
Antworten

zu den Fragen zu den einzelnen Kapiteln des Buches

"Java ist eine Sprache"

von Ulrich Grude.

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

"Java ist eine Sprache", Antworten auf die Fragen

1. Im ersten Kapitel werden einige besonders wichtige und allgemeine Grundbegriffe der Programmierung vorgestellt.
2. Die fünf Rollen: *Programmierer*, *Ausführer*, *Benutzer*, *Warter*, *Wiederverwender* (die letzten beiden werden auch als *die Kollegen* des Programmierers bezeichnet).
3. Besetzung 1: Ein (einzelner) *Mensch* übernimmt die Rollen Programmierer, Benutzer, Warter und Wiederverwender. Die Rolle des Ausführers wird einem *Computer* (mit Betriebssystem, Editor, Compiler, Interpreter etc.) übertragen. Besetzung 2: Ein *Team von ProgrammiererInnen* übernimmt die Rolle des Programmierers (die anderen Rollen entsprechend). Besetzung 3: Ein Mensch übernimmt zuerst die Rolle des *Programmierers*, schreibt ein Programm, schlüpft dann in die Rolle des *Ausführers* und führt das Programm "mit Papier und Bleistift" aus.
4. Der Programmierer *schreibt* Programme und *übergibt* sie dem Ausführer.
5. Der Ausführer *prüft* die Programme (die der Programmierer ihm übergibt), *lehnt* sie *ab* oder *akzeptiert* sie. Wenn der Benutzer ihn dazu auffordert, *führt* er zuvor akzeptierte Programme *aus*.
6. Ein Programm ist eine *Folge von Befehlen*, die von einem Programmierer geschrieben wurde und von einem Ausführer ausgeführt werden kann.
7. Zum Ausführer gehört alles was man braucht, um ein Programm zu erstellen und ausführen zu lassen, z. B. ein Computer mit Betriebssystem, ein Editor, ein Compiler etc. Der Ausführer kann aber auch ein Mensch sein, der Programme mit Papier und Bleistift ausführen kann.
8. Die konkreten Eigenschaften von Computern und Betriebssystemen ändern sich sehr schnell. Der abstrakte Begriff des *Ausführers* soll all die Eigenschaften zusammenfassen, die vermutlich auch in ein paar Jahren noch ähnlich sind wie heute und von "kurzlebigen Einzelheiten" abstrahieren.
9. Ein Hallo-Programm schreibt man typischerweise, wenn man damit beginnt, eine neue Programmiersprache zu lernen, oder wenn man einen neuen Ausführer kennenlernen und ausprobieren will.
10. Folgende Arbeitsschritte kann man mit einem Hallo-Programm ausprobieren: Ein Programm an den Ausführer übergeben, das Programm ausführen lassen, kleine Änderungen am Programm vornehmen, das Programm erneut übergeben.
11. In einem Java-Quellprogramm wird ein Text zu einem Kommentar, wenn man einen doppelten Schrägstrich // davor schreibt. Ein solcher Kommentar reicht bis zum Ende der betreffenden Zeile.

12. Kommentare sind für die *Kollegen* des Programmierers gedacht. Der *Ausführer* ignoriert Kommentare weitgehend (inwiefern nur *weitgehend* und nicht vollständig wird in einer späteren Antwort auf die gleiche Frage erläutert).
13. Jedes Java-Programm muss ein Unterprogramm namens *main* enthalten.
14. In einem Java-Programm darf man (unter anderem) an folgenden Stellen eine neue Zeile beginnen:
Vor und nach jedem *Schlüsselwort* wie z. B. *class* oder *static* etc. ,
vor und nach jedem *Klammerzeichen* () [] { } < > ,
vor und nach jedem *Punkt* . ,
vor und nach jedem *Operator* wie z. B. + - * / % = += == != etc.
15. Ein langes *String-Literal* kann man beliebig in kleinere *String-Literale* zerlegen, zwischen die man den Operator + schreibt, z. B. so:
"Dies ist der Anfang eines " +
"langen *String-Literals*, wel" +
"ches auf 3 Zeilen verteilt wurde"
Jedes der kleineren *String-Literale* muss vollständig auf einer Zeile stehen, die +-Operatoren kann man am Ende einer Zeile oder am Anfang der nächsten Zeile notieren. Wenn man ein solches "auf *mehrere* Zeilen verteilte" *String-Literal* zum Bildschirm ausgibt, erscheint es dort auf *einer* Zeile.
16. Die Dateien, in die der Programmierer seine Programme schreibt, bezeichnet man als *Quelldateien* (engl. source files).
17. Eine *Plattform* ist eine Kombination aus einem Computer (damit ist hier nur die Hardware gemeint, vor allem der Prozessor) und einem Betriebssystem, z. B. ein PC unter Linux oder ein PC unter Windows oder ein MacIntosh (PowerPC) unter OS X oder ein MacIntosh unter Linux etc.
18. Ein *Quelldateien-Interpreter* führt Java-Quelldateien aus (ohne dass der Programmierer die Dateien vorher compilieren lassen oder sonstwie bearbeiten muss).
19. Ein *nativer Java-Compiler* für eine bestimmte Plattform (z. B. für einen PC unter Linux) erzeugt aus Java-Quelldateien eine *ausführbare Datei*, die von der betreffenden Plattform (*direkt*, ohne weitere Umwandlungen und ohne die Hilfe eines Interpreterprogramms) ausgeführt werden kann.
20. Ein *normaler* (nicht-nativer) *Java-Compiler* erzeugt aus Java-Quelldateien sogenannte *Bytecodedateien* (oder: *class-Dateien*, weil ihre Namen die Erweiterung *.class* haben).
21. Die vier wichtigsten Grundkonzepte von Programmiersprachen: *Variablen* (deren Werte man beliebig oft verändern kann), *Typen*, *Unterprogramme* und *Module*. Das Konzept einer *Klasse* ist das fünfte dieser vier wichtigsten Grundkonzepte :-).

22. Eine *Variable* ist ein Wertebehälter, dessen Inhalt man beliebig oft durch einen neuen Wert ersetzen kann.
23. Eine Variable kann nie *leer* sein, und wenn man einen neuen Wert in eine Variable tut, wird dadurch der bisherige Wert der Variablen zerstört.
24. Auch einen Bildschirm, einen Drucker, eine Tastatur und eine Datei auf einer Festplatte etc. bezeichnen wir als *Wertebehälter*. In einen Bildschirm-Wertebehälter kann man Werte nur hineinschreiben (aber nicht herauslesen), aus einem Tastatur-Wertebehälter kann man von einem Programm aus Werte nur herauslesen, in einen Datei-Wertebehälter kann man Werte hineinschreiben und später wieder herauslesen.
25. **Def-1:** Ein *Typ* ist ein Bauplan für Variable.
Def-2: Ein *Typ* besteht aus einer Menge von Werten und einer Menge von Operationen, die man auf die Werte anwenden darf.
26. Der Typ einer Variablen legt fest, welche *Werte* man in die Variable hineintun darf (und welche man *nicht* hineintun darf) und welche *Operationen* man auf die Variable anwenden darf (und welche man *nicht* auf sie anwenden darf).
27. Der Programmierer muss ein Unterprogramm einmal *vereinbaren* (d. h. vom Ausführer *erzeugen* lassen) und darf es dann beliebig oft *aufrufen* (d. h. vom Ausführer *ausführen* lassen).
28. **Upro-Vorteil-1:** Ein gut gewählter *Name* erleichtert es den Kollegen zu erkennen, "was das Unterprogramm macht".
Upro-Vorteil-2: Wartungsarbeiten werden vereinfacht (weil Korrekturen und Veränderungen nur an *einem* Unterprogramm vorgenommen werden müssen und nicht an *vielen* Kopien der betreffenden Befehlsfolge).
Upro-Vorteil-3: Beim Lesen erkennt man leicht, wo überall ein bestimmtes Unterprogramm aufgerufen wird (Kopien einer bestimmten Befehlsfolge sind viel schwerer zu erkennen).
29. Ein *Modul* ist ein Behälter für Unterprogramme, Variablen, weitere Module und andere Dinge, der aus mindestens *zwei Teilen* besteht, einem *öffentlichen* (oder: ungeschützten, sichtbaren) und einem *privaten* (oder: geschützten, sichtbaren) Teil.
30. Wenn man eine Variable im *privaten* Teil eines Moduls M vereinbart und beim Testen feststellt, dass die Variable einen falschen Wert enthält, muss man die entsprechenden "falschen Befehle" nur im Modul M (und nicht in allen Modulen des Programms) suchen.
31. Eine *Klasse* ist ein Modul und ... (diese Definition ist noch unvollständig).
32. Die drei Befehlsarten auf Deutsch: *Vereinbarungen, Ausdrücke, Anweisungen*.
33. Die drei Befehlsarten auf Englisch: *declarations, expressions, statements*.
34. Eine Vereinbarung auf Deutsch: "Erzeuge ...".

35. Ein Ausdruck auf Deutsch: "Berechne den Wert des Ausdrucks ..."
36. Eine Anweisung auf Deutsch: "Tue die Werte ... in die Wertebehälter ..."
37. Der Befehl `int otto = 17;` ist eine *Vereinbarung*. Auf Deutsch: "Erzeuge eine Variable namens `otto` vom Typ `int` mit dem Anfangswert 17!"
38. Der Befehl `2 * otto + 35` ist ein *Ausdruck*. Auf Deutsch: "Berechne den Wert des Ausdrucks `2 * otto + 35`".
39. Der Befehl `otto = 2 * otto + 35;` ist eine *Anweisung* (genauer: eine Zuweisungsanweisung, oder kürzer: eine Zuweisung). Auf Deutsch: "Berechne den Wert des Ausdrucks `2 * otto + 35` und tue ihn in die Variable `otto`!"
40. Der Befehl `class Carl { ... }` ist eine *Vereinbarung*. Auf Deutsch: "Erzeuge eine Klasse namens `Carl` ...".
41. Der Befehl `otto` ist ein *Ausdruck*. Auf Deutsch: "Berechne den Wert des Ausdrucks `otto`".
42. Der Befehl `3.7` ist ein *Ausdruck*. Auf Deutsch: "Berechne den Wert des Ausdrucks `3.7`".
43. Ein normaler Ausdruck bewirkt, dass der Ausführer *einen Wert berechnet*.
44. Ein normaler Ausdruck bewirkt *nicht*, dass der Ausführer *den Inhalt von irgendeinem Wertebehälter verändert* (außerdem bewirkt ein normaler Ausdruck auch nicht, dass alle Pinguine am Südpol ins Wasser hüpfen, aber diese nicht-Wirkung war hier nicht gemeint :-).
45. Die nicht-normalen Ausdrücke, die nicht nur die *Berechnung eines Wertes*, sondern nebenbei auch noch die *Veränderung eines Wertebehälters* bewirken, bezeichnet man als *Ausdrücke mit Seiteneffekt*. Solche Befehle haben gleichzeitig den Charakter von *Ausdrücken* (sie bewirken, dass ein Wert berechnet wird) und den Charakter einer *Anweisung* (sie bewirken, dass der Inhalt eines Wertebehälters verändert wird).
46. Durch eine Anweisung wie `System.out.println("Hallo!");` wird der Wertebehälter *Bildschirm* verändert (eigentlich wird der Wertebehälter Standardausgabe verändert, aber meistens ist der Bildschirm die Standardausgabe).
47. Wenn `karl` eine `int`-Variable mit dem momentanen Wert 20 ist, dann hat der Ausdruck `++karl` den Wert 21 und den Seiteneffekt, dass der Wert von `karl` um 1 (auf 21) erhöht wird.
48. Wenn `karl` eine `int`-Variable mit dem momentanen Wert 20 ist, dann hat der Ausdruck `karl++` den Wert 20 und den Seiteneffekt, dass der Wert von `karl` um 1 (auf 21) erhöht wird.
49. Ein *Unterprogramm*, welches innerhalb einer *Klasse* vereinbart wurde, bezeichnet man als *Methode*. **Anmerkung:** In Java darf man Unterprogramme

- nur innerhalb von Klassen vereinbaren. Deshalb sind in Java *alle* Unterprogramme auch Methoden.
50. Zwei Arten von Unterprogrammen: *Prozeduren* und *Funktionen*
51. Antwort: Ja, beides. Die Reihenfolge, in der man Methoden innerhalb einer Klasse vereinbart, ist für den *Ausführer unwichtig*, für die *Kollegen* des Programmiers aber möglicherweise *wichtig* (um die Klasse zu verstehen).
52. *Funktionen* dienen dazu, Werte zu berechnen.
53. *Prozeduren* dienen dazu, die Inhalte von Wertebehältern (z. B. den Inhalt einer Variablen oder den Inhalt des Bildschirms) zu verändern.
54. Beim Vereinbaren einer Funktion muss man unmittelbar vor dem Namen der Funktion den *Ergebnistyp* (oder: Rückgabety) der Funktion angeben.
55. Beim Vereinbaren einer Prozedur muss man unmittelbar vor dem Namen der Prozedur das Schlüsselwort `void` angeben (damit kennzeichnet man die Methode als Prozedur und unterscheidet sie von allen Funktionen).
56. Der Prozeduraufruf `pln("Hallo!");` ist in vielen Beispielprogrammen eine Abkürzung für den Prozeduraufruf `System.out.println("Hallo!");`
57. Um in einer Methode einer Klasse `Hallo30` eine parameterlose Methode namens `formatiereDieFestplatte` aufzurufen, die in der Klasse `Hallo20` vereinbart wurde, müssen wir folgenden Befehl hinschreiben:
`Hallo20.formatiereDieFestplatte();`
58. Die Anweisung `if (a < b) pln("a ist kleiner!");` auf Deutsch:
"Berechne den Wert des Ausdrucks `a < b`. Wenn er gleich `true` ist, dann führe die Anweisung `pln("a ist kleiner!");` aus.
59. Wenn `a` und `b` `int`-Variablen mit den Werten 17 bzw. 25 sind, dann hat der Ausdruck `a < b` natürlich den Wert `true`. Dieser Wert (und somit auch der Ausdruck `a < b`) gehört zum Typ `boolean`. Ein Ausdruck wie `a < b < 30` ist verboten, weil man den `boolean`-Wert des Ausdrucks `a < b` nicht mit dem `int`-Wert 30 vergleichen kann (`true` ist weder kleiner noch nicht-kleiner als 30).
60. *Ausdrücke* werden vom *Programmierer* in Quellprogramme geschrieben. *Werte* existieren nur während der *Ausführung* eines Programms und werden vom *Ausführer* manipuliert (z. B. addiert, oder verglichen oder in Variablen gespeichert etc.).
61. Der Wert des Java-Ausdrucks `0.1` ist *nicht* gleich ein Zehntel (sondern ein klein bisschen größer).
62. Ein maschineller Java-Ausführer ist in aller Regel deutlich schneller als ein menschlicher Java-Ausführer. Manche Menschen drücken das so aus: Ein Computer kann pro Sekunde viel mehr Fehler machen als ein Mensch :-).

63. Wenn bei der Ausführung eines Java-Programms etwas schief geht, kann ein menschlicher Java-Ausführer häufig verstehen und erklären, was genau schief gegangen ist. Maschinelle Java-Ausführer sind meist ziemlich wortkarg, wenn man sie nach dem *Grund* eines Fehlers fragt :-).
64. Mit einer Anweisung befiehlt der Programmierer dem Ausführer, bestimmte Werte in bestimmte Wertebehälter zu schreiben.
65. Man unterscheidet *einfache Anweisungen* und *zusammengesetzte Anweisungen* (engl. simple statements und compound statements).
66. *Einfache Anweisungen* enthalten keine anderen Anweisungen als ihre Bestandteile. *Zusammengesetzte Anweisungen* *enthalten andere Anweisungen* als ihre Bestandteile.
67. Man unterscheidet bei den zusammengesetzten Anweisungen *Fallunterscheidungsanweisungen* (oder etwas kürzer: Fallunterscheidungen) und *Wiederholungsanweisungen* (oder etwas kürzer: Wiederholungen oder Schleifen).
68. Bei einer Ausführung einer Methode, die nur *einfache* Anweisungen enthält, wird jede Anweisung genau *einmal* ausgeführt (einige Fragen sind schwierig, weil sie so einfach sind).
69. Mit einer Fallunterscheidungsanweisung kann der Programmierer bewirken, dass (mindestens) eine darin enthaltene Anweisung *A weniger als einmal* (also *nicht*) ausgeführt wird.
70. Mit einer Wiederholungsanweisung kann der Programmierer bewirken, dass die darin enthaltene Anweisung *A mehr als einmal* ausgeführt wird.
71. In C, C++ und Java wird die *Zuweisungsanweisung* mit dem *Gleichheitszeichen* = bezeichnet. Warum die Entwickler für diese Festlegung nicht zu Haftstrafen verurteilt wurden? Tja, das weiss ich auch nicht :-). Vermutlich blieben sie unbestraft, weil sie sich ansonsten mit der Entwicklung ihrer Sprachen sehr hohe und bleibende Verdienste erworben haben.
72. Die *Gleichheitsoperation* wird in C, C++ und Java mit einer Kombination aus zwei *Gleichheitszeichen* == bezeichnet (z. B. so: `a == b`).
73. Eine Zuweisung wie `a = b + c`; sollte man möglichst *nicht* als "a gleich b plus c" vorlesen, sondern besser als "a wird zu b plus c" (weil das Gleichheitszeichen = nicht die *Gleichheitsoperation*, sondern die *Zuweisung* bezeichnet).
74. Die Anweisung `berta += 17;` ist eine Abkürzung für die Anweisung `berta = berta + 17;`. Ganz Entsprechendes gilt, wenn man anstelle des (aus + und =) *zusammengesetzten Zuweisungsoperators* += einen anderen zusammengesetzten Zuweisungsoperator wie -=, *=, /=, %= etc. verwendet. Im Buch findet man auf S. 144 *alle* 11 zusammengesetzten Zuweisungsoperatoren.

75. Aus einem *Ausdruck mit Seiteneffekt* wie etwa `++isidor` kann man durch Anhängen eines Semikolons die *Anweisung* `++isidor;` machen. Entsprechendes gilt für alle *Ausdrücke mit Seiteneffekt*, aber nicht für *normale Ausdrücke*.

76. Die Anweisung `System.out.println("Hallo!");` ist ein *Methodenaufruf*.

77. Zwei Aufrufe der Funktion `quad`:

```
1 int quad = hoch2(5); // Sinnvoller Aufruf von hoch2
2 hoch2(5);           // Sinnloser Aufruf von hoch2
```

Der zweite Aufruf der Methode `hoch2` (in Zeile 2) ist *sinnlos*, weil das Ergebnis des Aufrufs (die Zahl 25) vom Ausführer "weggeworfen" wird. Beim ersten Aufruf wird das Ergebnis zum Initialisieren der Variablen `quad` verwendet.

78. Die *break*-Anweisung darf man nur innerhalb von *Schleifenanweisungen* und von *switch*-Anweisungen verwenden.

79. Die *continue*-Anweisung darf man nur innerhalb von *Schleifenanweisungen* verwenden.

80. 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 }
```

besteht aus der Bedingung `(n < 7)` und dem Rumpf

```
1
2 {
3     pln(hoch2(n));
4     n = n + 1;
5 }
```

81. Der Rumpf dieser *while*-Schleife wird *viermal* ausgeführt.

82. Mit einer *break*-Anweisung kann man eine *Ausführung einer Schleife* ("eine Schleifenausführung") beenden. Mit einer *continue*-Anweisung kann man eine *Ausführung des Rumpfes einer Schleife* ("eine Rumpfausführung") beenden.

83. Die Befehle *break* und *continue* verändern den sogenannten *Befehlszähler* (engl. program counter oder instruction pointer) des Ausführer.

84. Funktionen *muss* man und Prozeduren *darf* man mit einer *return*-Anweisung beenden.

85. In einer Funktion muss nach *return* immer ein *Ausdruck* stehen (z. B. so: `return 2*otto +5;`). Dieser Ausdruck muss zum *Ergebnistyp* der Funktion gehören. In einer Prozedur muss eine *return*-Anweisung immer genau wie folgt aussehen (ohne Ausdruck dahinter): `return;`

86. Eine Funktion *muss* mindestens eine *return*-Anweisung (und darf beliebig viele) enthalten. Eine Prozedur *darf* null (als Zahl: 0) oder mehr *return*-Anweisungen enthalten.

87. Mit einer *Blockanweisung* kann der Programmierer *mehrere* Anweisungen (und Vereinbarungen) zu *einer* Anweisung zusammenfassen.

88. Variablen, die innerhalb einer Blockanweisung vereinbart wurden, bezeichnet man auch als *lokale Variablen* (des Blocks).

89. Lokale Variablen eines Blocks werden vom Ausführer *zerstört*, sobald er den Block fertig ausgeführt hat.

90. Eine *if*-Anweisung der einfachsten Varianten besteht aus *zwei* Teilen, die wir als *Bedingung* und als *dann-Rumpf* bezeichnen.

91. Die drei Teile einer einfachen *if*-Anweisung mit *else*-Teil nennen wir *Bedingung*, *dann-Rumpf* und *sonst-Rumpf*.

92. Von einer komplizierten *if*-Anweisung *ohne else*-Teil wird *mindestens eine* der Bedingungen ausgewertet und *höchstens einer* der Rümpfe ausgeführt (möglicherweise wird *keiner* der Rümpfe ausgeführt).

93. Von einer komplizierten *if*-Anweisung *mit else*-Teil wird *mindestens eine* der Bedingungen ausgewertet und *genau einer* der Rümpfe ausgeführt (alle anderen Rümpfe werden *nicht* ausgeführt).

94. Wenn Sie von den Anweisungen *if* und *switch* nur eine mit in den Urlaub nehmen können, sollten Sie sich für die *if*-Anweisung entscheiden. Alle Probleme, die man mit einer *switch*-Anweisung lösen kann, lassen sich auch mit einer *if*-Anweisung lösen (die *switch*-Anweisung ist nur etwas kürzer und eleganter). Es gibt aber Probleme, die man mit einer *if*-Anweisung, aber nicht (oder nur auf sehr umständliche und indirekte Weise) mit einer *switch*-Anweisung lösen kann. Als Beispiel kann man versuchen, die folgende *if*-Anweisung durch eine *switch*-Anweisung zu ersetzen:

```
1 int n = ... ; // Wird irgendwie initialisiert
2 if (n < 0) {
3     pln("n ist kleiner als 0!");
4 } else if (n < 1000) {
5     pln("n ist kleiner als 1000!");
6 } else if (n < 2000) {
7     pln("n ist kleiner als 2000!");
8 }
```

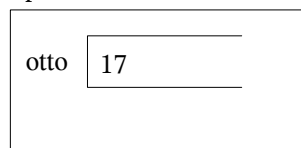
95. Ein *konstanter Ausdruck* ist einer, dessen Wert der Ausführer schon bei der *Übergabe* des Programms ("zur Compilezeit") berechnen kann (nicht erst bei der *Ausführung* des Programms, "zur Laufzeit"). Literale wie 0, 1, 17, 'a', 'b', 'c', 'A', 'B', 'C' und daraus mit Operatoren gebildete Ausdrücke wie -1, -17, 1+3, 'A' + 1 etc. sind Beispiele für konstante Ausdrücke.

96. Eine *Fallmarke* beginnt mit dem Schlüsselwort `case`. Dann kommt ein *konstanter Ausdruck* und zum Abschluss ein *Doppelpunkt*, z. B. so: `case 'A':`. Die Fallmarke `default:` ist ein Spezialfall.
97. Zwei kleine Schwächen der `switch`-Anweisung in Java:
1. Man muss (fast immer) jeden Fall mit einer `break`-Anweisung abschließen.
 2. Man kann viele "aufeinanderfolgende Fallmarken" wie z. B.
`case 1: case 2: case 3 ... case 100:`
nicht kurz und kompakt darstellen, etwa so: `case 1 to 100:`
98. Eine `while`-Schleife besteht aus zwei Teilen, einer *Bedingung* und einem *Rumpf*.
99. Man sollte (mindestens während der Entwicklung eines Programms) nur die *allgemeine Form* der `while`-Schleife
- ```

