?
20. Was macht ein *normaler* (nicht-nativer) Java-Compiler?

1.4 Die vier wichtigsten Grundkonzepte von Programmiersprachen

21. Wie heißen die vier wichtigsten *Grundkonzepte* von Programmiersprachen?
22. Was verstehen Informatiker unter einer *Variablen*? (S. 12)
23. Durch welche beiden Eigenschaften unterscheidet sich eine *Variable* (d.h. ein Wertebehälter) von normalen Behältern wie Schuhkartons und Marmeladengläsern? (S. 13)
24. Variablen sind Wertebehälter. Welche *anderen Dinge* (ausser Variablen) bezeichnen wir auch als Wertebehälter?
25. Was ist ein *Typ*? Geben Sie ein oder zwei Definitionen an. (S. 14)
26. Was wird durch den Typ einer Variablen festgelegt?
27. Was muss (und darf) der Programmierer mit einem *Unterprogramm* nur *einmal* machen? Und was darf er dann (mit dem Unterprogramm) *beliebig oft* machen? (S. 16)
28. Nennen Sie ein oder zwei *Vorteile* von Unterprogrammen.
29. Was ist ein *Modul*? (S. 18)
30. Beschreiben Sie einen wichtigen *Vorteil von Modulen* beim Testen eines Programms (welches aus vielen Modulen besteht).
31. Was ist eine *Klasse*? Geben Sie hier eine vorläufige und noch unvollständige Definition an. (S. 19)

1.5 Drei Arten von Befehlen

32. Im Grunde genommen gibt es in allen Programmiersprachen nur *drei Arten von Befehlen*. Wie heißen diese drei Befehlsarten auf Deutsch? (S. 19)
33. Wie heißen die drei Befehlsarten auf *Englisch*?
34. Wenn man eine *Vereinbarung* ins Deutsche übersetzt, wie beginnt dann der entsprechende deutsche Befehlssatz? (S. 20)
35. Ebenso für einen *Ausdruck*!
36. Ebenso für eine *Anweisung*!
37. Was für eine Art von Befehl ist `int otto = 17; ?` Übersetzen Sie diesen Befehl ins Deutsche.
38. Ebenso für den Befehl `2 * otto + 35`
39. Ebenso für den Befehl `otto = 2 * otto + 35;`

40. Ebenso für den Befehl `class Carl { ... }`
41. Ebenso für den Befehl `otto`
42. Ebenso für den Befehl `3.7`
43. Was bewirkt ein normaler Ausdruck?
44. Was bewirkt ein normaler Ausdruck nicht?
45. Mit welchem Fachbegriff bezeichnet man nicht-normale Ausdrücke (die etwas haben oder bewirken, was normale Ausdrücke nicht haben oder bewirken)?
46. Der Befehl `System.out.println("Hallo!");` ist eine Anweisung. Der Inhalt von welchem Wertebehälter wird durch diese Anweisung verändert?
47. Sei `karl` eine `int`-Variable mit dem momentanen Wert 25. Welchen Wert hat dann der Ausdruck `++karl` und welchen Seiteneffekt hat dieser Ausdruck? (S. 24)
48. Sei `karl` eine `int`-Variable mit dem momentanen Wert 25. Welchen Wert hat dann der Ausdruck `karl++` und welchen Seiteneffekt hat dieser Ausdruck?

2 Ein paar kleine Beispielprogramme

49. Wie bezeichnet man ein Unterprogramm, welches innerhalb einer Klasse vereinbart wurde? (S. 27)
50. Man unterscheidet ganz allgemein zwei Arten von Unterprogrammen. Wie heißen diese beiden Unterprogrammarten?
51. Ist die Reihenfolge, in der man Methoden innerhalb einer Klasse vereinbart, wichtig oder unwichtig?
52. Wozu dienen Funktionen? Oder: Was kann man mit einem Funktionsaufruf bewirken?
53. Wozu dienen Prozeduren? Oder: Was kann man mit einem Prozeduraufruf bewirken?
54. Was muss man beim Vereinbaren einer Funktion unmittelbar vor dem Namen der Funktion angeben?
55. Was muss man beim Vereinbaren einer Prozedur unmittelbar vor dem Namen der Prozedur angeben?
56. Ein Prozeduraufruf wie `pln("Hallo!");` ist in vielen Beispielprogrammen eine Abkürzung für welchen deutlich längeren Prozeduraufruf?
57. Wir wollen in der `main`-Methode einer Klasse `Hallo30` eine parameterlose Methode (genauer: Prozedur) namens `formatiereDieFestplatte` aufrufen, die in einer Klasse namens `Hallo20` vereinbart wurde. Welchen Befehl müssen wir hinschreiben?

58. Übersetzen Sie die Anweisung `if (a < b) pln("a ist kleiner!");` ins Deutsche.
59. Angenommen, `a` und `b` sind `int`-Variablen mit den Werten 17 bzw. 25. Welchen Wert hat dann der Ausdruck `a < b`? Zu welchem Typ gehört dieser Wert (und somit auch der Ausdruck `a < b`)? Warum ist somit ein Ausdruck wie `a < b < 30` in Java-Programmen *nicht* erlaubt?
60. Wo kommen *Ausdrücke* vor? Wann entstehen *Werte* und wer manipuliert *Werte*?
61. Was sollte man über den Wert des Java-Ausdrucks `0.1` wissen?

3 Programme selbst ausführen

62. Welchen Vorteil hat ein *maschineller* Java-Ausführer im Vergleich zu einem menschlichen Java-Ausführer? (S. 41)
63. Welchen Vorteil hat ein *menschlicher* Java-Ausführer im Vergleich zu einem maschinellen Java-Ausführer?

4 Anweisungen

64. Was befiehlt der Programmierer dem Ausführer mit einer *Anweisung*?
65. Man unterscheidet *zwei Arten von Anweisungen*. Wie heißen diese Anweisungsarten auf Deutsch und auf Englisch?
66. Wodurch unterscheiden sich diese beiden Arten von Anweisungen?
67. Man unterscheidet *zwei Arten von zusammengesetzten Anweisungen*. Mit welchen (ziemlich langen und unhandlichen) Worten bezeichnet man diese beiden Arten von zusammengesetzten Anweisungen?
68. Wenn ein Unterprogramm ausgeführt wird, welches nur *einfache Anweisungen* enthält, wie oft wird dann jede dieser einfachen Anweisungen ausgeführt? (Keine Angst vor ganz 1-fachen Antworten! :-).
69. Was kann der Programmierer mit einer Fallunterscheidungsanweisung bewirken? Zur Erinnerung: Eine Fallunterscheidungsanweisung ist eine zusammengesetzte Anweisung und enthält somit (mindestens) eine andere Anweisung `A`.
70. Was kann der Programmierer mit einer Wiederholungsanweisung bewirken? Zur Erinnerung: Eine Wiederholungsanweisung ist eine zusammengesetzte Anweisung und enthält somit eine andere Anweisung `A`.

4.1 Einfache Anweisungen (simple statements)

71. Welches Zeichen wird in den Sprachen C, C++ und Java zur Bezeichnung der Zuweisungsanweisung (oder kürzer: der Zuweisung) verwendet? Warum wurden die Entwickler der Sprachen C/C++/Java für diese Festlegung nicht zu Haftstrafen verurteilt?
72. Welche Zeichenkombination wird in den Sprachen C, C++ und Java zur Bezeichnung der Gleichheitsoperation verwendet?
73. Wie sollte man eine Zuweisung wie `a = b + c;` (beim lauten Vorlesen) *nicht* aussprechen, und wie sollte man sie aussprechen?
74. Für welche (etwas längere) Anweisung ist die Anweisung `berta += 17;` eine Abkürzung? (S. 49)
75. Sei `isidor` eine `int`-Variable. Wie kann man aus einem Ausdruck mit Seiteneffekt wie etwa `++isidor` eine Anweisung machen? (S. 50)
76. Geben Sie einen Aufruf der Methode `System.out.println` an.
77. Sei `hoch2` eine Funktion mit einem `int`-Parameter und dem Ergebnistyp `int` (wie in der Klasse `Hallo04` auf S. 29 vereinbart). Geben Sie einen sinnvollen und einen sinnlosen (aber erlaubten) Aufruf dieser Methode `hoch2` an.
78. Die `break`-Anweisung darf man nur an bestimmten Stellen eines Java-Programms verwenden. Wo?
79. Die `continue`-Anweisung darf man nur an bestimmten Stellen eines Java-Programms verwenden. Wo? (S. 51)
80. Aus welchen beiden Teilen besteht die `while`-Schleife (in Zeile 2 bis 5)

```

1  int n = 3;
2  while (n < 7) {
3      pln(hoch2(n));
4      n = n + 1;
5  }

```

81. Wie oft wird der Rumpf dieser `while`-Schleife ausgeführt (wenn man die Schleife einmal ausführt)? (S. 51)
82. Es ist wichtig, zwischen einer Ausführung einer *Schleife* und einer Ausführung des *Schleifenrumpfes* zu unterscheiden. Welche dieser beiden Ausführungen kann man mit einer `break`-Anweisung und welche mit einer `continue`-Anweisung beenden? (S. 52)
83. Anweisungen sind Befehle, die die Inhalte von Wertebehältern (z. B. den Inhalt einer Variablen oder den Inhalt eines Bildschirms) verändern. Wieso sind dann die Befehle `break` und `continue` Anweisungen? Welcher Wertebehälter wird durch sie verändert?

4.1 Einfache Anweisungen (simple statements)

84. Es gibt zwei Arten von Methoden: Funktionen und Prozeduren. Methoden welcher Art *muss* man mit einer `return`-Anweisung beenden und welche Methoden *darf* man mit einer `return`-Anweisung beenden?
85. Eine `return`-Anweisung in einer Prozedur muss anders aussehen als eine `return`-Anweisung in einer Funktion. Beschreiben Sie den Unterschied.
86. Wieviele `return`-Anweisungen muss eine *Funktion* mindestens enthalten? Wieviele `return`-Anweisungen darf eine *Prozedur* enthalten?

4.2 Zusammengesetzte Anweisungen (compound statements)

87. Wozu dienen *Blockanweisungen*? Oder: Was kann der Programmierer mit einer Blockanweisung bewirken? (S. 58)
88. Wie bezeichnet man Variablen, die *innerhalb einer Blockanweisung* vereinbart wurden (im Gegensatz zu Variablen, die direkt in einer Klasse vereinbart wurden)?
89. Wie lange (oder: bis wann) leben Variablen, die innerhalb einer Blockanweisung vereinbart wurden?
90. Es gibt verschiedene Varianten der `if`-Anweisung. Aus wieviel Teilen besteht die einfachste Variante und wie nennen wir diese Teile?
91. Eine einfache `if`-Anweisung mit `else`-Teil besteht aus drei Teilen. Wie nennen wir diese Teile?
92. Stellen Sie sich eine komplizierte `if`-Anweisung mit vielen Bedingungen und Rümpfen, aber ohne `else`-Teil vor. Genau wieviele der Bedingungen werden mindestens ausgewertet und wieviele der Rümpfe werden ausgeführt? (S. 63)
93. Stellen Sie sich eine komplizierte `if`-Anweisung mit vielen Bedingungen und Rümpfen und einem `else`-Teil vor. Genau wieviele der Bedingungen werden mindestens ausgewertet und wieviele der Rümpfe werden ausgeführt?
94. Wenn Sie mit kleinem Gepäck verreisen wollen und nur entweder die `if`-Anweisung oder die `switch`-Anweisung mitnehmen können (aber nicht beide), welche sollten Sie dann mitnehmen?
95. Was versteht man in Java-Programmen unter einem *konstanten Ausdruck*? Geben Sie ein paar Beispiele für konstante Ausdrücke an. (S. 67)
96. *Fallmarken* (engl. case labels) dürfen nur in `switch`-Anweisungen vorkommen. Beschreiben Sie den Aufbau einer Fallmarke.
97. Beschreiben Sie zwei kleine *Schwächen* der `switch`-Anweisung in Java.
98. Aus wieviel Teilen besteht eine `while`-Schleife und wie heißen diese Teile? (S. 68)

99. Welche Form der `while`-Schleife sollte man (zumindest während der Entwicklung eines Programms) verwenden und warum?
100. Welchen Vorteil haben die beiden Spezialformen der `while`-Schleife (Beispiele: `while (a < b) { ... }` und `do { ... } while (a < b);`)?
101. Aus wieviel Teilen besteht eine `for`-Schleife und wie haben wir diese Teile genannt?
102. In welcher Reihenfolge werden die Teile einer `for`-Schleife ausgeführt?
103. Wie oft wird der Rumpf einer `for`-Schleife mindestens ausgeführt?
104. Darf man eine `for`-Schleife auch durch eine `break`-Anweisung im Rumpf abbrechen?
105. Welche der vier Teile einer `for`-Schleife darf man leer lassen?
106. Übersetzen Sie die Variablenvereinbarung `int otto = 17;` ins Deutsche.
107. Angenommen, wir führen ein Java-Programm mit Papier und Bleistift (und einem Radiergummi) aus. Was sollten wir dann tun, wenn wir zu einer Variablenvereinbarung wie etwa `int otto = 17;` kommen? Wie sollten wir diese Vereinbarung ausführen? (S. 78)
108. Mit welcher Gefahr ist das Programmieren einer Schleife (fast) immer verbunden? Gemeint ist eine "typische Schleifen-Gefahr", die (fast) nur im Zusammenhang mit Schleifen auftritt, aber nicht bei anderen Befehlen.
109. Mit Java 5 wurden sichere `for`-Schleifen eingeführt, von denen der Ausführer (unter bestimmten Voraussetzungen) garantiert, dass sie keine Endlosschleifen sind. Was für Probleme kann man mit diesen sicheren `for`-Schleifen lösen?
110. Je nachdem, mit welchem Wert man die Variablen `n` (in Zeile 1) initialisiert, ist die nachfolgende `while`-Schleife eine *endliche* Schleife oder ein *Endlosschleife*:

```

1  int n = ... ; // Wird irgendwie initialisiert
2  while (n != 1) {
3      n = n / 2;
4      println("n: " + n);
5  }
```

Für welche Anfangswerte von `n` ist diese `while`-Schleife endlich und für welche Anfangswerte ist sie endlos?

111. Betrachten Sie die folgenden geschachtelten `for`-Schleifen:

```

1  for (int i=1; i<=5; i++) {
2      for (int j=1; j<=20; j++) {
3          p('*');
4      }
5  }
```

- Aus wieviel und welchen Befehlen besteht der Rumpf der äußeren Schleife ("der `for-i`-Schleife")? Aus wieviel und welchen Befehlen besteht der Rumpf der inneren Schleife ("der `for-j`-Schleife")? (S. 86)
112. Wenn man die `for-i`-Schleife einmal ausführt, wie oft wird dadurch ihr Rumpf ausgeführt?
Wenn man die `for-j`-Schleife einmal ausführt, wie oft wird dadurch ihr Rumpf ausgeführt?
Wenn man die `for-i`-Schleife einmal ausführt, wie oft wird dadurch der Rumpf der `for-j`-Schleife ausgeführt?
113. Warum wäre es sinnvoll und nützlich, auch in die Sprache Java einen `goto`-Befehl aufzunehmen?

5 Typen

114. In Java unterscheidet man (zuerst einmal) *zwei Arten von Typen*. Wie heißen diese Typarten? (S. 89)
115. Wieviele *primitive Typen* gibt es in Java und wie heißen sie?
116. Bei den *Referenztypen* unterscheidet man drei Unterarten. Wie heißen die?

5.1 Primitive Typen, eine Übersicht

117. Wieviele *nicht-numerische* Typen gibt es und wie heißen sie? (S. 91)
118. Wieviele *numerische* Typen gibt es und wie heißen sie?
119. Wieviele *Ganzzahltypen* gibt es und wie heißen sie?
120. Wieviele *Gleitpunktypen* gibt es und wie heißen sie?
121. Wie lang ist ein Wert des Typs `byte` (gemessen in Bits) und wieviele Werte gehören zum Typ `byte`? Geben Sie die Anzahl der Werte exakt an (z. B. als Zweierpotenz) und als "runde Zahl, die man sich merken kann". (S. 93)
122. Ebenso für den Typ `char`.
123. Ebenso für den Typ `short`.
124. Ebenso für den Typ `int`.
125. Ebenso für den Typ `long`.
126. Ebenso für den Typ `float`.
127. Ebenso für den Typ `double`.
128. Wieviele Werte gehören zum Typ `boolean` und mit welchen Schlüsselworten bezeichnet man sie?

129. Repräsentiert jeder `int`-Wert eine Zahl? Repräsentiert jeder `float`-Wert eine Zahl?
130. Zum Gleitpunkttyp `double` gehören nicht nur "normale Werte", die "normale Zahlen" wie 1.5 oder -12.345E6 repräsentieren, sondern auch einige "besondere Werte". Beschreiben Sie diese besonderen Werte.
131. Wieviele Hüllklassen (oder: Wickelklassen, engl. wrapper classes) gibt es?
132. Wie heißen die vier Hüllklassen für die vier (primitiven) Typen `byte`, `char`, `int` und `float`?
133. Zum Ganzzahltyp `int` gehören gleich viele Werte wie zum Gleitpunkttyp `float` (etwa 4 Milliarden). Wie ist es dann möglich, dass der größte `float`-Wert (etwa 3.402E38) viel, viel, also sehr viel größer ist, als der größte `int`-Wert (etwa 2 Milliarden oder 2E9)? Kann man jede Zahl, die sich als `int`-Wert darstellen läßt, auch als `float`-Wert darstellen?

5.2 Numerische Werte umwandeln (Cast-Befehle)

134. Was versteht man unter einer *Typumwandlung*? Wird dabei wirklich der Typ einer Variablen oder eines Wertes "umgewandelt", d.h. verändert? (S. 95)
135. Was versteht man unter einer *erweiternden* numerischen Typumwandlung?
136. Was versteht man unter einer *verengenden* numerischen Typumwandlung?
137. Sind *erweiternde* numerische Typumwandlungen im allgemeinen *harmlos* oder *gefährlich*? Sind *verengende* Typumwandlungen im allgemeinen *harmlos* oder *gefährlich*? (S. 96)
138. Welche numerischen Typumwandlungen führt der Ausführer auch *ohne* ausdrücklichen Befehl des Programmierers durch, *verengende* oder *erweiternde*?
139. Zwischen was für Typen kann man Typumwandlungen *mit Cast-Befehlen* veranlassen? (S. 100)
140. Wie nennt man die Verluste, die auch bei einer *erweiternden* numerischen Typumwandlung auftreten können? Verlust an ... ?
141. Wie nennt man die Verluste, die nur bei einer *verengenden* Typumwandlung (auf Grund eines Cast-Befehls) auftreten können? Verlust der ... ?
142. Sind in Java Typumwandlungen zwischen den Typen `boolean` und `int` möglich? (S. 101)
143. Sind in Java Typumwandlungen zwischen den (primitiven) numerischen Typen und dem Typ `String` möglich?

5.3 Mit primitiven Werten rechnen

144. Welche *Zahlenmengen* spielen in der *Mathematik* eine wichtige Rolle und in welcher Beziehung stehen diese Zahlenmengen zueinander?
145. Welche *Zahlenmengen* kennt der *Java-Ausführer* und in welcher Beziehung stehen diese Zahlenmengen zueinander?

5.4 Gleitpunktarithmetik

146. Welchen Wert hat der Ausdruck `-0.0 * -0.0`? (S. 107)
147. Welchen Wert hat der Ausdruck `17.0 / Double.POSITIVE_INFINITY`?
148. Welchen Wert hat der Ausdruck `Double.NEGATIVE_INFINITY * +0.0`?
149. Welchen Wert hat der Ausdruck `17.0 / +0.0`?
150. Welchen Wert hat der Ausdruck `17.0 / -0.0`?

5.5 Ganzzahlarithmetik

151. `Character.MIN_VALUE` (die Konstante `MIN_VALUE` im Modul `Character`) ist der *kleinste* Wert des primitiven Typs `char`. Entsprechend ist `Character.MAX_VALUE` der *größte* Wert des Typs `char`. Geben Sie die Werte der Konstanten `Character.MIN_VALUE` und `Character.MAX_VALUE` (exakt oder zumindest ungefähr) an.
152. `Integer.MIN_VALUE` (die Konstante `MIN_VALUE` im Modul `Integer`) ist der *kleinste* Wert des primitiven Typs `int`. Entsprechend ist `Integer.MAX_VALUE` der *größte* Wert des Typs `int`. Geben Sie die Werte der Konstanten `Integer.MIN_VALUE` und `Integer.MAX_VALUE` (zumindest ungefähr) an.
153. Welchen Wert hat der Ausdruck `Integer.MIN_VALUE - 1`? (S. 112)
154. Welchen Wert hat der Ausdruck `Integer.MAX_VALUE + 1`?
155. Welchen Wert hat der Ausdruck `-Integer.MIN_VALUE`?

5.6 Besonderheiten des Ganzzahltyps `char`

156. Welche Ganzzahlen gehören zum primitiven Typ `char`?
157. Was erscheint auf dem Bildschirm, wenn man den `int`-Wert 65 dorthin ausgibt?
158. Was erscheint auf dem Bildschirm, wenn man den `char`-Wert 65 dorthin ausgibt?

159. Was erscheint auf dem Bildschirm, wenn man die folgenden Zahlen als `char`-Werte dorthin ausgibt: 48? 49? 57? 90? 97? 98? 122?
160. Angenommen, Ihre Großmutter wird demnächst 68 Jahre alt und Sie wollen mit dieser Zahl eine `char`-Variable namens `alter` initialisieren. Geben Sie die Vereinbarung der Variablen `alter` an.
161. Angenommen, Sie wollen den Buchstaben `D` in einem String suchen und wollen deshalb eine `char`-Variable namens `buchstabe` mit diesem Buchstaben initialisieren. Geben Sie die Vereinbarung der Variablen `buchstabe` an.

5.7 Variablen als Bojen darstellen

162. Aus wieviel Teilen besteht jede Variable mindestens und wie heißen diese Teile? (S. 119)
163. Wieviel weitere Teile kann eine Variable haben und wie heißen diese optionalen Teile?
164. Wieviele und welche Teile einer Variablen gehören zur Welt des *Ausführers* (weil nur er sie festlegen und manipulieren kann)? Wieviele und welche Teile gehören zur Welt des *Programmierers*?
165. Wenn man eine Variable als Boje darstellt, wird ihre *Referenz* in ein Kästchen einer bestimmten Form eingezeichnet. Welche Kästchenform ist das?
166. Wenn man eine Variable als Boje darstellt, wird ihr Wert in ein Kästchen einer bestimmten Form eingezeichnet. Welche Kästchenform ist das?
167. Wenn man eine *Referenzvariable* als Boje darstellt, wird ihr Wert in ein viereckiges *und* in ein sechseckiges Kästchen eingezeichnet. Warum?
168. Welcher Teil der Variablen `x` wird durch eine *Zuweisung* `x = ...`; verändert, der Name, die Referenz, der Wert oder der Zielwert von `x`? (S. 122)
169. Welcher Teil der Variablen `x` wird durch einen *Ausgabebefehl* wie `println(x)`; oder `System.out.println(x)`; ausgegeben, der Name, die Referenz, der Wert oder der Zielwert von `x`? (S. 123)
170. Wenn man zwei Variablen mit dem *Gleichheitsoperator* `==` vergleicht, etwa so: `x == y`, welche Teile der Variablen werden dann verglichen, die Namen, die Referenzen, die Werte oder die Zielwerte?

5.8 Autohüllen (auto boxing)

171. Wer hat 1969 den Boxkampf zwischen einem VW-Käfer und einem Chevrolet Baujahr 58 gewonnen?

6 Ausdrücke, einfache und zusammengesetzte

172. Was befiehlt der Programmierer dem Ausführer mit einem Ausdruck (z. B. mit dem Ausdruck `x + 1`)? (S. 129)
173. Angenommen, `x` ist eine `int`-Variable und hat momentan den Wert 3. Welchen Wert hat dann der Ausdruck `5 * x / 10` und zu welchem Typ gehört dieser Wert?
174. Zu welchem Typ gehört der Ausdruck `123` und welchen Wert hat er?
175. Zu welchem Typ gehört der Ausdruck `0.1` und was sollte jede Java-ProgrammiererIn über seinen Wert wissen?
176. Wie berechnet der Ausführer den Wert einer Variablen `x`? (S. 129)
177. Sei `y` eine Variable des Typs `double`. Geben sie eine *Vereinbarung*, eine *einfache Anweisung* und eine *zusammengesetzte Anweisung* (insgesamt also drei Befehle) an, in denen der Ausdruck `2.5 * y + 3.2` vorkommt.
178. Was ist ein *Ausdruck mit Seiteneffekt*?
179. Sei `n` eine `int`-Variable mit dem Wert 5. Welchen Wert hat dann der Ausdruck `n+1` und welchen Wert hat die Variable `n`, nachdem der Ausführer den Wert des Ausdrucks berechnet hat?
180. Sei `n` eine `int`-Variable mit dem Wert 5. Welchen Wert hat dann der Ausdruck `n++` und welchen Wert hat die Variable `n`, nachdem der Ausführer den Wert des Ausdrucks berechnet hat?
181. Sei `n` eine `int`-Variable mit dem Wert 5. Welchen Wert hat dann der Ausdruck `++n` und welchen Wert hat die Variable `n`, nachdem der Ausführer den Wert des Ausdrucks berechnet hat?
182. Welchen *Seiteneffekt* hat der (Objekterzeugungs-) Ausdruck `new StringBuilder("ABC")`? (S. 131)

6.1 Einfache Ausdrücke (Literele und Variablennamen)

183. Was ist ein *Literal*? Ein lateinischer Buchstabe? Ein griechisches Wort? Ein Name für einen Wert? Eine Referenz auf ein Objekt? (S. 132)
184. Für welche Typen *gibt es* in Java *Literele*?
185. Für welche primitiven Typen gibt es in Java *keine* Literale?
186. Zu welchen Typen gehören die folgenden acht Literale? (S. 133)

Literal:	"true"	false	"7"	'7'	7	7L	7.0	7.0F
Typ:								

187. Bezeichnen Sie den `int`-Wert zehn durch *drei* Literale, ein *dezimales* Literal, ein *oktales* Literal und ein *hexadezimal*es Literal.
188. Bezeichnen Sie den größten und den kleinsten `int`-Wert jeweils durch ein hexadezimaler Literal.
189. Ist `-123` ein *Literal* und falls ja, zu welchem Typ gehört es?
190. Ist `+1.5` ein *Literal* und falls ja, zu welchem Typ gehört es?
191. Geben Sie drei verschiedene `char`-Literals an, die ein großes `a` (d.h. ein `A`) bezeichnen. (S. 134)
192. Geben Sie drei verschiedene `char`-Literals an, die *einen* Rückwärtsschrägstrich `\` bezeichnen.
193. Wieviele Zeichen enthält das `String`-Literal `"ABCD"` ?
194. Wieviele Zeichen enthält das `String`-Literal `"\\\\"` ?
195. Wieviele Zeichen enthält das `String`-Literal `"\""` ?
196. Welche Eigenschaft von Java-Ganzzahlliterals sollte möglichst bald verbessert werden? Welche unschöne Eigenschaft von Java-Ganzzahlliterals lässt sich wahrscheinlich nicht mehr verbessern? (S. 137)
197. Ein Literal ist *ein Name für einen Wert*. Eine Konstante, z. B.
- ```
1 final int MWST = 22; // Zukünftiger Mehrwertsteuersatz?
2 final double PI = 3.5 // π in einem nicht-euklidischen Raum?
```
- ist ebenfalls *ein Name für einen Wert*. Was ist der Unterschied zwischen diesen verschiedenen Arten von *Namen für Werte*?
198. Was ist der Unterschied zwischen einer *Konstanten* und einer *unveränderbaren Variablen*? (S. 138)

## 6.2 Zusammengesetzte Ausdrücke und Operatoren

199. Sei `n` eine `int`-Variable. Geben Sie (als Beispiele) zwei *einfache* und zwei *zusammengesetzte* Ausdrücke an.
200. Was ist ein *Operator*? (S. 139)
201. Was ist eine *Operation*?
202. Welche *Stelligkeit* haben die meisten arithmetischen Operationen, z. B. Additionsoperationen, Subtraktionsoperationen, Multiplikationsoperationen etc. (in Java und in vielen anderen Sprachen)? (S. 141)
203. Welche Notation für zweistellige Operatoren hat sich seit dem Mittelalter (ähnlich wie andere ansteckende Krankheiten) in Europa und dann auf der ganzen Erde verbreitet?
204. Nennen Sie zwei *Probleme*, die durch die *Infixnotation* aufgeworfen werden!
205. Womit hat man versucht, die Probleme der Infixnotation zu *lösen*?
206. Ist der Versuch, die Probleme der Infixnotation zu lösen, *geglückt*? (S. 142)

207. Der ungeklammerte Ausdruck `n1 + n2 * n3` entspricht dem voll geklammerten Ausdruck `(n1 + (n2 * n3))`. Welchem voll geklammerten Ausdruck entspricht der ungeklammerte Ausdruck `n1 - n2 - n3` ?
208. Welchem voll geklammerten Ausdruck entspricht der ungeklammerte Ausdruck `b1 || b2 && b3` ? (S. 144)
209. Welchem voll geklammerten Ausdruck entspricht der ungeklammerte Ausdruck `v1 = v2 = v3 = v4` ?
210. Zu welchem Typ müssen die Variablen `b1`, `b2` und `b3` gehören, damit der Ausdruck `b1 == b2 == b3` syntaktisch korrekt ("erlaubt") ist?
211. Welchem voll geklammerten Ausdruck entspricht der ungeklammerte Ausdruck `b1 == b2 == b3` ?
212. Wieviele verschiedene *Bindungsstärken* haben die Operatoren von Java? 3? 13? 23? 33?
213. In Java haben die meisten Operationen *keinen Seiteneffekt*. Welche Operatoren bezeichnen Operationen *mit Seiteneffekt*?
214. In Java sind alle Operatoren *überladen*, d.h. jeder Operator bezeichnet *mehrere Operationen*. Z. B. bezeichnet der Operator `*` vier verschiedene Multiplikationsoperationen (je eine für `int`-Werte, `long`-Werte, `float`-Werte und `double`-Werte). Wieviele Operationen bezeichnet der Operator `++` ?

## 7 Reihungen (arrays)

215. Wie kann man 5000 `double`-Variablen vereinbaren, ohne (vom vielen Schreiben) Blasen an den Händen zu bekommen? (S. 151)
216. Als was bezeichnen wir die Variablen, aus denen eine Reihung besteht? Als *Module* der Reihung? Als *Komponenten* der Reihung? Als *Elemente* der Reihung? Als *Teile* der Reihung?
217. Bei einer Reihung der Länge 20 laufen die Indizes von welcher Zahl bis zu welcher Zahl?
218. Aus wieviel Komponenten besteht eine Reihung, deren letzter (größter) Index gleich 99 ist?
219. Angenommen, `r` ist eine Reihung der Länge 5000. Was wissen wir über die Zeiten, die ein Zugriff auf die nullte Komponente `r[0]` bzw. auf die letzte Komponente `r[4999]` kostet?
220. Ersetzen Sie im folgenden Satz Auslassungen `"..."` durch geeignete Worte für Materialien: "Reihungen sind aus ... und nicht aus ... !"
221. In welcher Einheit misst man (in Java) die Länge einer Reihung? In Zentimeter? In Zoll? In Bits? In Bytes? In Komponenten? In Nanosekunden?
222. Geben Sie einen `println`-Befehl an, der die *Länge* der Reihung `r` ausgibt.

223. Zu welchem *Typ* gehört das `length`-Attribut einer Reihung? Wie lang kann eine Reihung deshalb *höchstens* sein?
224. Sei `zeilenReihung` eine Reihung von `String`-Variablen. Geben Sie eine sichere und einfache `for`-Schleife an, die die Komponenten dieser Reihung zur Standardausgabe ausgibt. Verwenden Sie die Prozedur `pln`, um die einzelnen `String`-Komponenten auszugeben. (S. 153)

## 7.1 Reihungen vereinbaren und als Bojen darstellen

225. Vereinbaren Sie eine Reihung namens `ir01`, die aus 5 `int`-Variablen mit den Werten 10, 20, 30, 40, 50 besteht. (S. 155)
226. Vereinbaren Sie eine Reihung namens `br01`, die aus 4 `boolean`-Variablen mit den Werten `true`, `true`, `false`, `true` besteht.
227. Vereinbaren Sie eine Reihung namens `sr02`, die die 3 Strings "Anna", "Bert" und "Carola" als Komponenten enthält. (S. 157)
228. Im Alltag sagt man von der Reihung aus der vorigen Frage: "`sr02` enthält drei Strings". Genau genommen enthält `sr02` aber keine `String`-Objekte, sondern andere (kleinere) "Gebilde". Was für "Gebilde" sind das?
229. Wenn man eine Reihung `r` in *vereinfachter Bojendarstellung* zeichnet, welche Teile kann man dann (im Zuge der Vereinfachung) weglassen? (S. 156/157)
230. Welche Werte haben die Komponenten der folgenden Reihungen (unmittelbar nachdem der Ausführer sie erzeugt hat):

```
3 float[] fr01 = new float[500];
4 boolean[] br02 = new boolean[750];
```

## 7.2 Die Erzeugung einer Reihung in 3 Schritten

231. Eine Reihung wie
- ```
5 String[] sr02 = {"Anna", "Bert", "Carola"};
```
- wird vom Ausführer in drei Schritten erzeugt. Beschreiben Sie diese Schritte. (S. 161)

7.3 Reihungen auf verschiedene Weisen kopieren

232. Eine Reihungsvariable wie `sr02` (siehe vorige Frage) kann man auf drei ganz verschiedene Weisen *kopieren*. Beschreiben Sie diese Kopierweisen. (S. 162)

7.4 Mehrstufige Reihungen (Reihungen von Reihungen)

233. Vereinbaren Sie eine Reihung namens `frrA`, die drei Reihungen enthält, von denen jede genau fünf `float`-Variablen enthält. (S. 165)
234. Vereinbaren Sie eine Reihung namens `frrB`, die drei Reihungen von `float`-Variablen enthält. Diese drei Reihungen sollen genau fünf bzw. drei bzw. vier `float`-Variablen enthalten.
235. Vereinbaren Sie eine Reihung namens `frrC`, die die gleiche Struktur hat wie `frrB` (siehe vorige Frage) und initialisieren Sie die 12 `float`-Variablen in `frrC` mit den Werten 1.5F, 2.5F, 3.5F, ..., 12.5F.
236. Die Summe aller `float`-Werte in der Reihung `frrC` soll ermittelt werden. Schreiben Sie eine entsprechende `for`-Schleife (mit einer geschachtelten `for`-Schleife darin).
237. Die Werte aller `float`-Variablen in der Reihung `frrC` (aus der vorigen und vorvorigen Frage) sollen verdoppelt werden. Schreiben Sie eine entsprechende `for`-Schleife (mit einer geschachtelten `for`-Schleife darin).
238. Wie werden *mehrstufige Reihungen* im Englischen bezeichnet?
239. Angenommen, `frrrA` ist eine Reihung vom Typ `float[][][]` (Reihung von Reihungen von Reihungen von `float`-Variablen). Von welchem Typ sind die *Komponenten* von `frrrA`? Und von welchem Typ sind die *elementaren Komponenten* von `frrrA`?
240. Angenommen, `frA` ist eine Reihung vom Typ `float[]`. Von welchem Typ sind die *Komponenten* von `frA`? Und von welchem Typ sind die *elementaren Komponenten* von `frA`?
241. Es gibt in Java Reihungen, bei denen der Begriff der *Stufigkeit* "zusammenbricht". Solche Reihungen sind z. B. gleichzeitig 2-stufig und 3-stufig. Vereinbaren Sie eine solche Reihung. (S. 167)
242. Vereinbaren Sie zwei Reihungen namens `dreiG` und `dreiId`, beide vom Typ `int[][]`. Die Reihung `dreiG` soll drei *gleiche* Komponenten und `dreiId` drei *identische* Komponenten ("dreimal dieselbe Komponente") enthalten.

7.5 Mehrdimensionale Reihungen

243. Gibt es in Java *mehrdimensionale Reihungen*?
244. Welchen Vorteil haben *mehrdimensionale* Reihungen im Vergleich zu mehrstufigen Reihungen?
245. Welchen Vorteil haben *mehrstufige* Reihungen im Vergleich zu mehrdimensionalen Reihungen?

7.6 Reihungen vergleichen und sortieren

246. Vereinbaren Sie drei Reihungsvariablen namens `r01`, `r02` und `r03` so, dass gilt: Die Reihungen `r01` und `r02` sind *gleich* (aber nicht identisch), und die Reihungen `r02` und `r03` sind sogar *identisch* (und nicht nur gleich). Den (Reihungs-) Typ der Variablen können Sie frei wählen (z. B. `float[]`). (S. 170)
247. Wenn man zwei Reihungsvariablen `r01` und `r02` mit der Gleichheitsoperation `==` vergleicht, etwa so: `r01 == r02`, welche Teile der Variablen werden dadurch verglichen, die *Namen*, die *Referenzen*, die *Werte* oder die *Zielwerte*?
248. Welche Eigenschaft müssen zwei Reihungen `r01` und `r02` haben, damit der Ausdruck `r01 == r02` den Wert `true` hat? Sie müssen äquivalent sein? Sie müssen identisch sein? Sie müssen gleich sein? Sie müssen gleich lang sein?

7.7 Die Klasse Arrays

249. Nennen Sie drei Methoden aus der Klasse `java.util.Arrays`.
250. Nennen Sie zwei Methoden aus der Klasse `java.util.Arrays`, die ausdrücklich zur Bearbeitung von *mehrstufigen* Reihungen geeignet sind.
251. Wieviele verschiedene Methoden namens `equals` gibt es in der Klasse `java.util.Arrays`?
252. Welche Eigenschaft müssen zwei Reihungen `r01` und `r02` haben, damit der Ausdruck `Arrays.equals(r01, r02)` den Wert `true` hat? Sie müssen äquivalent sein? Sie müssen identisch sein? Sie müssen gleich sein? Sie müssen gleich lang sein?
253. Wenn man zwei Reihungsvariablen `r01` und `r02` mit der Methode `Arrays.equals` vergleicht, welche Teile der Variablen werden dadurch verglichen, die *Namen*, die *Referenzen*, die *Werte* oder die *Zielwerte*?
254. Vereinbaren Sie zwei Reihungen namens `r03` und `r04` so, dass der Ausdruck `Arrays.equals(r03, r04)` den Wert `false` und der Ausdruck `Arrays.deepEquals(r03, r04)` den Wert `true` hat.
255. Vereinbaren Sie zwei *zweistufige* Reihungen namens `r05` und `r06` so, dass der Ausdruck `Arrays.equals(r05, r06)` den Wert `true` und der Ausdruck `Arrays.deepEquals(r05, r06)` auch den Wert `true` hat.

7.8 Initialisierungen sind keine Zuweisungen

256. Geben Sie eine *Vereinbarung* und eine *Anweisung* an, die beide mit `= 2.0F * (17.5F * 4.0F);` enden.
257. Warum ist es wichtig, zwischen dem *Initialisierungsteil einer Variablenvereinbarung* und einer *Zuweisung* zu unterscheiden, obwohl sich beide manchmal sehr ähnlich sehen (z. B. bei der vorigen Frage)?
258. Geben Sie einen *Initialisierungsteil* (als Teil einer Variablenvereinbarung) an, zu dem es keinen entsprechenden *Zuweisungsbefehl* gibt.