1 while (true) {
2 ...
3 if (...) break;
4 ...
5 }
```
- verwenden. Nur bei dieser Form kann man zusätzliche Befehle vor *und* nach der "Raussprungstelle" in Zeile 3 einfügen. Diese Möglichkeit ist z. B. beim Testen eines Programms wichtig, wenn man spezielle Testbefehle einfügen will.
100. Die beiden Spezialformen der `while`-Schleife haben keinen nennenswerten Vorteil (oder höchstens den, dass sie meist um eine Zeile kürzer sind als die allgemeine Form, und ältere Programmierer an ihre Jugend erinnern :-).
101. Eine `for`-Schleife besteht aus vier Teilen, die wir wie folgt benannt haben: *Initialisierungsteil*, *Bedingung*, *Fortschaltungsteil* und *Rumpf*. Die ersten drei Teile ("alle außer dem Rumpf") werden zusammen auch als *Kopf* der `for`-Schleife bezeichnet.
102. Die 4 Teile einer `for`-Schleife werden in folgender Reihenfolge ausgeführt:
- Initialisierungsteil, Bedingung, Rumpf, Fortschaltungsteil,  
 Bedingung, Rumpf, Fortschaltungsteil,  
 ...  
 Bedingung, Rumpf, Fortschaltungsteil,  
 Bedingung.
103. Der Rumpf einer `for`-Schleife wird mindestens null Mal ausgeführt (das soll heißen: Der Rumpf wird möglicherweise *nicht* ausgeführt).
104. Ja, man darf eine `for`-Schleife auch durch eine `break`-Anweisung abbrechen. Möglicherweise erleichtert ein Hinweis auf eine solche `break`-Anwei-

- sung in einem Kommentar am Anfang der `for`-Schleife den Kollegen das schnelle Verstehen der Schleife.
105. Im Prinzip darf man jeden der vier Teile einer `for`-Schleife leer lassen, aber wenn man alle vier Teile *gleichzeitig* leer läßt, hat man eine *Endlosschleife* programmiert, etwa so: `for (;;) {}` oder so: `for (;;) ;`. Die beiden Semikolons in den runden Klammern muss man *immer* notieren. Eine `for`-Schleife der Form `for (;;) { ... }` entspricht genau einer *while*-Schleife der Form `while (true) { ... }` und kann z. B. mit einer Anweisung der Form `if ( ... ) break;` im Rumpf beendet werden.
106. Die Variablenvereinbarung `int otto = 17;` auf Deutsch: "Erzeuge eine Variablen namens `otto` vom Typ `int` mit dem Anfangswert 17".
107. Mit Papier und Bleistift kann/sollten wir eine Variablenvereinbarung wie `int otto = 17;` ausführen, indem wir etwa Folgendes zeichnen:

Speicher



108. Wenn man eine Schleife programmiert, besteht (fast) immer die Gefahr, dass man aus Versehen eine *Endlosschleife* produziert. In Java besteht diese Gefahr nur dann *nicht*, wenn man die "neue und sichere `for`-Schleife" benutzt, die mit Java 5 eingeführt wurde. Die Gefahr der "Endlosigkeit" besteht bei *Schleifen* und bei sogenannten *rekursive Unterprogrammen* (die sich selbst aufrufen).
109. Mit einer sicheren `for`-Schleife wie etwa
- ```

6 int[] ir = ... ; // Wird irgendwie initialisiert
7 for (int n : ir) {
8     ... // Bearbeite n
9 }
```
- kann man nur Probleme lösen, bei denen alle Komponenten einer Reihung oder einer Sammlung bearbeitet werden sollen. Ausserdem kann man auf die Komponenten nur *lesend* zugreifen (man kann damit z. B. nicht alle Komponenten einer Reihung mit einem bestimmten Wert initialisieren).
110. Die folgende `while`-Schleife in den Zeilen 2 bis 5
- ```

1 int n = ... ; // Wird irgendwie initialisiert
2 while (true) {
3 ...
4 ...
5 }
```

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

ist *endlos*, wenn man *n* mit einem Wert initialisiert, der kleiner oder gleich 0 ist. Wird *n* mit einem *positiven* Wert initialisiert, wird der Schleifenrumpf nur *endlich* oft ausgeführt.

111. Der Rumpf der *for-i*-Schleife besteht aus einer einzigen Anweisung, nämlich aus der *for-j*-Schleife. Der Rumpf der *for-j*-Schleife besteht auch nur aus einer Anweisung, nämlich der *p*-Anweisung in Zeile 3, die ein Sternchen '\*' zur Standardausgabe ausgibt.

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

112. Eine Ausführung der *for-i*-Schleife besteht aus 5 Ausführungen ihres Rumpfes.

Eine Ausführung der *for-j*-Schleife besteht aus 20 Ausführungen ihres Rumpfes.

Eine Ausführung der *for-i*-Schleife bewirkt (5 mal 20 gleich) 100 Ausführungen des Rumpfes der *for-j*-Schleife.

113. Eine *goto*-Anweisung in Java wäre sinnvoll, weil nicht alle Java-Programme von *Menschen* geschrieben (und gelesen) werden. In Programmen, die von Generatoren oder Compilern erzeugt werden, sind *goto*-Befehle häufig sehr effizient und nützlich.

114. Man unterscheidet in Java (zuerst einmal) *primitive Typen* und *Referenztypen*.

115. Es gibt acht primitive Typen: *byte*, *char*, *short*, *int*, *long*, *float*, *double* und *boolean*.

116. Bei den Referenztypen unterscheidet man die folgenden drei Unterarten:

*Klassentypen*, *Schnittstellentypen* (engl. *interface types*) und *Reihungstypen* (engl. *array types*).

117. Es gibt nur *einen* (primitiven) nicht-numerischen Typ: *boolean*.

118. Es gibt sieben numerische Typen: *byte* bis *double* (in der obigen Aufzählung aller acht primitiven Typen).

119. Es gibt fünf Ganzzahltypen: *byte* bis *long*.

120. Es gibt zwei Gleitpunkttypen: *float* und *double*.

121. Ein *byte*-Wert ist 8 Bits lang. Somit gibt es  $2^8$  (gleich 256) *byte*-Werte.

122. Ein *char*-Wert belegt 16 Bits.

Somit gibt es  $2^{16}$  (etwa 65 Tausend) *char*-Werte.

123. Ein *short*-Wert belegt 16 Bits.

Somit gibt es  $2^{16}$  (etwa 65 Tausend) *short*-Werte.

124. Ein *int*-Wert belegt 32 Bits.

Somit gibt es  $2^{32}$  (etw 4 Milliarden) *int*-Werte.

125. Ein *long*-Wert belegt 64 Bits.

Somit gibt es  $2^{64}$  (etwa 18 Trillionen) *long*-Werte.

126. Ein *float*-Wert belegt 32 Bits.

Somit gibt es  $2^{32}$  (etw 4 Milliarden) *float*-Werte.

127. Ein *double*-Wert belegt 64 Bits.

Somit gibt es  $2^{64}$  (etwa 18 Trillionen) *double*-Werte.

128. Zum Typ *boolean* gehören zwei Werte, die man mit den Schlüsselworten *true* und *false* bezeichnet.

129. Leider repräsentiert *jeder* *int*-Wert eine *Zahl*. Es fehlt ein *int*-Wert, der *keine* Zahl repräsentiert, d.h. eine *int*-Unzahl (eng. *int-NaN*). Bei Gleitpunkttypen wie *float* und *double* hat es sich schon lange gut bewährt, dass einige Werte keine Zahlen repräsentieren, sondern Unzahlen (engl. *NaN values*, dabei steht *NaN* für *not a number*).

130. Zum Typ *double* gehören zwei Werte namens *+0* und *-0*, zwei weitere Werte namens *+infinity* und *-infinity* und zahlreiche Unzahlen (engl. *NaN values*). Bei den Namen *+0* und *+infinity* kann man das Vorzeichen *+* auch weglassen.

131. Es gibt *acht* Hüllklassen (für jeden primitiven Typ eine).

132. Die Hüllklassen für die (primitiven) Typen

*byte*, *char*, *int* und *float* heißen

*Byte*, *Character*, *Integer* und *Float*.

133. Nein, die meisten Ganzzahlen, die als *int*-Werte darstellbar sind, kann man *nicht* exakt durch *float*-Werte darstellen (man kann sie nur *näherungsweise* durch *float*-Werte darstellen, die z. B. um 1 oder um 60 zu groß oder zu klein sind).

134. Bei einer Typumwandlung wird aus einem Wert *w1* eines Typs *T1* ein entsprechender Wert *w2* eines Typs *T2* erzeugt (z. B. wird aus dem *double*-Wert 12.3 der *int*-Wert 12 erzeugt). Der *Typ* einer Variablen oder eines Wertes ist unveränderbar (und kann somit auch nicht *umgewandelt* werden).

135. Eine Typumwandlung von einem numerischen Typ *T1* zu einem numerischen Typ ist *erweiternd*, wenn es zu jedem Wert *w1* von *T1* einen Wert *w2* von *T2* gibt, der *gleich* *w1* oder zumindest *ungefähr gleich* *w1* ist.

136. Eine numerische Typumwandlung ist *verengend*, wenn sie nicht erweiternd ist.

137. *Erweiternde* numerische Typumwandlungen sind im allgemeinen *harmlos*, *verengende* Typumwandlungen sind im allgemeinen *gefährlich*.
138. Erweiternde numerische Typumwandlungen führt der Ausführer auch ohne ausdrücklichen Befehl des Programmierers durch (wenn er erkennt, das sie notwendig sind).
139. Nur zwischen *nah miteinander verwandten Typen* kann man Typumwandlungen mit *Cast-Befehlen* veranlassen. Zum Beispiel sind alle sieben (primitiven) *numerischen Typen* nah miteinander verwandt, aber keiner dieser numerischen Typen ist mit dem primitiven Typ `boolean` oder dem Referenztyp `String` nah verwandt.
140. Auch bei einer *erweiternden* numerischen Typumwandlung kann ein *Verlust an Genauigkeit* auftreten, wie etwa in den folgenden Beispielen:
- ```
1 float f01 = 2147483584;
2 float f02 = 2147483583;
```
- Beim Initialisieren der Variablen `f01` wird aus dem `int`-Wert 2147483584 der (um 64 größere) `float`-Wert 2147483648.0 erzeugt. Beim Initialisieren der Variablen `f02` wird aus dem `int`-Wert 2147483583 der (um 63 kleinere) `float`-Wert 2147483520.0 erzeugt.
141. Ein *Verlust der Größenordnung* kann bei *verengenden* numerischen Typumwandlungen (auf Grund eines *Cast-Befehls*) auftreten, wie etwa in den folgenden Beispielen:
- ```
1 char c1 = (char) -1;
2 short s1 = (short) 65535;
```
- Beim Initialisieren der Variablen `c1` wird aus dem `int`-Wert -1 (interne Darstellung: 1111 1111 1111 1111 1111 1111 1111 1111) der `char`-Wert 65535 (interne Darstellung: 1111 1111 1111 1111) erzeugt.
- Beim Initialisieren der Variablen `s1` wird aus dem `int`-Wert 65535 (interne Darstellung: 0000 0000 0000 0000 1111 1111 1111 1111) der `short`-Wert -1 (interne Darstellung: 1111 1111 1111 1111) erzeugt.
142. Ja, Typumwandlungen zwischen den Typen `boolean` und `int` sind möglich, aber nicht mit Hilfe von *Cast-Befehlen*, sondern nur mit selbst programmierten Befehlsfolgen (z. B. mit selbst geschriebenen Unterprogrammen). Welcher `boolean`-Wert dabei auf welchen `int`-Wert bzw. welche `int`-Werte auf welchen `boolean`-Wert abgebildet werden, muss vom Programmierer festgelegt werden (Java gibt keine spezielle Abbildung vor).

143. Ja, Typumwandlungen zwischen den (primitiven) numerischen Typen und dem Typ `String` sind möglich, aber nicht mit Hilfe von *Cast-Befehlen*, sondern mit Hilfe spezieller Methoden aus der Standardbibliothek (siehe z. B. die Methoden `valueOf` in der Klasse `String` und in den Hüllklassen) oder mit Hilfe selbst programmierter Methoden.
144. In der Mathematik spielen folgende *fünf* Zahlenmengen eine wichtige Rolle: Die Menge `N` der natürlichen Zahlen, die Menge `Z` der ganzen Zahlen, die Menge `Q` der rationalen Zahlen, die Menge `R` der reellen Zahlen und die Menge `C` der komplexen Zahlen. Jede dieser Zahlenmengen enthält *unendlich* viele Elemente und die Mengen sind *Teilmengen* voneinander wie folgt:  $N \subset Z \subset Q \subset R \subset C$ .
145. Der Java-Ausführer kennt *sieben* Zahlenmengen, nämlich die Wertemengen der numerischen Typen `byte`, `char`, `short`, `int`, `long`, `float` und `double`. Jede dieser Mengen enthält nur *endlich* viele Elemente und die Mengen sind paarweise *disjunkt* (kein Wert ist in mehreren dieser Mengen enthalten, denn die `byte`-3 unterscheidet sich in wichtigen Eigenschaften von der `char`-3, der `short`-3, der `int`-3, der `long`-3 und besonders von der `float`-3 und der `double`-3, und Entsprechendes gilt auch für alle anderen Werte).
146. Der Ausdruck `-0.0 * -0.0` hat den Wert `+0.0`.
147. Der Ausdruck `17.0 / Double.POSITIVE_INFINITY` hat den Wert `+0.0`.
148. Der Ausdruck `Double.NEGATIVE_INFINITY * +0.0` hat den Wert `NaN`.
149. Der Ausdruck `17.0 / +0.0` hat den Wert `Double.POSITIVE_INFINITY`.
150. Der Ausdruck `17.0 / -0.0` hat den Wert `Double.NEGATIVE_INFINITY`.
151. Die Konstanten `Character.MIN_VALUE` und `Character.MAX_VALUE` haben die Werte 0 bzw.  $2^{16}-1$  (ungefähr 65000).
152. Die Konstanten `Integer.MIN_VALUE` und `Integer.MAX_VALUE` haben die Werte  $-2^{31}$  (ungefähr -2 Milliarden) bzw.  $+2^{31}-1$  (ungefähr +2 Milliarden).
153. Der Ausdruck `Integer.MAX_VALUE + 1` hat den Wert `Integer.MIN_VALUE`.
154. Der Ausdruck `Integer.MIN_VALUE - 1` hat den Wert `Integer.MAX_VALUE`.



155. Der Ausdruck `-Integer.MIN_VALUE` (mit Minuszeichen davor) hat den Wert `Integer.MIN_VALUE` (ohne Minuszeichen davor).
156. Zum primitiven Typ `char` gehören die Ganzzahlen von 0 bis  $2^{16}-1$  (d. h. von 0 bis ungefähr 65000).
157. Wenn man den `int`-Wert 65 zum Bildschirm ausgibt, erscheint dort die Ziffernfolge 65.
158. Wenn man den `char`-Wert 65 zum Bildschirm ausgibt, erscheint dort das Zeichen A.
159. Wenn man die folgenden Zahlen als `char`-Werte zum Bildschirm ausgibt:  
 48 49 57 90 97 98 122 erscheinen dort die folgenden Zeichen:  
 0 1 9 Z a b z .
160. Vereinbarung einer `char`-Variablen `alter` mit dem Anfangswert 68:
- ```
1 char alter = 68; // So sollte man es machen!
2 char alter = 'D'; // So sollte man es nicht machen!
```
161. Vereinbarung einer `char`-Variablen `buchstabe` mit dem Anfangswert 'D':
- ```
3 char buchstabe = 'D'; // So sollte man es machen!
4 char buchstabe = 68; // So sollte man es nicht machen!
```
162. Jede Variable besteht aus mindestens zwei Teilen, aus einer *Referenz* und einem *Wert*.
163. Zusätzlich kann eine Variable einen *Namen* oder einen *Zielwert* (oder beides) haben.
164. Drei Teile einer Variablen gehören zur Welt des *Ausführers* (die Referenz, der Wert und der Zielwert, wenn vorhanden). Ein Teil einer Variablen gehört zur Welt des *Programmierers* (der Name, wenn vorhanden).
165. In einer Bojendarstellung werden Referenzen in *sechseckige* Kästchen eingezeichnet.
166. In einer Bojendarstellung werden die Werte von Variablen in *viereckige* Kästchen (genauer: in Rechtecke) eingezeichnet.
167. In der Bojendarstellung einer Referenzvariablen wird der Wert der Variablen in ein viereckiges Kästchen gezeichnet (weil er der Wert einer Variablen ist) und in ein sechseckiges Kästchen (weil dieser Wert eine Referenz ist).
168. Durch eine Zuweisung `x = ...` ; wird immer der *Wert* der Variablen `x` verändert (nicht ihr Name oder ihre Referenz). "Verändern" soll hier den Fall einschliessen, dass der neue Wert der Variablen `x` gleich ihrem alten Wert ist (zum Beispiel wenn man einer `int`-Variablen mit dem Wert 17 den Wert 17 zuweist). Wenn man den Wert einer Referenzvariablen ändert, ändert man damit meistens auch ihren Zielwert (ausser, wenn man einer Referenzvariablen mit dem Wert 123 den Referenzwert 123 zuweist).

169. Welcher Teil einer Variablen `x` durch einen Ausgabebefehl wie `println(x)` ; oder `System.out.println(x)` ; ausgegeben wird, hängt von der Art der Variablen `x` ab: Ist `x` eine *primitive* Variable, so wird ihr *Wert* ausgegeben. Ist `x` eine *Referenzvariable*, so wird ihr *Zielwert* ausgegeben.  
 Ausnahme: Wenn `x` den Wert `null` hat (was nur bei Referenzvariablen möglich ist) wird ihr Wert ausgegeben (in Form der Zeichenkette `null`).
170. Bei einem Vergleich `x == y` werden immer die *Werte* der Variablen verglichen (nicht ihre Namen oder Referenzen oder Zielwerte). Entsprechendes gilt natürlich auch für Vergleiche wie `x != y`. Solche Vergleiche sind nur erlaubt, wenn die Variablen `x` und `y` zum selben Typ (oder zu "nah verwandten" Typen) gehören.
171. Ein solcher Kampf hat nie stattgefunden, weil die meisten Autos gar nicht boxen können. Achten Sie auch in dieser Sammlung auf solche gefährlichen Fangfragen :-).
172. Mit einem *Ausdruck* befiehlt der Programmierer dem Ausführer, einen *Wert* zu *berechnen* (z. B. den Wert des Ausdrucks `x + 1`). Hier sind *Wert* und *berechnen* zwei Fachbegriffe, die man möglichst *nicht* durch ähnlich klingende Worte ersetzen sollte.
173. Wenn `x` eine `int`-Variable mit dem Wert 3 ist, dann hat der Ausdruck `5 * x / 10` den `int`-Wert 1 (wenn der Ausführer mit *Ganzzahlen* rechnet, ist `15 / 10` gleich 1, und nicht gleich 1.5!).
174. Der Ausdruck 123 gehört zum Typ `int` und hat den Wert 123.
175. Der Ausdruck 0.1 gehört zum Typ `double`. Jede Java-ProgrammiererIn sollte wissen, dass er *nicht* genau den Wert 0.1 hat (sondern einen etwas größeren Wert, S. 23 und 109).
176. Den Wert des Ausdrucks `x` berechnet der Ausführer, indem er in die Variable (oder: in den Wertebehälter) `x` hineinschaut. Der Wert, den er dort sieht, ist das Ergebnis seiner "Berechnung". Zur Erinnerung: Eine Variable kann nicht leer sein, sondern enthält immer einen Wert.
177. Drei Befehle (eine *Vereinbarung*, eine *einfache Anweisung* und eine *zusammengesetzte Anweisung*) in denen der Ausdruck `2.5 * y + 3.2` vorkommt:
- ```
1 double dora = 2.5 * y + 3.2 ;
2 dora = (2.5 * y + 3.2) / 2.0;
3 if (2.5 * y + 3.2 > 0.0) println( 2.5 * y + 3.2);
```
- In der `if`-Anweisung kommt der Ausdruck `2.5 * y + 3.2` sogar zweimal vor.
178. Mit einem normalen Ausdruck (z. B. `x + 1`) befiehlt man dem Ausführer nur, *einen Wert* zu *berechnen* (aber *nicht*, den Inhalt irgendwelcher Wertebehälter zu verändern). Mit einem *Ausdruck mit Seiteneffekt* (z. B. `x++`) befiehlt

man dem Ausführer, einen Wert zu berechnen *und* den Inhalt bestimmter Wertebehälter (im Beispiel `x++`: den Inhalt der Variablen `x`) zu verändern. Ein *Ausdruck mit Seiteneffekt* ist also ein *Ausdruck*, der gleichzeitig den Charakter einer *Anweisung* hat.

179. Wenn `n` eine `int`-Variable mit dem Wert 5 ist, dann hat der Ausdruck `n+1` den Wert 6, aber *nach* der Auswertung des Ausdrucks hat die Variable `n` immer noch den Wert 5.
180. Wenn `n` eine `int`-Variable mit dem Wert 5 ist, dann hat der Ausdruck `n++` den Wert 5, und *nach* der Auswertung des Ausdrucks hat die Variable `n` den Wert 6.
181. Wenn `n` eine `int`-Variable mit dem Wert 5 ist, dann hat der Ausdruck `++n` den Wert 6, und *nach* der Auswertung des Ausdrucks hat die Variable `n` den Wert 6.
182. Der Ausdruck `new StringBuilder("ABC")` hat den *Seiteneffekt*, dass ein neues `StringBuilder`-Objekt erzeugt wird. Dadurch wird "der Wertebehälter, der alle Objekte enthält" (Kurzname: Halde, engl. heap) verändert. Der Wert des Ausdrucks `new StringBuilder("ABC")` ist eine Referenz, die auf das neu erzeugte `StringBuilder`-Objekt zeigt.
183. Ein *Literal* ist ein Name für einen Wert.
184. Für jeden Referenztyp gibt es *ein* Literal `null`. Außerdem gibt es für jeden der folgenden Typen (mehrere, meist ziemlich viele) Literale:
`char`, `int`, `long`, `float`, `double`, `boolean`, `String`.
185. Für die primitiven Typen `byte` und `short` gibt es *keine* Literale.
186. Acht Literale und ihre Typen:

Literal:	"true"	false	"7"	'7'	7	7L	7.0	7.0F
Typ:	String	boolean	String	char	int	long	double	float

187. Die folgenden drei Literale gehören alle zum Typ `int` und jedes hat den Wert zehn: `10` `012` `0xA` (ein dezimales, ein oktales und ein hexadezimaler Literal).
188. Die hexadezimalen Literale `0x7FFFFFFF` und `0x80000000` bezeichnen den größten bzw. den kleinsten `int`-Wert (`+2147483647` bzw. `-2147483648`).
189. Nein, `-123` ist kein Literal, sondern ein *zusammengesetzter Ausdruck*, der aus dem Minusoperator `-` und dem `int`-Literal `123` besteht. Der Wert dieses Ausdrucks gehört natürlich zum Typ `int`.
190. Nein, `+1.5` ist kein Literal, sondern ein *zusammengesetzter Ausdruck*, der aus dem Plusoperator `+` und dem `double`-Literal `1.5` besteht. Der Wert dieses Ausdrucks gehört natürlich zum Typ `double`.

191. Drei `char`-Literale, die ein großes A bezeichnen:
`'A'` // Ein normales Zeichenliteral
`'\u0041'` // Ein Unicode-Literal (hex-Zeichenliteral)
`'\101'` // Ein oktales Zeichenliteral
192. Drei `char`-Literale die je *einen* Rückwärtsschrägstrich bezeichnen:
`'\'` // Ein normales Zeichenliteral
`'\u005C\u005C'` // Ein Unicode-Literal (hex-Zeichenliteral)
`'\134'` // Ein oktales Zeichenliteral
193. Das String-Literal `"ABCD"` enthält 4 Zeichen (4 Buchstaben).
194. Das String-Literal `"\\\\"` enthält 2 Zeichen (2 Rückwärtsschrägstriche).
195. Das String-Literal `"\" \" \" "` enthält 3 Zeichen (3 doppelte Anführungszeichen).
196. Verbesserung von Ganzzahlliteralen: Es sollte eine Möglichkeit geschaffen werden, auch große Ganzzahlliterale *lesbar* zu notieren, z. B. `1_000_000_000` anstelle von `1000000000`. Da eine solche Erweiterung der Sprache keinen Einfluß auf schon existierende Java-Programme hätte, ist sie nicht prinzipiell unmöglich. Eine Abschaffung der *oktalen Literale* (z. B. `012` mit dem Wert 12) ist leider kaum möglich, weil sie in zahlreichen bereits existierenden Programmen vorkommen.
197. Welchen Wert ein bestimmtes *Literal* (z. B. `012` oder `0.1` etc.) bezeichnet, wird von der Sprache Java (oder: vom Java-Ausführer) festgelegt. Welchen Wert eine *Konstante* (z. B. `MWST` oder `PI`) bezeichnet, wird vom *Programmierer* festgelegt.
198. Eine ("echte") *Konstante* hat (ähnlich wie ein Literal) *keine Referenz*, sondern besteht nur aus einem *Namen* und einem *Wert*. Eine *unveränderbare Variable* besteht (wie andere Variablen auch) mindestens aus einer *Referenz* und einem *Wert* (und hat meistens auch einen Namen). Da es in Java keine "echten Konstanten" gibt, bezeichnen Java-Programmierer *unveränderbare Variablen* häufig als *Konstanten*. In C/C++ gibt es echte Konstanten und unveränderbare Variablen.
199. Zwei einfache Ausdrücke: `17` und `n`. Zwei zusammengesetzte Ausdrücke: `n+17` und `5 * (n + 3) / (n * n)`.
200. Ein *Operator* ist ein Name (für eine Operation), der typischerweise aus Sonderzeichen besteht (z. B. `+`, `-`, `+=`, `==`, `!=`, `<=` etc.) und auf eine besondere Weise notiert wird (in Präfix-, Infix-, Postfix- oder Mixfix-Notation).
201. Eine *Operation* ist eine Funktion (ein Unterprogramm, welches einen Wert liefert) mit einem *Operator* als Namen.
202. In Java (und in vielen anderen Sprachen) sind die meisten arithmetischen Operationen *zweistellig*.

203. Die *Infixnotation* hat sich seit dem Mittelalter in Europa und auf der ganzen Erde verbreitet (und bis heute gibt es noch keinen Impfstoff dagegen :-).
204. Die *Infixnotation* wirft das Problem der *Assoziativität* auf ("Ist $9-3-2$ gleich 8 oder gleich 4?") und das Problem der *Bindungsstärke* ("Ist $2+4*3$ gleich 18 oder gleich 14?").
205. Man hat versucht, die Probleme der Infixnotation durch *Konventionen* (d. h. durch zusätzliche, ziemlich willkürliche Regeln) zu lösen.
206. Der Versuch, die Probleme der Infixnotation zu lösen, ist weitgehend (aber nicht ganz vollständig) geglückt. Er kostet allerdings jedes Jahr viele Arbeitsstunden (wenn SchülerInnen unter anderem die Regel "Punktrechnung geht vor Strichrechnung" büffeln, statt Mathematik zu lernen oder eine Disco zu besuchen :-).
207. Dem ungeklammerte Ausdruck $n1 - n2 - n3$ entspricht der voll geklammerte Ausdruck $((n1 - n2) - n3)$ (weil der Subtraktionsoperator $-$ linksassoziativ ist).
208. Dem ungeklammerte Ausdruck $b1 || b2 \&\& b3$ entspricht der voll geklammerte Ausdruck $(b1 || (b2 \&\& b3))$ (weil der Und-Operator $\&\&$ stärker bindet als der Oder-Operator $||$).
209. Dem ungeklammerte Ausdruck $v1 = v2 = v3 = v4$ entspricht der voll geklammerte Ausdruck $(v1 = (v2 = (v3 = v4)))$ (weil der Zuweisungsoperator $=$ rechtsassoziativ ist).
210. Der Ausdruck $b1 == b2 == b3$ ist genau dann syntaktisch korrekt, wenn die Variablen $b1$, $b2$ und $b3$ zum Typ `boolean` gehören.
211. Dem ungeklammerten Ausdruck $b1 == b2 == b3$ entspricht der voll geklammerte Ausdruck $((b1 == b2) == b3)$ (weil der Gleichheitsoperator $==$ linksassoziativ ist).
212. In Java gibt es Operatoren mit 13 verschiedenen Bindungsstärken.
213. Die folgenden 14 Operatoren bezeichnen Operationen *mit Seiteneffekt*:
 $++$ $--$ $=$ $+=$ $-=$ $*$ $/=$ $<=<$ $>>=$ $>>>=$ $\% =$ $\& =$ $\wedge =$ $| =$
 Alle anderen Operatoren ($+$ $-$ $*$ $/$ $\%$ $<<$ $>>$ $>>>$ $==$ $!=$ etc.) bezeichnen Operationen *ohne Seiteneffekt*.
214. Den Operator $++$ darf man *vor* bzw. *nach* Variablen aller (sieben) numerischen Typen verwenden. Also bezeichnet er (2 mal 7 gleich) 14 verschiedene Operationen.
215. 5000 double-Variablen kann man etwa so in Form einer *Reihung* vereinbaren:
- ```
1 double[] ronald = new double[5000];
```
- Alternativ dazu kann man die Variablen auch einzeln vereinbaren und dabei Blasen z. B. mit Hilfe von Lederhandschuhen vermeiden :-).

216. Die Variablen, aus denen eine Reihung besteht, bezeichnen wir als die *Komponenten* der Reihung. Einige Autoren ziehen die Bezeichnung *Elemente* vor. Wir werden die Größen, die innerhalb einer *Klasse* vereinbart werden (engl. the members of the class) als *die Elemente der Klasse* bezeichnen) und die Komponenten einer Reihung deshalb *nicht* als Elemente bezeichnen.
217. Bei einer Reihung der Länge 20 laufen die Indizes von 0 bis 19.
218. Eine Reihung, deren letzter Index gleich 99 ist, besteht aus 100 Komponenten (oder: eine solche Reihung hat die Länge 100).
219. Ein Zugriff auf die nullte Komponente  $r[0]$  einer Reihung  $r$  kostet *gleich viel Zeit* wie ein Zugriff auf die 4999-ste Komponente  $r[4999]$  oder ein Zugriff auf irgendeine andere Komponente von  $r$ .
220. Reihungen sind aus *Beton*, nicht aus *Gummi* (d. h. wenn der Ausführer eine Reihung fertig erzeugt hat, liegt ihre Länge fest und kann nicht mehr geändert werden).
221. Die Länge von Reihungen misst man (in Java) immer in *Komponenten*, nicht in Zentimetern, in Bytes oder in Bits (die *Länge einer Reihung* ist gleich der *Anzahl ihrer Komponenten*).
222. Der Befehle `pln(r.length);` gibt die Länge der Reihung  $r$  aus.
223. Das `length`-Attribut `r.length` einer Reihung  $r$  gehört zum Typ `int`. Deshalb kann eine Reihung höchstens `Integer.MAX_VALUE` viele Komponenten enthalten (das sind etwa 2.15 Milliarden Komponenten oder ca. 2 Giga Komponenten).
224. Die folgende sichere und einfache `for`-Schleife gibt die Komponenten der Reihung `zeilenReihung` (mit Komponenten vom Typ `String`) zur Standardausgabe aus:
- ```
1 for (String zeile: zeilenReihung) {
2     pln(zeile);
3 }
```
225. Es folgt die Vereinbarung einer Reihung von 5 `int`-Variablen:
- ```
4 int[] ir01 = {10, 20, 30, 40, 50};
```
226. Es folgt die Vereinbarung einer Reihung von 4 `boolean`-Variablen:
- ```
5 boolean[] br01 = {true, true, false, true};
```
227. Es folgt die Vereinbarung einer Reihung von 3 `String`-Variablen:
- ```
1 String[] sr02 = {
2 new String("Anna"),
3 new String("Bert"),
4 new String("Carola"),
5 };
```

Das Komma am Ende von Zeile 4 ist nicht notwendig, aber erlaubt. Es folgt die deutlich kürzere Vereinbarung einer ganz ähnlichen Reihung namens `sr03`:

```
6 String[] sr03 = {"Anna", "Bert", "Carola"};
```

Der Unterschied zwischen einem `String`-Objekt wie "Anna" und einem `String`-Objekt wie `new String("Anna")` wird im Buch im Beispiel-02 auf Seite 230 (Abschnitt 10.2) erläutert.

228. Eine Reihung wie `sr02` (siehe vorige Antwort) enthält genau genommen keine Objekte, sondern nur *Referenzvariablen*, die auf Objekte zeigen (oder den Wert `null` haben).

229. In einer vereinfachten Bojendarstellung einer Reihung kann man die *Referenzen der Komponenten* und das `length`-Attribut weglassen. Von den Komponenten zeichnet man dann nur noch die Wertekästchen (und in der Nähe davon die zugehörigen Indizes).

230. Die Komponenten einer Reihung des Typs `float[]` werden vom Ausführer automatisch mit dem Wert `0.0F` initialisiert und die Komponenten einer `boolean`-Reihung mit dem Wert `false`.

231. Die Reihung `sr02` wird in den folgenden drei Schritten erzeugt:

Schritt 1: Eine Referenzvariable `sr02` wird erzeugt und mit `null` initialisiert.

Schritt 2: Eine Reihung von Referenzvariablen wird erzeugt, ihre Komponenten werden mit `null` initialisiert und die Referenz der gesamten Reihung wird der Variablen `sr02` als neuer Wert zugewiesen.

Schritt 3: Die Komponenten der Reihung werden mit Referenzen initialisiert, die auf die `String`-Objekte "Anna", "Bert" und "Carola" zeigen.

232. Drei Weisen, eine Reihungsvariable wie `sr02` (mit Referenzvariablen als Komponenten, nicht mit primitiven Komponenten) zu kopieren:

**Kopierweise 1:** Man vereinbart eine neue Reihungsvariable und initialisiert sie mit dem Wert von `sr02` (oder kopiert den Wert der Variablen `sr02` mit einer Zuweisungsanweisung in die neue Reihungsvariable). Danach zeigen `sr02` und die neue Reihungsvariable auf dieselbe Reihung.

**Kopierweise 2:** Man erzeugt eine neue Reihungsvariable, die auf eine neue Reihung zeigt, und kopiert mit der Prozedur `System.arraycopy` die Werte der Komponenten von `sr02` in die neue Reihung. Danach zeigen die Komponenten von `sr02` und die Komponenten der neuen Reihung auf dieselben Komponentenobjekte (im Falle von `sr02` aus der vorigen Antwort sind das drei `String`-Objekte). Diese Kopierweise wird auch als *flaches Kopieren* bezeichnet.

**Kopierweise 3:** Man erzeugt eine neue Reihungsvariable, die auf eine neue Reihung zeigt, deren Komponenten auf Kopien der alten Komponentenobjekte zeigen. Diese Kopierweise wird auch als *tiefes Kopieren* bezeichnet.

233. Es folgt die Vereinbarung einer Reihung `frra` von drei Reihungen, von denen jede fünf `float`-Variablen enthält:

```
7 float[][] frrA = new float[3][5];
```

Die (insgesamt 15) `float`-Variablen werden vom Ausführer automatisch mit dem Wert `0.0F` initialisiert.

234. Es folgt die Vereinbarung einer Reihung `frrB`, die drei Reihungen enthält, die fünf bzw. drei bzw. vier `float`-Variablen enthalten:

```
1 float[][] frrB = {
2 new float[5],
3 new float[3],
4 new float[4],
5 };
```

Die (insgesamt 5+3+4 gleich 12) `float`-Variablen werden vom Ausführer automatisch mit dem Wert `0.0F` initialisiert. Das Komma am Ende von Zeile 4 ist nicht notwendig, aber erlaubt.

235. Es folgt die Vereinbarung einer Reihung `frrC`, die ebenfalls (wie `frrB` in der vorigen Antwort) drei Reihungen enthält, die fünf bzw. drei bzw. vier `float`-Variablen enthalten. Die 12 `float`-Variablen in `frrC` werden mit den Werten `1.5F`, `2.5F`, `3.5F`, ..., `12.5F` initialisiert:

```
1 float[][] frrC =
2 {{1.5F, 2.5F, 3.5F, 4.5F, 5.5F},
3 {6.5F, 7.5F, 8.5F},
4 {9.5F, 10.5F, 11.5F, 12.5F},
5 };
```

Das Komma am Ende von Zeile 4 ist nicht notwendig, aber erlaubt.

236. Eine (sichere, einfache) `for`-Schleife (mit einer geschachtelten `for`-Schleife darin), die die Summe aller `float`-Werte in der Reihung `frrC` (siehe vorige Antwort) berechnet:

```
6 float summe = 0;
7 for (float[] fr: frrC) {
8 for (float f: fr) {
9 summe += f;
10 }
11 }
```

237. Eine (unsichere?, alte?, klassische?) `for`-Schleife (mit einer geschachtelten `for`-Schleife darin), die die Werte aller `float`-Variablen in der Reihung `frrC` verdoppelt:

```
12 for (int i=0; i<frrC.length; i++) {
13 for (int j=0; j<frrC[i].length; j++) {
```

```
14 frrC[i][j] *= 2;
15 }
16 }
```

238. *Mehrstufige Reihungen* werden im Englischen als *nested arrays* bezeichnet (und damit klar von *multidimensional arrays* unterschieden).

239. Bei einer Reihung vom Typ `float[][][]` sind die *Komponenten* vom Typ `float[][]` und die *elementaren Komponenten* vom Typ `float`.

240. Bei einer einstufigen Reihung wie `frA` (vom Typ `float[]`) sind die Komponenten bereits elementar. Somit sind die Komponenten und die elementaren Komponenten von `frA` identisch und vom Typ `float`.

241. Im Abschnitt 7.4 des Buches findet man auf S. 167 im Beispiel-06 eine Reihung `orXs`, die gleichzeitig ein-, zwei- und dreistufig ist. Wenn man in der Vereinbarung der Reihung die beiden elementaren Komponenten `new Long(17)` und "Ende" aus der Initialisierung entfernt, ist die Reihung "nur noch" zwei- und dreistufig.

242. Die Reihung `dreig` vom Typ `int[][]` (siehe unten Zeile 20) enthält drei *gleiche* Komponenten und die Reihung `dreiid` (ebenfalls vom Typ `int[][]`) drei *identische* Komponenten ("dreimal dieselbe Komponente"):

```
17 int[] komp1 = {2, 4, 6, 8};
18 int[] komp2 = {2, 4, 6, 8};
19 int[] komp3 = {2, 4, 6, 8};
20 int[][] dreig = {komp1, komp2, komp3};
21 int[][] dreiid = {komp1, komp1, komp1};
```

243. Nein und Ja. Im Kern der Sprache Java gibt es nur *mehrstufige* Reihungen, aber *keine mehrdimensionalen* Reihungen. Die Firma IBM hat aber einige Klassen geschrieben (und allen Java-ProgrammiererInnen zur Verfügung gestellt), deren Objekte sich genau wie mehrdimensionale Reihungen verhalten (und die man deshalb als mehrdimensionale Reihungen bezeichnen kann, nach der Regel: "Wenn etwas aussieht wie ein Elefant und riecht wie ein Elefant, dann ist es ein Elefant" :-).

244. *Mehrdimensionale* Reihungen haben (im Vergleich zu *mehrstufigen* Reihungen) den Vorteil, dass Zugriffe auf ihre elementaren Komponenten schneller sind und die Reihungen insgesamt (ein bisschen) weniger Speicherplatz belegen.

245. *Mehrstufige* Reihungen haben (im Vergleich zu mehrdimensionalen Reihungen) den Vorteil, dass sie flexibler und vielseitiger sind, weil ihre Komponenten (die selbst ja auch Reihungen sind) gleich lang oder *verschieden lang* sein können ("Reihungen mit Flatterrand sind erlaubt"). Die Komponenten von *mehrdimensionalen* Reihungen müssen dagegen immer Reihungen gleicher Länge sein (man sagt auch: Sie müssen *rechteckig* sein, wie eine Matrix).

246. Drei Reihungsvariablen:

```
22 float[] r01 = {1.5F, 2.6F, 3.7F};
23 float[] r02 = {1.5F, 2.6F, 3.7F};
24 float[] r03 = r02;
```

Die Reihungen `r01` und `r02` sind gleich (aber nicht identisch). Die Reihungen `r02` und `r03` sind (nicht nur gleich sondern) identisch. Man kann auch sagen: Die Referenzvariablen `r02` und `r03` haben gleiche Werte und zeigen damit auf dieselbe Reihung (von `float`-Variablen).

247. Wenn man zwei Reihungsvariablen `r01` und `r02` mit dem Gleichheitsoperator `==` vergleicht, etwa so: `r01 == r02`, werden ihre *Werte* miteinander verglichen (nicht ihre Namen, Referenzen oder Zielwerte). Die Werte von Reihungsvariablen sind *Referenzen*, die auf eine Reihung zeigen (oder gleich `null` sind).

248. Zwei Reihungen `r01` und `r02` müssen *identisch* sein, damit der Ausdruck `r01 == r02` den Wert `true` hat.

249. Methoden aus der Klasse `java.util.Array`: `equals`, `deepEquals`, `fill`, `sort`, `binarySearch`, `toString`, `deepToString`.

250. Für die Bearbeitung von *mehrstufigen* Reihungen sind die Methoden `deepEquals` und `deepToString` gedacht.

251. In der Klasse `java.util.Arrays` gibt es 9 Methoden namens `equals`, mit denen man jeweils zwei Objekte der folgenden Typen vergleichen kann: `byte[]`, `char[]`, `short[]`, `int[]`, `long[]`, `float[]`, `double[]`, `boolean[]`, `Object[]`.

252. Zwei Reihungen `r01` und `r02` müssen ("Komponente für Komponente") *gleich* sein, damit der Ausdruck `Arrays.equals(r01, r02)` den Wert `true` hat.

253. Wenn man zwei Reihungsvariablen `r01` und `r02` mit der Methode `Arrays.equals` vergleicht, werden ihre *Zielwerte* miteinander verglichen ("Komponente für Komponente")!

254. Zwei Reihungen `r03` und `r04`:

```
1 int[][] r03 = {{1, 2}, null};
2 int[][] r04 = {{1, 2}, null};
```

Für diese Reihungen gilt, dass der Ausdruck

`Arrays.equals(r03, r04)` den Wert `false` und der Ausdruck

`Arrays.deepEquals(r03, r04)` den Wert `true` hat.

Diese Ergebnisse ändern sich nicht, wenn man beim Initialisieren von `r03` und `r04` (in Zeile 1 und 2) anstelle von `null` z. B. den Reihungsinitialisierer

{3, 4, 5} angibt. Indem man die Reihungen als Bojen darstellt, kann man sich die Vergleichsergebnisse veranschaulichen.

255. Zwei Reihungen r05 und r06:

```
3 int[][] r05 = {{1, 2}, null};
4 int[][] r06 = {r05[0], r05[1]};
```

Für diese Reihungen gilt, dass der Ausdruck `Arrays.equals(r05, r06)` den Wert `true` und der Ausdruck `Arrays.deepEquals(r05, r06)` auch den Wert `true` hat. Diese Ergebnisse ändern sich nicht, wenn man beim Initialisieren von `r05` (in Zeile 3) anstelle von `null` z. B. den Reihungsinitialisierer {3, 4, 5} angibt. Indem man die Reihungen als Bojen darstellt, kann man sich die Vergleichsergebnisse veranschaulichen.

256. Eine Vereinbarung und eine Anweisung, die beide mit  
= 2.0F \* (17.5F \* 4.0F); enden:

```
5 float florian = 2.0F * (17.5F * 4.0F); // Eine Vereinbarung
6 florian = 2.0F * (17.5F * 4.0F); // Eine Anweisung
```

257. Zwischen dem *Initialisierungsteil einer Variablenvereinbarung* und einer *Zuweisung* muss man (trotz oder gerade wegen ihrer Ähnlichkeit) sorgfältig unterscheiden, weil es Regeln gibt, die nur für das eine oder für das andere Konstrukt gelten.

258. Ein *Initialisierungsteil*, zu dem es keinen entsprechenden *Zuweisungsbefehl* gibt:

```
7 int[] r07 = {1, 2, 3}; // Initialisierung, erlaubt
8 r07 = {1, 2, 3}; // Zuweisung, verboten!
```

259. Bei einem Auto ist der Benzinverbrauch pro Stunde in aller Regel *kovariant* zur Geschwindigkeit: Je schneller das Auto fährt, desto mehr Benzin verbraucht es pro Stunde.

260. Bei einem Auto ist der Füllstand des Tanks *kontravariant* zur (seit dem letzten Tankstopp) zurückgelegten Strecke: Je größer die zurückgelegte Strecke ist, desto kleiner ist der Füllstand.

261. Das Alter eines Autos (gemessen in Minuten) und sein Gewicht ("mit allem drum und dran") sind in aller Regel weder *ko-* noch *kontravariant*, da das Alter stetig zunimmt, während das Gewicht hin- und herschwankt: Wenn man tankt oder viele Passagiere einsteigen lässt, nimmt das Gewicht zu, wenn man Benzin verbraucht oder Passagiere aussteigen nimmt das Gewicht ab etc.

262. Fasst man Typen als *Wertemengen* auf, so liegt es nahe, Folgendes festzulegen: Ein Typ T1 ist *größer* als ein Typ T2, wenn die Wertemenge T1 in der

Wertemenge T2 *enthalten* ist. Nach dieser Festlegung gibt es viele Typen T1 und T2, bei denen T1 weder größer noch kleiner als T2 ist. Zum Beispiel ist die Wertemenge des Typs `boolean` weder größer noch kleiner als die Wertemenge des Typs `int` und Entsprechendes gilt auch für die Wertemengen der Typen `short` und `char` (es gibt keinen `short`-Wert 65000, aber auch keinen `char`-Wert -1). Dagegen ist die Wertemenge von `int` in der Wertemenge von `long` enthalten und damit ist der Typ `long` größer als der Typ `int`.

263. Eine *Methode* ist ein Unterprogramm, welches innerhalb einer Klasse vereinbart wurde. In Java sind *alle* Unterprogramme auch Methoden, in C++ gibt es außer Methoden auch Unterprogramme, die *keine* Methoden sind.

264. In einem Java-Programm darf man Methoden an folgenden Stellen vereinbaren (Ja) bzw. nicht vereinbaren (Nein):

|                                         |         |
|-----------------------------------------|---------|
| Unmittelbar am Anfang einer Quelldatei? | Nein    |
| Innerhalb einer anderen Methode?        | Nein!!! |
| Innerhalb einer Klasse?                 | Ja!     |
| Innerhalb von Schleifen?                | Nein    |
| Unmittelbar nach jeder Zuweisung?       | Nein    |

265. Der Programmierer muss eine Methode *vereinbaren* (vom Ausführer erzeugen lassen) und er darf sie *aufrufen* (vom Ausführer ausführen lassen).

266. Der Programmierer muss eine Methode genau *einmal* vereinbaren (weil sie sonst nicht erzeugt wird) und darf sie dann *beliebig oft* (nullmal, einmal, zweimal, dreimal, ..., 10000 Mal, ... etc.) aufrufen.

267. Der folgende Befehl

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

ist eine *Vereinbarung* (genauer: eine Methodenvereinbarung). Ins Deutsche übersetzt lautet dieser Befehl etwa so:

"Erzeuge eine Methode namens `printHallo` mit dem Ergebnistyp `void` und null Parametern. Der Rumpf der Methode soll nur aus dem einen Befehl

```
System.out.println("Hallo!");
```

bestehen!"

Der Vereinbarungsbefehl in Zeile 1 bis 3 bewirkt also, dass eine Methode (namens `printHallo`) *erzeugt* wird. Er bewirkt aber *nicht*, dass die Methode `printHallo` ausgeführt wird. Er bewirkt insbesondere *nicht*, dass irgendetwas (z. B. der String "Hallo!") zum Bildschirm ausgegeben wird.

268. Mit dem folgenden Befehl kann der Programmierer dem Ausführer befehlen, die Methode `printHallo` (die oben in Zeile 1 bis 3 vereinbart wurde) auszuführen:

```
4 printHallo();
```

Dieser Methodenaufruf ist eine *Anweisung*, denn er verändert den Inhalt des Wertebehälters Bildschirm.

269. Jeder der folgenden drei Befehle

```
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(); }
```

ist eine (Methoden-) *Vereinbarung*. Ins Deutsche übersetzt lauten diese Befehle etwa so:

Zeile 1: "Erzeuge eine Methode namens `pln` mit dem Ergebnistyp `void` und einem Parameter `ob` vom Typ `Object`. Der Rumpf dieser Methode soll ... !"

Zeile 2: "Erzeuge eine Methode namens `p` mit dem Ergebnistyp `void` und einem Parameter `ob` vom Typ `Object`. Der Rumpf dieser Methode soll ... !"

Zeile 3: "Erzeuge eine Methode namens `pln` mit dem Ergebnistyp `void` und null Parametern. Der Rumpf dieser Methode soll ... !"

Die drei Befehle bewirken, dass drei Methoden (zwei namens `pln` und eine namens `p`) erzeugt werden.

270. Das folgende Beispiel dient zur Illustration (der Frage und) der Antwort. Es enthält *eine Methodenvereinbarung* (in Zeile 1 bis 3) und *zwei Methodenaufrufe* (in den Zeilen 7 und 8):