7.9 Das Kovarianzproblem

259. Wir betrachten bei einem Auto den *Benzinverbrauch pro Stunde* und die *Geschwindigkeit*, mit der das Auto fährt. Sind diese beiden Größen eher *kovariant* oder *kontravariant*? (S. 174)
260. Wir betrachten bei einem Auto den *Füllstand des Tanks* (z. B. 100% voll, 90%, 80%, ..., 0% leer) und die *Strecke*, die das Auto seit dem letzten Tankstopp zurückgelegt hat. Sind diese beiden Größen eher *kovariant* oder *kontravariant*?
261. Wir betrachten das *Alter eines Autos* (gemessen in Minuten, seit es von seinem Besitzer gekauft wurde) und sein *Gewicht* ("mit allem drum und dran"). Sind diese beiden Größen eher *kovariant* oder *kontravariant*?
262. Einen Typ kann man (unter anderem) als eine *Menge von Werten* verstehen. Wann ist bei dieser Auffassung ein Typ `T1` *größer* als ein Typ `T2`?

8 Methoden vereinbaren und aufrufen

263. Was ist (in Java und anderen objektorientierten Programmiersprachen) eine *Methode* und was haben Methoden mit *Unterprogrammen* zu tun? (S. 182)
264. Wo in einem Java-Programm darf man Methoden *vereinbaren*? Unmittelbar am Anfang einer Quelldatei? Innerhalb einer anderen Methode? Innerhalb einer Klasse? Innerhalb von Schleifen? Unmittelbar nach jeder Zuweisung?
265. Was darf/muss der Programmierer mit einer Methode machen? Eigentlich gibt es hier nur zwei zulässige und sinnvolle Tätigkeiten (d. h. Sie sollen davon absehen, dass einige ProgrammiererInnen ihre Methoden auch verkaufen, in Schubladen verstecken, mit Kaffee begießen oder zu Papierfliegern zusammenfalten :-).
266. *Wie oft* darf der Programmierer die beiden (zulässigen und sinnvollen) Tätigkeiten aus der vorigen Frage auf eine Methode anwenden?
267. Es gibt *drei Arten von Befehlen*. Zu welcher Art gehört der folgende Befehl (in Zeile 1 bis 3)? Wie lautet dieser Befehl, wenn man ihn ins Deutsche übersetzt? Was bewirkt dieser Befehl?

```
1 static void printHallo() {
2     System.out.println("Hallo!");
3 }
```

268. Wie kann der Programmierer dem Ausführer befehlen, die Methode `printHallo` (die oben in Zeile 1 bis 3 vereinbart wurde) *auszuführen*? Wie sieht der entsprechende Befehl aus? Um was für eine Art von Befehl (Vereinbarung, Ausdruck oder Anweisung) handelt es sich dabei? Welcher Wertebehälter wird durch diesen Befehl verändert?
269. Zu welcher Art von Befehlen (Vereinbarung, Ausdruck, Anweisung) gehören die folgenden drei Befehle? Wie lauten sie, wenn man sie ins Deutsche übersetzt? Was bewirken sie?

```
1 static void pln(Object ob) {System.out.println(ob);}
2 static void p (Object ob) {System.out.print (ob);}
3 static void pln()          {System.out.println(); }
```

8.1 Parameterübergabe per Wert

270. Das folgende Beispiel dient zur Illustration der nachfolgenden Frage (und der Antwort). Es enthält eine *Methodenvereinbarung* und zwei *Methodenaufrufe*. In Zeile 1 bis 3 wird eine Methode namens `hoch2` vereinbart und in den Zeilen 7 und 8 wird diese Methode zweimal *aufgerufen*:

8.1 Parameterübergabe per Wert

```
1 static public int hoch2(int basis) {
2     return basis * basis;
3 }
4 ...
5 static public void main(String[] sonja) {
6     ...
7     int quad = hoch2(3); // Ein Aufruf der Methode hoch2
8     pln(hoch2(quad+1)); // Noch ein Aufruf der Methode hoch2
9     ...
10 }
```

- In der Vereinbarung der Methode `hoch2` wird in Zeile 1 ein *Parameter* namens `basis` vereinbart. Im Aufruf der Methode `hoch2` in Zeile 7 der Programmierer als *Parameter* 3 angegeben. Im Aufruf der Methode `hoch2` in Zeile 8 hat er als *Parameter* `quad+1` angegeben. Durch welche Bezeichnungen unterscheidet man Parameter in einer *Methodenvereinbarung* (wie `basis` in Zeile 1) und Parameter in *Methodenaufrufen* (wie 3 in Zeile 7 und `quad+1` in Zeile 8)? (S. 184)
271. Mit welchen anderen Befehlen hat die Vereinbarung eines formalen Parameters (z. B. die Vereinbarung des formalen Parameters `basis` oben in Zeile 1) große Ähnlichkeit? (S. 185)
272. Die formalen Parameter einer Methode sind also spezielle Variablen. *Wann* werden diese Variablen vom Ausführer *erzeugt*? Wenn der Programmierer das betreffende Programm dem Ausführer übergibt? Wenn der Ausführer das Programm akzeptiert? Wenn der Benutzer das Programm ausführen lässt? Jedesmal, wenn der Ausführer einen Aufruf der betreffenden Methode ausführt? Jeden Freitag bei Sonnenaufgang? Sonntags nie?
273. Bis wann leben die formalen Parameter einer Methode? Bis der Ausführer das umgebende Programm fertig ausgeführt hat? Bis der Ausführer nicht mehr genug Speicher hat? Bis der Ausführer den betreffenden Methodenaufruf fertig ausgeführt hat? Bis zum nächsten Sonnenuntergang?
274. Wenn der Ausführer einen Methodenaufruf ausführt (z. B. den Aufruf der Methode `hoch2` oben in Zeile 7, oder den Aufruf der Methode `hoch2` oben in Zeile 8), erzeugt er die formalen Parameter als Variablen und *initialisiert sie*. *Womit* initialisiert er sie? Je nach Typ mit 0 bzw. 0.0 bzw. `false` etc.? Immer mit dem kleinsten Wert des betreffenden Typs? Mit dem Wert des entsprechenden aktuellen Parameters? Mit einem beliebigen Wert, den der Ausführer je nach Laune auswählt?
275. Die Methode `hoch2` hat einen *formalen Parameter* vom Typ `int`. Deshalb muss man in jedem Aufruf der Methode `hoch2` einen aktuellen Parameter vom Typ `int` angeben. Was darf man als aktuellen Parameter alles angeben? Nur Literale vom Typ `int` wie 17 oder 123? Nur Namen von `int`-Variablen wie

- quad? Beliebige Ausdrücke des Typs `int` wie `1+quad` oder `2*quad+123`? Nur Strings, die der Ausführer in einen passenden Wert umwandeln kann?
276. Betrachten Sie noch einmal die Zeile 7 im obigen Beispiel: Mit welchem Wert wird die Variablen `quad` dort initialisiert?
277. Betrachten Sie noch einmal die Zeile 8 im obigen Beispiel: Welcher Wert wird dort durch den `println`-Befehl zum Bildschirm ausgegeben?
278. Welchen Vorteil haben *Methoden mit Parametern* im Vergleich zu *Methoden ohne Parameter*? Was passiert, wenn man eine Methode ohne Parameter *mehrmals* ausführen läßt? Was passiert, wenn man eine Methode mit Parameter *mehrmals* mit unterschiedlichen aktuellen Parametern ausführen läßt?
279. In Java werden die formalen Parameter einer Methode bei jedem Aufruf neu erzeugt und mit den *Werten* der aktuellen Parameter initialisiert. Wie nennt man diese Art der Parameterübergabe?
280. Wie heißt eine andere wichtige Art der Parameterübergabe (die es auch in Java gibt bzw. nicht gibt, je nachdem, "wie man die Dinge sehen möchte")? Wie funktioniert diese andere Art der Parameterübergabe? Welche Dinge sind als aktuelle Parameter erlaubt? Was von diesen Dingen wird dann an die betreffende Methode übergeben?
281. Können Sie einen Methodenaufruf mit *einem* Parameter angeben, bei dem man sich wahlweise vorstellen kann, dass der Parameter *per Wert* oder dass er *per Referenz* übergeben wird?
282. Wenn man davon ausgeht, dass es in Java die beiden Parameterübergabearten *per Wert* und *per Referenz* gibt, welche Parameter werden dann immer *per Wert* und welche werden immer *per Referenz* übergeben? (S. 189)
283. Wenn man davon ausgeht, dass es in Java die beiden Parameterübergabearten *per Wert* und *per Referenz* gibt, welche Parameter kann man dann nicht *per Wert* und welche kann man nicht *per Referenz* übergeben?

8.2 Prozeduren und Funktionen

284. Wozu dienen Funktionen? (S. 190)
285. Wozu dienen Prozeduren?
286. Was für eine Art von Befehl ist ein Funktionsaufruf?
287. Was für eine Art von Befehl ist ein Prozeduraufruf?
288. Wodurch kann man besonders leicht eine *Funktionsvereinbarung* von einer *Prozedurvereinbarung* unterscheiden?
289. Eine Ausführung einer Prozedur kann entweder durch einen `return`-Befehl oder *normal* beendet werden. Was ist hier mit "normal beendet werden" gemeint?

290. Wie darf eine Funktionsausführung *nie* beendet werden? Wie werden die meisten Funktionsausführungen beendet? Wodurch werden einige Funktionsausführungen abgebrochen?
291. Jede Methodenvereinbarung darf viele `return`-Anweisungen enthalten. Aber wieviele `return`-Anweisungen muss eine *Funktionsvereinbarung* (bzw. eine *Prozedurvereinbarung*) *mindestens* enthalten?
292. Wodurch unterscheidet sich eine `return`-Anweisung in einer Funktion von einer `return`-Anweisung in einer Prozedur? (S. 193)
293. Wodurch unterscheiden sich *funktionale* Programmiersprachen von *prozeduralen* Programmiersprachen? Welche Grundkonzepte und Befehle gibt es nur in Sprachen der einen bzw. der anderen Art?

8.3 Methodennamen überladen

294. Welche Angaben gehören zur *Signatur* einer Methode?
295. Welche Angaben (von denen man leicht denken könnte, dass sie auch zur Signatur einer Methode gehören) gehören *nicht* zur Signatur einer Methode?
296. Welche Signatur hat die folgende Methode:
- ```
1 static public void gibAus(boolean b, float f1, float f2) {...}
```
297. Welche Dinge kann man in Java *überladen*? Operatoren? Methoden? Den Programmierer? Die Namen von Methoden? Den Frühstückstisch des Programmierers?
298. Innerhalb einer Klasse darf man beliebig viele Methoden mit gleichen Namen vereinbaren, aber nur, wenn diese Methoden eine Bedingung erfüllen. Welche Bedingung?

## 8.4 Methoden mit variabel vielen Parametern

299. Wieviele aktuelle Parameter vom Typ `int` darf/muss man angeben, wenn man die folgende Methode aufruft (wurzieren ist eine neue Grundrechenart, die sich noch in Entwicklung befindet :-):
- ```
1 static public int wurziere(int... fir) {
2     println("Wieso ausgerechnet " + fir.length + " Parameter?");
3     return fir.length;
4 }
```
300. Wieviele aktuelle Parameter darf/muss man angeben, wenn man die folgende Methode aufruft (performieren ist eine alte Rechenmethode, die bald abgeschafft werden soll :-):

```

1  static public double performiere(boolean b, int... fir) {
2      if (b) {
3          return fir.length + 42.0;
4      } else {
5          return 999.9;
6      }
7  }

```

301. Was ist an der folgenden Methodenvereinbarung falsch?

```

8  static public int finiere(int... f1, boolean b, int... f2) {
9      return 17;
10 }

```

8.5 Kommentare für Methoden

302. Für wen schreibt der Programmierer *Kommentare* in seine Programme und wer ignoriert Kommentare weitgehend? (S. 197)
303. Wo sollte bei fast jeder Methode ein Kommentar stehen und was sollte er beschreiben?
304. *Wo* genau (in welcher Zeile) sollte der *Anfangskommentar* einer Methode eigentlich stehen und warum?
305. Unter welchen Umständen kann man den Anfangskommentar einer Methode *nicht* da hinschreiben, wo er hingehört?
306. Ganz allgemein und abstrakt: *Wo* sollte man Kommentare einfügen und was für Kommentare sollte man einfügen? (S. 199)
307. Wie haben wir eine *populäre Art von Kommentaren* genannt, die man möglichst vermeiden sollte?

9 Klassen

308. Was ist ein *Modul*? (S. 201)

309. Was ist ein *Typ*? Geben Sie möglichst nur die einfachere und kürzere Definition (aus dem Abschnitt 1.4.2 des Buches) an.

310. Was ist eine *Klasse*? (S. 201, 210)

9.1 Eine kleine Beispielklasse

311. Woran kann man (innerhalb einer Klassenvereinbarung) einen *Konstruktor* erkennen? (S. 203)

312. Übersetzen Sie den folgenden Befehl ins Deutsche:

```
1  class Zaehler01 { ... }
```

313. Wenn der Ausführer eine Klasse namens *Zaehler01* erzeugt, erzeugt er nur einen *Modul* namens *Zaehler01*. Welche *Elemente* der Klasse enthält dieser Modul? Alle Elemente der Klasse? Nur die mit *static* gekennzeichneten Elemente? Nur die *nicht* mit *static* gekennzeichneten Elemente? Nur die mit *static* gekennzeichneten Elemente und die Konstruktoren? Keine Elemente?

314. Wenn der Ausführer ein *Objekt* der Klasse *Zaehler01* erzeugt, welche Elemente der Klasse baut er dann in dieses Objekt ein? Alle Elemente der Klasse? Nur die mit *static* gekennzeichneten Elemente? Nur die *nicht* mit *static* gekennzeichneten Elemente? Nur die mit *static* gekennzeichneten Elemente und die Konstruktoren? Keine Elemente?

315. Womit befiehlt man dem Ausführer normalerweise, ein Objekt zu erzeugen? Mit einer *break*-Anweisung? Mit einem Konstruktor? Mit dem Operator *new*? Mit der Funktion *Math.sin*? (S. 205)

316. Mit welchem Wert initialisiert die *new*-Operation jede *int*-Variable in einem neu erzeugten Objekt standardmäßig, wenn der Programmierer keine andere Initialisierung angegeben hat? Wie initialisiert der *new*-Befehl *boolean*-Variablen im neu erzeugten Objekt?

317. Wo in einem Java-Programm muss man immer einen Konstruktor angeben und was ist die Aufgabe eines Konstruktors? (S. 207)

9.2 Eine Klasse mit Attributen eines Referenztyps

318. Angenommen, ein Objekt namens *anna* (siehe S. 121 im Buch) enthält unter anderem eine Variable (oder: ein Attribut) vom Typ *String*. Diese *String*-

Variable besteht aus einer *Referenz*, einem *Wert* und einem *Zielwert*. Wieviele und welche dieser Teile liegen *innerhalb* des Objekts *anna* und wieviele und welche dieser Teile liegen *ausserhalb* des Objekts *anna*?

Anmerkung: Variablen innerhalb eines Objekts haben (für Betrachter, die von ausserhalb in das Objekt hineinschauen) *keine einfachen Namen* und können (von ausserhalb) nur mit *zusammengesetzten Namen* wie *anna.vorName* (die Variable *vorName* im Objekt *anna*) oder *anna.nachName* (die Variable *nachName* im Objekt *anna*) bezeichnet werden. Solche zusammengesetzte Namen zeichnen wir in Bojen *nicht* in Paddelboote, sondern schreiben sie nur in die Nähe der anderen Teile der Variable.

9.3 Extreme Klassen

- 319. Jede Klasse hat einen *Modulaspekt* und einen *Bauplanaspekt*. Welche Elemente der Klasse gehören zum Modulaspekt, und welche gehören zum Bauplanaspekt? (S. 213)
- 320. Jede Klasse hat einen *Modulaspekt* und einen *Bauplanaspekt*. Warum können beide Aspekte nie ganz leer, sondern nur *fast leer* sein? Die Behauptung im Buch auf S. 213, dass der Modulaspekt *ganz leer* sein könne, ist leider falsch, sorry.
- 321. Wann bekommt eine Klasse vom Ausführer einen Konstruktor geschenkt (sozusagen als "Sozialhilfe")?
- 322. Was versteht man unter einem Standardkonstruktor?
- 323. Von wem bekommt eine Klasse einen Standardkonstruktor?

9.4 Klassische Fachbegriffe

- 324. Was für Dinge kann man innerhalb einer Klasse vereinbaren? Nennen Sie möglichst die allgemeinsten und abstraktesten Begriffe. (S. 216)
- 325. Die Elemente einer Klasse haben wir auf drei verschiedene Weisen in Gruppen eingeteilt, oder nach drei verschiedenen Kriterien. Wie heißen diese Kriterien?
- 326. Welche Gruppen ergeben sich, wenn man die Elemente einer Klasse nach ihrer *Art* unterscheidet?
- 327. Welche Gruppen ergeben sich, wenn man die Elemente einer Klasse nach ihrer *Apektzugehörigkeit* unterscheidet?
- 328. Welche Gruppen ergeben sich, wenn man die Elemente einer Klasse nach ihrer *Erreichbarkeit* unterscheidet?

9.5 Objektorientierte Programmierung

- 329. Angenommen, wir wollen ein Verwaltungsprogramm für eine Hochschule schreiben. Was versuchen wir (als objektorientierte ProgrammiererInnen) als erstes herauszufinden?
- 330. Angenommen, wir wollen ein Verwaltungsprogramm für eine Hochschule schreiben und haben herausgefunden, dass an einer Hochschule vor allem StudentInnen, ProfessorInnen, Räume, Lehrveranstaltungen, Notenlisten, Abschlusszeugnisse etc. wichtig sind. Was machen wir dann?

10 Ein paar Standardklassen und Methoden

- 331. Ungefähr wieviele Klassen und Schnittstellen gehörten am 30. Juni 2005 um 23 Uhr 59 zur Standardbibliothek von Java 5? Die genaue Zeitangabe soll verdeutlichen, dass die Standardbibliothek explodiert und spätestens nach ein paar Stunden wieder größer geworden ist.
- 332. Wodurch unterscheiden sich Objekte der Klassen *String* und *StringBuilder*? Wann sollte man Objekte der einen bzw. der anderen Art verwenden? (S. 221)

10.1 Die Klasse *String*

- 333. Sei *s* eine *String*-Variable mit dem Zielwert "ABCDEF". Welchen Wert hat dann der Ausdruck *s.charAt(0)*? Und der Ausdruck *charAt(5)*? Und der Ausdruck *s.length()*?
- 334. Wieviel *char*-Werte und wieviel Zeichen enthält ein *String*-Objekt der Länge 10? (S. 222)
- 335. Sei *s* eine *String*-Variable mit dem Zielwert "ABCDEF". Welchen Zielwert hat dann der Ausdruck *s.substring(2)*? Und welchen Zielwert hat der Ausdruck *s.substring(3, 4)*? Und der Ausdruck *s.substring(3,3)*? (S. 223)
- 336. Sei *s* eine *String*-Variable mit dem Zielwert "ABCDEF". Welchen Wert hat dann der Ausdruck *s.startsWith("ABC")*? Und die Ausdrücke *s.startsWith("A")*, *s.startsWith("BC")*?, *s.endsWith("DEF")*, *s.endsWith("DF")*?
- 337. Sei *s* eine *String*-Variable mit dem Zielwert "ABCABCABC". Welche Werte haben dann die Ausdrücke *s.indexOf('B')*, *s.indexOf("BC")*, *s.la-*

`stIndexOf('B'), s.lastIndexOf("BC"), s.indexOf('A', 1)` und `s.lastIndexOf("BC", 6), s.indexOf('X'), s.lastIndex("BA")`?

338. Wenn man zwei `String`-Variablen `s11` und `s12` vergleichen will um festzustellen, ob der Zielwert von `s11` kleiner, gleich oder größer als der Zielwert von `s12` ist, darf man dazu nicht die Vergleichsoperationen `<` oder `<=` etc. verwenden, sondern muss die Funktion `compareTo` benutzen (z. B. so: `s1.compareTo(s2)`). Welchen *Ergebnistyp* (oder: Rückgabetypp) hat die Funktion `compareTo`? (S. 227)
339. Warum ist es vernünftig, dass man `String`-Objekte nicht mit den Vergleichsoperationen `<` oder `<=` etc. vergleichen darf, sondern dazu die Funktion `compareTo` verwenden muss?