```
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 }
```

Parameter in einer *Methodenvereinbarung* bezeichnet man auch als *formale* Parameter (der Methode). Die Methode `hoch2` hat (nur) *einen* formalen Parameter namens `basis`, der in Zeile 1 vereinbart und in Zeile 2 (zweimal) benutzt wird.

Parameter in einem *Methodenaufruf* bezeichnet man auch als *aktuelle* Parameter. In Zeile 7 wird die Methode `hoch2` mit dem aktuellen Parameter 3 aufgerufen. In Zeile 8 wird dieselbe Methode `hoch2` mit dem aktuellen Parameter `quad+1` aufgerufen.

271. Eine *Vereinbarung eines Parameters* in einer Methodenvereinbarung hat große Ähnlichkeit mit einer *Variablenvereinbarung*: Man gibt einen Typ ("einen Bauplan für Variablen") und einen Namen an.

272. Die formalen Parameter einer Methode sind *Variablen*, die jedesmal neu erzeugt werden, wenn der Ausführer einen *Aufruf der Methode* ausführt.

273. Die formalen Parameter leben nur so lange, bis der Ausführer den betreffenden Methodenaufruf *fertig ausgeführt* hat. Dann werden sie von ihm zerstört.

274. Wenn der Ausführer einen Methodenaufruf ausführt und dabei die formalen Parameter der Methode als Variablen erzeugt, initialisiert er sie mit den Werten der entsprechenden *aktuellen Parameter* aus dem Aufruf.

**Beispiel-1:** Wenn der Ausführer den Aufruf der Methode `hoch2` oben in Zeile 7 ausführt, erzeugt er den formalen Parameter `basis` als `int`-Variable und initialisiert sie mit 3 (dem aktuellen Parameter aus Zeile 7).

**Beispiel-2:** Wenn der Ausführer den Aufruf der Methode `hoch2` oben in Zeile 8 ausführt, erzeugt er den formalen Parameter `basis` als `int`-Variable und initialisiert sie mit dem Wert des Ausdrucks `quad+1` (dem aktuellen Parameter aus Zeile 8).

275. Für einen formalen Parameter vom Typ `int` (in der Vereinbarung der Methode), darf man in jedem Aufruf der Methode einen *beliebigen Ausdruck* vom Typ `int` angeben. Man darf sogar einen Ausdruck eines anderen Typs angeben, wenn der Ausführer den Wert des Ausdrucks "ohne Probleme" in einen Wert des betreffenden Typs (im Beispiel: `int`) umwandeln kann. Als aktuellen Parameter für einen formalen `int`-Parameter darf man z. B. auch einen `char`-Ausdruck oder einen `short`-Ausdruck angeben, aber keinen `long`-Ausdruck oder `String`-Ausdruck.

276. Im obigen Beispiel wird in Zeile 7 eine `int`-Variable namens `quad` vereinbart und mit dem Wert des Ausdrucks `hoch2(3)`, also mit dem Wert 9 initialisiert.

277. Im obigen Beispiel wird in Zeile 8 der Wert des Ausdrucks `hoch2(quad+1)`, also der Wert 100, zum Bildschirm ausgegeben.

278. Eine Methode *mit Parametern* ist im Allgemeinen flexibel und kann "mehrere Probleme lösen". Eine Methode *ohne Parameter* ist dagegen "starr" und kann nur "ein Problem lösen". Wenn man eine Methode ohne Parameter mehrmals ausführen lässt, passiert "jedesmal das Gleiche" (das ist eine grobe Vereinfachung, die aber doch in vielen Fällen zutrifft). Wenn man eine Methode mit Parametern aufruft, kann jedesmal "etwas anderes passieren", je nach den aktuellen Parametern.

279. Die Art und Weise, in der in Java Parameter an Methoden übergeben werden, wird als Parameterübergabe *per Wert* (*pass by value*) bezeichnet. Die englische Bezeichnung *call by value* hat sich zwar weit verbreitet, ist aber sprachlich verunglückt, da man das englische Verb *to call* zwar auf Methoden anwenden

kann (*to call a method*), aber nicht auf die einzelnen Parameter einer Methode (*to call a parameter* ist im Englischen unsinnig).

280. Eine andere wichtige Art der Parameterübergabe ist die Übergabe *per Referenz* (pass by reference). Bei dieser Art der Parameterübergabe sind als aktuelle Parameter nur *Variablen* (statt beliebige Ausdrücke) erlaubt, und an die Methode übergeben wird die *Referenz* der Variablen, nicht ihr *Wert*.

281. Im folgenden Beispiel findet eine Parameterübergabe statt, die man wahlweise als Übergabe *per Wert* oder als Übergabe *per Referenz* interpretieren kann:

```
1 String s = new String("Hallo Sonja!");
2 println(s);
```

Offenbar wird in Zeile 2 eine Methode namens `println` aufgerufen. Als aktuellen Parameter hat der Programmierer die Referenzvariable `s` angegeben. Was bei der Ausführung des Methodenaufrufs passiert, kann man auf zwei verschiedene Weisen sehen:

**Sichtweise-1:** Der Methode `println` wird die Variable `s` *per Wert* übergeben.

**Sichtweise-2:** Der Methode `println` wird der Zielwert der Variablen `s` (das String-Objekt "Hallo Sonja!") *per Referenz* übergeben.

Die beiden Sichtweisen widersprechen sich nicht, weil der Wert der Variablen `s` gleichzeitig die Referenz des String-Objekts "Hallo Sonja!" ist (wie man sich anhand einer Bojendarstellung von `s` anschaulich klar machen kann). Der Kernunterschied zwischen den beiden Sichtweisen: Bei der ersten wird in Zeile 2 eine Variable übergeben, und bei der zweiten ein Objekt (nämlich das, auf welches die Variable zeigt).

282. Wenn man davon ausgeht, dass es in Java die beiden Parameterübergabearten *per Wert* und *per Referenz* gibt, dann werden *primitive Parameter* (Parameter, die zu einem primitiven Typ wie `int` oder `double` gehören) immer *per Wert* übergeben und *Objekte* werden immer *per Referenz* übergeben.

283. Wenn man davon ausgeht, dass es in Java die beiden Parameterübergabearten *per Wert* und *per Referenz* gibt, dann kann man *primitive Parameter* nicht *per Referenz* übergeben und *Objekte* kann man nicht *per Wert* übergeben. **Hinweis:** In der Programmiersprache C++ kann man z. B. `int`-Werte wahlweise *per Wert* oder *per Referenz* übergeben, und `string`-Objekte kann man ebenfalls wahlweise *per Wert* oder *per Referenz* übergeben (`string`-Objekte in einem C++-Programm entsprechen weitgehend `String`-Objekten in einem Java-Programm).

284. Eine Funktion dient dazu, einen *Wert* zu berechnen.

285. Eine Prozedur dient dazu, die *Inhalte von Wertebehältern* (von Variablen, von Bildschirmen, Dateien etc.) zu verändern.

286. Ein Funktionsaufruf ist ein *Ausdruck*.

287. Ein Prozeduraufruf ist eine *Anweisung*.

288. In einer *Prozedurvereinbarung* muss unmittelbar vor dem Namen der Prozedur der pseudo-Ergebnistyp `void` stehen. In einer *Funktionsvereinbarung* muss unmittelbar vor dem Namen der Funktion einer richtiger Typ (z. B. `int` oder `String` oder `float[]` oder ...) stehen. Daran kann man die beiden Arten von Unterprogrammen besonders leicht unterscheiden.

289. Wenn der Ausführer bei der *Ausführung einer Prozedur* die abschliessende geschweifte Klammer des Prozedurrumpfes erreicht, beendet er die Prozedurausführung *normal*.

290. Eine *Funktionsausführung* darf nie *normal* (durch Erreichen der abschliessenden geschweiften Klammer des Funktionsrumpfes) beendet werden. Die meisten Funktionsausführungen werden durch eine `return`-Anweisung (mit einem Ausdruck nach `return`) beendet. Einige Funktionsausführungen werden durch eine *Ausnahme* (siehe Kapitel 15 im Buch) abgebrochen.

291. Eine Funktionsvereinbarung muss *mindestens eine* `return`-Anweisung enthalten. Eine Prozedurvereinbarung muss *mindestens null* `return`-Anweisungen enthalten (d. h. sie braucht keine `return`-Anweisungen zu enthalten).

292. In einer Funktion muss nach `return` immer ein *Ausdruck des Ergebnistyps* der Funktion stehen. Der Wert dieses Ausdrucks wird dann von der Funktion als Ergebnis geliefert. In einer Prozedur darf nach `return` nie ein Ausdruck stehen. Jede `return`-Anweisung (egal ob in einer Funktion oder Prozedur) muss mit einem *Semikolon* abgeschlossen werden (wie die meisten anderen Anweisungen auch, nur *Blockanweisungen* haben *kein* abschließendes Semikolon).

293. In einer *prozeduralen Programmiersprache* gibt es veränderbare Variablen, eine Zuweisungsanweisung und (außer Funktionen auch) Prozeduren. In einer *funktionalen Sprache* gibt es nur unveränderbare Variablen, keine Zuweisungsanweisung und Funktionen (aber keine Prozeduren).

294. Zur *Signatur* einer Methode gehören der Name der Methode und die Namen der Typen ihrer Parameter.

295. *Nicht* zur Signatur einer Methode gehören die *Namen der Parameter*, Modifizierer wie `static` und `public` und der *Name des Ergebnistyps*. Auch die Haarfarbe des Programmierers einer Methode gehört nicht zu ihrer Signatur, aber das Gegenteil hätte ja wohl auch kaum jemand vermutet :-).

296. Die folgende Methode:

```
1 static public void gibAus(boolean b, float f1, float f2) {
2 ...
3 }
```

hat die Signatur `gibAus boolean float float`.



297. In Java kann man nur *Namen von Methoden* überladen. Bei dieser Metapher sind *Methoden* die Gewichte, mit denen man *Namen* belädt. Die Methoden können selbst aber nicht *beladen* oder *überladen* werden.

298. Innerhalb einer Klasse darf man nur dann mehrere Methoden mit gleichen Namen vereinbaren, wenn die Methoden sich (paarweise) durch ihre *Signaturen* unterscheiden. Da die Methoden nach Voraussetzung gleiche Namen haben sollen, müssen sie sich also durch die *Anzahl* und/oder die *Typen* ihrer Parameter unterscheiden.

299. Die folgende Methode

```
1 static public int wurziere(int... fir) {
2 pln("Wieso ausgerechnet " + fir.length + " Parameter?");
3 return fir.length;
4 }
```

darf man mit beliebig vielen (0, 1, 2, 3, ...) aktuellen `int`-Parametern aufrufen.

300. Wenn man die folgende Methode

```
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 }
```

aufruft, muss man *mindestens einen* aktuellen Parameter (vom Typ `boolean`) angeben. Zusätzlich darf man beliebig viele (0, 1, 2, 3, ...) aktuelle `int`-Parameter angeben.

301. Die folgende Methodenvereinbarung

```
8 static public int finiere(int... f1, boolean b, int... f2) {
9 if (b) return fir1.length; else fir2.length;
10 }
```

enthält zwei Fehler: 1. Sie hat mehr als einen formalen `...`-Parameter (nämlich `f1` und `f2`) und 2. nach dem `...`-Parameter `f1` folgen noch weitere Parameter (`b` und `f2`), was auch verboten ist.

302. Der Programmierer schreibt Kommentare für seine *Kollegen* (den Warter und den Wiederverwender). Der *Ausführer* ignoriert Kommentare *weitgehend* (manchmal scheint er sie gar nicht zu verstehen :-), aber nicht ganz: Manchmal lehnt er einen Kommentar als falsch ab, wie z. B. den folgenden:

```
14 // Das Zeichen LF (\u000D) bewirkt einen Zeilenwechsel
```

Dieser Kommentar ist inhaltlich völlig richtig (das Zeichen `\u000D` bewirkt wirklich einen Zeilenwechsel), trotzdem (oder gerade deshalb) lehnt ihn der Compiler `javac` von Sun mit folgender Fehlermeldung ab:

```
1 D:\Java\v05.java:14: illegal start of expression
2 // Das Zeichen LF (\u000D) bewirkt einen Zeilenwechsel:
3 ^
```

**Erklärung:** Bevor der Ausführer ein Programm genau prüft, ersetzt er darin alle Unicode-Literale (wie z. B. `\u000D`) durch das Zeichen, für das sie stehen. Aus der obigen Zeile 14 werden dadurch *zwei Zeilen*, etwa so:

```
14a // Das Zeichen LF (
14b) bewirkt einen Zeilenwechsel
```

Die Zeile 14a ist eine Kommentarzeile, aber das Kommentarzeichen `//` wirkt nur bis zum Ende von 14a. Damit ist die Zeile 14b *kein* Kommentar mehr und da sie keine korrekten Java-Befehle enthält, ist sie falsch. Wenn man den Kommentar in Zeile 14 als Schrägstrich-Stern-Kommentar notiert, etwa so:

```
14 /* Das Zeichen LF (\u000D) bewirkt einen Zeilenwechsel */
```

akzeptiert der Ausführer ihn. **Merke:** Unicode-Literale wie `\u000D` sollte man nur sehr sparsam und mit großer Vorsicht verwenden.

303. Bei fast allen Methoden sollte am Anfang ein Kommentar stehen, der kurz erklärt, was die Methode "im Großen und Ganzen" macht.

304. Der Anfangskommentar einer Methode sollte "nach der ersten Zeile der Methode stehen". Genauer: Er sollte nach dem Kopf der Methode am Anfang des Rumpfes (nach der öffnenden geschweiften Klammer) stehen. Begründung: Die Kollegen sollen zuerst den Kopf der Methode und dann den Anfangskommentar lesen. Im Anfangskommentar kann man dann davon ausgehen, dass die Kollegen schon folgendes wissen:

1. Wie die Methode heißt
2. Ob es sich um eine Funktion oder Prozedur handelt
3. Wieviele Parameter die Methode hat, wie jeder einzelne Parameter heißt und zu welchem Typ er gehört.

Diese Informationen sollte man dann im Anfangskommentar einfach *voraussetzen* (und *nicht* noch einmal erwähnen oder erläutern).

305. Wenn man das (im Prinzip sehr nützliche und leistungsfähige) Programm `javadoc` dazu verwendet, aus dem Quelltext einer Klasse oder aus den Texten vieler Klassen eine übersichtliche Dokumentation zu erzeugen, muss man den Anfangskommentar einer Methode *vor die Methode* (statt vor den *Rumpf* der Methode) schreiben, weil `javadoc` ihn nur dort erwartet und erkennt. Das gilt

- natürlich nur solange, bis Sie eine noch bessere Variante von `javadoc` geschrieben und damit `javadoc` auf dem Weltmarkt verdrängt haben :-).
306. Man sollte Kommentare überall da in ein Programm einfügen, wo die Leser des Programms *Hilfe brauchen* (nicht da, wo der Programmierer gerade Lust zum Schreiben hat). Man sollte nur solche Kommentare einfügen, die den Lesern *helfen*.
307. Sogenannte *Nacherzählungen* sind Kommentare, die man möglichst vermeiden sollte. Sie drücken nur *das* noch einmal in Worten aus, was jeder Java-Programmierer auch den Java-Befehlen entnehmen kann.
308. Ein *Modul* ist ein Behälter für Variablen, Unterprogramme und andere Module, der aus mindestens zwei Teilen besteht, einem öffentlichen (ungeschützten, sichtbaren) Teil und einem privaten (geschützten, nicht-sichtbaren) Teil. Auf die Dinge, die sich im *privaten* Teil eines Moduls befinden, kann man von Stellen außerhalb des Moduls *nicht* zugreifen.
309. Ein *Typ* ist ein Bauplan für Variablen.
310. Eine *Klasse* ist ein Modul und ein Bauplan für Module. Oder: Eine Klasse ist ein Modul und ein Typ, dessen Werte Module sind.
311. Die *Vereinbarung eines Konstruktors* sieht einer Methodenvereinbarung nicht völlig unähnlich. Einen Konstruktor kann man an zwei Merkmalen erkennen:
1. Sein Name ist gleich dem Namen der Klasse, in der er vereinbart wurde.
  2. Unmittelbar vor seinem Namen steht *kein* Ergebnistyp (auch nicht der pseudo-Typ `void`, an dem man Prozeduren erkennen kann).
- Ausserdem gilt: Ein Konstruktor ist nie mit dem Schlüsselwort `static` gekennzeichnet.
312. Ins Deutsche übersetzt lautet der folgende Befehl
- ```
1 class Zaehler01 { ... }
```
- etwa so: "Erzeuge eine Klasse namens `Zaehler01`, die folgende Elemente enthält: ...".
313. Wenn der Ausführer eine Klasse `Zaehler01` (als Modul namens `Zaehler01`) erzeugt, erzeugt er nur die mit `static` gekennzeichneten Elemente und die Konstruktoren der Klasse, und tut sie in den Modul `Zaehler01`. Die nicht mit `static` gekennzeichneten Elemente der Klasse werden (wenn überhaupt jemals) erst später erzeugt und werden nie in den Modul `Zaehler01` getan, weder jetzt noch später!
314. Wenn der Ausführer ein Objekt der Klasse `Zaehler01` erzeugt, baut er nur die *nicht* mit `static` gekennzeichneten Elemente der Klasse in das Objekt ein.

- Das macht er *jedesmal*, wenn er ein Objekt der Klasse `Zaehler01` erzeugt (also möglicherweise sehr oft, möglicherweise aber auch nullmal).
315. Ein neues Objekt zu erzeugen befiehlt man dem Ausführer normalerweise mit dem Operator `new`. Sehr viel seltener werden zwei Alternativen dazu benutzt: Man kann Objekte auch mit Methoden namens `clone` erzeugen lassen. Ausserdem kann man Objekte in Dateien schreiben. Wenn man sie von dort wieder einliest, wird dadurch auch ein (für das einlesende Programm) neues Objekt erzeugt (siehe dazu im Buch den Abschnitt 19.6).
316. Die `new`-Operation initialisiert in jedem neu erzeugten Objekt alle `int`-Variablen, für die der Programmierer keine andere Initialisierung angegeben hat, standardmäßig mit `0`. Entsprechend initialisiert die `new`-Operation `boolean`-Variablen standardmäßig mit `false` (und `double`-Variablen mit `0.0`, `float`-Variablen mit `0.0F` und Referenzvariablen mit `null` etc.).
317. Unmittelbar nach dem `new`-Operator muss man immer einen *Konstruktor* aufrufen, etwa so: `new Zaehler01(5)`. Nachdem die `new`-Operation das neue Objekt erzeugt und standardmäßig initialisiert hat, wird der Konstruktor ausgeführt. Seine Aufgabe ist es, *die* Variablen des neuen Objekts noch einmal zu initialisieren, die eine "individuelle und/oder besondere Initialisierung erfordern" (z. B. die Variable `punkte` in den Objekten der Klasse `Zaehler01`, siehe im Buch S. 203). Zusammenfassung: Ein Konstruktor dient *nicht* dazu, ein Objekt zu "konstruieren oder zu erzeugen" (das erledigt die `new`-Operation). Ein Konstruktor dient normalerweise dazu, bestimmte Variablen in einem neuen Objekt zu *initialisieren*.
318. Wenn ein Objekt `anna` eine Variable des Typs `String` enthält, liegen die Referenz und der Wert dieser Variablen immer *innerhalb* des Objekts `anna`. Falls die `String`-Variable (nicht den Wert `null` hat, sondern) auf einen Zielwert zeigt, liegt der *außerhalb* des Objekts `anna`.
319. Zum Modulaspekt einer Klasse gehören alle mit `static` gekennzeichneten Elemente der Klasse (und die Konstruktoren, die offiziell *nicht* zu den *Elementen* gehören). Zum Bauplanaspekt einer Klasse gehören alle *nicht* mit `static` gekennzeichneten Elemente der Klasse.
320. Der Modulaspekt einer Klasse enthält immer mindestens einen Konstruktor. Der Bauplanaspekt enthält immer mindestens 11 Elemente, die jede Klasse von der Klasse `Object` erbt (was "erben" heißt wird im Kapitel 12 des Buches behandelt).
321. Wenn der Programmierer in der Klasse keinen einzigen Konstruktor vereinbart hat, bekommt die Klasse vom Ausführer einen Konstruktor geschenkt (als "Sozialhilfe" :-).
322. Ein Standardkonstruktor ist ein Konstruktor mit 0 Parametern.

323. Eine Klasse kann einen Standardkonstruktor vom *Programmierer* bekommen oder bekommt ihn (wenn der Programmierer keinen einzigen Konstruktor vereinbart) vom *Ausführer*.
324. Innerhalb einer Klasse kann man *Elemente* und *Konstrukturen* vereinbaren. Die Konstrukturen zählen bei Java offiziell *nicht* zu den Elementen der Klasse.
325. Die Elemente einer Klasse haben wir nach ihrer *Art*, nach ihrer *Aspektzugehörigkeit* und nach ihrer *Erreichbarkeit* in Gruppen eingeteilt. Diese drei Einteilungen sind unabhängig voneinander.
326. Wenn man die Elemente einer Klasse nach ihrer *Art* unterscheidet, gibt es vier Gruppen:
Attribute (d. h. Variablen),
Methoden (d. h. Unterprogramme),
Klassen und
Schnittstellen.
 Innerhalb einer Klasse K kann man also weitere Klassen und Schnittstellen als Elemente von K vereinbaren.
327. Wenn man die Elemente einer Klasse nach ihrer *Aspektzugehörigkeit* unterscheidet, gibt es zwei Gruppen:
Klassenelemente (mit `static` gekennzeichnet) und
Objektelemente (nicht mit `static` gekennzeichnet).
328. Wenn man die Elemente einer Klasse nach ihrer *Erreichbarkeit* unterscheidet, gibt es vier Gruppen:
öffentliche Elemente (mit `public` gekennzeichnet),
geschützte Elemente (mit `protected` gekennzeichnet),
paketweit erreichbare Elemente (ohne Erreichbarkeitsmodifizierer) und
private Elemente (mit `private` gekennzeichnet).
329. Angenommen, wir wollen ein Verwaltungsprogramm für eine Hochschule schreiben. Als objektorientierte ProgrammiererInnen versuchen wir als erstes herauszufinden, welche Dinge (im allgemeinsten Sinne des Wortes) bei der Verwaltung einer Hochschule wichtig sind. Möglicherweise stellen wir fest, dass an einer Hochschule vor allem StudentInnen, ProfessorInnen, Räume, Lehrveranstaltungen, Notenlisten, Abschlusszeugnisse etc. wichtig sind.
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. Dann entwerfen wir für diese Dinge entsprechende Klassen namens `StudentIn`, `ProfessorIn`, `Raum`, `Lehrveranstaltung` etc. Dabei streben wir folgendes Ziel an: Ein Objekt der Klasse `StudentIn` sollte möglichst alle für die Verwaltung der Hochschule wichtigen

- Eigenschaften einer `StudentIn` repräsentieren oder widerspiegeln. Für die anderen Klassen `ProfessorIn`, `Raum` etc. ebenso.
331. Im Jahr 2005 gehörten ungefähr 2500 Klassen und ungefähr 700 Schnittstellen zur Standardbibliothek von Java 5.
332. Objekte der Klasse `String` sind *unveränderbar*. Dagegen sind Objekte der Klasse `StringBuilder` *veränderbar* (man kann z. B. einzelne Zeichen einfügen, löschen oder ersetzen). Wenn man eine Zeichenkette nie (oder nur ganz selten) verändern will, sollte man sie als `String`-Objekt darstellen, sonst als `StringBuilder`-Objekt.
333. Wenn `s` eine `String`-Variable mit dem Zielwert `"ABCDEF"` ist, dann hat
 Der Ausdruck `s.charAt(0)` den `char`-Wert `'A'`,
 der Ausdruck `s.charAt(5)` den `char`-Wert `'F'`,
 der Ausdruck `s.length()` den `int`-Wert `6`.
334. Ein `String`-Objekt der Länge 10 enthält immer genau 10 `char`-Werte. Die meisten Unicode-Zeichen werden durch je *einen* `char`-Wert dargestellt. Es gibt aber auch einige (selten gebrauchte) Zeichen, die durch *zwei* `char`-Werte dargestellt werden. Ein `String`-Objekt der Länge 10 enthält also normalerweise 10 Zeichen, möglicherweise aber auch nur 9, 8, 7, 6 oder 5 Zeichen.
335. Sei `s` eine `String`-Variable mit dem Zielwert `"ABCDEF"`. Dann hat
 der Ausdruck `s.substring(2)` den `String`-Zielwert `"CDEF"`,
 der Ausdruck `s.substring(3, 4)` den `String`-Zielwert `"D"` und
 der Ausdruck `s.substring(3,3)` den `String`-Zielwert `" "`.
336. Sei `s` eine `String`-Variable mit dem Zielwert `"ABCDEF"`. Dann hat
 der Ausdruck `s.startsWith("ABC")` den `boolean`-Wert `true`,
 der Ausdruck `s.startsWith("A")` den `boolean`-Wert `true`,
 der Ausdruck `s.startsWith("BC")`? den `boolean`-Wert `false`
 der Ausdruck `s.endsWith("DEF")` den `boolean`-Wert `true`
 der Ausdruck `s.endsWith("DF")`? den `boolean`-Wert `false`.
337. Sei `s` eine `String`-Variable mit dem Zielwert `"ABCABCABC"`. Dann hat
 der Ausdruck `s.indexOf('B')` den `int`-Wert `1`,
 der Ausdruck `s.indexOf("BC")` den `int`-Wert `1`,
 der Ausdruck `s.lastIndexOf('B')` den `int`-Wert `7`,
 der Ausdruck `s.lastIndexOf("BC")` den `int`-Wert `7`,
 der Ausdruck `s.indexOf('A', 1)` den `int`-Wert `3`,
 der Ausdruck `s.lastIndexOf("BC", 6)` den `int`-Wert `4`,
 der Ausdruck `s.indexOf('X')` den `int`-Wert `-1`,
 der Ausdruck `s.lastIndexOf("BA")` den `int`-Wert `-1`.

338. Die Methode `compareTo` (mit der man z. B. zwei `String`-Objekte auf kleiner, gleich oder größer vergleichen kann) hat den Ergebnistyp `int` (dagegen haben die Vergleichsoperationen `<`, `<=`, `>` und `>=` den Ergebnistyp `boolean`).
339. Aus folgendem Grunde ist es vernünftig, dass man `String`-Objekte *nicht* mit den Vergleichsoperationen `<` oder `<=` etc. vergleichen darf, sondern statt dessen die Funktion `compareTo` verwenden muss:
1. Ein Vergleich zweier `String`-Objekte `s11` und `s12` kann insgesamt *drei* verschiedene Ergebnisse haben: `s11` ist kleiner, gleich oder größer `s12`.
 2. Ein Vergleich zweier `String`-Objekte kann *sehr teuer* sein (d. h. viel Zeit kosten, z. B. wenn die Strings sehr lang sind und sich erst durch das Zeichen mit dem Index 1 Million unterscheiden).
 3. Mit den Operationen `<` oder `<=` etc. kann man jeweils nur *zwei* Vergleichsergebnisse unterscheiden (weil diese Operationen nur einen `boolean`-Wert als Ergebnis liefern). Um alle *drei* möglichen Vergleichsergebnisse (kleiner, gleich und größer) zu unterscheiden braucht man deshalb *zwei* solche Vergleiche-mit-`boolean`-Ergebnis.
 4. Die Funktion `compareTo` hat den Ergebnistyp `int` und kann mit *einem* Vergleich alle *drei* möglichen Vergleichsergebnisse (kleiner, gleich und größer) unterscheiden. *Ein* solcher Vergleich-mit-`int`-Ergebnis ist *billiger* als *zwei* Vergleiche-mit-`boolean`-Ergebnis.
340. Jedes `String`-Objekt enthält 47 öffentliche Methoden.
341. Jedes `String`-Objekt enthält 0 öffentliche Attribute.
342. Ist `r` eine Reihung, so ist `r.length` ein *Attribut* (auf das man ohne irgendwelche runden Klammern zugreift).
343. Ist `s` ein `String`-Objekt, so ist `s.length` eine *Methode* (die man so aufrufen muss: `s.length()`, d. h. *mit* einem Paar runder Klammern).
344. Wenn wir eine `String`-Variable als Boje darstellen, deuten wir im Zielwertkasten meistens nur das private Attribut an, in dem die einzelnen Zeichen des Objekts gespeichert sind, und lassen alle öffentlichen Methoden des Objekts weg (weil die zu zahlreich sind und jederzeit in der Dokumentation der Klasse `String` nachgesehen werden können).
345. Die Zeichenkette eines `String`-Objekts steht in einem privaten Attribut und wir können nicht direkt darauf zugreifen. Wir können aber *indirekt* darauf zugreifen, indem wir die öffentlichen Methoden des `String`-Objekts aufrufen. Da die sich im selben Modul (im selben Objekt) wie das private Attribut befinden, dürfen sie darauf zugreifen.
346. Ein `String`-Literal wie `"Hallo!"` ist in Java der *Name einer unveränderbaren Stringvariablen*. In der Bojendarstellung einer solchen Variablen steht das

Literal sowohl ganz oben im Namenspaddelboot als auch ganz unten im Zielwertkasten.

347. Angenommen, drei `String`-Variablen wurden wie folgt vereinbart:

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

Dann wissen wir, dass die Werte von `anna` und `bert` *gleich* sind und dass die Werte von `bert` und `carl` *verschieden* sind. Die Variablen `anna` und `bert` werden mit dem Wert des Literals `"ABC"` initialisiert (dieser Wert ist eine *Referenz*, die auf ein `String`-Objekt `"ABC"` zeigt).

348. Jedes `StringBuilder`-Objekt enthält 48 öffentliche Methoden. Die Zeichenkette des Objekts steht in einem privaten Attribut.
349. Die Zeichenkette eines `StringBuilder`-Objekts steht in einem privaten Attribut und wir können nicht direkt darauf zugreifen. Wir können aber *indirekt* darauf zugreifen, indem wir die öffentlichen Methoden des `StringBuilder`-Objekts aufrufen. Da die sich im selben Modul wie das private Attribut befinden, dürfen sie darauf zugreifen.
350. Das private Attribut eines `StringBuilder`-Objekts, in dem die Zeichenkette des Objekts steht, ist vermutlich vom Typ `char[]`. Wir bezeichnen dieses Attribut als den *Puffer* des `StringBuilder`-Objekts. In der offiziellen javadoc-Dokumentation von Klassen werden private Elemente nicht beschrieben (aber bei manchen Java-Distributionen kann man sich auch die Quelldateien der Standardklassen besorgen und darin nachsehen).
351. Unter der *Länge* eines `StringBuilder`-Objekts versteht man (ganz ähnlich wie bei einem `String`-Objekt oder einer Reihung vom Typ `char[]`) die Anzahl der `char`-Werte, die es "offiziell enthält". Unter der *Kapazität* versteht man die Länge des privaten Puffers vom Typ `char[]`. Die Kapazität eines `StringBuilder`-Objekts ist immer größer oder gleich seiner Länge. **Achtung:** Weil einige (selten gebrauchte) Zeichen zwei `char`-Werte für ihre Darstellung erfordern, ist die *Anzahl der Zeichen* in einem `StringBuilder`-Objekt manchmal kleiner als seine *Länge*.
352. Sei `sb` ein ziemlich langes `StringBuilder`-Objekt. Von den folgenden beiden Befehlen
- ```
sb.insert(0, "AB");
sb.insert(sb.length(), "ABCDEFGHJKLMNOPQRSTUVWXYZ");
```
- braucht der *zweite* (bei heute üblichen maschinellen Java-Ausführern) meistens *weniger Zeit* als der erste, obwohl er mehr Zeichen einfügt. Weil der erste Befehl den String `"AB"` *ganz vorn* einfügt, müssen alle schon vorhandenen `char`-Werte um zwei Positionen nach rechts verschoben werden. Allerdings: Wenn

- der zweite Befehl den privaten Puffer von `sb` zum Platzen bringt (d. h. wenn die Kapazität des Puffers vergrößert werden muß), braucht der zweite Befehl wahrscheinlich mehr Zeit als der erste.
353. Das Gleiche wie der Befehl `sb.insert(sb.length(), "XYZ");` leistet auch der kürzere Befehl `sb.append("XYZ");`.
354. Eine *Sammlung* ist ein Objekt, in das man Objekte *einfügen*, in dem man nach einem Objekt *suchen* und aus dem man Objekte *entfernen* kann.
355. Die Objekte in einer Sammlung bezeichnen wir als die *Komponenten* der Sammlung (aber *gesammelte Objekte* ist auch nicht schlecht). Im Englischen werden die Objekte in einer Sammlung als *elements of the collection* bezeichnet. Das ist möglich, weil die Elemente einer Klasse *members of the class* heißen. Um Mißverständnisse zu vermeiden, sollte man im Deutschen klar zwischen den *Elementen* eines (Sammlungs-) Objekts und den *Komponenten* eines Sammlungsobjekts unterscheiden.
356. Eine normale Klasse wie `String`, `StringBuilder` oder `Object` repräsentiert genau *einen* Typ (und es ist nicht unbedingt notwendig, immer zwischen der *Klasse* `String` und dem *Typ* `String` zu unterscheiden). Eine *generische* Klasse wie `ArrayList<K>` repräsentiert dagegen (unbegrenzt) *viele* Typen (und deshalb sollte man zwischen der *einen* Klasse und den *vielen* Typen unterscheiden).
357. Wenn einem beim Besuch einer Kneipe ein ziemlich roh aussehender Typ begegnet, sollte man den Umgang mit ihm möglichst meiden (andererseits: Manchmal entpuppen sich auch sehr roh aussehende Typen als erfahrene Menschen und interessante Gesprächspartner :-).
358. Beim Schreiben von Java-Programmen sollte man rohe Typen wenn irgend möglich vermeiden. Man sollte sie nur benutzen, wenn sich das nicht vermeiden läßt, weil man z. B. ein älteres Java-Programm (mit rohen Typen darin) erweitern oder auf andere Weise verwenden muss.
359. Drei p-`ArrayList`-Typen: `ArrayList<String>`, `ArrayList<Integer>`, `ArrayList<ArrayList<String>>`.  
In Sammlungen des Typs `ArrayList<String>` kann man ausschließlich `String`-Objekte sammeln.  
In Sammlungen des Typs `ArrayList<Integer>` kann man ausschließlich `Integer`-Objekte sammeln.  
In Sammlungen des Typs `ArrayList<ArrayList<String>>` kann man ausschließlich Sammlungen des Typs `ArrayList<String>` sammeln.  
In Sammlungen des Typs `ArrayList<Object>` kann man alle möglichen Objekte sammeln.

360. Die Klasse `ArrayList` heisst mit vollem Namen `java.util.ArrayList`. Die Klasse selbst hat keinen Adelstitel, aber einen p-`ArrayList`-Typ wie `ArrayList<String>` liest man als *ArrayList von String* :-)
361. Sei `samma` eine Objekt des Typs `ArrayList<String>`. Dann bewirkt die Anweisung `samma.add("Hallo!");` dass das `String`-Objekt `"Hallo!"` in die Sammlung `samma` eingefügt wird, und zwar unmittelbar *hinter* alle schon vorher eingefügten Objekte.
362. Sei `samma` eine Objekt des Typs `ArrayList<String>`. Dann bewirkt die Anweisung `samma.add(samma.size(), "Hallo!");` das Gleiche wie die Anweisung in der vorigen Frage.
363. Wenn `samma` eine Objekt des Typs `ArrayList<String>` ist, welches bereits *fünf String-Objekte* enthält, dann enthält es nach Ausführung der folgenden beiden Anweisungen:  
`samma.add(3, "Hallo!");`  
`samma.remove("Hallo!");`  
wieder *fünf String-Objekte* (dieselben wie vorher).
364. Sammlungen eines `ArrayList`-Typs sind *flexibel* ("aus Gummi"), d. h. man kann jederzeit noch ein weiteres Objekt einfügen (bis der Ausführer nicht mehr in der Lage ist, weitere Variablen zu erzeugen und "out of memory" meldet). Dagegen hat eine Reihung eine *feste Länge* ("Reihungen sind aus Beton"). Außerdem sind Sammlungen eines `ArrayList`-Typs insgesamt komfortabler als Reihungen, d. h. sie enthalten "viele nützliche Methoden".
365. Wenn man sehr viele primitive Werte (z. B. `double`-Werte) speichern muss, wird man möglicherweise eine *Reihung* vorziehen. Primitive Werte können grundsätzlich nicht in eine Sammlung eingefügt werden, sondern müssen vorher in Objekte der zuständigen Hüllklasse eingehüllt (und später wieder enthüllt) werden. In manchen Anwendungen kostet das Einhüllen und Enthüllen zu viel Zeit und man speichert die primitiven Werte lieber direkt in einer Reihung.
366. Zufallszahlen werden unter anderem zum automatischen *Erzeugen von Testdaten* und in *Computerspielen* verwendet.
367. Eine innerhalb eines Programms `P` erzeugte Folge von Zufallszahlen nennt man reproduzierbar, wenn sie bei jeder Ausführung des Programms gleich ist und nicht-reproduzierbar, wenn sie sich von Ausführung zu Ausführung (in praktisch nicht vorhersehbarer Weise) verändert.
368. Beim Erzeugen von Testdaten verwendet man in aller Regel *reproduzierbare* Zufallszahlen (um einen mit den Testdaten gefundenen Fehler beliebig oft reproduzieren zu können). In Computerspielen (in denen die Spieler z. B. würfeln können oder Lose ziehen dürfen) verwendet man in aller Regel *nicht-re-*

produzierbare Zufallszahlen (außer während der Entwicklung und beim Testen des Spiels, siehe oben, oder wenn man Mitspieler betrügen will :-).

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();
```