10.2 Objekte mit Hilfe von Bojen genauer verstehen

340. Wieviele öffentliche *Methoden* enthält jedes `String`-Objekt? (S. 228)
341. Wieviele öffentliche *Attribute* enthält jedes `String`-Objekt?
342. Jedes Reihungsobjekt enthält ein Element namens `length`. Von welcher *Art* ist dieses Element, *Attribut* oder *Methode*?
343. Jedes `String`-Objekt enthält ein Element namens `length`. Von welcher *Art* ist dieses Element, *Attribut* oder *Methode*?
344. In der Bojendarstellung einer `String`-Variablen ist der Zielwert-Kasten das eigentliche `String`-Objekt. Aus praktischen Gründen stellen wir meist nicht alle Elemente dieses Objekts dar. Welche Elemente lassen wir meistens weg? Welches Element zeichnen wir meistens als einziges ein?
345. Die Zeichenkette eines `String`-Objekts steht in einem privaten Attribut und wir können nicht direkt darauf zugreifen. Warum sind `String`-Objekte trotzdem nicht völlig nutzlos für uns?
346. Was ist ein `String`-Literal wie "Hallo!" in Java? Eine Methode, die ein `String`-Objekt liefert? Eine Reihung von `char`-Werten? Der Name einer unveränderbaren `String`-Variablen? Der Wert einer veränderbaren `String`-Builder-Variablen? (S. 229)
347. Angenommen, drei `String`-Variablen wurden wie folgt vereinbart:

```
1 String anna = "ABC";
2 String bert = "ABC";
3 String carl = new String("ABC");
```

Was wissen wir dann über die *Werte* dieser Variablen? (S. 230)

10.3 Die Klasse `StringBuilder`

348. Wieviele öffentliche Methoden enthält jedes `StringBuilder`-Objekt? Und wo steht die Zeichenkette, die von dem Objekt repräsentiert wird? (S. 233)
349. Die Zeichenkette eines `StringBuilder`-Objekts steht in einem privaten Attribut und wir können nicht direkt darauf zugreifen. Warum sind `StringBuilder`-Objekte trotzdem nicht völlig nutzlos für uns?
350. Von welchem Typ ist das private Attribut eines `StringBuilder`-Objekts, in dem die Zeichenkette des Objekts steht, vermutlich? Und wie bezeichnen wir dieses private Attribut?
351. Was versteht man unter der *Länge* eines `StringBuilder`-Objekts? Und was versteht man unter seiner *Kapazität*?
352. Sei `sb` ein ziemlich langes `StringBuilder`-Objekt. Welcher der beiden folgenden Befehle braucht dann *mehr* und welcher *weniger* Zeit?
- ```
sb.insert(0, "AB");
sb.insert(sb.length(), "ABCDEFGHJKLMNOPQRSTUVWXYZ");
```
353. Welcher andere und kürzere Befehl leistet das Gleiche wie der Befehl `sb.insert(sb.length(), "XYZ");`?

## 10.4 Die Klasse `ArrayList`

354. Was ist eine *Sammlung* (engl. collection)? (S. 237)
355. Wie bezeichnen wir die Objekte in einer Sammlung? Als gesammelte Objekte? Als Elemente der Sammlung? Als Inhalte der Sammlung? Als Komponenten der Sammlung? Als Antiquitäten?
356. *Wieviele Typen* repräsentiert eine *normale* Klasse wie `String`, `StringBuilder` oder `Object`? *Wieviele Typen* repräsentiert eine *generische* Klasse wie `ArrayList<K>`?
357. Wenn Ihnen beim Besuch einer Kneipe ein ziemlich roh aussehender Typ begegnet, wie sollten Sie sich dann verhalten?
358. Wenn Ihnen beim Schreiben von Java-Programmen ein roher Typ begegnet, wie sollten Sie sich dann verhalten?
359. Geben Sie die Namen von drei *p-ArrayList-Typen* (parametrisierten `ArrayList`-Typen) an und beschreiben Sie kurz, was für Objekte man in Sammlungen dieser Typen sammeln kann.
360. Wie heißt die Klasse `ArrayList` mit vollem Namen? Und hat sie einen Adelstitel?
361. Sei `samma` ein Objekt des Typs `ArrayList<String>`. Beschreiben Sie kurz, was die Anweisung `samma.add("Hallo!");` bewirkt.

362. Sei `samma` eine Objekt des Typs `ArrayList<String>`. Beschreiben Sie kurz, was die Anweisung `samma.add(samma.size(), "Hallo!");` bewirkt.
363. Sei `samma` eine Objekt des Typs `ArrayList<String>`, welches bereits *fünf* *String-Objekte* enthält. Wieviele Objekte enthält die Sammlung dann nach Ausführung der folgenden beiden Anweisungen:
- ```
samma.add(3, "Hallo!");
samma.remove("Hallo!");
```
364. Nennen Sie einen Vorteil, den Sammlungen eines `ArrayList`-Typs im Vergleich zu entsprechenden *Reihungen* haben.
365. Skizzieren Sie einen Fall, in dem man eher eine *Reihung* verwenden würde als eine Sammlung (eines `ArrayList`-Typs).

10.5 Die Klasse Random und der Zufall

366. Nennen Sie zwei typische Anwendungsgebiete für *Zufallszahlen*. (S. 245)
367. Wann nennt man eine innerhalb eines Programms *P* erzeugte Folge von Zufallszahlen *reproduzierbar*? Wann nennt man die Folge *nicht-reproduzierbar*?
368. Nennen Sie eine Anwendung, bei der man in der Regel *reproduzierbare* Zufallszahlen verwendet und eine Anwendung, bei der man *nicht-reproduzierbare* Zufallszahlen bevorzugt.
369. Im folgenden werden zwei Objekte erzeugt, die beide als Quellen von Zufallszahlen geeignet sind:
- ```
1 Random quelle01 = new Random(7381);
2 Random quelle02 = new Random();
```

Welches der beiden Objekte ist eine Quelle von *nicht-reproduzierbaren* Zufallszahlen und welches liefert *reproduzierbare* Zufallszahlen?

370. Wie kann man aus der `quelle01` einen zufälligen `int`-Wert schöpfen?
371. Wie kann man aus der `quelle02` einen zufälligen `boolean`-Wert schöpfen?
372. Was für eine Zufallszahl liefert der Aufruf `quelle01.nextInt(50)`?

## 10.6 Die Klassen BigInteger und BigDecimal

373. Wie groß sind die Zahlen `Long.MIN_VALUE` und `Long.MAX_VALUE` (die kleinste bzw. größte Zahl des Typs `long`) ungefähr? Aus wieviel Dezimalziffern bestehen diese Zahlen?
374. Beim Verschlüsseln von Daten muss man manchmal mit Ganzzahlen rechnen, die wesentlich größer sind als die größten `long`-Werte (z. B. mit Ganzzahlen, die in dezimaler Darstellung aus 150 oder aus 600 Ziffern bestehen). Was

- spricht dagegen, solche Ganzzahlen durch Werte des Typs `double` darzustellen? Wie kann man solche Ganzzahlen besser darstellen?
375. Was sind die Vor- und Nachteile des Typs `BigInteger` im Vergleich zum primitiven Typ `long`?
376. Was sind die Vor- und Nachteile des Typs `BigDecimal` im Vergleich zum primitiven Typ `double`?
377. Gibt es zu jedem endlichen Dezimalbruch (z. B. 123.4567) einen endlichen Binärbruch mit exakt demselben Wert?
378. Gibt es zu jedem endlichen Binärbruch (z. B. 101101.1101) einen endlichen Dezimalbruch mit exakt demselben Wert?

## 10.7 Die Klasse Formatter und die print-Methode

379. Wie heißen die zwei wichtigste Methode eines `Formatter`-Objekts?
380. Welche beiden Methoden in einem `PrintStream`-Objekt (z. B. im `PrintStream`-Objekt namens `System.out`) leisten ganz Ähnliches wie die Methoden namens `format` in einem `Formatter`-Objekt?
381. Was gibt der Befehl `System.out.printf("Nr. %0+4dXYZ%n", 5)` zum Bildschirm aus?

## 11 Struktur und Ausführung eines Java-Programms

382. Ein Java-Programm besteht aus *Klassen*. Welche Klassen gehören zu einem Java-Programm namens *Abrechnung*?
383. Wann wird die *Hauptklasse* eines Java-Programms *erzeugt*?
384. Wann wird eine *Nebenklasse* eines Java-Programms *erzeugt*?
385. Welche Klassen eines Java-Programms *müssen* eine *main*-Methode enthalten? Welche *dürfen* eine *main*-Methode enthalten?

## 12 Klassen erweitern (beerben) und Typgraphen

386. Wenn man eine Variante einer schon vorhandenen Klasse `K` braucht, könnte man `K` mit einem Editor *kopieren* und die Kopie `K1` dann ändern (d.h. an die neuen Bedürfnisse anpassen). Warum ist das *keine* gute Idee?
387. Was bedeutet es, eine neue Klasse `K1` als eine *Erweiterung* einer schon vorhandenen Klasse `K` zu vereinbaren?

### 12.1 Ein kleines Beispiel für Beerbung

388. Im Beispiel-02 des Abschnitts 12.1 (auf S. 280 des Buches) wird eine neue Klasse `Person02` als *Erweiterung* einer alten Klasse `Person01` vereinbart (man beschreibt das auch so: Die neue Klasse `Person02` *beerbt* die alte Klasse `Person01`). Wieviele Elemente erbt die Klasse `Person02` von `Person01` und wie heißen diese Elemente? (S. 278, 280)
389. Mit dem `new`-Operator kann der Programmierer (unter anderem) neue `Person02`-Objekte erzeugen lassen. Wieviele werden in jedes solche Objekt eingebaut und wie heißen diese Elemente?
390. Wieviele Elemente enthält der Modul `Person02` und wie heißen sie?
391. In der Vereinbarung der Klasse `Person01` (siehe S. 278 des Buches) ist *nicht* angegeben, dass diese Klasse eine andere erweitert (nach dem Klassennamen `Person01` steht *nicht* `extends ...`). Trotzdem erweitert `Person01` eine andere Klasse. Welche?
392. Wieviele *Objektelemente* sind in der Klasse `Object` vereinbart? Sehen Sie in Ihrer Lieblingsdokumentation der Java-Standardbibliothek nach. Wieviele *Klassenelemente* (d.h. mit `static` gekennzeichnete Elemente) sind in der Klasse `Object` vereinbart?

### 12.2 Allgemeine Regeln zum Beerben und Vererben

393. Wieviele alte Klassen darf eine neue Klasse *beerben*? Entweder null oder eine? Genau eine? Mindestens eine? Höchstens 3? (S. 283)
394. An wieviele neue Klassen darf eine alte Klasse ihre Elemente *vererben*? An genau eine? An null oder eine? An beliebig viele?
395. Wenn man eine neue Klasse `K` vereinbart, darf man dabei fast jede Klasse beerben, mit einer Ausnahme. Welche Klasse darf man *nicht* beerben?

396. Die *erweitert*-Relation kann man sich so vorstellen: Wenn die Klasse `K2` eine Erweiterung der Klasse `K1` ist, führt ein Pfeil von `K2` nach `K1`. Was für eine Struktur bilden alle Java-Klassen zusammen mit dieser *erweitert*-Relation?
397. Sei `K2` eine Erweiterung (oder: Unterklasse) der Klasse `K1`. Welche der beiden Regeln gilt dann:  
 R1: Dann ist jedes `K1`-Objekt auch ein `K2`-Objekt.  
 R2: Dann ist jedes `K2`-Objekt auch ein `K1`-Objekt. (S. 286)

### 12.3 Ein größeres Beispiel für Beerbung

398. Wenn man ein Objekt als Zwiebel darstellt (d.h. als ein System von ineinander enthaltenen Objekten), was für ein Objekt bildet dann immer den innersten *Kern* der Zwiebel? (S. 291)

### 12.4 Wozu Untertypen gut sind

399. In einer Reihung vom Typ `E01Punkt[]` (d.h. in eine Reihung von `E01Punkt`-Variablen) kann man eigentlich nur `E01Punkt`-Objekte speichern. Wieso (auf Grund welcher Regel) kann man in einer solchen Reihung trotzdem auch `E01Ellipse`-Objekte und `E01Rechteck`-Objekte speichern? (S. 295)
400. Was ist an einer Reihung vom Typ `E01Punkt[]` (mit Objekten der Klassen `E01Ellipse`, `E01Rechteck` etc. darin) so interessant und nützlich?

### 12.5 Kleinere und größere Klassen

401. Wenn man das Gefühl hat, eine Oberklasse sei *größer* als eine Unterklasse, welches Maß (für "die Größe" einer Klasse) kann man dann wählen, um dieses Gefühl zu bestätigen? (S. 300)
402. Wenn man das Gefühl hat, eine Unterklasse sei *größer* als eine Oberklasse, welches Maß (für "die Größe" einer Klasse) kann man dann wählen, um dieses Gefühl zu bestätigen?
403. Welches Maß für die Größe einer Klasse verwenden wir normalerweise?
404. Wo zeichnen wir somit in einem Typgraphen die *größeren* Klassen ein, weiter *oben* oder weiter *unten*?
405. Wann sagen wir von zwei Objekten `ob1` und `ob2`, das Objekt `ob2` sei größer als `ob1`? Hier ist nicht die Größe entsprechend einer Vergleichsfunktion gemeint, sondern eine, die mit der Größe von Klassen zu tun hat.

406. Wenn eine Referenzvariable auf Objekte einer bestimmten Klasse zeigen darf, darf sie dann auch auf *größere Objekte* oder auf *kleinere Objekte* zeigen?

## 12.6 Konstruktoren und Unterklassen

407. Wieviele Konstruktoren muss/darf der Programmierer innerhalb einer Klasse vereinbaren? Er muss mindestens einen und darf ansonsten beliebig viele vereinbaren? Er muss mindestens einen und darf höchstens fünf vereinbaren? Er muss keinen und darf höchstens fünf vereinbaren? Er muss keinen, sondern darf beliebig viele vereinbaren? (S. 302)
408. Wieviele Konstruktoren enthält jede Klasse mindestens? Mindestens null? Mindestens einen? Mindestens fünf?
409. Wenn eine Klasse einen Standardkonstruktor enthält, von wem stammt der dann? Immer vom Programmierer (der ihn vereinbart hat)? Immer vom Ausführer (der ihn der Klasse "geschenkt" hat)? Manchmal vom Programmierer und manchmal vom Ausführer?
410. Hat ein Standardkonstruktor immer einen leeren Rumpf?
411. Was bewirkt der erste Befehl im Rumpf eines Konstruktors immer? Dass eine Erfolgsmeldung zur Standardausgabe ausgegeben wird? Dass alle Attribute des neuen Objekts standarmäßig (d.h. mit 0, 0.0, false bzw. null) initialisiert werden? Dass ein Konstruktor der direkten Oberklasse aufgerufen wird? Der erste Befehl im Rumpf eines Konstruktors hat keine festgelegte Wirkung?
412. Welches *Schlüsselwort* muss der Programmierer verwenden, wenn er in einem Konstruktor einen *Konstruktor der direkten Oberklasse* aufrufen will?
413. Welches *Schlüsselwort* muss der Programmierer verwenden, wenn er in einem Konstruktor einen *anderen Konstruktor derselben Klasse* aufrufen will?

## 12.7 Der Typ und der Zieltyp einer Referenzvariablen

414. Auf Objekte *welcher Klassen* darf eine Variable vom Typ `E01Punkt` zeigen (wenn man den auf S. 301 des Buches abgebildeten Typgraphen zu Grunde legt)? (S. 303)
415. Welche *Zieltypen* kann eine Variable des Typs `E01Punkt` haben?
416. Geben Sie einen Befehl an, mit dem man den *Zieltyp* einer Referenzvariablen verändern kann.
417. Sei `p02` eine Variable des Typs `E01Punkt`. Dann kann der (zusammengesetzte) Methodenname `p02.toString` verschiedene Methoden bezeichnen. Wovon hängt es ab, welche Methode dieser Name tatsächlich bezeichnet? Wann bezeichnet der Name keine Methode?

## 12.8 Referenzen umdeuten (Cast-Befehle)

418. Sei `d01` eine Variable des primitiven Typs `double`, die momentan den Wert 7.8 enthält. Wie sieht ein Cast-Befehl aus, der aus dem Wert von `d01` einen entsprechenden `int`-Wert erzeugt? Welchen `int`-Wert erzeugt der Cast-Befehl?
419. Sei `s01` eine Variable des primitiven Typs `short`, die momentan den Wert -1 enthält. Wie sieht dieser `short`-Wert in binärer Darstellung aus? Welchen Wert erzeugt der Cast-Befehl `(char) s01` aus dem Wert von `s01`? Wie sieht dieser `char`-Wert in binärer Darstellung aus?
420. Zwischen welchen *primitiven* Typen sind Typumwandlungen mit Cast-Befehlen *erlaubt*? (S. 305)
421. Zwischen welchen *primitiven* Typen sind Typumwandlungen mit Cast-Befehlen *nicht* erlaubt?
422. Typumwandlungen zwischen *primitiven Typen* und *Referenztypen* sind (erst seit Java 5 und) nur in relativ *wenigen Fällen* erlaubt. In welchen Fällen (d.h. zwischen *welchen* Typen)?

**Hinweis:** Die Beschreibung aller erlaubten Cast-Befehle durch die **Cast-Regel-01** (im Buch auf S. 305) ist *unvollständig*, da sie die in der vorigen Frage erwähnten Umwandlungen nicht erwähnt (die **Cast-Regel-01** war bis Java 1.4 gültig, wurde aber mit Java 5 unvollständig).

423. Eine Typumwandlung mit Cast-Befehl zwischen zwei Referenztypen *Quelle* und *Ziel* ist nur in bestimmten Fällen erlaubt. In welchen Fällen?
424. Ein Cast-Befehl wie z. B. `(int) 3.8` erzeugt aus dem `double`-Wert 3.8 den `int`-Wert 3. Das entspricht noch einigermaßen der üblichen Bedeutung des Wortes "Umwandlung" (der Wert 3.8 wird in den Wert 3 *umgewandelt*). Cast-Befehle zwischen Referenztypen entsprechen kaum der üblichen Bedeutung von "Umwandlung". Mit welchem Begriff wird im Buch die Wirkung solcher Cast-Befehle beschrieben?
425. In einem Typgraphen stehen *Obertypen* weiter *oben* als ihre Untertypen. Welche Cast-Befehle zwischen Referenztypen sind harmlos, die *von oben nach unten* oder die *von unten nach oben*?
426. Sind Cast-Befehle von oben nach unten harmlos oder können sie eine Ausnahme auslösen?
427. Was bewirkt der Cast-Befehl `(StringBuilder) "Hallo!"`?

## 12.9 Geerbte Elemente ersetzen

428. In welcher Situation kann der Programmierer ein Element einer Klasse *ersetzen*? Was muss er in einer solchen Situation tun, um das Element zu ersetzen?
429. Welche Angaben gehören zur *Signatur* einer Methode?
430. Welche Angaben (von denen man leicht denken könnte, dass sie auch zur Signatur einer Methode gehören) gehören *nicht* zur Signatur einer Methode?
431. Woraus besteht das *Profil* einer Methode?
432. Wann sind zwei *Attribute* ("Variablen, die in einer Klasse vereinbart wurden") *homonym* ("bezeichnungsgleich")?
433. Wann sind zwei *Methoden* ("Unterprogramme, die in einer Klasse vereinbart wurden") *homonym* ("bezeichnungsgleich")?
434. Welche der folgenden Methoden sind *homonym* bzw. *nicht homonym*?
- ```

1 static public float tuWas(int i, int j, String s) {...}
2 private float tuWas(int anz, int preis, String t) {...}
3 static public short tuWas(int xxx, int yyyy, String s) {...}

```
435. Es ist nicht erlaubt, in einer Klasse zwei homonyme Elemente zu *vereinbaren*. Wie kann es trotzdem passieren, dass zwei homonyme Elemente zu einer Klasse *gehören*?
436. Was für Elemente einer Klasse werden häufiger ersetzt, *Klassenelemente* oder *Objektelemente*?
437. Es gibt zwei *Arten von ersetzen*. Mit welchen Verben bezeichnet man diese Ersetzungsarten auf Deutsch und auf Englisch?
438. Wovon hängt es ab, ob ein Element e1 von einem Element e2 *verdeckt* oder *überschrieben* wird? Von der Absicht des Programmierers? Von der Art der Elemente (Methode oder Attribut)? Von der Aspektzugehörigkeit der Elemente (Klassenelement oder Objektelemente)? Von der Erreichbarkeit der Elemente (public, protected, paketweit sichtbar oder private)? Von der Laune des Ausführers?
439. Wird eine Objektmethode m1 von einer homonymen Objektmethode m2 *verdeckt* oder *überschrieben*?
440. Wird ein Objektattribut a1 von einem homonymen Objektattribut a2 *verdeckt* oder *überschrieben*?
441. Welche Tätigkeit ist für den *Unter-Programmierer* charakteristisch? Und welche Tätigkeit ist für den *Unter-Verwender* charakteristisch?
442. Auf welche Elemente der Klasse ober kann der *Unter-Programmierer* zugreifen, auch wenn er sie gerade *ersetzt* hat? Nur auf *verdeckte* Elemente? Nur auf *überschriebene* Elemente? Auf *beide* Arten von Elementen?

443. Auf welche ersetzten Elemente in seinen *Unter-Objekten* kann der *Unter-Verwender* zugreifen? Nur auf *verdeckte* Elemente? Nur auf *überschriebene* Elemente? Auf *beide* Arten von Elementen? (S. 313)
444. Was passiert, wenn man versucht, eine geerbte Objektmethode m1 mit einer neuen Objektmethode m2 zu überschreiben, so dass m1 und m2 zwar *gleiche Signaturen*, aber *unterschiedliche Ergebnistypen* haben, etwa so:
- ```

1 int add(int n1, int n2) {...} // Geerbte Objektmethode m1
2 long add(int n1, int n2) {...} // Neue Methode m2

```
- In diesem Beispiel haben beide Methoden die Signatur `add int int`, aber unterschiedliche Ergebnistypen (`int` bzw. `long`).
445. Darf man ein *public-Element* durch ein *private-Element* ersetzen? (S. 316)
446. Selbst wenn eine geerbte Methode m1 und eine neue Methode m2 gleiche Profile und die gleiche Erreichbarkeit haben, ist es in einigen Fällen verboten, m1 mit m2 zu ersetzen. In welchen Fällen? (S. 316)
447. Die Verben "überladen" und "überschreiben" klingen ziemlich ähnlich, haben aber ganz verschiedene Bedeutungen. Was für Dinge kann man *überladen*, und was für Dinge kann man *überschreiben*? Wie überlädt man und wie überschreibt man?

## 12.10 Überschriebene Methoden in geerbten Methoden aufrufen

448. In welchem Sinne sind überschriebene Objektemethoden "*unwiederruflich*" überschriebenen, verdeckte Objektattribute dagegen nur "*wiederruflich*" verdeckt? Für wen gelten die Eigenschaftsworte *unwiederruflich* bzw. *wiederruflich*?

## 12.11 Erweitern oder instanziiieren

449. Wieviele Klassen darf eine neue Klasse *erweitern*? Und wiviele Klassen darf man in einer (neuen) Klasse *instanziiieren*?
450. Häufig kann man sich "die Leistungen und Vorteile" einer Klasse auf zwei sehr verschiedene Weisen nutzbar machen:
1. Indem man die Klasse *erweitert* (d.h. indem man sie beerbt) oder
  2. indem man die Klasse *instanziiert* (d.h. ein Objekt von ihr erzeugen lässt).
- Welche dieser beiden Vorgehensweisen sollte man in aller Regel *bevorzugen*? (S. 321)

451. Wie kann der Autor einer Klasse sicherstellen, dass seine Klasse *nicht* erweitert werden kann? Nennen Sie möglichst zwei Verfahren.

## 12.12 Klassen- und Objektinitialisierer

452. Wo dürfen *Klasseninitialisierer* (engl. static initialiser) und *Objektinitialisierer* (engl. non static initialiser) stehen? Nur unmittelbar vor einer Klassenvereinbarung? Nur innerhalb von Methoden? Nur direkt innerhalb einer Klassenvereinbarung? Nur in einer separaten Quelldatei? (S. 324)
453. Wie sieht ein Klasseninitialisierer aus? Wie sieht ein Objektinitialisierer aus?
454. Welche Art von Initialisierer ist *leicht entbehrlich* (weil man alle damit lösbar Probleme auch leicht anders lösen könnte)? Und welche Art ist *schwer entbehrlich* (weil man einige der damit lösbar Probleme nur schwer auf andere Weise lösen könnte)?
455. Wie oft und wann werden die *Klasseninitialisierer* einer Klasse ausgeführt?
456. Wie oft und wann werden die *Objektinitialisierer* einer Klasse K ausgeführt?
457. Wozu werden Klasseninitialisierer typischerweise verwendet?
458. Wozu werden Objektinitialisierer typischerweise verwendet?

## 13 Aufzählungstypen

459. Wo im Buch wird gezeigt (oder zumindest angedeutet), wie man in Java schon immer (d.h. vor Java 5) *Aufzählungstypen vereinbaren* konnte? (S. 328 und 214)
460. Im Abschnitt 9.3 des Buches, im Beispiel-02 auf S. 214, wird ein Aufzählungstyp als eine Klasse namens `Farbe` vereinbart. Diese Klasse enthält (fast) keine Objektelemente (nur die von `Object` geerbten). Die drei `Farbe`-Objekte `ROT`, `GRUEN` und `BLAU` sind also im wesentlichen leer. Wodurch unterscheiden sich die drei Objekte dann aber? Oder: Wodurch werden die drei Farben eigentlich dargestellt und voneinander unterschieden?

### 13.1 Ein paar einfache Beispiele

461. Anstelle eines Aufzählungstyps wie z. B.

```
1 enum Farbe {rot, gruen, blau, farblos}
```

könnte man auch vier `int`-Konstanten vereinbaren, etwa so:

### 13.1 Ein paar einfache Beispiele

```
2 final int ROT = 1;
3 final int GRUEN = 2;
4 final int BLAU = 3;
5 final int FARBLOS = 4;
```

Welche Vorteile hat der Aufzählungstyp `Farbe` gegenüber den vier `int`-Konstanten?

### 13.2 Ein Aufzählungstyp ist eine Klasse

462. Sei `Farbe` als Aufzählungstyp vereinbart wie in der vorigen Frage. Aus der Vereinbarung des Aufzählungstyps erzeugt der Ausführer eine Klasse als *Erweiterung* einer Standardklasse. Wie heißt diese Standardklasse? (S. 330)
463. Als was für Elemente der Klasse `Farbe` werden die Aufzählungswerte `rot`, `gruen`, `blau` und `farblos` (vom Ausführer, automatisch) implementiert? Als private Objektmethoden? Als öffentliche Klassenmethoden? Als öffentliche, unveränderbare Klassenattribute? Als öffentliche, veränderbare Objektattribute?
464. Was liefert die Funktion `Farbe.values()`? (S. 330, unten)
465. Mit welchen Operationen und Methoden kann man Objekte des Typs `Farbe` miteinander *vergleichen*?

### 13.3 Aufzählungstypen und static import

466. Mit `import`-Vereinbarungen führt man *einfache Namen* als Abkürzungen für *zusammengesetzte Namen* ein. Für welchen *zusammengesetzten Namen* führt die `import`-Vereinbarung `import javax.swing.JButton;` welchen *einfachen Namen* als Abkürzung ein?
467. Welchen (zusammengesetzten) Namen muss man angeben, wenn man (ohne die Hilfe einer `import`-Vereinbarung) eine Klassenmethode namens `sort` aus der Klasse `Arrays` (die zum Paket `util` im Paket `java` gehört) aufrufen will?
468. Welchen kürzeren Namen kann man angeben, wenn man die `sort`-Methode aus der vorigen Frage im Wirkungsbereich der normalen `import`-Vereinbarung `import java.util.Arrays;` aufrufen will?
469. Wir möchten beim Aufrufen der `sort`-Methode aus der vorigen Frage nur den *einfachen Namen* `sort` angeben, etwa so: `sort( ... );` Welche `import`-Vereinbarung müssen wir vorher angeben, damit das möglich ist?
470. Für die (zusammengesetzten) Namen von *was für Größen* kann man mit einer normalen `import`-Vereinbarung *Abkürzungen* einführen? Für die Namen von

Attributen? Von Klassenelementen? Von Schnittstellen? Von Methoden? Von Klassen und Schnittstellen?

471. Für die (zusammengesetzten) Namen von *was für Größen* kann man mit einer `import-static`-Vereinbarung *Abkürzungen* einführen? Für die Namen von Attributen? Von Klassenelementen? Von Schnittstellen? Von Methoden? Von Klassen und Schnittstellen?

472. Welcher der beiden `import-static`-Vereinbarungen ist "stärker" als die andere?

```
import static Farbe.rot;
import static Wein.*; // (S. 334)
```

## 14 Abstrakte Klassen und Schnittstellen

473. Wenn der Programmierer zum Geburtstag eine Klasse geschenkt bekommt, was kann er damit machen? Beschreiben Sie die beiden wichtigsten Tätigkeiten. Die Tätigkeiten "weiterverschenken" und "in den Müll werfen" sind auch möglich, gelten aber nicht als besonders wichtig :-).

474. Wenn eine Klasse als *abstrakte* Klasse vereinbart wurde, etwa so:

```
abstract class Carl extends Oskar { ... }
```

kann der Programmierer von den beiden Tätigkeiten aus der vorigen Frage nur noch *eine* durchführen. Welche? (S. 335)

475. In einer normalen (nicht-abstrakten) Klasse darf man Konstruktoren, Attribute, konkrete Methoden, Klassen und Schnittstellen vereinbaren. Was darf man in einer *abstrakten Klasse* vereinbaren?

### 14.1 Abstrakte Klassen

476. Angenommen, Sie programmieren eine Klasse `K`, die von ihrer abstrakten Oberklasse `A` eine abstrakte Methode `am` erbt. Wenn Sie dem Ausführer ihre Klasse `K` übergeben, lehnt er sie ab mit einer Fehlermeldung, die sich auf die abstrakte Methode `am` bezieht. Sie haben zwei ganz verschiedene Möglichkeiten, diese Fehlermeldung zu beseitigen. Welche?

477. Eine abstrakte Klasse `K` darf man *nicht* instanziiieren (ein Objekterzeugungsausdruck wie `new K(...)` ist verboten). Trotzdem kann man Objekte der Klasse `K` erzeugen lassen. Wie?

## 14.2 Schnittstellen (interfaces)

478. Welche der folgenden Größen darf man in einer *Schnittstelle* vereinbaren? Konkrete Objektmethoden? Abstrakte Objektmethoden? Konkrete Klassenmethoden? Abstrakte Klassenmethoden? (S. 341)

479. *Wieviele Schnittstellen* darf eine *Schnittstelle erweitern*? Null? Null oder eine? Null oder mehr? Eine oder mehr?

480. *Wieviele Schnittstellen* darf eine *Klasse implementieren*? Null? Null oder eine? Null oder mehr? Eine oder mehr? (S. 343)

481. Sei `S` eine Schnittstelle. Was versteht man dann unter einer *S-Klasse*? Und unter einem *S-Objekt*?

482. Sei *Ausgebbar* eine Schnittstelle. Was für Objekte darf man dann als Komponenten einer Reihung des Typs `Ausgebbar[]` speichern?

### 14.3 Schnittstellen dürfen auch Konstanten enthalten

483. Ausser *abstrakten Objektmethoden* darf eine Schnittstelle noch eine andere Art von Elementen enthalten? Welche? Konkrete Objektmethoden? Unveränderbare Objektattribute? Unveränderbare Klassenattribute? Abstrakte Klassenmethoden?

484. Die Schnittstelle `Runnable` enthält nur *eine* abstrakte Objektmethode namens `run`. Wozu ist eine konkrete Klasse `K` verpflichtet, die diese Schnittstelle implementiert, etwa so: `class K implements Runnable { ... }`?

485. Die Schnittstelle `javax.swing.SwingConstants` enthält nur Konstanten namens `BOTTOM`, `CENTER`, `EAST` etc. (sie enthält keine abstrakten Objektmethoden). Wozu ist eine Klasse `K` verpflichtet, die diese Schnittstelle implementiert, etwa so: `class K implements SwingConstants { ... }`?

### 14.4 Schnittstellen als Parametertypen

486. Die Schnittstelle `Runnable` (aus der Java-Standardbibliothek) wird unter anderem von den Standardklassen `FutureTask`, `Thread` und `TimerTask` implementiert. Welchen Vorteil hat somit eine Methode wie

```
void bearbeite01(Runnable r) { ... }
```

im Vergleich zu einer Methode wie

```
void bearbeite02(Thread t) { ... }?
```

## 14.5 Leere Schnittstellen als Markierungen

487. Was ist eine *Markierungsschnittstelle* (engl. a marker interface) und wozu dient sie? (S. 352)
488. Geben Sie zwei wichtige Markierungsschnittstellen aus der Java-Standardbibliothek an. (S. 353)

## 14.6 Anmerkungen (annotations) statt leere Schnittstellen

489. Mit Hilfe einer *Markierungsschnittstellen* S kann man nur *eine* Art von Dingen unterscheiden. Welche Art von Dingen? (S. 353)
490. Welche Arten von Dingen kann man mit Hilfe von *Anmerkungen* (engl. annotations) unterscheiden?
491. Warum kann es nützlich sein, bestimmte Pakete, Klassen, Methoden etc. mit Anmerkungen wie *Autor*, *Firma*, *NurGegenAufpreis* etc. zu markieren?
492. Welche Anmerkung wird *schon heute* vom Compiler *javac* von Sun beachtet, und was für Programmteile kann man mit dieser Anmerkung markieren? (S. 354)
493. Was ist eine *Anmerkung* technisch gesehen? Eine Methode? Eine Klasse? Ein Objekt? Ein Attribut? (S. 354)
494. Womit haben Anmerkungstypen einiges gemeinsam? Mit primitiven Typen? Mit Methoden? Mit Schnittstellen? Mit Klassen?
495. Wodurch unterscheiden sich Anmerkungstypen von Schnittstellen? Mit Hilfe von was für Konstrukten vereinbart man Anmerkungstypen?
496. Man darf ein Programmteil mit beliebig vielen Anmerkungen versehen. Welche Einschränkung muss man aber trotzdem beachten?
497. Wie bezeichnet Anmerkungen, mit denen man nur *Anmerkungstypen* versehen darf (aber keine anderen Programmteile wie Pakete, Methoden, Attribute etc.)? Wie bezeichnet man die Typen solcher Anmerkungen? (S. 359)
498. Der Programmierer schreibt Anmerkungen in seine Quellprogramme. Er kann dem Ausführer aber auch befehlen, die Anmerkungen "an andere Orte mitzunehmen". An welche Orte? Mit Anmerkungen von welchem Typ kann der Programmierer diese "Mitnahme" seiner Anmerkungen festlegen?

## 15 Ausnahmen fangen und werfen

499. Was ist eine *Ausnahmeklasse* (engl. exception class) technisch gesehen? Wodurch wird eine Klasse zu einer Ausnahmeklasse? (S. 363)
500. Was ist ein *Ausnahmeobjekt* technisch gesehen? Was kann man nur mit Ausnahmeobjekten machen (aber nicht mit anderen, normalen Objekten)?
501. Jedes Ausnahmeobjekt enthält mindestens 11 Methoden, die seine Klasse direkt von der Klasse *Throwable* geerbt hat. Nennen Sie eine oder zwei dieser Methoden. **Hinweis:** Es gilt grundsätzlich *nicht* als unanständig, Informationen in der HTML-Dokumentation der Java-Standardbibliothek nachzusehen :-).
502. Geben Sie Beispiele für Java-Befehle an, die unter bestimmten Bedingungen eine *Ausnahme werfen*. (S. 363, 364)

### 15.1 Ausnahmen fangen (der try-catch-Befehl)

503. Was passiert mit einer Programmausführung, in deren Verlauf eine Ausnahme *geworfen*, aber *nicht gefangen* wird?
504. Wie kann man Ausnahmen, die von einem gefährlichen Befehl GB *geworfen* werden, fangen und behandeln? Der gefährliche Befehl GB könnte dabei z. B. der Methodenaufruf `Integer.parseInt("ABC")` sein, der eine Ausnahme des Typs `NumberFormatException` wirft. (S. 366)
505. Angenommen, ein *try-catch*-Befehl mit drei *catch*-Blöcken wird ausgeführt. Wieviele der *catch*-Blöcke werden dabei ausgeführt? Alle drei? Keiner? Genau einer? Höchstens einer? Mindestens einer?
506. Angenommen, in einem Java-Programm ruft die *main*-Methode eine Methode *m1* auf und *m1* ruft eine Methode *m2* auf. Die Methode *m2* enthält einen gefährlichen Befehl, der möglicherweise eine Ausnahme wirft. An welchen Stellen des Programms kann man *try-catch*-Befehle einbauen, um diese Ausnahme zu fangen und zu behandeln?

### 15.2 Flugregeln für Ausnahmen

507. Die Ausführung von Methoden organisiert der Ausführer mit Hilfe eines *Stapels* (engl. stack). Über welche Methoden befinden sich in einem bestimmten Moment Informationen auf diesem Stapel? Über alle, deren Ausführung schon einmal begonnen wurde? Über alle, deren Ausführung schon abgeschlossen wurde? Über alle, deren Ausführung schon begonnen, aber noch nicht abgeschlossen wurde? Über alle Klassenmethoden? (S. 369)



508. Was prüft der Ausführer immer, wenn an einer bestimmten Stelle eine Programms eine Ausnahme AUS auftritt ("angeflogen kommt")?
509. Was macht der Ausführer, wenn in einer nicht-main-Methode eines Programms eine Ausnahme auftritt ("angeflogen kommt") und dort nicht gefangen wird?
510. Was macht der Ausführer, wenn in der main-Methode eines Programms P eine Ausnahme auftritt ("angeflogen kommt") und dort nicht gefangen wird?

### 15.3 Der try-catch-finally-Befehl

511. Was sollte man sicherstellen, wenn man eine Herdplatte (elektrisch oder mit Gas) einschaltet? (S. 373)
512. Was garantiert der Ausführer einem in Bezug auf einen try-catch-finally-Befehl?
513. Es gibt einen Befehl (nur einen), der die try-catch-finally-Garantie des Ausführers zum Erlöschen bringt. Welcher Befehl ist das?

### 15.4 Ausnahmen beim Behandeln von Ausnahmen

514. Welche Java-Anweisungen darf man innerhalb eines try-Blocks bzw. innerhalb eines catch-Blocks verwenden?
515. Was macht der Ausführer, wenn während der Ausführung eines catch-Blocks eine *Ausnahme* auftritt ("angeflogen kommt")? (S. 375)

### 15.5 Geprüfte und ungeprüfte Ausnahmen

516. Was sollte am Anfang jeder Methode in Bezug auf Ausnahmen dokumentiert werden? (S. 377)
517. Was für Ausnahmen sollte man *nicht* am Anfang jeder Methode als "wird möglicherweise ausgelöst" dokumentieren?
518. Welche negative Eigenschaft haben praktisch alle *Kommentare* und *Dokumentationen* in einem Programm?
519. Eine throws-Klausel darf man (nur) am Anfang einer Methodenvereinbarung hinschreiben (unmittelbar vor der öffnenden geschweiften Klammer, mit der der Rumpf der Methode beginnt), z. B. so:

```
6 void tuEs(int n) throws IOException, FileNotFoundException {
7 ...
8 }
```

Dagegen darf man throw-Anweisungen überall da hinschreiben, wo auch andere Anweisungen erlaubt sind, d.h. im Rumpf von Methoden und Konstruktor und in Initialisierern.