Das Objekt `quelle02` ist eine Quelle von *nicht-reproduzierbaren* Zufallszahlen und `quelle01` liefert *reproduzierbare* Zufallszahlen.

370. So kann man aus der `quelle01` einen zufälligen `int`-Wert schöpfen:

```
int becher1 = quelle01.nextInt();
```

371. So kann man aus der `quelle02` einen zufälligen `boolean`-Wert schöpfen:

```
boolean becher2 = quelle02.nextBoolean();
```

372. Der Aufruf `quelle01.nextInt(50)` liefert eine Zufallszahl zwischen 0 (einschließlich) und 50 (ausschließlich).

373. Die Zahlen `Long.MIN_VALUE` und `Long.MAX_VALUE` haben ungefähr die Werte -9 Trillionen und +9 Trillionen? Die Zahl 1 Trillion besteht aus einer 1 gefolgt von 18 Nullen.

374. Wenn man sehr große Ganzzahlen durch Werte des Typs `double` darstellt, werden die meisten Ganzzahlen nur *näherungsweise* dargestellt, z. B. durch einen `double`-Wert, der um 1000 oder um 100000 zu groß oder zu klein ist (siehe dazu im Buch den Abschnitt 5.1, insbesondere S. 94). In Java kann man Ganzzahlen auch durch Objekte der Klasse `BigInteger` darstellen. Solche Zahlen können (in dezimaler Darstellung) ohne weiteres aus 100 oder aus 1000 Ziffern bestehen und werden immer exakt dargestellt (nicht nur näherungsweise).

375. Eine Berechnung mit `long`-Werten kostet (bei heute üblichen, maschinellen Java-Ausführern) nur einen Bruchteil der Zeit, die eine entsprechende Berechnung mit `BigInteger`-Objekten kostet. Aber bei einigen Anwendungen spielt dieser Vorteil von `long`-Berechnungen kaum eine Rolle, weil die Berechnung so oder so nur Bruchteile einer Sekunde dauern.

Der Typ `BigInteger` hat im Vergleich zu `long` zwei Vorteile: 1. Den erheblich größeren Wertebereich und 2. die Sicherheit, dass keine "unentdeckten Überläufe" auftreten können. In der Praxis gibt es bei `BigInteger`-Berechnungen keine Überläufe, und wenn es doch mal so etwa Ähnliches geben sollte, wird eine Ausnahme geworfen (d. h., etwas vereinfacht gesagt, eine klare Fehlermeldung ausgegeben).

376. Eine Berechnung mit `double`-Werten kostet (bei heute üblichen, maschinellen Java-Ausführern) nur einen Bruchteil der Zeit, die eine entsprechende Berechnung mit `BigDecimal`-Objekten kostet. Aber bei einigen Anwendungen

spielt dieser Vorteil von `double`-Berechnungen kaum eine Rolle, weil die Berechnung so oder so nur Bruchteile einer Sekunde dauern.

Der Typ `BigDecimal` hat im Vergleich zu `double` zwei Vorteile: 1. Die Möglichkeit, mit fast *beliebig großen* und fast *beliebig genauen* Bruchzahlen zu rechnen. 2. `BigDecimal`-Objekte werden vom Java-Ausführer so dargestellt, dass sie beim Rechnen (und insbesondere beim Runden) die Eigenschaften von *Dezimalbrüchen* haben, und nicht (wie `float`- und `double`-Werte) die Eigenschaften von *Binärbrüchen*. Deshalb eignen sich `BigDecimal`-Objekte insbesondere für *kommerzielle* Rechnungen, bei denen auch Rundungsfehler mit einem dezimalen Taschenrechner nachvollziehbar sein müssen.

377. Nein, es gibt endliche Dezimalbrüche (z. B. 0.1), zu denen es keine endlichen Binärbrüche mit exakt denselben Werten gibt.

378. Ja, es gibt zu jedem endlichen Binärbruch einen endlichen Dezimalbruch mit exakt demselben Wert (und es gilt: Wenn der Binärbruch  $n$  Stellen nach dem Binärpunkt hat, dann hat der Dezimalbruch ebenfalls  $n$  Stellen nach dem Dezimalpunkt).

379. Die zwei wichtigste Methode in einem `Formatter`-Objekt heißen beide `format`.

380. Die beiden Methoden namens `printf` in einem `PrintStream`-Objekt (z. B. im `PrintStream`-Objekt `System.out`) leisten ganz Ähnliches wie die beiden Methoden namens `format` in einem `Formatter`-Objekt.

381. Der Befehl `System.out.printf("Nr. %0+4dXYZ%n", 5);` gibt folgenden String (gefolgt von einem Zeilenwechsel) zum Bildschirm aus:  
Nr. +005XYZ

382. Zu einem Java-Programm namens `AbRechnung` gehört zunächst nur *eine* Klasse namens `AbRechnung`, die *Hauptklasse* des Programms. Die *Hauptklasse* muss eine `main`-Methode enthalten. Alle Klassen, die zur Ausführung dieser `main`-Methode benötigt werden, gehören als *Nebenklassen* zum Programm `AbRechnung`. Welche Klassen das sind, kann von Ausführung zu Ausführung verschieden sein (je nach den Eingabedaten des Benutzers).

383. Die *Hauptklasse* eines Programms wird erzeugt, wenn der Benutzer das Programm startet (d. h. den Ausführer dazu auffordert, das Programm auszuführen).

384. Eine *Nebenklasse* wird erzeugt, wenn der Ausführer sie zum ersten Mal *benötigt*.

385. Die *Hauptklasse* eines Java-Programms *muss* eine `main`-Methode enthalten. Jede *Nebenklasse* *darf* eine `main`-Methode enthalten, muss aber nicht (in aller Regel enthalten *Nebenklassen keine* `main`-Methode).

386. Eine vorhandene Klasse K mit einem Editor zu kopieren und die Kopie K1 dann zu ändern, um so eine benötigte Variante von K zu erhalten, ist aus folgendem Grund *keine* gute Idee: Die Wartung von K und K1 kann aufwendig werden. Wenn man in K einen Fehler entdeckt und korrigiert, muss man auch K1 untersuchen und evtl. korrigieren. Falls man K auf andere Weise verbessert oder erweitert gilt ähnliches. Wenn man die Klasse K nicht nur einmal, sondern mehrmals kopiert, wird das Wartungsproblem möglicherweise zu einem Alptraum.
387. Wenn man eine neue Klasse K1 als Erweiterung einer schon vorhandenen Klasse K vereinbart, kopiert der Ausführer (so kann man sich zumindest vorstellen) alle Elemente von K in die neue Klasse K1. Zusätzlich kann man dann in K1 noch weitere Elemente vereinbaren.
388. Die Klasse `Person02` erbt insgesamt 6 Elemente von `Person01`, 3 Klassenattribute (ein Klassenattribut `anzahlPersonen` und zwei Klassenmethoden `druckeAnzahlPersonen` und `p`) und 3 Objektelemente (zwei Objektattribute `vorName` und `nachName` und eine Objektmethode `druckeName`). *Konstruktor* werden grundsätzlich *nicht* ver- und geerbt.
389. In jedes `Person02`-Objekt werden die folgenden 5 Elemente eingebaut:
- 2 Attribute `vorName` und `nachName` (hat `Person02` von `Person01` geerbt)
  - 1 Methode `druckeName` (hat `Person02` von `Person01` geerbt)
  - 1 Attribut `personalNr` (wurde in `Person02` neu vereinbart)
  - 1 Methode `druckeAlles` (wurde in `Person02` neu vereinbart)
390. Der *Modul* `Person02` enthält die folgenden 5 Elemente:
- 1 Attribut `anzahlPerson01` (hat `Person02` von `Person01` geerbt)
  - 2 Methoden `drucke...son01` und `p` (hat `Person02` von `Person01` geerbt)
  - 1 Attribut `anzahlPerson02` (wurde in `Person02` neu vereinbart)
  - 1 Methode `drucke...son02` (wurde in `Person02` neu vereinbart)
391. Jede Klasse, die nicht ausdrücklich eine andere Klasse erweitert, erweitert ("automatisch und stillschweigend") die Klasse `Object`.
392. In der Klasse `Object` sind 11 Objektelemente (`clone` bis `wait`) und 0 (in Worten: null) Klassenelemente vereinbart.
393. Die Klasse `Object` beerbt *null* Klassen. Alle anderen Java-Klassen müssen *genau eine* Klasse beerben.
394. Eine alte Klasse darf ihre Elemente an *beliebig viele* (null, eine, zwei, drei, ...) neue Klassen vererben.
395. Eine Klasse darf sich weder direkt noch indirekt *selbst* selbst beerben, etwa so:
- ```

1 class Karl extends Karl { ... } // Verboten
2 class Hin extends Her { ... }
```

- ```

3 class Her extends Hin { ... } // Verboten
```
396. Alle Java-Klassen mit der erweitert-Relation bilden einen *Baum*. Die Klasse `Object` ist die *Wurzel* dieses Baums (und wird in grafischen Darstellungen des Baums meistens ganz *oben* gezeichnet).
397. Wenn K2 eine Erweiterung (oder: Unterklasse) der Klasse K1 ist, dann gilt die Regel  
R2: Jedes K2-Objekt ist auch ein K1-Objekt.
398. Wenn man ein Objekt als Zwiebel darstellt, ist der Kern der Zwiebel immer ein Objekt der Klasse `Object` (mit 11 Objektmethoden darin).
399. In einer Reihung vom Typ `E01Punkt[]` (d. h. in eine Reihung von `E01Punkt`-Variablen) kann man nicht nur `E01Punkt`-Objekt speichern, sondern auch `E01Ellipse`-Objekte und `E01Rechteck`-Objekte, weil die Klassen `E01Ellipse` und `E01Rechteck` Unterklassen von `E01Punkt` sind und deshalb jedes `E01Ellipse`-Objekt (und jedes `E01Rechteck`-Objekt) auch als ein `E01Punkt`-Objekt gilt.
400. Die Komponenten einer Reihung vom Typ `E01Punkt[]` kann man "einheitlich bearbeiten", obwohl die Reihung Objekte *unterschiedlicher* Klassen (`E01Ellipse`, `E01Rechteck` etc.) enthält. Beim Bearbeiten der Komponenten braucht man sie *nicht* mit komplizierten `if`-Anweisungen nach ihren genauen Typen zu unterscheiden. Und wenn man eine neue Unterklasse von `E01Punkt` vereinbart und Objekte dieser Klasse in die Reihung (vom Typ `E01Punkt[]`) einfügt, funktioniert die "einheitliche Bearbeitung" auch für diese neuen Objekte, ohne dass man irgendwelche Änderungen oder Anpassungen vornehmen muss.
401. Eine *Oberklasse* ist größer als eine Unterklasse, wenn man als Vergleichsmaß die *Anzahl der Objekte* wählt, die zu den Klassen gehören.
402. Eine *Unterklasse* ist größer als eine Oberklasse, wenn man als Vergleichsmaß die *Anzahl der Elemente* (Attribute und Methoden) wählt, die zu den Klassen gehören.
403. Normalerweise verwenden wir die Anzahl der Objekte, die zu einer Klasse gehören, als Maß für die Größe der Klasse.
404. In einem Typgraphen zeichnen wir größere ("allgemeinere") Klassen weiter oben und kleinere ("speziellere") Klassen weiter unten ein. Die Klasse `Object` ist bei dieser Auffassung von "Größe" die *größte* Java-Klasse und steht ganz oben.
405. Je größer die Klassen, desto kleiner ihre Objekte. Ein Objekt `k2` einer Klasse K2 ist *größer* als ein Objekt `k1` einer Klasse K1, wenn die Klasse K1 eine *Oberklasse* von K2 ist. Oder: Wenn die Menge der Elemente des Objekts `k2` eine *Obermenge* der Elemente des Objekts `k1` ist. Objekte der Klasse `Object`

- sind bei dieser Betrachtung die *kleinsten* Objekte die es gibt (und enthalten nur die minimalen 11 Objektmethoden, die in der Klasse Object vereinbart sind und die jedes Objekt enthält).
406. Wenn eine Referenzvariable auf Objekte einer bestimmten Klasse zeigen darf, dann darf sie auch auf *größere Objekte* (d. h. auf Objekte mit einer größeren Elementmenge) zeigen.
407. Der Programmierer muss in einer Klasse keinen Konstruktor vereinbaren und darf beliebig viele vereinbaren
408. Jede Klasse enthält mindestens einen Konstruktor, oder sovielen, wie der Programmierer vereinbart hat (die größere der beiden Zahlen gewinnt). Wenn der Programmierer keinen Konstruktor vereinbart hat, bekommt die Klasse vom Ausführer einen Standardkonstruktor "geschenkt" ("als Sozialhilfe").
409. Ein Standardkonstruktor in einer Klasse kann vom *Programmierer* vereinbart oder vom *Ausführer* geschenkt worden sein.
410. Nein. Ein vom Ausführer geschenkter Standardkonstruktor hat immer einen leeren Rumpf. Aber ein vom Programmierer vereinbarter Standardkonstruktor kann einen leeren oder einen nicht-leeren Rumpf haben.
411. Der erste Befehl im Rumpf eines Konstruktors bewirkt immer, dass ein Konstruktor der direkten Oberklasse aufgerufen wird. Wenn der Programmierer nicht ausdrücklich einen solchen Befehl als ersten Befehl in den Konstruktor schreibt, fügt der Ausführer einen Aufruf des *Standardkonstruktors* der Oberklasse ein. Falls die Oberklasse keinen Standardkonstruktor besitzt, meldet der Ausführer einen Fehler, weil dann der von ihm eingefügte Aufruf (des Standardkonstruktors) falsch ist.
412. Wenn der Programmierer in einem Konstruktor einen Konstruktor der direkten Oberklasse aufrufen will, muss er (anstelle des üblichen Namens des Konstruktors) das Schlüsselwort *super* verwenden. Ein solcher *super*-Aufruf ist nur als *erster Befehl* im Rumpf eines Konstruktors erlaubt.
413. Wenn der Programmierer in einem Konstruktor einen anderen Konstruktor derselben Klasse aufrufen will, muss er (anstelle des üblichen Namens des Konstruktors) das Schlüsselwort *this* verwenden. Ein solcher *this*-Aufruf ist nur als *erster Befehl* im Rumpf eines Konstruktors erlaubt.
414. Eine Variable des Typs *E01Punkt* darf nicht nur auf *E01Punkt*-Objekte zeigen, sondern auch auf Objekte der Unterklassen *E01Rechteck*, *E01Quadrat*, *E01Ellipse* und *E01Kreis*.
415. Eine Variable des Typs *E01Punkt* kann (je nachdem, auf was für ein Objekt sie gerade zeigt) einen der folgenden Typen als Zieltyp haben: *E01Punkt*, *E01Rechteck*, *E01Quadrat*, *E01Ellipse* und *E01Kreis*. Wenn die Varia-

- ble den Wert *null* hat (und somit nicht auf ein Objekt zeigt), ordnen wir ihr den Zieltyp *void* zu.
416. Den Zieltyp einer Referenzvariablen *rv* kann man mit einer *Zuweisung* verändern. Zum Beispiel hat *rv* nach der Zuweisung *rv = null;* den Zieltyp *void*.
417. Sei *p02* eine Variable des Typs *E01Punkt*. Dann kann der (zusammengesetzte) Methodenname *p02.toString* eine der fünf *toString*-Methoden bezeichnen, die in den Klassen *E01Punkt*, *E01Rechteck*, *E01Quadrat*, *E01Ellipse* und *E01Kreis* vereinbart wurden. Welche dieser Methoden der Name tatsächlich bezeichnet, hängt vom *momentanen Zieltyp* der Variablen *p02* (d.h. vom Typ des Objekts, auf das sie zeigt) ab. Wenn die Variable *p02* den Wert *null* hat, bezeichnet der Name *p02.toString* *keine* Methode.
418. Sei *d01* eine *double*-Variable, die momentan den Wert 7.8 enthält. Dann erzeugt der Cast-Befehl (*int*) *d01* aus dem Wert von *d01* den entsprechenden *int*-Wert 7 (es wird nicht gerundet, alle Nachpunktstellen werden abgeschnitten).
419. Sei *s01* eine *short*-Variable, die momentan den Wert -1 enthält. In binärer Darstellung sieht dieser *short*-Wert etwas so aus: 1111 1111 1111 1111 (sechzehn mal die Binärziffer 1). Der Cast-Befehl (*char*) *s01* erzeugt aus diesem Wert den *char*-Wert 65535. In binärer Darstellung sieht dieser *char*-Wert etwa so aus: 1111 1111 1111 1111 (sechzehn mal die Binärziffer 1).
420. Typumwandlungen mit Cast-Befehlen zwischen primitiven Typen sind erlaubt, wenn beide Typen *numerisch* sind (numerisch sind *byte*, *short*, *char*, *int*, *long*, *float*, *double*). Ausserdem sind nur triviale Cast-Befehle von *boolean* nach *boolean* erlaubt.
421. Typumwandlungen mit Cast-Befehlen zwischen dem Typ *boolean* und anderen primitiven Typen (oder Referenztypen) sind *nicht* erlaubt.
422. Typumwandlungen mit Cast-Befehlen zwischen *primitiven Typen* und *Referenztypen* sind nur in (2 \* 8 gleich) 16 Fällen erlaubt, nämlich nur zwischen einem primitiven Typ und dem entsprechenden Hüllklassentyp (z. B. von *float* nach *Float* oder von *Integer* nach *int* etc.).
423. Eine Typumwandlung mit Cast-Befehl zwischen zwei Referenztypen *Quelle* und *Ziel* ist nur dann erlaubt, wenn einer der Typen ein Untertyp des anderen ist (wenn also *Quelle* ein Untertyp von *Ziel* oder *Ziel* ein Untertyp von *Quelle* ist).
424. Im Buch wird ein Cast-Befehl zwischen zwei Referenztypen als eine *Umdeutung* bezeichnet, weil der Begriff Umwandlung möglicherweise falsche Vorstellungen (einer "Veränderung") weckt.



425. In einem Typgraphen stehen Obertypen weiter oben als ihre Untertypen. Cast-Befehle zwischen Referenztypen *von unten nach oben* (von einem Untertyp zu einem Obertyp) sind harmlos, d. h. sie lösen keine Ausnahme aus.
426. Cast-Befehle *von oben nach unten* (von einem Obertyp zu einem Untertyp) sind nicht immer harmlos, sondern können eine Ausnahme auslösen.
427. Der Cast-Befehl (StringBuilder) "Hallo!" wird vom Ausführer (schon bei der Übergabe des Programms, "zur Compilezeit"), *abgelehnt*, weil der Ausdruck "Hallo!" vom Typ String ist und keiner der beiden Typen String und StringBuilder ein Untertyp des anderen ist.
428. Wenn der Programmierer eine Klasse Unter schreibt, die von ihrer direkten Oberklasse Ober ein Element e erbt, kann der Programmierer dieses Element *ersetzen*, indem er in der Klasse Unter ein *homonymes* Element vereinbart.
429. Zur *Signatur* einer Methode gehören der Name der Methode und die Namen der Typen ihrer Parameter.
430. *Nicht* zur Signatur einer Methode gehören die *Namen der Parameter*, Modifizierer wie `static` und `public` und der *Name des Ergebnistyps*.
431. Das *Profil* einer Methode besteht aus dem *Namen ihres Ergebnistyps* gefolgt von ihrer *Signatur*.
432. Zwei *Attribute* sind homonym, wenn ihre Namen gleich sind (ihre Typen können gleich oder ungleich sein).
433. Zwei *Methoden* sind homonym, wenn ihre *Profile* gleich sind.
434. Von den folgenden drei Methoden
- ```

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) {...}

```
- sind die erste und die zweite *homonym*, aber die zweite und die dritte sind *nicht* homonym (weil sie unterschiedliche Ergebnistypen float bzw. short und somit unterschiedliche *Profile* haben).
435. Es ist nicht erlaubt, in einer Klasse zwei homonyme Elemente zu vereinbaren. Es ist aber erlaubt, in einer Klasse Ober ein Element e und in einer Unterklasse Unter von Ober ein zu e homonymes Element e' zu vereinbaren. In einer solchen Situation gehören die homonymen Elemente e und e' beide zur Klasse Unter (e als geerbtes Element und e' als neu vereinbartes Element).
436. *Objektelemente* werden *häufig* ersetzt, *Klassenelemente* werden dagegen nur relativ *selten* ersetzt.
437. Es gibt zwei unterschiedliche Weisen, ein Element zu ersetzen: *verdecken* (engl. to hide) und *überschreiben* (engl. to override).

438. Ob ein Element e2 von einem Element e1 *verdeckt* oder *überschrieben* wird, hängt von der *Art* der Elemente (Methode oder Attribut) und von ihrer *Aspekt-zugehörigkeit* (Klassenelement oder Objektelement) ab.
439. Eine *Objektmethode* wird von einer homonymen Objektmethode *überschrieben*.
440. Ein *Objektattribut* wird von einem homonymen Objektattribut *verdeckt*.
441. Der *Unter-Programmierer* programmiert eine Klasse Unter (und ersetzt dabei eventuell gewisse Elemente, die die Klasse Unter von ihrer direkten Oberklasse Ober geerbt hat). Der *Unter-Verwender* vereinbart Objekte der Unterklasse Unter und versucht, auch auf die *ersetzten Elemente* in diesen Objekten zuzugreifen.
442. Der *Unter-Programmierer* kann in seiner Klasse Unter auf *alle* Elemente zugreifen, auch wenn er sie gerade ersetzt hat. Auf ein *verdecktes* Element kann er mit Hilfe einer Variablen des entsprechenden Obertyps zugreifen und auf ein *überschriebenes* Element, indem er `super.` vor den Namen des Elements schreibt.
443. Der *Unter-Verwender* kann (in seinen Unter-Objekten) nur auf *verdeckte* Elemente zugreifen (mit Hilfe von Ober-Variablen), aber nicht auf *überschriebene* Elemente. Ausserdem kann er natürlich auf nicht-ersetzte Elemente zugreifen, wenn sie erreichbar (z. B. `public`) sind.
444. 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, wie etwa im folgenden Beispiel:
- ```

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

```
- dann meldet der Ausführer (schon bei der Übergabe des Programms, "zur Compilezeit") einen Fehler. Eine Ausnahme zu dieser Regel wird im Abschnitt 12.9 des Buches (Beispi-07 auf S. 315) erläutert und in der Literatur als *kovariante Erweiterung des Ergebnistyps* bezeichnet.
445. Nein, ein `public`-Element darf man nicht durch ein `private`-Element ersetzen. Allgemein gilt: Das ersetzende Element muss immer mindestens so erreichbar sein wie das ersetzte Element (oder "erreichbarer").
446. Selbst wenn eine geerbte Methode m1 und eine neue Methode m2 gleiche Profile und die gleiche Erreichbarkeit haben, ist es in den folgenden Fällen verboten, m1 durch m2 zu ersetzen:
- Fall-1: Die geerbte Methode m1 ist in der erbenden Klasse *nicht sichtbar* (z. B. weil m1 als `private` vereinbart wurde).
- Fall-2: Die neue Methode m2 *wirft mehr geprüfte Ausnahmen* als die geerbte Methode m1. Genauer: Die Methode m2 wirft (mind.) eine *geprüfte Ausnahme*

- (und erwähnt sie entsprechend in ihrer `throws`-Klausel), die von der Methode `m1` nicht geworfen (und nicht in ihrer `throws`-Klausel erwähnt) wird.
- Fall-3: Die geerbte Methode `m1` wurde als `final` ("darf nicht ersetzt werden") vereinbart.
447. *Überladen* kann man einen *Namen* (indem man ihn "mit mehreren Methoden als Bedeutungen belädt"), *überschreiben* kann man eine *Methode*. Überladen wird ein Methodenname in einer Klasse `K`, wenn man in `K` mehrere Methoden mit diesem Namen, aber *unterschiedlichen Signaturen* erbt und/oder vereinbart. Überschreiben kann man in einer Klasse `K` nur eine geerbte Objektmethode, indem man in `K` eine Methode mit *gleichem Profil* vereinbart.
448. Der *Unter-Verwender* (der Objekte einer Klasse `Unter` verwendet und versucht, auch auf die vom *Unter-Programmierer* ersetzten Elemente dieser Objekte zuzugreifen) kann mit keinem Trick auf *überschriebene* Objektmethoden in den *Unter-Objekten* zugreifen, er kann aber durchaus auf *verdeckte* Objektattribute zugreifen. In diesem Sinne sind *überschriebene* Elemente für den *Unter-Verwender* *unwiederruflich* überschrieben, *verdeckte* Elemente dagegen nur *wiederruflich* verdeckt.
449. Eine neue Klasse darf (und muss) nur *eine* alte Klasse *erweitern*, darf aber *beliebig viele* Klassen *instanzieren*.
450. Faustregel: Eine Klasse *instanzieren* ist meistens *besser* als die Klasse zu *erweitern*. Eine Klasse zu *erweitern* ist im Allgemeinen fehlerträchtiger, als ein Objekt der Klasse zu vereinbaren und zu benutzen.
451. Mit den folgenden beiden Verfahren kann der Autor einer Klasse sicherstellen, dass seine Klasse nicht erweitert werden kann:
- Verfahren-1:** Er vereinbart die Klasse als `final`, etwa so:
- ```
final class Karl { ... }
```
- Verfahren-2:** Der Programmierer vereinbart einen oder mehrere Konstruktoren und vereinbart *alle* Konstruktoren als `private`.
452. *Initialisierer* (Klassen- und Objekt-) dürfen nur direkt in einer Klassenvereinbarung stehen (nicht außerhalb einer Klassenvereinbarung und nicht innerhalb einer Methode).
453. Ein *Klasseninitialisierer* sieht so aus:
- ```
static { ... }
```
- Ein *Objektinitialisierer* sieht so aus:
- ```
{ ... }
```
- Bei beiden darf innerhalb der Blockanweisung `{ ... }` eine beliebige Folge von Vereinbarungen und/oder Anweisungen stehen.
454. *Objektinitialisierer* sind *leicht entbehrlich*, weil man alle damit lösbaren Probleme leicht auch durch Befehlsfolgen in einem Konstruktor lösen kann. *Klas-*

- seninitialisierer* sind dagegen *schwer entbehrlich*, weil man einige damit lösbaren Probleme nur umständlich und wenig elegant lösen kann.
455. Die *Klasseninitialisierer* einer Klasse werden genau *einmal* ausgeführt, und zwar dann, wenn die Klasse geladen (d. h. erzeugt) wird. Sie werden in der Reihenfolge ausgeführt, in der sie in der Klasse vereinbart wurden.
456. Die *Objektinitialisierer* einer Klasse `K` werden jedesmal ausgeführt, wenn mit dem `new`-Befehl ein neues Objekt der Klasse `K` erzeugt wurde (und noch bevor ein Konstruktor für dieses neue Objekt aufgerufen wurde). Auch die *Objektinitialisierer* werden in der Reihenfolge ausgeführt, in der sie in `K` vereinbart wurden.
457. *Klasseninitialisierer* werden typischerweise zum Initialisieren solcher Klassenattribute verwendet, die sich nicht einfach durch einen Initialisierungsteil in ihrer Vereinbarung initialisieren lassen.
- Beispiel: Eine `int`-Reihung der Länge 100 soll mit 100 Zufallszahlen initialisiert werden.
458. *Objektinitialisierer* werden typischerweise zum Initialisieren solcher Objektattribute verwendet, die sich nicht einfach durch einen Initialisierungsteil in ihrer Vereinbarung initialisieren lassen. Beispiel:
- Eine `int`-Reihung der Länge 100 soll mit 100 Zufallszahlen initialisiert werden.
- Objektattribute kann man auch durch "komplizierte Befehlsfolgen in einem Konstruktor" initialisieren. Statt ein Attribut von *allen* Konstruktoren einer Klasse durch gleiche "komplizierte Befehlsfolgen" initialisieren zu lassen, kann es vorteilhaft sein, die "komplizierte Befehlsfolge" in einen *Objektinitialisierer* (oder in eine *private* Objektmethode) auszulagern.
459. Auf S. 214 des Buches (dort im Beispiel-02) wird gezeigt (oder zumindest angedeutet), wie man in Java schon immer (d. h. vor Java 5) *Aufzählungstypen vereinbaren* konnte.
460. Die drei *Farbe*-Objekte namens `ROT`, `GRUEN` und `BLAU` unterscheiden sich nur durch ihre *Referenzen*. Diese Referenzen (die Werte der unveränderbaren Variablen `ROT`, `GRUEN` und `BLAU`, vereinbart in den Zeilen 5 bis 7 der Klasse `Farbe`, repräsentieren die Farben (nicht die leeren Objekte, auf die die Referenzen zeigen).
461. Ein *Aufzählungstyps* wie z. B.

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

hat gegenüber vier int-Konstanten wie etwa

2  final int ROT      = 1;
3  final int GRUEN    = 2;
```

```
4 final int BLAU = 3;
5 final int FARBLOS = 4;
```

folgende Vorteile:

Einer int-Variablen, in der man Farben speichern will, kann man die Werte der Konstanten ROT, GRUEN, BLAU und FARBLOS zuweisen, aber auch beliebige andere int-Werte, die keine Farben darstellen. Einer int-Variablen, in der man *keine* Farben speichern will (sondern z. B. Euro-Beträge) kann man trotzdem die Werte der Konstanten ROT, GRUEN, BLAU und FARBLOS zuweisen. Einer Variablen vom Typ Farbe kann man nur die Werte Farbe.rot, Farbe.gruen, Farbe.blau und Farbe.farblos zuweisen, und diese Werte kann man nur einer Variablen des Typs Farbe zuweisen. Außerdem gilt:

Ein Ausgabebefehl wie `println(BLAU)`; gibt die Zahl 3 zum Bildschirm aus. Eine Ausgabebefehl wie `println(Farbe.blau)`; gibt blau zum Bildschirm aus.

462. Aus der Vereinbarung eines Aufzählungstyps Farbe erzeugt der Ausführer eine Klasse, die die Standardklasse Enum erweitert. Enum ist eine *generische* Klasse. Generische Klassen werden im Kapitel 16 des Buches behandelt.

463. Die Aufzählungswerte rot, gruen, blau und farblos der Klasse Farbe werden (vom Ausführer, automatisch) als *öffentliche, unveränderbare Klassenattribute* implementiert, z. B. so:

```
static final public Farbe rot = new Farbe("rot", 0); !
```

464. Wenn man die Funktion Farbe.values() aufruft, liefert sie eine Reihung vom Typ Farbe[], die alle vier Farbe-Objekte rot, gruen, blau und farblos enthält.

465. Ob zwei Farbe-Objekte *gleich* oder *ungleich* sind, kann man mit den Operationen == und != feststellen. Wenn man wissen will, ob ein Farbe-Objekt f1 kleiner, gleich oder größer als ein Farbe-Objekt f2 ist, muss man die Methode compareTo aufrufen, etwa so:

```
1 int erg = f1.compareTo(f2);
```

Die int-Variable erg bekommt einen negativen Wert, den Wert 0 bzw. einen positiven Wert, je nachdem ob f1 kleiner, gleich oder größer als f2 ist. Dabei gilt rot als der kleinste und farblos als der größte Farbe-Wert (entsprechend der Reihenfolge in der Vereinbarung des Typs Farbe).

466. Die normale import-Vereinbarung

```
import javax.swing.JButton;
```

führt den *einfachen Namen* JButton als Abkürzung für den *zusammengesetzten Namen* javax.swing.JButton ein.

467. 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, muss man den folgenden zusammengesetzten Namen angeben: `java.util.Arrays.sort(...)`.

468. Wenn man die sort-Methode aus der vorigen Antwort im Wirkungsbereich der normalen import-Vereinbarung `import java.util.Arrays;` aufrufen will, kann man auch den kürzeren Namen `Arrays.sort(...)` angeben.

469. Wir möchten die sort-Methode aus der vorigen Antwort wie folgt aufrufen: `sort(...);` Damit das möglich ist, müssen wir vorher eine der folgenden static-import-Vereinbarungen angeben:

```
static import java.util.Arrays.sort; // Gute Lösung
static import java.util.Arrays.*;    // Weniger gut!
```

470. Mit einer normalen import-Vereinbarung kann man für die (zusammengesetzten) Namen von *Klassen* und *Schnittstellen* Abkürzungen einführen.

471. Mit einer import-static-Vereinbarung kann man für die (zusammengesetzten) Namen von *Klassenelementen* Abkürzungen einführen.

472. Von den beiden import-static-Vereinbarungen

```
import static Farbe.rot;
import static Wein.*;
```

ist die *erste* (die einzel-import-Vereinbarung) *stärker* als die zweite (die Pauschal-import-Vereinbarung).

473. Der Programmierer kann eine Klasse K *instanziierten* (d. h. als Bauplan für Objekte benutzen, etwa so: `new K(...)`) und er kann K *beerben* (indem er andere Klassen als *Erweiterungen* von K vereinbart, etwa so: `class Neu extends K { ... }`). Die dritte der zwei wichtigsten Tätigkeiten: Er kann *den Modulaspekt von K benutzen* und mit Namen wie `K.attribut1` oder `K.methode3` etc. auf Elemente in diesem Modul zugreifen.

474. Eine *abstrakte* Klasse kann der Programmierer nur *beerben* (oder: *erweitern*), aber nicht *instanziierten*. Ausserdem kann er *den Modulaspekt von K benutzen* und mit Namen wie `K.attribut1` oder `K.methode3` etc. auf Attribute und konkrete Methoden in diesem Modul zugreifen.

475. In einer *abstrakten* Klasse darf man alles vereinbaren, was auch in einer *konkreten* Klasse erlaubt ist (Konstruktoren, Attribute, konkrete Methoden, Klassen und Schnittstellen). Zusätzlich darf man in einer abstrakten Klasse auch noch *abstrakte Methoden* vereinbaren.

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. Dann haben Sie folgende Möglichkeiten, diese Fehlermeldung zu beseitigen:

1. Sie überschreiben die geerbte, abstrakte Methode am in der Klasse K durch

- eine konkrete Methode.
2. Sie vereinbaren ihre Klasse K als *abstrakte Klasse*.
477. Von einer abstrakten Klasse K kann man mit den folgenden beiden Schritten Objekte erzeugen lassen:
1. Man vereinbart eine konkrete Unterklasse U von K .
 2. Man läßt Objekte von U erzeugen.
- Da jedes Objekt einer Unterklasse auch als ein Objekt seiner Oberklasse(n) gilt, hat man auf diese Weise auch Objekte der Klasse K erzeugen lassen.
478. An Methoden darf man in einer Schnittstelle nur *abstrakte Objektmethoden* vereinbaren (keine konkreten Objektmethoden und keine Klassenmethoden).
479. Eine Schnittstelle *darf beliebig* viele (null oder mehr) Schnittstellen erweitern.
480. Eine Klasse darf *beliebig viele* (null oder mehr) Schnittstellen implementieren.
481. Wenn S eine Schnittstelle ist, dann gilt:
- Eine *S-Klasse* ist eine Klasse, die die Schnittstelle S implementiert.
- Ein *S-Objekt* ist ein Objekt einer S -Klasse.
482. Sei *Ausgebbbar* eine Schnittstelle. Als Komponenten einer Reihung des Typs *Ausgebbbar*[] darf man dann beliebige *Ausgebbbar*-Objekte (d. h. Objekte beliebiger Klassen, die nur die Schnittstelle *Ausgebbbar* implementieren müssen) speichern.
483. Ausser *abstrakten Objektmethoden* darf eine Schnittstelle auch unveränderbare Klassenattribute (Klassenkonstanten) enthalten. (S. 348)
484. Die Schnittstelle *Runnable* enthält (nur) *eine* abstrakte Objektmethode namens *run*. Jede konkrete Klasse K , die diese Schnittstelle implementiert, ist dazu verpflichtet, die abstrakte Methode *run* zu *implementieren*, d. h. durch eine *konkrete Methode zu überschreiben*.
485. Die Schnittstelle *javax.swing.SwingConstants* enthält nur Konstanten namens *BOTTOM*, *CENTER*, *EAST* etc. (sie enthält keine abstrakten Objektmethoden). Eine Klasse K , die diese Schnittstelle implementiert, ist dadurch zu *nichts* verpflichtet. Im Gegenteil: Innerhalb von K darf man auf die Konstanten der Schnittstelle mit deren einfachen Namen *BOTTOM*, *CENTER*, *EAST* etc. zugreifen und muss nicht unbedingt die zusammengesetzten Namen *SwingConstants.BOTTOM*, *SwingConstants.CENTER*, *SwingConstants.EAST* etc. verwenden.
486. Die Schnittstelle *Runnable* (aus der Java-Standardbibliothek) wird unter anderem von den Standardklassen *FutureTask*, *Thread* und *TimerTask* implementiert. Somit hat eine Methode mit einem *Runnable*-Parameter wie
- ```
void bearbeite01(Runnable r) { ... }
```

- im Vergleich zu einer Methode mit einem *Thread*-Parameter wie
- ```
void bearbeite02(Thread t) { ... }
```
- folgenden Vorteil:
- Die Methode *bearbeite01* kann man auf alle *Runnable*-Objekte anwenden (unter anderem auf Objekte der Klassen *FutureTask*, *Thread* und *TimerTask*). Die Methode *bearbeite02* kann man nur auf *Thread*-Objekte anwenden. Anschaulich gilt: Es gibt viel mehr *Runnable*-Objekte (die zu *verschiedenen* Klassen gehören) als *Thread*-Objekte (die alle zu der *einen* Klasse *Thread* oder zu einer ihrer Unterklassen gehören müssen).
487. Eine *Markierungsschnittstelle* (engl. a marker interface) ist eine leere Schnittstelle S , die nur dazu dient, S -Objekte und andere Objekte zu unterscheiden.
488. Die Schnittstellen *Serializable* und *Cloneable* sind zwei wichtige Markierungsschnittstellen aus der Java-Standardbibliothek.
489. Mit Hilfe einer Markierungsschnittstellen S kann man nur *Objekte* unterscheiden, nämlich S -Objekte und alle anderen. Man kann damit nicht Pakete, Klassen, Methoden, Attribute, Konstruktoren etc. unterscheiden.
490. Mit Hilfe von *Anmerkungen* (engl. annotations) kann man folgende Arten von Dingen unterscheiden: Objekte, Pakete, Klassen, Methoden, Attribute, Konstruktoren und Parameter von Methoden. Ausserdem kann man *Anmerkungstypen* mit Anmerkungen versehen.
491. Bestimmte Pakete, Klassen, Methoden etc. mit Anmerkungen wie *Autor*, *Firma*, *NurGegenAufpreis* etc. zu markieren kann nützlich sein, weil dann geeignete Werkzeugprogramme diese Pakete, Klassen, Methoden etc. erkennen und speziell bearbeiten können.
492. Der Java-Compiler *javac* von Sun reagiert schon heute auf die Anmerkung *@Override*. Mit dieser Anmerkung sollte man Methoden versehen von denen man weiß oder annimmt, dass sie eine geerbte Methode *überschreiben*. Falls das dann doch nicht der Fall sein sollte (z. B. weil man den Methodennamen nicht ganz richtig geschrieben hat), warnt einen der Compiler.
493. Eine Anmerkung ist technisch gesehen ein *Objekt* eines speziellen *Anmerkungstyps*.
494. Anmerkungstypen haben einiges gemeinsam mit *Schnittstellen*.
495. Anmerkungstypen unterscheiden sich von Schnittstellen vor allem durch die *Notation* ihrer Vereinbarungen. Anmerkungstypen werden mit Hilfe von *Anmerkungen* vereinbart.
496. Man darf ein Programmteil mit *beliebig vielen* Anmerkungen versehen, aber nicht mit *mehreren* Anmerkungen des *selben Typs*. Zum Beispiel darf man eine Methode *nicht* mit zwei Anmerkungen des Typs *@Autor* versehen. (S. 357)

497. *Anmerkungen*, mit denen man nur Anmerkungstypen versehen darf, bezeichnet man als *Meta-Anmerkungen*. Ihre Typen bezeichnet man als *Meta-Anmerkungstypen*.
498. Der Programmierer schreibt Anmerkungen in seine *Quellprogramme*. Er kann dem Ausführer aber auch befehlen, die Anmerkungen an zwei andere Orte mitzunehmen:
1. Der Compiler kann Anmerkungen aus einer Quelldatei in die daraus erzeugte *Bytecodedatei* (.class-Datei) übernehmen.
 2. Der Bytecodeinterpreter kann Anmerkungen aus einer *Bytecodedatei laden*, so dass sie während der Ausführung des betreffenden Programms ("zur Laufzeit") zur Verfügung stehen.
- Diese "Mitnahme" seiner Anmerkungen kann der Programmierer mit Anmerkungen des Typs `@Retention` festlegen.
499. Eine *Ausnahmeklasse* ist eine Unterklasse der Standardklasse *Throwable*. Da jede Klasse auch als Unterklasse von sich selbst gilt, ist damit *Throwable* selbst auch eine Ausnahmeklasse. Eine Klasse wird also dadurch zu einer Ausnahmeklasse, dass sie die Klasse *Throwable* (direkt oder indirekt) *beerbt*. (S. 363)
500. Ein *Ausnahmeobjekt* ist ein Objekt einer *Ausnahmeklasse*. Mit der *throw*-Anweisung kann man nur Ausnahmeobjekte werfen (keine anderen, normalen Objekte).
501. Jedes *Ausnahmeobjekt* enthält (zusätzlich zu den 11 von *Object* geerbten Methoden) mindestens 11 Methoden, die seine Klasse von der Klasse *Throwable* geerbt hat, darunter die Methoden *getMessage*, *printStackTrace*, *initCause* und *getCause*.
502. Eine *Ganzzahldivision* *dend/dor* wirft eine Ausnahme (des Typs *ArithmeticException*), wenn der Divisor *dor* den Wert 0 hat.
- Die Methode *Integer.parseInt*, mit der man einen String wie z. B. `" +123"` in einen entsprechenden *int*-Wert umwandeln lassen kann, wirft eine Ausnahme (des Typs *NumberFormatException*), wenn man sie auf einen ungeeigneten String wie z. B. `"ABC"` anwendet.
503. Wird im Verlauf einer Programmausführung eine Ausnahme *geworfen*, aber nicht *gefangen*, so wird die Programmausführung vom Ausführer mit einer Fehlermeldung *abgebrochen*.
504. *Ausnahmen*, die von einem gefährlichen Befehl *GB* geworfen werden, kann man mit einem *try-catch*-Befehl *fangen und behandeln*, etwa so:
1. Man schreibt den gefährlichen Befehl *GB* in einen *try*-Block:
 2. Hinter diesen *try*-Block schreibt man einen oder mehrere *catch*-Blöcke,

von denen jeder Ausnahmen eines bestimmten Ausnahmetyps fängt und behandelt, z. B. so:

```
1  try {
2      Integer.parseInt("ABC");
3  } catch(NumberFormatException ex) {
4      pln("Eine Ausnahme ist aufgetreten: " + ex);
5  }
```

505. Angenommen, ein *try-catch*-Befehl mit drei *catch*-Blöcken wird ausgeführt. Dann wird dabei *höchstens einer* der *catch*-Blöcke ausgeführt.
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. Dann kann man an folgenden Stellen des Programms *try-catch*-Befehle einbauen, um diese Ausnahme zu fangen und zu behandeln:
1. In der Methode *m2* kann man den gefährlichen Befehl in den *try*-Block eines *try-catch*-Befehls schreiben.
 2. In der Methode *m1* kann man den Aufruf der Methode *m2* in den *try*-Block eines *try-catch*-Befehls schreiben.
 3. In der *main*-Methode kann man den Aufruf der Methode *m1* in den *try*-Block eines *try-catch*-Befehls schreiben.
507. Auf dem Stapel des Ausführers befinden sich in jedem Moment Informationen über alle Methoden, deren Ausführung schon *begonnen*, aber noch *nicht abgeschlossen* wurde. Bei einer rekursiven Methode *RM* kann es vorkommen, dass mehrere Ausführungen von *RM* schon begonnen, aber noch nicht abgeschlossen sind. In einem solchen Fall stehen entsprechende viele Einträge für *RM* auf dem Stapel.
508. Wenn an einer bestimmten Stelle eines Programms eine Ausnahme *AUS* auftritt ("angeflogen kommt"), prüft der Ausführer immer: *Trat die Ausnahme AUS in einem try-Block auf, dem ein zum Fangen von AUS geeigneter catch-Block folgt?*
509. Wenn in einer nicht-*main*-Methode eines Programms eine Ausnahme auftritt ("angeflogen kommt") und dort *nicht* gefangen wird, bricht der Ausführer die Ausführung der nicht-*main*-Methode ab, kehrt an die Programmstelle zurück, an der die Methode aufgerufen wurde, und wirft die Ausnahme dort erneut.
510. Wenn in der *main*-Methode eines Programms *P* eine Ausnahme auftritt ("angeflogen kommt") und dort nicht gefangen wird, bricht der Ausführer die Ausführung von *P* ab und gibt eine Fehlermeldung aus. Üblicherweise gibt diese Fehlermeldung auch den Inhalt des Stapels wieder (und besteht deshalb häufig aus vielen Zeilen).

511. Wenn man eine Herdplatte einschaltet sollte man sicherstellen, dass die Platte auch wieder ausgeschaltet wird (selbst wenn man durch Telefonanrufe oder andere dringende Ausnahmefälle unterbrochen und abgelenkt wird).
512. In Bezug auf einen `try-catch-finally`-Befehl garantiert der Ausführer einem: Wenn er den `try`-Block betritt (und nicht wegen einer Endlosschleife darin hängen bleibt), betritt er auch den zugehörigen `finally`-Block (unabhängig davon, was im `try`-Block passiert).
513. Nur der Befehle `System.exit(n)` bringt die `try-catch-finally`-Garantie des Ausführers zum Erlöschen. Wenn dieser Befehl im `try`-Block oder in einem `catch`-Block ausgeführt wird, wird das Programm *sofort* beendet und ein eventuell "noch offener `finally`-Block" wird *nicht* ausgeführt.
514. Innerhalb eines `try`-Blocks bzw. innerhalb eines `catch`-Blocks darf man *alle* Java-Anweisungen verwenden, insbesondere auch `try-catch`-Blöcke.
515. Wenn während der Ausführung eines `catch`-Blocks eine Ausnahme `AUS` auftritt ("angeflogen kommt"), mach der Ausführer dasselbe wie immer, wenn eine Ausnahme auftritt, er prüft: *Trat die Ausnahme AUS in einem try-Block auf, dem ein zum Fangen von AUS geeigneter catch-Block folgt?* etc. (siehe S. 372)
516. Am Anfang jeder Methode sollte dokumentiert werden, welche Ausnahmen durch Aufrufe der Methode möglicherweise ausgelöst werden (oder: welche Ausnahmen möglicherweise zu einem Aufruf dieser Methode "geflogen kommen").
517. Ausnahmen, die praktisch *an jeder Stelle* eines Programms ausgelöst werden können (z. B. Ausnahmen des Typs `OutOfMemoryError`) sollten *nicht* am Anfang von jeder Methode als "wird möglicherweise ausgelöst" dokumentiert werden.
518. Kommentare und Dokumentationen in einem Programm werden vom Ausführer nicht geprüft. Deshalb sind sie unzuverlässig. Wenn man in einem Programmtext nach Fehlern sucht, muss man immer damit rechnen, dass die Kommentare Fehler enthalten und kann sich nicht auf deren Inhalt verlassen.
519. Wo darf man eine *throws-Klausel* und wo darf man *throw-Anweisungen* hinschreiben?
520. Wenn der Programmierer einen gefährlichen Befehl, der möglicherweise eine geprüfte Ausnahme `A` auslöst, in eine Methode `m` schreibt, muss er eine der folgenden beiden Maßnahmen treffen:
1. Er muss in einer *throws-Klausel* am Anfang von `m` "ehrlich zugeben", dass ein Aufruf von `m` möglicherweise die Ausnahme `A` auslöst.
 2. Oder er muss dafür sorgen, dass die Ausnahme `A` innerhalb von `m` gefangen und behandelt wird und nicht "nach draussen dringt".

521. Der Programmierer kann eine von ihm vereinbarte Klasse `K` zu einer ungeprüften (bzw. zu einer geprüften) Ausnahmeklasse machen, indem er `K` als Erweiterung der Standardklasse `Error` oder der Standardklasse `RuntimeException` (bzw. der Standardklasse `Exception`) vereinbart.
522. Die obersten drei Klassen des Baums aller Ausnahmeklassen und die Klasse `RuntimeException` bilden folgende Struktur (*Unterklassen* sind relativ zu ihrer direkten Oberklasse um eine Stufe *eingerrückt*):
- | | | |
|---|-------------------------------|-------------|
| 6 | <code>Throwable</code> | (geprüft) |
| 7 | <code>Exception</code> | (geprüft) |
| 8 | <code>RuntimeException</code> | (ungeprüft) |
| 9 | <code>Error</code> | (ungeprüft) |
523. Die Klasse `Exception` ist eine *geprüfte* Ausnahmeklasse, hat aber die *ungeprüfte* direkte Unterklasse `RuntimeException`.
524. Die Zusicherung `assert n%2==0;` kann man etwa wie folgt ins Deutsche übersetzen: "Stelle sicher, dass die Bedingung `n%2==0` *erfüllt* ist (d. h. dass `n` gerade ist). Werfe eine Ausnahme des Typs `AssertionError`, falls die Bedingung (wider Erwarten) *nicht erfüllt* sein sollte".
525. Der Unterschied zwischen der Zusicherung `assert n%2==0;` und der Anweisung `if (n%2==0) throw AssertionError();` ist der folgende: *Zusicherungen* werden vom Ausführer normalerweise *nicht* ausgeführt (und werfen dann auch keine Ausnahmen). *Zusicherungen* werden *nur* ausgeführt, wenn der Benutzer das beim Starten eines Programms *ausdrücklich befiehlt*. Eine normale *if*-Anweisung kann dagegen nicht beim Start eines Programms *aktiviert* oder *deaktiviert* werden.
526. Durch das Konzept der *geprüften Ausnahmen* soll der Java-Programmierer dazu angeregt bzw. gezwungen werden, bestimmte Ausnahmen entweder zu behandeln oder am Anfang jeder Methode in einer *throws-Klausel* vollständig aufzuzählen. Der Ausführer überprüft die Vollständigkeit der *throws-Klausel*.
527. Das Konzept der geprüften Ausnahmen hat nicht in allen Fällen dazu geführt, dass geprüfte Ausnahmen immer behandelt oder in *throws-Klauseln* übersichtlich dokumentiert werden. In einigen Programmen werden Ausnahmen zwar gefangen und mit einem leeren `catch`-Block "formal", aber nicht "wirklich, inhaltlich" behandelt.
528. Betrachten Sie das Beispiel-01 im Abschnitt 15.9 des Buches (auf S. 387). Das dort skizzierte Problem sollte man lösen, indem man die *geprüfte(n)* Ausnahmen fängt und in den `catch`-Blöcken entsprechende *ungeprüfte* Ausnahmen wirft (d. h. indem man die geprüften Ausnahmen in ungeprüfte Ausnahmen "übersetzt"). Vielen Dank an meinen Kollegen Prof. Christoph Knabe, der mich auf diese Lösung aufmerksam machte.

529. Java-Befehle, von denen der Java-Ausführer *schon bei der Übergabe* des Programms ("zur Compilezeit") feststellen kann, ob bei ihrer Ausführung Typfehler auftreten können oder nicht, bezeichnen wir als *stark typsicher*. Befehle, bei denen *erste während der Ausführung* ("zur Laufzeit") entschieden werden kann, ob sie einen Typfehler enthalten oder nicht, bezeichnen wir als *schwach typsicher*.
530. Die folgenden beiden Java-Befehle sind nur schwach typsicher:
1. Zuweisungen an die Komponenten einer Reihung von Objekten (Reihungen mit primitiven Komponenten sind davon nicht betroffen). Siehe dazu den Abschnitt 7.9 über das Kovarianzproblem bei Reihungen.
 2. Cast-Befehle, angewendet auf Operanden eines Referenztyps (angewendet auf primitive Operanden sind Cast-Befehle zumindest formal typsicher, obwohl dabei inhaltlich sehr "merkwürdige" Umwandlungen stattfinden können (z. B. von -1 nach 65535)).
531. Folgende Festlegung ("Regel der Sprache Java") ist der Grund dafür, dass schreibende Zugriffe auf nicht-primitive Reihungskomponenten nur schwach typsicher sind: Wenn ein Typ U ein Untertyp eines Typs T ist, dann ist der Typ $U[]$ ein Untertyp des Typs $T[]$.
- Beispiel:** Weil der Typ `String` ein Untertyp des Typs `Object` ist, ist der Typ `String[]` ein Untertyp des Typs `Object[]`. Warum diese Regel (bei der Entwicklung von Java) trotz ihrer Gefährlichkeit kaum zu vermeiden war, wird im Abschnitt 7.9 (S. 178) des Buches erläutert.
532. Beim vorgenerischen Java mußte man im Zusammenhang mit *Sammlungen* nur schwach typsichere Cast-Befehle anwenden. Wenn man ein Objekt aus einer Sammlung holte, hatte es grundsätzlich "nur den Typ `Object`", und mußte in aller Regel mit einem Cast-Befehl "als Objekt einer spezielleren Klasse" umgedeutet werden.
533. Das prominenteste Anwendungsgebiet für generische Einheiten ist (auch in Java) das Gebiet der *Sammlungen* (engl. collections).
534. Die generische Klasse `PairB<K>` hat *einen* formalen Parameter namens `K`. Die generische Klasse `PairC<S, T>` hat *zwei* formale Parameter namens `S` und `T`. Für die formalen Parameter einer generischen Klasse verwendet man in aller Regel *sehr kurze Namen*, die nur aus einem oder zwei großen Buchstaben bestehen, z. B. `K`, `S`, `T` etc. Länger Namen (z. B. `ElementType` oder `TypWolfgangAmadeus` etc.) sind erlaubt, aber unüblich.
535. Beim Instanzieren einer generischen Klasse darf (und muss) man für jeden formalen Parameter einen *Referenztyp* als aktuellen Parameter angeben. Man

- darf keine primitiven Typen (wie `int`, `float`, `boolean` etc.) als aktuelle Parameter angeben. Man darf auch keine Fahrräder mit 7-Gang-Schaltung als aktuelle Parameter angeben (aber das hätte wohl auch kaum jemand vermutet :-).
536. Eine normale Klasse wie `String` repräsentiert genau *einen* Typ (entsprechend für `StringBuilder` und `Double` etc.). Eine generische Klasse wie zum Beispiel `PairB<K>` repräsentiert *vielen* Typen (man kann auch sagen: *unbegrenzt vielen* oder *unendlich vielen* Typen). Auch eine generische Klasse mit mehreren formalen Parametern wie z. B. `PairC<S, T>` repräsentiert *vielen* (unbegrenzt vielen, unendlich vielen) Typen.
537. Wenn man die generische Klasse `PairB<K>` mit dem Typ `String` als aktuellem Parameter instanziiert, erhält man den Typ namens `PairB<String>`. Diesen Typnamen spricht man so: *Paar Beh von String*.
538. Den parametrisierten Typ `PairC<Double, Number>` erhält man, indem man die generische Klasse `PairC<S, T>` mit den Typen `Double` und `Number` als aktuelle Parameter instanziiert (oder: parametrisiert). Den Typnamen `PairC<Double, Number>` spricht man so: *Paar Ceh von Double und Number*.
539. Von den drei Sätzen
1. Der p-`PairB`-Typ `PairB<String>` ist ein Untertyp von `PairB<Object>`.
 2. Von den beiden p-`PairB`-Typen `PairB<String>` und `PairB<Object>` ist keiner ein Untertyp des anderen.
 3. Der p-`PairB`-Typ `PairB<Object>` ist ein Untertyp von `PairB<String>`.
- ist nur 2. richtig, 1. und 3. sind falsch.
540. Von den drei Sätzen
1. Der p-`PairB`-Typ `PairB<String>` ist ein Untertyp des p-`PairC`-Typs `PairC<String, String>`.
 2. Von den beiden p-Typen `PairB<String>` und `PairC<String, String>` ist keiner ein Untertyp des anderen.
 3. Der p-`PairC`-Typ `PairC<String, String>` ist ein Untertyp des p-`PairB`-Typs `PairB<String>`.
- ist nur 2. richtig, 1. und 3. sind falsch.
541. Die generische Schnittstelle `java.lang.Comparable<K>` hat *einen* formalen Parameter. Sie enthält nur *eine* (abstrakte Objekt-) Methode namens `compareTo`.
542. Jedes `String`-Objekt enthält eine Methode namens `compareTo` mit *einem* Parameter vom Typ `String` und dem Ergebnistyp `int`, etwa so:
- ```
public int compareTo(String that) { ... }
```
543. Seien `s1` und `s2` zwei `String`-Variablen, die auf zwei `String`-Objekte zeigen. Dann hat der Ausdruck `s1.compareTo(s2)` einen negativen Wert, den

- Wert 0 bzw. einen positiven Wert, je nachdem ob der String *s1* kleiner, gleich oder größer *s2* ist. Ein String *s1* ist *kleiner* als ein String *s2*, wenn *s1* in einem Lexikon *weiter vorne* stehen muss als *s2*.
544. Unter einem Comparable<String>-Objekt versteht man ein Objekt einer Klasse, die die Schnittstelle Comparable<String> implementiert.
545. Drei Instanzen der generischen Schnittstelle Comparable<K>:  
Comparable<String>, Comparable<Integer>, Comparable<Number>.  
Noch eine Instanz: Comparable<Comparable<String>>.
546. Eine Klasse namens MeineKlasse darf eine beliebige Instanz der generischen Schnittstelle Comparable<K> implementieren (aber nur *eine* Instanz). Dies gilt sogar für *alle* Klassen und für *alle* generischen Schnittstellen.
547. Wenn eine Klasse namens MeineKlasse eine Instanz der generischen Schnittstelle Comparable<K> implementiert, dann ist das in aller Regel die Instanz Comparable<MeineKlasse>.
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. Dabei sind alle *Referenztypen* als aktuelle Parameter erlaubt (aber keine primitiven Typen).
549. Die Typparameter (oder: formalen Parameter) vieler generischer Einheiten sind *unbeschränkt*. Das bedeutet, dass man sie durch *beliebige Referenztypen* ersetzen darf.
550. Die folgende Vereinbarung einer generischen Klasse
- ```
1 class KarlHeinz<T extends Otto & Druckbar & Loeschbar> { ... }
```
- kann man etwa wie folgt ins Deutsche übersetzen: "Erzeuge eine generische Klasse namens KarHeinz mit einem formalen Typparameter T. Als aktuelle Parameter für T sollen nur solche Typen eingesetzt werden dürfen, die die Klasse Otto erweitern und die Schnittstellen Druckbar und Loeschbar implementieren. ..."
- Aus der Vereinbarung folgt, dass Druckbar und Loeschbar *Schnittstellen* sein müssen. Außerdem folgt aus ihr, dass man für T nur *Klassentypen* einsetzen darf (aber keine Schnittstellentypen und keine Reihungstypen), denn nur eine *Klasse* kann die Klasse Otto erweitern.
551. Die folgende Vereinbarung einer nicht-generischen Klasse
- ```
2 class KarlHeinrich extends Otto { ... }
```
- kann man etwa wie folgt ins Deutsche übersetzen: "Erzeuge eine Klasse namens KarlHeinrich als Erweiterung der Klasse Otto. ..."
- Hier hat das Schlüsselwort extends eine etwas andere Bedeutung als in der vorigen Frage/Antwort.

552. Die folgende Vereinbarung einer generischen Klasse

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

kann man etwa wie folgt ins Deutsche übersetzen: "Erzeuge eine generische Klasse Namens PaarD mit zwei formalen Typparametern S und T. Als aktuelle Parameter für S soll ein beliebiger (Referenz-) Typ RT erlaubt sein, aber für S soll nur eine *Erweiterung* von RT (oder: ein *Untertyp* von RT) eingesetzt werden dürfen. ..."

553. Wir bezeichnen zwei Typen als *unabhängig* voneinander, wenn keiner der beiden ein Untertyp (oder Obertyp) des anderen ist.

554. Die folgende generische Klasse PaarX<K> ist fehlerhaft:

```
1 class PaarX<K> {
2 static K a; // falsch
3 static void tuNix1(K b) {} // falsch
4 K c;
5 void tuNix2(K d) {}
6 }
```

Das Element a ist ein Klassenattribut.

Das Element tuNix1 ist eine Klassenmethode.

Das Element c ist ein Objektattribut.

Das Element tuNix2 ist eine Objektmethode.

Die Vereinbarungen der Elemente a und tuNix1 sind *falsch*, weil *Klassenelemente* nicht im Gültigkeitsbereich von *formalen Typparametern* wie K liegen.

555. Die Vereinbarung einer generischen Klassenmethode (mit einem Typparameter namens T) kann etwa so aussehen:

```
1 static <T> void machWas(T t) { ... }
```

556. Die Typen Double, Object, Integer, Number und die Relation ist-ein-Untertyp-von (dargestellt durch Einrückung):

```
Object
 Number
 Double
 Integer
```

557. Normale Typparameter wie K, S, T etc. darf man (mit extends) nur nach *oben* beschränken.

558. Die Jokervariable ? darf man (mit extends bzw. super) nach oben bzw. unten beschränken.

559. Sei PaarH<K> eine generische Klasse. Dann sind die Typen PaarH<Double>, PaarH<Integer> und PaarH<Number> Untertypen des Typs PaarH<? extends Number>, weil die Typen Double, Integer und Number



*Erweiterungen* (oder: Untertypen) von `Number` sind.

**Anmerkung:** Die Klasse `Number` hat außer sich selbst und den Klassen `Integer` und `Double` noch acht weitere Unterklassen.

560. Sei `PaarH<K>` eine generische Klasse. Dann sind z. B. die Typen

`PaarH<Double>`, `PaarH<Number>` und `PaarH<Object>`

Untertypen des Typs `PaarH<? super Double>`, weil die Typen

`Double`, `Number` und `Object`

*Obertypen* (engl. *supertypes*) von `Double` sind.

**Anmerkung:** `Double`, `Number` und `Object` sind die *einzigsten* Obertypen von `Double`.

561. Nein, der Typ `ArrayList<String>` ist kein Untertyp von `ArrayList<Object>` (obwohl `String` natürlich ein Untertyp von `Object` ist). Der Typ `ArrayList<String>` ist aber ein Untertyp von `ArrayList<?>`!

562. Sei die Variablen `sam1` wie folgt vereinbart:

```
ArrayList<String> sam1 = new ArrayList<String>;
```

Dann ist ein schreibender Zugriff wie z. B.

```
sam1.add("Hallo!");
```

erlaubt!

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

```
ArrayList<?> sam2 = new ArrayList<String>;
```

Dann ist ein schreibender Zugriff wie z. B.