520. Wenn der Programmierer einen gefährlichen Befehl, der möglicherweise eine geprüfte Ausnahme A auslöst, in eine Methode m schreibt, muss er eine von zwei Maßnahmen treffen. Zwischen welchen beiden Maßnahmen kann er wählen?

### 15.6 Eigene Ausnahmeklassen vereinbaren

521. Wodurch kann der Programmierer eine von ihm vereinbarte Klasse zu einer *ungeprüften* bzw. zu einer *geprüften* Ausnahmeklasse machen?

### 15.7 Übersicht über ein paar wichtige Ausnahmeklassen

522. Geben Sie "die Wurzeln" (die obersten drei Klassen) des *Baums aller Ausnahmeklassen* an und zeichnen Sie als vierte Klasse auch noch die Klasse RuntimeException ein. Geben Sie von jeder der vier Klassen an, ob sie geprüft oder ungeprüft (engl. checked or unchecked) ist.
523. Es gibt nur *eine* geprüfte Ausnahmeklasse, die eine *ungeprüfte* direkte Unterklasse hat. Wie heißt die Klasse und ihre Unterklasse?

### 15.8 Zusicherungen (assertions)

524. Sei n eine int-Variable. Übersetzen Sie die Zusicherung `assert n%2==0;` ins Deutsche. (S. 384)
525. Was ist der Unterschied zwischen der Zusicherung `assert n%2==0;` (aus der vorigen Frage) und der folgenden Anweisung:  
`if (n%2==0) throw AssertionError();`

### 15.9 Kritik am Konzept der geprüften Ausnahmen

526. Wozu soll der Java-Programmierer durch das Konzept der *geprüften Ausnahmen* angeregt werden (wenn man es freundlich sagt) bzw. gezwungen werden (wenn man es weniger freundlich ausdrückt)?
527. Hat das Konzept der *geprüften Ausnahmen* dazu geführt, dass in Java-Programmen geprüfte Ausnahmen immer behandelt oder in throws-Klauseln übersichtlich dokumentiert werden?

528. Betrachten Sie das Beispiel-01 im Abschnitt 15.9 des Buches (auf S. 387). Wie kann und sollte man das dort skizzierte Problem lösen?

## 16 Generische Einheiten

529. Alle Befehle in Java sind *typsicher*. Welche Java-Befehle bezeichnen wir als *stark typsicher* und welche als *schwach typsicher*? (S. 388)

### 16.1 Warum generische Einheiten?

530. Java ist eine im Wesentlichen *stark typsichere Sprache* (d.h. fast alle Befehle sind stark typsicher im Sinne der vorigen Frage). Beschreiben Sie zwei Java-Befehle, die (abweichend von der Regel) nur *schwach typsicher* sind. (S. 389)
531. Welche Festlegung ("Regel der Sprache Java") ist der Grund dafür, dass schreibende Zugriffe auf nicht-primitive Reihungskomponenten nur *schwach typsicher* sind?
532. In welchem Zusammenhang musste man im vorgenerischen Java (bei allen Java-Versionen vor Java 5) besonders intensiv Cast-Befehle auf Referenzvariablen anwenden (d.h. nur schwach typsichere Befehle verwenden)? (S. 391)
533. Was ist das *prominenteste* Anwendungsgebiet für *generische Einheiten* (auch in Java)? (S. 392)

### 16.2 Generische Klassen selbst vereinbaren

534. Betrachten Sie die Vereinbarungen der Klassen `PaarA`, `PaarB<K>` und `PaarC<S, T>` auf S. 393 des Buches. Ähnlich wie bei Methoden kann man auch bei generischen Klassen *formale Parameter* (in der Vereinbarung der generischen Klasse) und *aktuelle Parameter* (beim Instanzieren der Klasse) unterscheiden. Wieviele formale Parameter hat die generische Klasse `PaarB<K>` und wie heißen sie? Wieviele formale Parameter hat die generische Klasse `PaarC<S, T>` und wie heißen sie? Was für Namen werden (entsprechend einer verbreiteten Konvention) für die formalen Parameter einer generischen Klasse verwendet?
535. Was darf (und muss) man beim Instanzieren einer generischen Klasse als aktuelle Parameter (für die formalen Parameter aus der Klassenvereinbarung) angeben? Was darf man *nicht* als aktuelle Parameter angeben? (S. 394)

536. Wieviele Typen repräsentiert eine normale Klasse wie `StringBuilder`, `String` oder `Double`? Wieviele Typen repräsentiert eine generische Klasse wie z. B. `PaarB<K>` oder `Paar<S, T>`?
537. Wenn man die generische Klasse `PaarB<K>` mit dem Typ `String` als aktuellem Parameter instanziiert (oder: parametrisiert), erhält man einen Typ. Wie schreibt und spricht man dessen Namen?
538. Wie erhält man den parametrisierten Typ `PaarC<Double, Number>`? In dem man welche Klasse mit welchen Typen als aktuelle Parameter instanziiert (oder: parametrisiert)?
539. Welcher der folgenden drei Sätze ist richtig?
1. Der p-PaarB-Typ `PaarB<String>` ist ein Untertyp von `PaarB<Object>`.
  2. Von den beiden p-PaarB-Typen `PaarB<String>` und `PaarB<Object>` ist keiner ein Untertyp des anderen.
  3. Der p-PaarB-Typ `PaarB<Object>` ist ein Untertyp von `PaarB<String>`.
540. Welcher der folgenden drei Sätze ist richtig?
1. Der p-PaarB-Typ `PaarB<String>` ist ein Untertyp des p-PaarC-Typs `PaarC<String, String>`.
  2. Von den beiden p-Typen `PaarB<String>` und `PaarC<String, String>` ist keiner ein Untertyp des anderen.
  3. Der p-PaarC-Typ `PaarC<String, String>` ist ein Untertyp des p-PaarB-Typs `PaarB<String>`.

### 16.3 Generische Schnittstellen implementieren

541. Wieviele formale Parameter hat die generische Schnittstelle `java.lang.Comparable` aus der Java-Standardbibliothek? Und wieviele (abstrakte Objekt-) Methoden enthält diese Schnittstelle? (S. 399)
542. Die Klasse `String` implementiert die Schnittstelle `Comparable<String>`. Deshalb enthält jedes `String`-Objekte eine Methode namens `compareTo`. Wieviele Parameter hat diese Methode und von welchen Typen sind die Parameter? Welchen *Ergebnistyp* hat diese Methode?
543. Seien `s1` und `s2` zwei `String`-Variablen, die auf zwei `String`-Objekte zeigen. Was wissen wir dann über den Ausdruck `s1.compareTo(s2)`? Wie hängt sein Wert von den `String`-Objekten `s1` und `s2` ab?
544. Was versteht man unter einem `Comparable<String>`-Objekt?
545. Geben Sie drei Instanzen der generischen Schnittstelle `Comparable<K>` an.
546. Wieviele und welche Instanzen der generischen Schnittstelle `Comparable<K>` darf eine Klasse namens `MeineKlasse` implementieren? Keine?

Höchstens eine (beliebige) Instanz? Beliebige viele Instanzen? Nur die eine Instanz `Comparable<MeineKlasse>`?

547. Wenn eine Klasse namens `MeineKlasse` eine Instanz der generischen Schnittstelle `Comparable<K>` implementiert, dann ist das in aller Regel *welche* Instanz?

## 16.4 Typparameter mit extends beschränken

548. Die generische Schnittstelle `Comparable<K>` hat einen formalen Parameter (oder: Typparameter) `K`. Wenn man `Comparable<K>` instanziiert, muss man für den Typparameter `K` einen aktuellen Parameter angeben. Welche Typen sind dabei als aktuelle Parameter erlaubt? (S. 401)

549. Die Typparameter (oder: formalen Parameter) vieler generischer Einheiten sind *unbeschränkt*. Was bedeutet "unbeschränkt" in diesem Zusammenhang?

550. Sei `Otto` eine Klasse. Übersetzen Sie die folgende Vereinbarung einer generischen Klasse (insbesondere das Wort `extends`) ins Deutsche:

```
1 class KarlHeinz<T extends Otto & Druckbar & Loeschbar> {...}
```

Was müssen `Druckbar` und `Loeschbar` in dieser Vereinbarung sein (weil der Ausführer sie sonst ablehnen würde)?

551. Übersetzen Sie die folgende Vereinbarung einer nicht-generischen Klasse (insbesondere das Wort `extends`) ins Deutsche:

```
2 class KarlHeinrich extends Otto {...}
```

552. Übersetzen Sie die folgende Vereinbarung einer generischen Klasse ins Deutsche:

```
3 class PaarD<S, T extends S> {...}
```

## 16.5 Generische Methoden

553. Wann bezeichnen wir zwei Typen als *unabhängig* voneinander? (S. 402)

554. Die folgende generische Klasse `PaarX<K>` ist *fehlerhaft*. Sie enthält vier Vereinbarungen von Elementen und in jeder dieser Vereinbarungen wird der formale Typparameter `K` benutzt:

```
1 class PaarX<K> {
2 static K a;
3 static void tuNix1(K b) {}
4 K c;
5 void tuNix2(K d) {}
```

```
6 }
```

Geben Sie von jedem der vier Elemente seinen Namen, seine Art (Methode oder Attribut) und seine Aspektzugehörigkeit (Objektelement, Klassenelement) an. Welche der vier Elementvereinbarungen sind falsch, weil darin der Typparameter `K` benutzt wird (die Elemente aber nicht im Gültigkeitsbereich solcher Typparameter liegen)? (S. 404)

555. Wie sieht die Vereinbarung einer generischen Klassenmethode (namens `machWas` mit *einem* Typparameter namens `T`) aus? Es genügt, wenn Sie ein typisches Beispiel andeuten.

## 16.6 Die Jokervariable ? im Vergleich mit anderen Typvariablen

556. Schreiben Sie die folgenden Typnamen untereinander (Untertypen unter Obertypen) und stellen Sie dabei die Relation ist-ein-Untertyp-von durch *Einrückung* dar: `Double`, `Object`, `Integer`, `Number`. (S. 406)

557. Wie darf man *normale Typparameter* wie `K`, `S`, `T` etc. (mit `extends`) beschränken? Nach links? Nach oben? Nach rechts? Nach unten? Gar nicht?

558. Wie darf man die *Jokervariable* ? (mit `extends` bzw. `super`) beschränken? Nach links bzw. rechts? Nach oben bzw. unten? Nach unten bzw. oben? Nach rechts bzw. unten? Nur nach unten? Gar nicht?

559. Sei `PaarH<K>` eine generische Klasse. Geben Sie ein paar *Untertypen* des Typs `PaarH<? extends Number>` an und begründen Sie kurz, warum es sich wirklich um Untertypen handelt.

560. Sei `PaarH<K>` eine generische Klasse. Geben Sie ein paar *Untertypen* des Typs `PaarH<? super Double>` an und begründen Sie kurz, warum es sich wirklich um Untertypen handelt.

561. Ist der Typ `ArrayList<String>` ein Untertyp von `ArrayList<Object>`? Ist der Typ `ArrayList<String>` ein Untertyp von `ArrayList<?>`?

## 16.7 Die Jokervariable ? und schreibende Zugriffe auf Parameter

562. Sei die Variable `saml` wie folgt vereinbart:

```
ArrayList<String> saml = new ArrayList<String>;
```

Ist ein schreibender Zugriff wie z. B.

```
saml.add("Hallo!");
```

dann erlaubt oder nicht?

563. Sei die Variable `sam2` wie folgt vereinbart:

```
ArrayList<?> sam2 = new ArrayList<String>;
Ist ein schreibender Zugriff wie z. B.
sam2.add("Hallo!");
dann erlaubt oder nicht?
```

## 16.8 Joker fangen (wildcard capture)

## 16.9 Rohe Typen

564. Beschreiben Sie einen wichtigen Unterschied zwischen dem *parametrisierten Typ* (oder: *p-Typ*) `ArrayList<Object>` und dem *rohen Typ* `ArrayList`. (S. 416)

565. Ein *p-Wert* bzw. ein roher *Wert* ist ein Wert eines parametrisierten Typs wie `ArrayList<String>`, `ArrayList<Integer>` etc. bzw. eines rohen Typs wie `ArrayList`. Für *p-Variablen* bzw. rohe *Variablen* gilt Entsprechendes. Welche Art der Zuweisung ist *typsicher* und welche ist *typunsicher*:  
Eine Zuweisung eines *rohen* Wertes an eine *p-Variable*?  
Eine Zuweisung eines *p-Wertes* an eine *rohe Variable*?

## 16.10 Typgraphen roher und parametrisierter Typen

566. Welche der folgenden Methodenvereinbarungen sind korrekt und welche nicht?

```
1 static <K> ArrayList<K> machWas1(ArrayList<K> al) {return al;}
2 static ArrayList<?> machWas2(ArrayList<?> al) {return al;}
3 static <K> ArrayList<K> machWas3(ArrayList<?> al) {return al;}
4 static ArrayList<?> machWas4(ArrayList<K> al) {return al;}
```

## 16.11 Falsche und fragwürdige Befehle

567. Was für Befehle werden in diesem Abschnitt als *fragwürdig* bezeichnet?

## 17 Pakete

568. Was ist ein *Paket* (in Java, nicht bei der Post :-)? (S. 424)

569. Wie heißt das *Top-Paket* java mit *vollem Namen*?

570. Wie heißt das Paket `sax` im Paket `xml` im *Top-Paket* `org` mit *vollem Namen*?

571. Aus welchen Zeichen bestehen Paketnamen üblicherweise? Warum?

### 17.1 Pakete und Klassen

572. Zu *wieviele* Paketen kann eine Klasse gehören? Zu höchstens einem Paket? Zu mindestens einem Paket? Zu beliebig vielen (0 oder mehr) Paketen? Zu genau einem Paket? (S. 425)

573. Was muss der Programmierer tun, damit eine Klasse `AbRechnung` zum Paket `de.meyerundsohn.test` gehört?

574. Wie lautet der *volle Name* einer Klasse `AbRechnung`, die zum Paket `de.meyerundsohn.test` gehört?

### 17.2 Öffentliche und nur paketweit erreichbare Klassen

575. Jedes *Element* (und jeder Konstruktor) einer Klasse gehört zu einer von vier verschiedenen *Erreichbarkeitsstufen* (`public`, `protected`, paketweit erreichbar und `private`). Wieviele und welche Erreichbarkeitsstufen gibt es für *Klassen*? (S. 427)

576. Von wo aus kann man auf eine nur *paketweit erreichbare* Klasse zugreifen?

577. Von wo aus kann man auf eine öffentliche Klasse (engl. `public class`) zugreifen, die zu einem Paket namens `de.meyerundsohn.test` gehört?

578. Von wo aus kann man auf öffentliche Klassen (engl. `public classes`) zugreifen, die zum *namenlosen Paket* gehören?

579. In welchen Fällen darf man zwei Klassen *nicht* in das selbe Paket tun?

580. Ein Paket ist zwar ein *Behälter* für Klassen, Schnittstellen und Pakete, aber kein richtiger *Modul*. Warum nicht?

### 17.3 Mit import Abkürzungen vereinbaren

581. Welche der folgenden Aussagen über `import`-Vereinbarungen treffen zu bzw. nicht zu?

1. `import`-Vereinbarungen sind notwendig, wenn man auf Klassen in bestimmten Paketen zugreifen will.

- 2. `import`-Vereinbarungen bewirken, dass Klassen aus bestimmten Paketen in ein Programm eingebunden werden.
- 3. Wenn der Programmierer viele `import`-Vereinbarungen in sein Programm schreibt, benötigt das Programm entsprechend viel Speicherplatz.
- 4. Die `import`-Vereinbarungen in einem Programm legen fest, welche Klassen zu diesem Programm gehören.

582. Was bewirkt die folgende `import`-Vereinbarung?

```
import java.util.ArrayList;
```

583. Was wird durch *pauschale* `import`-Vereinbarungen ("import-Vereinbarungen mit einem Sternchen") *vereinfacht* und was wird durch sie *erschwert*?

584. Was ist falsch an der folgenden Klassenvereinbarung:

```
1 import java.util.*;
2 class KarlHeiz extends Executors { ... }
```

Die Klasse `Executors` gehört zur Java-Standardbibliothek. Falls Sie noch nicht alle Einzelheiten dieser Klasse auswendig gelernt haben, sollten Sie sich jetzt vielleicht (in der Dokumentation der Java-Standardbibliothek) über sie informieren (nach dem Motto: "Wer lesen kann ist eindeutig im Vorteil!" :-).

- 585. Angenommen, wir schreiben eine Klasse namens `Karl` in einem Paket namens `patrick.test`. Welche Klassen und Schnittstellen dürfen wir dann mit ihren *einfachen Namen* bezeichnen, auch *ohne* entsprechende `import`-Vereinbarung?
- 586. Welche Klassen und Schnittstellen *darf man nicht* in einer `import`-Vereinbarung angeben?

## 17.4 Der Klassenbaum und der Paketwald

- 587. Ein *Baum* besteht aus *Knoten* und *Pfeilen*. Was sind beim *Baum aller Java-Klassen* die *Knoten* und wann führt von einem Knoten B ein *Pfeil* zu einem Knoten A? (S. 432)
- 588. Was ist ein *Wald*?
- 589. Erläutern Sie, warum alle Java-Pakete einen *Wald* bilden und nicht nur einen *Baum*.
- 590. Sei `p1` ein Unterpaket des Paketes `p0`.  
Seien außerdem `K1` eine Unterklasse der Klasse `K0`.  
Wenn die Klasse `K0` zum Paket `p0` gehört, zu welchem Paket muss dann `K1` gehören?
- 591. Welche Elemente einer Klasse sind "stärker geschützt" (d.h. von *weniger* Stellen des Programms aus erreichbar), die mit `protected` gekennzeichneten

Elemente oder die *paketweit erreichbaren* Elemente (die mit *keinem* Erreichbarkeitsmodifizierer gekennzeichnet sind)? (S. 434)

## 17.5 Programme in Paketen compilieren und ausführen

- 592. Angenommen, ein großes Java-Programm besteht aus vielen Klassen, die zu verschiedenen Paketen gehören. Die Quelldateien des Programms sollen im Windowsverzeichnis `d:\projekt27` bzw. im Unixverzeichnis `/projekt27` (und seinen Unterverzeichnissen) gespeichert werden. In welchem Unterverzeichnis sollte man dann die Quellen von Klassen ablegen, die zu einem Paket namens `einausgabe.basis` gehören?

## 18 Sammlungen (collections) und Abbildungen (maps)

- 593. Welche Vorteile hat es, wenn man 500 Variablen eines bestimmten Typs als eine *Reihung* (engl. `array`) vereinbart statt als 500 *separate Variablen*? (S. 437)
- 594. Welche Vorteile haben *Sammlungen* (engl. `collections`) im Vergleich zu *Reihungen* (engl. `arrays`)?
- 595. Die "Dinger, die in einer Sammlung gesammelt werden", bezeichnet man im Englischen als *the elements of the collection*. Warum sollte man diese "Dinger" im Deutschen nicht als *Elemente der Sammlung* bezeichnen, sondern besser als die *Komponenten der Sammlung*?
- 596. Was ist eine *Sammlung* inhaltlich? Was muss man mit einem Objekt tun können, damit man es als Sammlung benutzen kann? (S. 437)
- 597. Wie kann man eine Sammlung (ganz kurz und einfach) *formal* definieren? (S. 438)
- 598. Was ist eine Sammlungsklasse (kurz und formal definiert)?
- 599. Mit welcher Methode aus welcher Klasse kann man bewirken, dass eine *Reihung* wie eine *richtige Sammlung* aussieht?
- 600. Die Sammlungsklassen in der Java-Standardbibliothek bilden ein zusammenhängendes System. Beschreiben Sie eine wichtige Eigenschaft dieses Systems.
- 601. Von einer Reihung mit nicht-primtiven Komponenten wie z. B. `String[] sr = {"Anna", "Bert", "Claudia"};` sagt man häufig: Sie enthält (drei `String`-) *Objekte*. Was enthält die Reihung tatsächlich, wenn man sie technisch etwas genauer beschreibt?

602. Angenommen, `s` ist ein `String`-Objekt, welches etwa 100 Tausend Zeichen enthält. Wieviele Bytes verändert oder kopiert ein heute üblicher, maschineller Java-Ausführer, wenn er das Objekt `s` in eine Sammlung einfügt?

## 18.1 Schnittstelle sind Verträge

603. Eine Schnittstelle kann man als einen *Vertrag* auffassen. Zwischen wem gilt dieser Vertrag?

604. Eine Schnittstelle kann man als eine Menge von *Bedingungen* auffassen (die der Implementierer erfüllen muss und auf die der Anwender sich verlassen kann). Dabei unterscheidet man ganz allgemein *zwei Arten* von Bedingungen. Welche Bedingungsarten sind das?

605. Betrachten Sie die folgende Schnittstelle:

```
1 interface Plus1 {
2 int plus1(int n);
3 // Liefert den Wert n+1, wenn bei dessen Berechnung kein
4 // Ueberlauf auftritt. Liefert sonst den Wert n.
5 }
```

Beschreiben Sie ein paar *harte* Bedingungen, die jeder Implementierer dieser Schnittstelle `Plus1` erfüllen muss.

606. Beschreiben Sie eine *weiche* Bedingung, die jeder Implementierer der Schnittstelle `Plus1` erfüllen sollte.

607. Was für Objekte kann man in einer Sammlung des Typs `HashSet<String>` sammeln? Und in einer Sammlung des Typs `HashSet<Number>`? Und in einer Sammlung des Typ `HashSet<Object>`?

## 18.2 Die Sammlungsschnittstelle Collection (ohne s)

608. Was bezeichnet der Name `Collection` (*ohne s* am Ende)? Und was bezeichnet der Name `Collections` (*mit* einem `s` am Ende)? (S. 441, 459)

609. Die Schnittstelle `Collection` ist generisch und hat *einen* Typparamter. Der wird im Englischen häufig mit dem Buchstaben `E` bezeichnet und im Deutschen mit dem Buchstaben `K`. Woran soll `K` bzw. `E` erinnern?

610. Zur Erinnerung: Sei `S` eine Schnittstelle. Was ist dann eine *S-Klasse*? Und ein *S-Objekt*?

611. Wenn der Programmierer eine Klasse `KlausDieter` wie folgt vereinbart:

```
1 class KlausDieter implements Collection<String> { ... }
```

dann muss er in dieser Klasse unter anderem eine Objekt-Methode namens `add` vereinbaren. Mit wieviel Parametern von welchen Typen muss er diese `add`-Methode vereinbaren? (S. 441)

612. Jedes `Collection`-Objekt muss unter anderem eine Methode namens `add` enthalten (das schreibt die Schnittstelle `Collection` als harte Bedingung vor). Was sollte diese Methode `add` machen oder bewirken? Ebenso für die Methode `addAll`. Beschreiben Sie die Wirkungen der Methoden möglichst kurz und einfach. (S. 441, 442)

613. Wieviele *Konstruktoren* sollte jede `Collection`-Klasse mindestens haben? Und was sollen diese Konstruktoren leisten? (S. 445)

614. Wieviele Methoden der Schnittstelle `Collection` sind als *optional* gekennzeichnet? Was verbindet diese Methoden inhaltlich? Was dürfen diese optionalen Methoden bewirken, wenn man sie aufruft? (S. 446)

## 18.3 Die Sammlungsschnittstellen Set, Queue und List

615. Nennen Sie vier wichtige Erweiterungen der Sammlungsschnittstelle `Collection<K>`. (S. 447)

### 18.3.1 Die Schnittstelle Set

616. Die Sammlungsschnittstelle `Set<K>` ist eine Erweiterung der Sammlungsschnittstelle `Collection<K>`. Um wieviele Methoden erweitert die Schnittstelle `Set` die Schnittstelle `Collection`?

617. Um welche *weiche Vertragsbedingung* erweitert die Schnittstelle `Set` die Schnittstelle `Collection`?

618. Wann gelten zwei Objekte `ob1` und `ob2` als *Doppelgänger* voneinander?

619. Seien `s1` und `s2` zwei `String`-Objekte. Erläutern Sie, unter welchen Umständen der Ausdruck `s1.equals(s2)` den Wert `true` hat. (S. 448)

620. Seien `b1` und `b2` zwei `StringBuilder`-Objekte. Erläutern Sie, unter welchen Umständen der Ausdruck `b1.equals(b2)` den Wert `true` hat.

621. Geben Sie ein oder zwei Klassen aus der Java-Standardbibliothek an, die die Schnittstelle `Set<K>` implementieren.

### 18.3.2 Die Schnittstelle Queue

622. Die Sammlungsschnittstelle `Queue<K>` ist eine Erweiterung der Sammlungsschnittstelle `Collection<K>`. Um wieviele Methoden erweitert die Schnittstelle `Queue` die Schnittstelle `Collection`? (S. 449)

623. Um welche *weiche Vertragsbedingung* erweitert die Schnittstelle `Queue` die Schnittstelle `Collection`?
624. Welche `Queue`-Sammlungen bezeichnet man als *Schlangen*?
625. Welche `Queue`-Sammlungen bezeichnet man als *Prioritätsschlangen*?

### 18.3.3 Die Schnittstelle `List`

626. Die Sammlungsschnittstelle `List<K>` ist eine Erweiterung der Sammlungsschnittstelle `Collection<K>`. Um wieviele Methoden erweiter die Schnittstelle `List` die Schnittstelle `Collection`? (S. 450)
627. Um welche *Vorstellung* erweitert die Schnittstelle `List` die Schnittstelle `Collection`?
628. Angenommen, Sie haben ein Objekt `obl` in ein `Collection`-Objekt `samC` eingefügt und wollen es jetzt mit der Methode `remove` wieder entfernen. Was müssen Sie beim Aufruf der Methode `samC.remove(...)` als Parameter angeben?
629. Angenommen, Sie haben ein Objekt `obl` in ein `List`-Objekt `samL` eingefügt und wollen es jetzt wieder *entfernen*. Dann stehen Ihnen dazu zwei `remove`-Methoden zur Verfügung (eine aus der Schnittstelle `Collection` und eine aus der Schnittstelle `List`). Was für Parameter erwarten diese beiden `remove`-Methoden?
630. Sei `samL` ein `List`-Objekt, in das schon zahlreiche Objekte eingefügt wurden. Wieviele und welche Komponenten der Sammlung `samL` enthält dann das `List`-Objekt `samL.subList(3, 5)`?

## 18.4 Die Vergleichsschnittstellen `Comparable` und `Comparator`

631. Geben Sie die sechs Worte Bert, Zoo, Ballkleid, Albert, Berta, Anna 1. in der *lexikografischen* und 2. in der *lexikalischen* Reihenfolge an.
632. Geben Sie drei Kriterien an, nach denen man *Autos* sortieren kann.
633. Geben Sie Mengen an, die man nicht ohne weiteres sortieren kann, weil für sie keine allgemein bekannte totale Ordnung oder mehrere totale Ordnungen definiert sind.
634. Wenn man eine Java-Klasse entwirft, deren Objekte sortierbar sein sollen, muss man eine entsprechende *Vergleichsmethode* schreiben. Welche beiden Schnittstellen aus der Java-Standardbibliothek beschreiben solche Vergleichsmethoden?

635. Die Schnittstelle `Comparable` enthält nur eine einzige Methode. Wie heißt diese Methode, wieviele Parameter und welchen Ergebnistyp hat sie?
636. Die Schnittstelle `Comparator` enthält nur eine einzige Methode. Wie heißt diese Methode, wieviele Parameter und welchen Ergebnistyp hat sie?
637. Angenommen, wir wollen die Objekte einer Klasse `Auto` nach fünf verschiedenen Kriterien vergleichbar (und damit sortierbar) machen. Mit Hilfe welcher Schnittstelle sollten wir die wichtigste, "natürliche" Ordnung für `Auto`-Objekte definieren? Mit Hilfe welcher Schnittstelle sollten wir die übrigen vier Ordnungen für `Auto`-Objekte definieren?
638. Angenommen, wir wollen die Objekte einer Klasse `Auto` nach fünf verschiedenen Kriterien vergleichbar (und damit sortierbar) machen. In welcher Klasse sollten wir die Schnittstelle `Comparable` implementieren? Und in welchen Klassen sollten wir die Schnittstelle `Comparator` implementieren?
639. Die Schnittstellen `Comparable<String>`, `Comparable<Object>`, `Comparable<Auto>` etc. sind *Instanzen* der generischen Schnittstelle `Comparable`. Wenn man in der Klasse `Auto` eine Instanz der Schnittstelle `Comparable` implementieren will, *welche* Instanz sollte man dann (in aller Regel) implementieren? Welche andere Instanz kommt in einigen Fällen auch in Frage?
640. Wieviele Instanzen einer generischen Schnittstelle darf man in einer Klasse implementieren? Keine? Höchstens eine? Höchtens drei? Beliebig viele?
641. Sammlungen von welchem Typ werden vom Ausführer automatisch "in einem sortierten Zustand gehalten"? (S. 455)

## 18.5 Die Sammlungsschnittstelle `SortedSet`

642. Die Sammlungsschnittstelle `SortedSet<K>` ist eine Erweiterung der Sammlungsschnittstelle `Set<K>`. Um wieviele Methoden erweiter die Schnittstelle `SortedSet` die Schnittstelle `Set`? (S. 455)
643. Welche weiche Bedingung *erbt* die Schnittstelle `SortedSet` von der Schnittstelle `Set`?
644. Welche weitere (*nicht geerbte*) weiche Bedingung gehört auch noch zur Schnittstelle `SortedSet`?

## 18.6 Standard-Sammlungsklassen

645. Nennen Sie alle Klassen aus der Java-Standardbibliothek, die die Schnittstelle `SortedSet<K>` implementieren. (S. 457)

646. Welche Sammlungsklasse aus der Java-Standardbibliothek implementiert als einzige zwei (voneinander unabhängige) Schnittstellen? Und welche Schnittstellen sind das?
647. Sind Sammlungen des Typs `HashSet<K>` im allgemeinen *schnell* oder *langsam*? Sind sie *sortiert* oder *nicht sortiert*?

## 18.7 Die Klasse Collections (mit s)

648. Gibt es (in der Java-Standardbibliothek) eine Klasse `Collection` (ohne s)?
649. Was kann man mit den Methoden im Modul (oder: in der Klasse) `Collections` machen? Wozu sind sie gedacht und geeignet? Hier wird eine ganz kurze, abstrakte und einfache Antwort erwartet.
650. Was leistet die Methode `Collections.shuffle`? Was kann man mit ihr machen?
651. Was leistet die Methode `Collections.reverse`? Was kann man mit ihr machen?

## 18.8 Reihungen und Sammlungen

652. Sind Reihungen Sammlungen?
653. Mit Hilfe welcher Methode kann man eine Reihung als eine Sammlung sehen und bearbeiten? (S. 461)
654. Mit welchen Methoden kann man aus einer Sammlung eine Reihung erzeugen? (S. 460)

## 18.9 Abbildungen (map objects)

655. Woraus besteht (oder: was enthält) eine *Abbildung*?
656. Woraus besteht (oder: was enthält) ein *Eintrag*?
657. Was ist eine *Abbildungsklasse* (technisch, formal gesehen)?
658. Wieviele (formale) *Typparameter* hat die generische Schnittstelle `Map`?
659. Warum hat (in der Schnittstelle `Map<S, W>`) die Methode `keySet` den Ergebnistyp `Set<S>`, die Methode `values` dagegen den Ergebnistyp `Collection<W>`?
660. Was kann passieren, wenn man z. B. ein `StringBuilder`-Objekt `sb` als Schlüssel in eine Abbildung einfügt und es (während es in der Abbildung eingetragen ist) verändert?

661. Ist in einer Abbildung `null` als *Schlüssel* erlaubt? Ist `null` als *Wert* erlaubt? (S. 468)

## 19 Ein-/Ausgabe mit Strömen (streams)

662. Was passiert, wenn man mit der Methode `EM.liesInt()` versucht, einen `int`-Wert von der Tastatur einzulesen, der Benutzer aber eine ungeeignete Zeichenkette (z. B. ABC) eingibt?
663. Was passiert, wenn man mit der Methode `EM.liesDouble()` versucht, einen `double`-Wert von der Tastatur einzulesen, und der Benutzer die Zeichenkette `inf` eingibt?
664. Wenn man einen Ausgabebefehl wie z. B. `pln("Hallo wie geht es?");` ausführen läßt, werden die Daten häufig nicht wirklich ausgegeben. Was macht der Ausführer statt dessen mit den auszugebenden Daten? Warum macht er das?

### 19.1 Erste Beispiele mit Strömen

665. Ströme dienen dazu, Daten ein- bzw. auszugeben. Welche darüber hinausgehende Eigenschaft haben Ströme (und sind deshalb recht nützlich)? (S. 471)
666. Eine Kombination aus drei Strömen stellen wir etwa so wie in den folgenden beiden Beispielen dar:

```
1 ??? <-- felix <-- oskar <-- bruno <-- ???
2 ??? --> fiona --> ilse --> britta --> ???
```

Wofür stehen die Fragezeichen ??? ganz links und ganz rechts in diesen beiden simplen "Diagrammen"? Warum zeichnen wir Ströme immer in der hier angegebenen Reihenfolge (und nicht in der umgekehrten Reihenfolge, was ja auch leicht möglich wäre)? (S. 472, 477)

### 19.2 Zeichenorientierte und byteorientierte Ströme

667. Zu einem *Bildschirm* kann man eigentlich nur Daten *zweier Typen* ausgeben (Daten aller anderen Typen müssen vor der Ausgabe zu einem Bildschirm in Werte eines dieser beiden Typen umgewandelt werden). Welche beiden Typen sind das? (S. 474)
668. Von einer *Tastatur* kann man eigentlich nur Daten *zweier Typen* einlesen. Welche beiden Typen sind das?



669. Was passiert, wenn man z. B. den `int`-Wert 17 mit einem Ausgabebefehl wie `println(17)`; zum Bildschirm ausgibt (obwohl man `int`-Wert doch gar nicht zum Bildschirm ausgeben kann)?
670. Was passiert, wenn man z. B. den `int`-Wert 17 von der Tastatur einliest (obwohl man `int`-Werte doch gar nicht von der Tastatur einlesen kann)?
671. In der Java-Standardbibliothek gibt es vier *abstrakte Stromklassen*, von denen jede die Wurzel eines kleinen Typgraphen ist. Wie heißen diese vier abstrakten Stromklassen?

### 19.3 Ströme, Quellen und Senken miteinander verbinden

672. In was für *Datensenken* kann man mit Java-Stromobjekten Daten *schreiben*?
673. Aus was für *Datenquellen* kann man mit Java-Stromobjekten Daten *lesen*?
674. Nennen Sie mindestens eine *Stromklasse*, mit deren Objekten man Daten aus einer Datei lesen kann. (S. 478)
675. Nennen Sie mindestens eine *Stromklasse*, mit deren Objekten man Daten in eine Datei schreiben kann.
676. Objekte verschiedener Stromklassen kann man mit einer *Datei* verbinden (siehe die vorigen beiden Fragen). Geben Sie drei verschiedene Weisen an, auf die man eine Datei angeben ("beschreiben") kann.

### 19.4 Die Standardströme `System.out`, `-.err` und `-.in`

677. Wie hat man in Java die Standardströme `out`, `err` und `in` realisiert? Als Klassenmethoden der Klasse `System`? Als Objektmethoden der Klasse `System`? Als Klassenattribute der Klasse `System`? Als Objektattribute der Klasse `System`? (S. 482)
678. Mit welchen Geräten sind die Standardströme `System.out`, `System.err` und `System.in` (unter Windows und Unix) normalerweise verbunden?
679. Mit welchen Methoden kann man die Standardströme `System.out`, `System.err` und `System.in` mit anderen Geräten oder Dateien verbinden?
680. Objekte welcher *Typen* sind die Standardströme `System.out`, `System.err` und `System.in`?

### 19.5 Brücken zwischen `char`-Werten und `byte`-Werten

681. In welchem Code werden Zeichen (`char`-Werte) innerhalb eines Java-Programms während der Ausführung immer dargestellt?

682. Was für eine Abbildung führt ein `OutputStreamWriter` mit Hilfe einer *Encodierung* durch? Was für Werte werden dabei auf was für Werte abgebildet?
683. Was für eine Abbildung führt ein `InputStreamReader` mit Hilfe einer *Encodierung* durch? Was für Werte werden dabei auf was für Werte abgebildet?

### 19.6 Objekte in Ströme schreiben/aus Strömen lesen

684. Was bedeutet es, ein Objekt zu *serialisieren*? Was passiert dabei mit dem Objekt? (S. 484)
685. Was bedeutet es, ein Objekt zu *deserialisieren*?
686. Sei `oos` ein Objekt der Klasse `ObjectOutputStream` und sei `ob` irgendein Objekt. Was bewirkt dann der Befehl `oos.writeObject(ob)`? (S. 486)
687. Sei `ois` ein Objekt der Klasse `ObjectInputStream`, welches als nächstes ein (serialisiertes) Objekt einer Klasse *Bescheinigung* enthält, und sei `bs` eine Variable des Typs *Bescheinigung*. Was bewirkt dann der Befehl `bs = ois.readObject()`?

### 19.7 Programmteile mit Röhren (pipes) verbinden

688. Bei was für Programmteilen ist eine Verbindung durch eine Röhre (pipe) besonders interessant und möglicherweise vorteilhaft?
689. Sei `pw` ein `PipedWriter`-Objekt und `pr` ein `PipedReader`-Objekt und seien `pw` und `pr` zu einer Röhre (pipe) verbunden. Stellen Sie den Datenfluss durch diese Röhre als "simple ASCII-Graphik" dar. (S. 489)

### 19.8 Kurze Hinweise auf weitere Stromklassen

690. Mit Stromobjekten welcher Klassen kann man *mehrere Dateien* zu *einem Archiv* zusammenfassen und Daten beim Schreiben komprimieren und beim Lesen dekomprimieren?
691. Mit Stromobjekten welcher Klasse kann man *mehrere* `InputStream`-Objekte zu *einem* Eingabestrom zusammenfassen?
692. Mit Stromobjekten welcher Klasse kann man besonders bequem Zeichenfolgen lesen, die durch bestimmte Trennzeichen (z. B. durch Kommas oder Blanks) voneinander getrennt sind?
693. Stromobjekte welcher Klasse verfolgen automatisch die Anzahl der bereits eingelesenen Zeilenendemarkierungen (d.h. "die Zeilen-Nummer")?

694. Stromobjekte welcher Klasse erlauben es, bereits gelesene Daten zu "entlesen" (d.h. in den Eingabestrom zurückzulegen, damit man sie später und evtl. an einer anderen Stelle eines Programms erneut einlesen kann)?

## 19.9 Ein-/Ausgabe ohne Ströme (mit RandomAccessFile und nio)

695. In der Java-Standardbibliothek gibt es zwei Pakete mit Ein-/Ausgabe-Klassen: `java.io` (seit Java 1.0) und `java.nio` (seit Java 1.4). Welche sich widersprechenden *Ziele* hat man beim Entwickeln dieser Pakete (auf unterschiedliche Weise) zu erreichen versucht?
696. Welches der beiden Pakete `java.io` und `java.nio` ist stärker auf *Schnelligkeit* ausgerichtet und welches stärker auf *Plattformunabhängigkeit*?
697. Die Klassen im Paket `java.io` unterstützen die Ein-/Ausgabe von Daten mit Hilfe von *Strömen* (Stromobjekten). Mit was für Objekten unterstützen die Klassen im Paket `java.nio` die Ein-/Ausgabe?
698. Beschreiben Sie kurz den Unterschied zwischen einer *sequenziellen Datei* und einer *Direktdatei* (engl. random access file).
699. Welche "Strukturdaten" muss eine Datei enthalten, damit man sie als Direktdatei bearbeiten kann?

## 20 Steuerfäden (threads of control)

700. Was ist ein *sequentielles* Programm? (S. 494)
701. Skizzieren Sie ein Problem, welches man mit einem sequentiellen Programm nicht wirklich lösen kann.
702. Wenn der Programmierer dem Ausführer befiehlt, zwei Blöcke (Block1 und Block2) *nebenläufig* zueinander auszuführen, welche Wahl hat der Ausführer dann? Wie darf er die Blöcke dann ausführen?
703. Weil die Bezeichnung "nebenläufiges Programm" mehrdeutig ist, haben wir den Begriff *Proma* eingeführt. Was bedeutet er?
704. Wie kann man (nach Einführung des Begriffs *Proma* und als Gegenstück dazu) den Begriff *sequentielles Programm* kurz definieren? (S. 497)
705. Ein *Proma* ist in aller Regel erheblich schwerer zu testen als ein sequenzielles Programm, weil bestimmte Fehler nur sporadisch auftreten und man sie nur sehr schwer oder gar nicht reproduzieren ("zum erneuten Auftreten bewegen") kann.

706. Welche Java-Programme sind automatisch *Promas* ("Programme mit mehreren Ausführern"), obwohl der Programmierer darin keine Fäden gestartet hat?

## 20.1 Prozesse und Fäden

707. Bestimmte theoretische Untersuchungen beschäftigen sich mit nebenläufigen Einheiten (engl. concurrent units). In der Praxis unterscheidet man zwei Arten von nebenläufigen Einheiten. Wie bezeichnet man diese Einheiten?
708. Was ist die wichtigste, grundlegende ("definierende") Eigenschaft sowohl von Prozessen als auch von Fäden?
709. Was ist der wichtigste *Unterschied* zwischen Prozessen und Fäden?
710. Warum verwendet man (wenn man nebenläufige Einheiten benötigt) nicht ausschliesslich Fäden? Welchen Nachteil haben Fäden im Vergleich zu Prozessen?
711. Beschreiben Sie einen (möglichst einfachen) Fall, in dem zwei Fäden versuchen, eine gemeinsam genutzte Variable zu verändern und sich dabei stören. (S. 499)
712. Prozesse können sich nicht dadurch stören, dass sie eine gemeinsam genutzte Variable in ihren Adressräumen zu verändern versuchen. Was muss man tun, um trotzdem (einigermaßen zuverlässig) eine Störung zwischen zwei Prozessen zu provozieren?

## 20.2 Die Klasse Threads und die Schnittstelle Runnable

713. In Java kann man Fäden auf zwei verschiedene Weisen programmieren. Auf welche beiden Weisen?
714. Woraus besteht ein *Threads*-Objekt oder ein *Runnable*-Objekt im Wesentlichen? Welcher "Teil" eines solchen Objekts wird nebenläufig zu ändern, ähnlichen "Teilen" ausgeführt?
715. Wie kann man einem Java-Ausführer befehlen, sich in zwei Ausführer aufzuspalten (oder: einen neuen, zusätzlichen Ausführer zu erzeugen)?

## 20.3 Der Hauptfaden (main thread) eines Programms

716. Wenn der Java-Ausführer ein Programm ausführt, besteht er aus mindestens einem Faden, dem sogenannten *Hauptfaden* (engl. main thread). Welche Aufgaben hat dieser Hauptfaden?

## 20.4 Reihungen und Gruppen von Fäden

717. Wie kann man viele (z. B. 500) Fäden eines bestimmten Typs vereinbaren und starten lassen, ohne Blasen an den Fingern zu riskieren? Indem man die Vereinbarungen und Start-Befehle mit einem Editor und einer fingerfreundlichen Tastatur schreibt? Indem man einen Editor verwendet, der das Kopieren und Verändern von Zeilen besonders gut unterstützt? Indem man das Schreiben der Vereinbarungen und Start-Befehle in einem Land durchführen lässt, in dem das Lohnniveau deutlich niedriger ist als in Deutschlang? Auf andere Weise? (S. 505)
718. Was ist der Sinn und Zweck von Fadengruppen (ThreadGroup-Objekten)?

## 20.5 Nebenläufigkeitsfehler und synchronized-Blöcke

719. Stellen Sie sich ein Java-Programm vor, bei dessen Ausführung der Ausführer sich in mehrere Fäden aufgespalten hat. Welche Variablen werden von und für *jeden Faden* neu erzeugt? Und welche Variablen werden nur *einmal* erzeugt und von allen Fäden gemeinsam genutzt? (S. 506, 507)
720. Wann ist eine Methode *fadensicher* (engl. thread safe)? (S. 510)
721. Wie kann man in Java bewirken, dass eine Methode nur von *einem* Faden auf einmal ausgeführt werden kann (und damit automatisch fadensicher ist)?
722. Geben Sie ein (möglichst einfaches) Beispiel einer *nicht fadensicheren* Methode an.
723. Geben Sie ein (möglichst einfaches) Beispiel einer Methode an, die auch ohne *synchronized*-Befehl *fadensicher* ist.

## 20.6 Synchronized-Blöcke und Verklemmungen

724. Welche Art von *Programmfehlern* kann man nur mit Hilfe von *synchronized*-Befehlen erzeugen?
725. Was braucht man mindestens, um eine *Verklemmung* zu programmieren?
726. Nach welchem einfachen Schema kann man Verklemmungen sicher *verhindern*?

## 21 Reflexion

727. Welche Dinge gehören zur sogenannten *Reflektionsschnittstelle* von Java (die keine Schnittstelle im Sinne des Schlüsselwortes *interface* ist)?
728. Welche Programme, Klassen, Methoden etc. bezeichnen wir als *reflektiv*?

### 21.1 Class-Objekte reflektieren Klassen

729. Jedes Objekt der Klasse `Class` reflektiert eine Klasse. Erläutern Sie ein bisschen genauer, was damit gemeint ist.
730. In welcher Form (oder: "als was") existieren die Klassen eines Programms, während es ausgeführt wird? (S. 516)
731. Wie kann der Programmierer sich Zugriff auf das `Class`-Objekt verschaffen, welches die Klasse `StringBuilder` reflektiert?
- Indem er es mit dem Befehl `new Class("StringBuilder")` erzeugen lässt?
  - Mit Hilfe des Ausdrucks `StringBuilder.class` ?
  - Mit Hilfe des Ausdrucks `sb.getClass()` (wobei `sb` irgendein `StringBuilder`-Objekt sein muss) ?
  - Mit Hilfe des Ausdrucks `Class.forName("StringBuilder")` ? (S. 517)
732. Wie kann man auf das `Class`-Objekt zugreifen, welches den Reihungstyp `long[][]` reflektiert? Geben Sie drei verschiedene Ausdrücke an. Dabei dürfen Sie voraussetzen, dass die Variable `lr1` bereits existiert und auf eine Reihung des Typs `long[][]` zeigt.
733. Wie kann man auf das `Class`-Objekt zugreifen, welches den Reihungstyp `short[][][]` reflektiert? Geben Sie drei verschiedene Ausdrücke an. Dabei dürfen Sie voraussetzen, dass die Variable `sr1` bereits existiert und auf eine Reihung des Typs `short[][][]` zeigt. Achtung: `Short` ist ein Referenztyp, kein primitiver Typ!
734. Wie kann man auf das `Class`-Objekt zugreifen, welches den primitiven Typ `double` reflektiert? Und wie kann man auf das `Class`-Objekt zugreifen, welches die Hüllklasse `Double` reflektiert? Geben Sie je einen Ausdruck an.

### 21.2 Methoden einer beliebigen Klasse ausführen

735. Aus *wieviele* Methoden besteht die Lösung für die in diesem Abschnitt behandelte Aufgabe, und wie *heißen* die Methoden?

## 21.3 Oberklassen, Konstruktoren, Methoden, Attribute etc.

736. Angenommen, wir wollen *reflektiv* auf die Oberklassen, Konstruktoren, Methoden oder Attribute einer Klasse `AbRechnung` zugreifen. Zugriff auf welches Objekt müssen wir uns dann zuerst einmal verschaffen? (S. 523)
737. Angenommen, Sie haben in einem Java-Programm die drei parametrisierten Typen `Vector<String>`, `Vector<Integer>` und `Vector<Double>` verwendet. Wieviele `Class`-Objekte existieren dann, die diese Typen reflektieren?
738. Sei `kob` das `Class`-Objekt, welches die Klasse `AbRechnung` reflektiert. Welche Methoden liefert dann die Funktion `kob.getDeclaredMethods` (als Reihung von `Method`-Objekten) und welche Methoden liefert die Funktion `kob.getMethods`?
739. Sei `kob` das `Class`-Objekt, welches die Klasse `AbRechnung` reflektiert. Was liefert dann die Funktion `kob.getSuperclass`?

## 22 Grafische Benutzeroberflächen (Grabos)

740. Wodurch unterscheidet sich ein *Grabo-Programm* (ein Programm mit einer grafischen Benutzeroberfläche) von einem *Konsolen-Programm*?
741. Was für Programme kann man als eine Zwischenstufe zwischen Konsolen-Programmen und Grabo-Programmen verstehen?

### 22.1 Ein paar kleine Grabo-Programme

742. Wie (oder: als was) wird ein Objekt der Klasse `javax.swing.JFrame` auf dem Bildschirm dargestellt? (S. 530)
743. Nachdem man (in einem Java-Programm) ein `JFrame`-Objekt `job` hat erzeugen lassen, ist es auf dem Bildschirm noch nicht sofort sichtbar. Welchen Befehl muss man noch geben, damit es sichtbar wird? (S. 531)
744. Mit welchem Befehl kann man die anfänglich Größe und Position (der Bildschirmdarstellung) eines `JFrame`-Objekts `job` festlegen? Warum enthält diese Frage das vorsichtige Wort "anfänglich"?
745. Wenn der Java-Ausführer ein Grabo-Programm ausführt, spaltet er sich in mindestens zwei Fäden (Ausführer) auf. Wie heißen diese beiden Fäden?

746. Welcher dieser beiden Fäden ist normalerweise für das Zeichnen von Fenstern, Knöpfen, Menüs etc. verantwortlich? Welcher Faden führt die `main`-Methode aus?
747. Welche bemerkenswerte Eigenschaft hat die Methode `paint`, die man z. B. in einer Erweiterung der Klasse `JFrame` programmieren kann? (S. 533)
748. Wann wird die `paint`-Methode aufgerufen und vom wem?
749. Wenn man eine `paint`-Methode programmiert, welche Prozedur sollte man dann immer als erstes aufrufen?
750. Was passiert normalerweise, wenn der Benutzer mit seiner Maus auf den Fenster-Schließen-Knopf eines `JFrame`-Fensters klickt? Was passiert in so einem Fall aber nicht?
751. Sei `job` ein `JFrame`-Objekt. Mit welchem Befehl kann der Programmierer die normale Wirkung eines Klicks auf den Fenster-Schließen-Knopfs von `job` modifizieren? (S. 535)
752. Sei `job` ein `JFrame`-Objekt. Wenn der Benutzer auf den Fenster-Schließen-Knopf von `job` klickt, soll der Befehl  
`pln("Der Fenster-Schliessen-Knopf wurde angeklickt!");`  
 ausgeführt werden. Wie kann der Programmierer erreichen, dass das passiert? (S. 537)

### 22.2 Grabo-Objekte, -Klassen, -Pakete und Behälter

753. Was ist ein *Grabo-Objekt*? Geben Sie bitte eine *Java-unabhängige, inhaltliche* Definition an (keine Java-abhängige, formale Definition). (S. 541)
754. Was ist (wenn man ganz einfach auf der vorigen Definition aufbaut) eine *Grabo-Klasse*?
755. Was ist eine *Grabo-Klasse*? Geben Sie bitte eine *Java-abhängige, formale* Definition an (keine Java-unabhängige, inhaltliche Definition).
756. Was ist (wenn man ganz einfach auf der vorigen Definition aufbaut) ein *Grabo-Objekt*?
757. Wie heißt also die oberste Klasse im Typgraphen aller Java-Grabo-Klassen (sie kam schon in der Antwort auf die vorige Frage vor)? (S. 542)
758. Was ist eine (Grabo-) *Behälterklasse* (engl. *container class*)? Geben Sie bitte eine *Java-abhängige, formale* Definition an (keine Java-unabhängige, inhaltliche Definition).
759. Welcher der folgenden Sätze trifft zu?
1. Die Klasse `Component` ist eine Erweiterung der Klasse `Container`.
  2. Die Klasse `Container` ist eine Erweiterung der Klasse `Component`.
  3. Keine der beiden Klassen ist eine Erweiterung der anderen.

760. Was für Objekte darf man in ein *Behälterobjekt* hineintun?  
 761. Was für Objekte darf man in ein *Sammlungsobjekt* hineintun?  
 762. Was folgt aus den folgenden beiden Tatsachen:  
 1. Jedes Container-Objekt ist auch ein Component-Objekt.  
 2. Ein Container-Objekt darf beliebige Grabo-Objekte enthalten.  
 763. Zu wieviel Sammlungen (engl. collection objects) darf ein Objekt gleichzeitig gehören? Zu wievielen Behältern (engl. container objects) darf ein (Grabo-) Objekt gleichzeitig gehören?

## 22.3 Die Pakete awt und swing, schwere und leicht Komponenten

764. Wie heißen die beiden obersten Grabo-Pakete awt und swing mit vollem Namen?  
 765. Skizzieren Sie zwei unterschiedliche Strategien, nach denen man ein System von Grabo-Klassen und -Objekten unter einem bestimmten Betriebssystem BS implementieren kann. (S. 543)  
 766. Nach welcher dieser beiden Strategien wurde das Paket java.awt entwickelt? Und javax.swing?  
 767. Wie heißt die oberste Klasse im Typgraphen (fast) aller awt-Klassen (und damit auch fast aller Java-Grabo-Klassen)?  
 768. Wie heißt die oberste Klasse im Typgraphen aller swing-Klassen?  
 769. Was folgt aus der Tatsache, dass die Klasse javax.swing.JComponent eine Erweiterung der Klasse java.awt.Container ist? (S. 544, 545)  
 770. Mit welchem Buchstaben fangen die Namen vieler swing-Klassen an?

## 22.4 Aktionen, Ereignisse, Listener-Schnittstellen und Adapter-Klassen

771. Wer führt *Aktionen* durch? Der Ausführer? Der Programmierer? Der Benutzer? Die Kollegen?  
 772. Skizzieren Sie ein paar typische *Aktionen*.  
 773. Was ist ein *Ereignis* (engl. event)? Geben Sie ein typisches Beispiel an.  
 774. Ereignisse werden zu *Arten* und *Oberarten* zusammengefasst. Nennen Sie ein paar *Oberarten* von Ereignissen.  
 775. Geben Sie ein paar *Arten von Ereignissen* an, die zur Oberart *Fensterereignis* gehören.

776. Wieviele Methoden enthält die Schnittstelle WindowListener und wie heißen diese Methoden? (S. 548)  
 777. Objekte welcher Klassen können Quellen von *Fensterereignissen* sein?  
 778. Welche *Schnittstelle* und welche *Adapterklasse* gehören zusammen? Füllen Sie die folgende Tabelle aus:

| Schnittstelle       | Anzahl Methoden | Adapterklasse |
|---------------------|-----------------|---------------|
|                     |                 | WindowAdapter |
|                     |                 | MouseAdapter  |
| MouseMotionListener |                 |               |
| MouseWheelListener  |                 |               |
| ActionListener      |                 |               |

779. Zu welchen Schnittstellen gibt es keine Adapterklasse? Warum?  
 780. Was verstehen wir (im Zusammenhang mit Adapterklassen) unter einer *Strohpuppe*?  
 781. Objekte welcher Klassen können Quellen von *Mausereignissen* sein? (S. 551)  
 782. Objekte welcher Klassen können Quellen von *Mausbewegungsereignissen* sein? (S. 552)  
 783. Objekte welcher Klassen können Quellen von *Mausradereignissen* sein? (S. 553)  
 784. Wie bezeichnen wir Objekte, die Quellen von *Aktionsergebnissen* sein können? Nennen Sie ein paar Klassen, deren Objekte diese Eigenschaft haben. (S. 554)

## 22.5 Wettläufe zwischen Haupt- und Ereignisfaden

785. Wenn der Java-Ausführer ein Grabo-Programm (ein Programm mit einer grafischen Benutzeroberfläche) ausführt, spaltet er sich in mindestens *zwei* Fäden (Ausführer) auf, in einen *Hauptfaden* und in einen *Ereignisfaden*. Welchen Wertebehälter können beide Fäden verändern und sich dadurch (wenn der Programmierer sich ein bißchen Mühe gibt) gegenseitig stören?  
 786. Wie werden Nebenläufigkeitsfehler (z. B. Störungen zwischen dem Haupt- und dem Ereignisfaden eines Java-Grabo-Programms) im Englischen bezeichnet?  
 787. Was für Befehle sollte man nicht in die main-Methode eines Grabo-Programms schreiben (weil sonst Nebenläufigkeitsfehler zu befürchten sind)?

## 23 Java und der Unicode

788. Was ist ein *Zeichencode* (Kurzdefinition)? (S. 558)

### 23.1 Der ASCII-Code, der Unicode und der UTF-8-Code

789. Wievielen Zeichen ordnet der 7-Bit-ASCII-Code eine Codezahl zu?
790. Was wissen Sie über die Verbreitung des 7-Bit-ASCII-Code? Hat er sich als einfaches und gut handhabbares Werkzeug durchgesetzt?
791. Wievielen Zeichen ordnet ein 8-Bit-ASCII-Code eine Codezahl zu?
792. Was wissen Sie über die Verbreitung von 8-Bit-ASCII-Codes? Haben sie sich als einfache und gut handhabbare Werkzeuge durchgesetzt?
793. Wievielen Zeichen ordnet der Unicode Codezahlen zu?
794. Was versteht man unter CJK-Zeichen und in welchen Ländern werden sie vor allem benutzt?
795. Nennen Sie zwei Vorteile des Unicode (im Vergleich zu den ASCII-Codes).
796. Nennen Sie einen Nachteil des Unicode (im Vergleich zu den ASCII-Codes).
797. Welchen Zeichen ordnet der UTF-8-Code Codezahlen zu? Versuchen Sie nicht, diese Zeichen einzeln aufzuzählen (es sind ziemlich viele), sondern beschreiben Sie die Menge ganz allgemein, "mit einem billigen Trick".
798. Wodurch unterscheiden sich der Unicode und der UTF-8-Code vor allem?
799. Welchen Zeichen ordnet der UTF-8-Code besonders kurze Codezahlen (d.h. 8-Bit-Codezahlen) zu?
800. Was verbindet den Unicode und den UTF-8-Code?
801. Bei was für Dateien ist es vorteilhaft, sie im UTF-8-Code abzuspeichern statt im Unicode?

### 23.2 Unicode-Editoren und Compilationskommandos

802. Was sollte ein Unicode-Editor können? (S. 562)
803. Skizzieren Sie zwei verschiedenen Verfahren zum Eingeben von Unicode-Zeichen.

### 23.3 Java-Konsolen-Programme und Unicode-Zeichen

804. Welche Zeichen darf der Java-Programmierer für die Namen seiner Klassen, Methoden, Attribute etc. verwenden? (S. 563)

805. Wie kann man chinesische Schriftzeichen in ein Java-Programm einfügen (als Teil eines Namens oder in einem `String`-Literal oder als `char`-Literal), wenn einem *kein* Unicode-Editor zur Verfügung steht?



<http://www.springer.com/978-3-528-05914-9>

Java ist eine Sprache

Java lesen, schreiben und ausführen — Eine präzise  
und verständliche Einführung

Grude, U.

2005, XII, 604 S. 612 Abb., Softcover

ISBN: 978-3-528-05914-9