```
sam2.add("Hallo!");
```

nicht erlaubt (weil der Typ der Variable `sam2` nicht garantiert, dass `sam2` auf ein Objekt des Typs `ArrayList<String>` zeigt)!

564. Der rohe Typ `ArrayList` ist ein *Obertyp* der Typen `ArrayList<String>`, `ArrayList<Integer>`, `ArrayList<Number>` etc. Dagegen ist der parametrisierte Typ `ArrayList<Object>` *kein* Obertyp der parametrisierten Typen `ArrayList<String>`, `ArrayList<Integer>`, `ArrayList<Number>` etc.

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. Es gilt dann allgemein:

Eine Zuweisung eines *rohen* Wertes an eine *p*-Variable ist *typunsicher*!

Eine Zuweisung eines *p*-Wertes an eine *rohe* Variable ist *typsicher*!

566. Von den folgenden Methodenvereinbarungen

```
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;}
```

sind `machWas1`, `machWas2` und `machWas4` korrekt, nur `machWas3` ist nicht korrekt.

567. Als *fragwürdig* werden in diesem Abschnitt (16.11) Befehle bezeichnet, die der Ausführer mit einer Warnung akzeptiert.

568. Ein *Paket* ist ein Behälter für Klassen, Schnittstellen und weitere Pakete.

569. Das Top-Paket `java` heißt mit vollem Namen `java`.

570. Das Paket `sax` im Paket `xml` im Top-Paket `org` heißt mit vollem Namen `org.xml.sax`.

571. Paketnamen bestehen üblicherweise nur aus *kleinen Buchstaben* (und evtl. Ziffern). Damit kann man sie leicht von den Namen von Klassen und Schnittstellen unterscheiden, die üblicherweise mit einem *großen Buchstaben* anfangen.

572. Jede Klasse gehört zu *genau einem* Paket. Beispiele:

Die Klasse `java.lang.String` gehört zum Paket `java.lang` und die Beispielsklasse `Hallo01` gehört zum namenlosen Paket.

573. Damit eine Klasse `AbRechnung` zu einem Paket `de.meyerundsohn.test` gehört, muss der Programmierer am Anfang *der* Quelldatei, in der die Klasse vereinbart wird, den Befehl `package de.meyerundsohn.test;` einfügen. Vor diesem `package`-Befehl dürfen in der Quelldatei nur Kommentare, aber keine anderen Befehle stehen.

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

575. Für *Klassen* gibt es nur *zwei* Erreichbarkeitsstufen: *public* und *paketweit erreichbar*.

576. Auf eine nur paketweit erreichbare Klasse kann man nur von Stellen innerhalb ihres Paketes zugreifen.

577. Auf eine *öffentliche* Klasse (engl. *public class*), die zu einem *Paket mit Namen* gehört (oder: die *nicht* zum *namenlosen Paket* gehört), kann man im Prinzip "von überall her" zugreifen.

578. Auf *öffentliche* Klassen (engl. *public classes*), die zum *namenlosen Paket* gehören, kann man nur von Stellen innerhalb des namenlosen Paketes aus zugreifen. *Öffentliche* Klassen im *namenlosen Paket* sind also "nicht wirklich öffentlich", sondern nur *paketweit erreichbar*.

579. Wenn zwei Klassen gleiche Namen haben, darf man sie nicht in das selbe Paket tun.

580. Ein *Paket* ist zwar ein *Behälter* für Klassen, Schnittstellen und Pakete, aber kein richtiger *Modul*, weil man Pakete innerhalb eines Paketes *p* nicht in den geschützten (privaten, unsichtbaren) Teil von *p* tun kann (oder: weil Pakete grundsätzlich öffentlich sind).

581. Von den folgenden Aussagen über `import`-Vereinbarungen trifft keine zu (alle sind falsch):

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.

Zu 1: Man kann immer *ohne* `import`-Vereinbarungen auskommen (wenn man bereit ist, alle Klassen mit ihren vollen Namen zu bezeichnen).

Zu 2: `import`-Vereinbarungen bewirken *nicht*, dass irgendwelche Klassen "importiert", d. h. in ein Programm eingebunden werden.

Zu 3: `import`-Vereinbarungen haben *keinen* Einfluß auf den Speicherbedarf eines Programms.

Zu 4: `import`-Vereinbarungen haben *keinen* Einfluß darauf, welche Klassen zu einem Programm gehören.

582. Die `import`-Vereinbarung

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

bewirkt, dass der Programmierer den Klassennamen `ArrayList` als Abkürzung für den vollen Klassennamen `java.util.ArrayList` verwenden darf.

583. Pauschale `import`-Vereinbarungen ("import-Vereinbarungen mit einem Sternchen") *erleichtern* das *Schreiben* eines Programms und *erschweren* das *Lesen*. Wenn ein Programm nur einfache `import`-Vereinbarungen (wie in der vorigen Antwort) enthält, kann ein Leser, der eine benutzte Klasse nicht kennt, allein anhand der `import`-Vereinbarungen herausfinden, zu welchem Paket sie gehört. Wenn ein Programm zwei oder mehr pauschale `import`-Vereinbarungen enthält, entfällt dieser Vorteil.

584. Die Klassenvereinbarung:

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

ist falsch, weil die Klasse `Executors` zum Paket `java.util.concurrent` gehört und die `import`-Vereinbarung in Zeile 1 deshalb *keine* Abkürzung für sie einführt. Richtig wäre z. B. folgende Variante der Klassenvereinbarung:

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

585. Angenommen, wir schreiben eine Klasse namens `Karl` in einem Paket namens `patrick.test`. Dann dürfen wir alle Klassen und Schnittstellen, die zum Paket `java.lang` oder zum Paket `patrick.test` gehören mit ihren *einfachen Namen* bezeichnen (auch *ohne* entsprechende `import`-Vereinbarung).

586. Klassen und Schnittstellen, die zum *namenlosen Paket* gehören, darf man *nicht* in einer `import`-Vereinbarung angeben. Deshalb kann man in (Klassen und Schnittstellen in) *Paketen mit Namen* solche Klassen und Schnittstellen nicht benutzen.

587. Beim *Baum aller Java-Klassen* sind Klassen die Knoten. Ein Pfeil führt von einer Klasse `B` zu einer Klasse `A`, wenn `A` die direkte Oberklasse von `B` ist.

588. Ein *Wald* ist eine Menge von Bäumen.

589. Weil es in Java mehrere Top-Pakete gibt (d. h. Pakete, die in keinem Paket enthalten sind), bilden alle Pakete zusammen einen *Wald* 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, dann kann `K1` zum Paket `p1` oder zum Paket `p0` oder zu irgendeinem anderen Paket gehören.

591. Die mit `protected` gekennzeichneten Elemente einer Klasse sind *weniger* geschützt (von *mehr* Stellen des Programms aus erreichbar) als die nur *paketweit erreichbaren* Elemente (die mit *keinem* Erreichbarkeitsmodifizierer gekennzeichnet sind). Nur im Vergleich mit `public`-Elementen sind `protected`-Elemente *besser* geschützt (von *weniger* Stellen des Programms aus erreichbar).

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. Dann sollte man die Quellen von Klassen, die zu einem Paket namens `einausgabe.basis` gehören, im Unterverzeichnis `d:\projekt27\einausgabe\basis` bzw. `/projekt27/einausgabe/basis` ablegen.

593. Um 500 Variablen eines bestimmten Typs als *separate Variablen* zu vereinbaren, muss man 500 Vereinbarungen hinschreiben. Um diese Variablen zu bearbeiten (z. B. auszugeben) muss man ebenfalls etwa 500 Bearbeitungsbefehle (z. B. Ausgabebefehle wie `println`) hinschreiben. Wenn das Programm später um 5 Variablen erweitert werden soll, muss man 5 Vereinbarungen und 5 Bearbeitungsbefehle hinzufügen.

Eine Reihung mit 500 Komponenten kann man dagegen mit *einer* einzigen

- Vereinbarung (oft in einer Zeile oder mit wenigen Zeilen) vereinbaren. Um alle Komponenten einer solchen Reihung zu bearbeiten braucht man häufig nur eine kleine `for`-Schleife zu schreiben. Wenn das Programm später um 5 Variablen erweitert werden soll, muss man häufig nur die Vereinbarung der Reihung ein bisschen ändern, die `for`-Schleifen zur Bearbeitung funktionieren typischerweise ohne Änderungen auch für die erweiterte Reihung.
594. Sammlungen sind für den Programmierer in aller Regel *komfortabler* als Reihungen. Reihungen sind in Java *Objekte*, aber *Reihungstypen* sind *keine Klassentypen*, die man beerben und (um nützliche Bearbeitungsmethoden) erweitern könnte. Dagegen sind Sammlungsklassen Klassen, die man beerben und erweitern kann, und von denen es in der Java-Standardbibliothek schon viele für verschiedene Anwendungsfälle optimierte Varianten gibt. Nach dieser ziemlich abstrakten Charakterisierung hier ein konkretes Beispiel für die Vorteile von Sammlungen: Reihungen sind aus Beton (d. h. ihre Länge lässt sich nach ihrer Erzeugung nicht mehr verändern). Viele Sammlungen sind dagegen aus Gummi (d. h. ihre Länge kann nach ihrer Erzeugung noch verändert werden).
595. Sammlungen sind *Objekte*. Die deutsche Bezeichnung "*Elemente eines Objekts*" ist schon als Übersetzung von "*the members of an object*" sehr weit verbreitet. Deshalb sollte man "die Dinger, die in einer Sammlung gesammelt werden" nicht auch noch als *Elemente* bezeichnen, sondern besser als *Komponenten*. Die *Komponenten* eines Sammlungsobjekts sind immer *Objekte*, die *Elemente* des Sammlungsobjekts sind dagegen *Methoden* oder *Attribute* etc.
596. *Inhaltlich* ist eine Sammlung ein Objekt, in das man Objekte *einfügen* ("hineintun") kann, von dem man prüfen kann, ob es ein bestimmtes Objekt *enthält* und aus dem man Objekte wieder *entfernen* kann.
597. *Formal* ist eine Sammlung ein Objekt einer *Sammlungsklasse*.
598. Eine Sammlungsklasse ist eine Klasse, die die Schnittstelle `Collection` implementiert (d. h. eine *Collection-Klasse*).
599. Mit der Methode `asList` aus der Klasse `Arrays` kann man bewirken, dass eine Reihung `r` wie eine richtige Sammlung aussieht (der Ausdruck `Arrays.asList(r)` bezeichnet diese Sammlung). Im Alltag sagt man auch: "Die Methode `Arrays.asList` wandelt eine Reihung in eine Sammlung um". Dabei ist aber zu beachten, dass die Sammlung keine "Kopie der Reihung" ist. Vielmehr verändert sich die Reihung `r`, wenn man die Sammlung `Arrays.asList(r)` verändert, und umgekehrt verändert sich die Sammlung `Arrays.asList(r)`, wenn man die Reihung `r` verändert.
600. Die Sammlungsklassen in der Java-Standardbibliothek bilden ein zusammenhängendes System. Eine wichtige Eigenschaft dieses Systems ist die folgende:

- Wenn man eine Sammlung *eines* Typs hat, kann man daraus mit sehr wenig Programmierarbeit eine Sammlung eines beliebigen anderen Typs erzeugen lassen. Beispiel: Aus einer Sammlung des Typs `ArrayList<String>` kann man ganz leicht eine Sammlung des Typs `TreeSet<String>` erzeugen lassen.
601. Von einer Reihung mit nicht-primitiven Komponenten wie z. B.  

```
String[] sr = {"Anna", "Bert", "Claudia"};
```

sagt man häufig: Sie enthält (drei `String`-) *Objekte*. Tatsächlich enthält die Reihung (wenn man sie technisch etwas genauer beschreibt) nur drei *Referenzen*, die auf (`String`-) Objekte zeigen.
602. Angenommen, `s` ist ein `String`-Objekt, welches etwa 100 Tausend Zeichen enthält. Wenn ein heute üblicher, maschineller Java-Ausführer dieses Objekt in eine Sammlung einfügt, verändert er nur sehr wenige Bytes (z. B. 4 oder 8 Bytes), weil er ja (wenn man den Vorgang technisch etwas genauer beschreibt) nicht das große `String`-Objekt, sondern nur seine relativ kleine *Referenz* in die Sammlung einfügt.
603. Eine Schnittstelle `s` kann man als einen *Vertrag* auffassen. Dieser Vertrag gilt zwischen dem *Implementierer* (der `s` in einer Klasse `K` implementiert) und dem Anwender, der Objekte der Klasse `K` erzeugen lässt.
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 *harte* (oder: syntaktische) Bedingungen (deren Einhaltung der Java-Ausführer *überwacht*) und *weiche* (oder: semantische) Bedingungen (deren Einhaltung heutige Java-Ausführer noch *nicht* überwachen können).
605. Wir betrachten die folgende Schnittstelle:
- ```
1 interface Plus1 {
2     public int plus1(int n);
3     // Liefert den Wert n+1, wenn bei dessen Berechnung kein
4     // Ueberlauf auftritt. Liefert sonst den Wert n.
5 }
```
- Jeder Implementierer dieser Schnittstelle muss folgende *harten* Bedingungen erfüllen:
1. Er muss eine öffentliche (`public`) Methode namens `plus1` schreiben.
 2. Die Methode muss eine *Funktion* mit dem Ergebnistyp `int` sein.
 3. Die Methode muss genau *einen* Parameter vom Typ `int` haben.
- Die Einhaltung dieser Bedingungen wird vom Java-Ausführer *überwacht*.
Anmerkung: Ob man hier von *drei* Bedingungen spricht, oder sie zusammen als "eine komplexe Bedingung" auffasst, ist unwesentlich.
606. Jeder Implementierer der Schnittstelle `Plus1` sollte auch folgende *weiche* Bedingung erfüllen: Er sollte eine Funktion schreiben, die der Beschreibung in

den Zeilen 3 und 4 entspricht.

Die Einhaltung dieser Bedingung wird vom Java-Ausführer *nicht* überwacht (heute übliche, maschinelle Java-Ausführer können Java-Programme zwar erstaunlich schnell ausführen, aber leider nicht wirklich "verstehen").

607. In einer Sammlung des Typs `HashSet<String>` kann man (nur) `String`-Objekte sammeln.

Anmerkung: Die Klasse `java.lang.String` ist als `final`-Klasse vereinbart. Das bedeutet, dass man *keine* Unterklassen von ihr vereinbaren kann.

In einer Sammlung des Typs `HashSet<Number>` kann man beliebige `Number`-Objekte sammeln. Da unter anderem `Double` und `Integer` Unterklassen von `Number` sind, sind alle `Double`- und `Integer`-Objekte auch `Number`-Objekte und man kann sie in einer `HashSet<Number>`-Sammlung sammeln.

In einer Sammlung des Typs `HashSet<Object>` kann man alle möglichen Objekte (aber keine primitiven Werte) sammeln.

608. Der Name `Collection` (*ohne* `s` am Ende) bezeichnet eine *Schnittstelle*. Der Name `Collections` (*mit* einem `s` am Ende) bezeichnet eine *Klasse* (die nützliche Methoden zur Bearbeitung von `Collection`-Objekten enthält).

609. Die Schnittstelle `Collection` ist generisch und hat *einen* Typparameter, der für den Typ der *Komponenten* der Sammlung steht. Dieser Typparameter wird im Englischen häufig mit dem Buchstaben `E` (wie *element*) und im Deutschen mit dem Buchstaben `K` (wie *Komponente*) bezeichnet.

610. Zur Erinnerung: Sei `S` eine Schnittstelle. Dann ist eine *S-Klasse* eine Klasse, die die Schnittstelle `S` implementiert. Und ein *S-Objekt* ist ein Objekt einer *S-Klasse*.

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` mit *einem* Parameter vom Typ `String` vereinbaren. Das folgt aus der Vereinbarung der Schnittstelle `Collection` (siehe S. 441, die mit 1 nummerierte Zeile) und aus dem "Versprechen" `implements Collection<String>` in der Vereinbarung von `KlausDieter`.

612. Jedes `Collection`-Objekt muss unter anderem zwei Methoden namens `add` und `addAll` enthalten. Die Methode `add` sollte *eine* Komponente in die aktuelle Sammlung einfügen. Die Methode `addAll` hat eine Sammlung `c` als Parameter und sollte jede Komponente von `c` in die aktuelle Sammlung einfügen.

613. **Zur Erinnerung:** Aufgabe eines Konstruktor ist es, ein neues Objekt der betreffenden Klasse zu *initialisieren*.

Jede `Collection`-Klasse sollte mindestens *zwei* Konstruktoren enthalten:

1. Einen Standardkonstruktor, der das neue Sammlungsobjekt als *leere Sammlung* initialisiert.

2. Ein Konstruktor mit einer Sammlung `sam` als Parameter, der jede Komponente von `sam` in das neue Sammlungsobjekt einfügt.

614. Die Schnittstelle `Collection` enthält 15 Methoden. Davon sind 6 als *optional* gekennzeichnet. Das sind all die Methoden, mit denen man eine Sammlung *verändern* (vergrößern, verkleinern oder sonstwie verändern) kann. Wenn man eine dieser optionalen Methoden aufruft, dann darf dadurch eine Ausnahme des Typs `OperationNotSupportedException` ausgelöst werden.

615. Die vier Schnittstellen `Set<K>`, `SortedSet<K>`, `Queue<K>` und `List<K>` sind Erweiterungen der Sammlungsschnittstelle `Collection<K>`. Außerdem ist `SortedSet<K>` eine Erweiterung von `Set<K>`.

616. Die Sammlungsschnittstelle `Set<K>` ist eine Erweiterung der Sammlungsschnittstelle `Collection<K>`. Die Schnittstelle `Set` erweitert die Schnittstelle `Collection` um 0 (in Worten: um *null*) Methoden.

617. Die Schnittstelle `Set` erweitert die Schnittstelle `Collection` um die folgende *weiche Vertragsbedingung*: Jede `Set`-Klasse soll sicherstellen, dass ihre Sammlungsobjekte *keine Doppelgänger* enthalten können (weil `Set`-Objekte mathematische Mengen repräsentieren sollen und die auch keine Doppelgänger enthalten dürfen).

618. Zwei Objekte `ob1` und `ob2` gelten als Doppelgänger voneinander, wenn der Ausdruck `ob1.equals(ob2)` den Wert `true` hat. Außerdem sollte auch der Ausdruck `ob2.equals(ob1)` den Wert `true` haben (sonst ist mindestens eine `equals`-Methode falsch programmiert).

619. Seien `s1` und `s2` zwei `String`-Objekte. Dann hat der Ausdruck `s1.equals(s2)` den Wert `true`, wenn die *Zielwerte* der Variablen `s1` und `s2` gleich sind (oder: wenn `s1` und `s2` gleiche Zeichenketten repräsentieren).

620. Seien `b1` und `b2` zwei `StringBuilder`-Objekte. Dann hat der Ausdruck `b1.equals(b2)` den Wert `true`, wenn die *Werte* der Variablen `b1` und `b2` gleich sind (und damit der Zielwert von `b1` identisch ist mit dem Zielwert von `b2`).

621. Die Java-Standardklassen `HashSet<K>`, `LinkedHashSet<K>` und `TreeSet<K>` implementieren die Schnittstelle `Set<K>`.

622. Die Sammlungsschnittstelle `Queue<K>` ist eine Erweiterung der Sammlungsschnittstelle `Collection<K>`. Die Schnittstelle `Queue` erweitert die Schnittstelle `Collection` um 5 Methoden.

623. Die Schnittstelle `Queue` erweitert die Schnittstelle `Collection` um die folgende *weiche Vertragsbedingung*: In jeder (nicht-leeren) `Queue`-Sammlung soll

- in jedem Moment eine Komponente als *Kopf* der Sammlung (engl. head of the collection) ausgezeichnet sein.
624. Als *Schlangen* bezeichnet man `Queue`-Sammlungen, bei denen jeweils die *zuletzt eingefügte* Komponente als Kopf ausgezeichnet ist.
625. Als *Prioritätsschlangen* bezeichnet man `Queue`-Sammlungen, bei denen jeweils die *kleinste* Komponente als Kopf ausgezeichnet ist.
626. Die Sammlungsschnittstelle `List<K>` ist eine Erweiterung der Sammlungsschnittstelle `Collection<K>`. Die Schnittstelle `List` erweitert die Schnittstelle `Collection` um 10 Methoden.
627. Die Schnittstelle `List` erweitert die Schnittstelle `Collection` um die *Vorstellung*, dass die Komponenten einer Sammlung an bestimmten *Positionen* liegen, die durch *Indizes* gekennzeichnet sind (ähnlich wie bei einer Reihung).
628. Angenommen, Sie haben ein Objekt `obl` in ein `Collection`-Objekt `samC` eingefügt und wollen es jetzt mit der Methode `remove` wieder entfernen. Dann müssen Sie beim Aufruf der Methode `samC.remove` als Parameter das Objekt `obl` angeben. Sie können ein bestimmtes Objekt `obl` also nur dann aus einem `Collection`-Objekt entfernen, wenn Sie `obl` als Parameter angeben können.
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: Die `remove`-Methode aus der Schnittstelle `Collection` erwartet das Objekt `obl` als Parameter (wie in der vorigen Antwort). Die `remove`-Methode aus der Schnittstelle `List` erwartet als Parameter "nur" den *Index*, an dem das zu entfernende Objekt steht.
630. Sei `samL` ein `List`-Objekt, in das schon zahlreiche Objekte eingefügt wurden. Dann enthält das `List`-Objekt `samL.subList(3, 5)` *zwei* Komponenten der Sammlung `samL`, nämlich die an den Indexpositionen 3 und 4 (aber nicht die Komponente an der Indexposition 5!).
631. Die sechs Worte Bert, Zoo, Ballkleid, Alber, Berta, Anna in *lexikografischer* und in *lexikalischer* Reihenfolge:
- | | |
|-----------|-----------|
| Albert | Zoo |
| Anna | Anna |
| Ballkleid | Bert |
| Bert | Berta |
| Berta | Albert |
| Zoo | Ballkleid |
632. Verschiedene Kriterien, nach denen man Autos sortieren kann:
- Preis, Name des Besitzers, Wohnort des Besitzers, Baujahr, Hubraum in cm³, Leistung in KW, Name der Farbe, Gewicht, Durchschnittlicher Kraftstoffver-

- brauch pro 100 km etc. Nach jedem dieser (mehr oder weniger eindeutig definierten) Kriterien kann man Autos aufsteigend oder absteigend sortieren.
633. Beispiel für Mengen, die man nicht ohne weiteres sortieren kann:
- Die Punkte (x, y) einer Ebene, Farben (werden häufig nicht total geordnet, sondern auf einer Ebene oder als Farbkörper angeordnet), die Knoten eines Baumes (sind nur partiell geordnet, aber nicht total).
634. Die Schnittstellen `Comparable` und `Comparator` beschreiben Vergleichsfunktionen, mit deren Hilfe man die Objekte einer Klasse sortieren kann.
635. Die Schnittstelle `Comparable` enthält nur eine einzige Methode namens `compareTo`. Diese Methode hat *einen* Parameter und den Ergebnistyp `int`.
636. Die Schnittstelle `Comparator` enthält nur eine einzige Methode namens `compare`. Diese Methode hat *zwei* Parameter und den Ergebnistyp `int`.
637. Angenommen, wir wollen die Objekte einer Klasse `Auto` nach fünf verschiedenen Kriterien vergleichbar (und damit sortierbar) machen. Dann sollten wir die wichtigste, "natürliche" Ordnung für `Auto`-Objekte mit Hilfe der Schnittstelle `Comparable` definieren. Die übrigen vier Ordnungen für `Auto`-Objekte sollten wir mit Hilfe der Schnittstelle `Comparator` definieren.
638. Angenommen, wir wollen die Objekte einer Klasse `Auto` nach fünf verschiedenen Kriterien vergleichbar (und damit sortierbar) machen. Dann sollten (oder: müssen) wir die Schnittstelle `Comparable` in der Klasse `Auto` implementieren. Die Schnittstelle `Comparator` sollten wir in vier verschiedenen, separaten Klassen implementieren.
639. Wenn man in der Klasse `Auto` eine Instanz der Schnittstelle `Comparable` implementiert, dann sollte das in aller Regel die Instanz `Comparable<Auto>` sein. In einigen Fällen kommt auch eine Instanz wie `Comparable<Kraftfahrzeug>` in Frage, wobei `Kraftfahrzeug` eine Oberklasse von `Auto` ist.
640. Von einer generischen Schnittstelle darf man in einer Klasse *höchstens eine* Instanz implementieren.
641. Sammlungen des Typs `SortedTree` werden vom Ausführer automatisch "in einem sortierten Zustand gehalten", auch wenn man wiederholt neue Komponenten einfügt und alte Komponenten aus der Sammlung entfernt.
642. Die Sammlungsschnittstelle `SortedSet<K>` ist eine Erweiterung der Sammlungsschnittstelle `Set<K>`. Die Schnittstelle `SortedSet<K>` erweitert die Schnittstelle `Set<K>` um sechs Methoden.
643. Die Schnittstelle `SortedSet` erbt von der Schnittstelle `Set` folgende weiche Bedingung: Auch jede `SortedSet`-Klasse soll sicherstellen, dass ihre Sammlungsobjekte *keine Doppelgänger* enthalten können (und damit Ähnlichkeit mit mathematischen Mengen haben).

644. Zur Schnittstelle `SortedSet` gehört auch noch folgende (nicht geerbte) weiche Bedingung: Die Komponenten einer `SortedSet`-Sammlung sollten immer sortiert sein.
645. Die Klasse `TreeSet` ist die einzige Klasse aus der Java-Standardbibliothek (Stand 13.11.2005, 8.30 Uhr GMT :-), die die Schnittstelle `SortedSet<K>` implementiert.
646. Die Sammlungsklasse `LinkedList` ist die einzige Standardklasse, die zwei (voneinander unabhängige) Schnittstellen implementiert, nämlich die Schnittstellen `Queue<K>` und `List<K>`.
647. Sammlungen des Typs `HashSet<K>` sind im allgemeinen sehr *schnell*, aber *nicht sortiert*.
648. Nein, es gibt (in der Java-Standardbibliothek) keine *Klasse* namens `Collection` (ohne s), aber eine *Schnittstelle* mit diesem Namen.
649. Mit den Methoden im Modul (oder: in der Klasse) `Collections` kann man *Sammlungen bearbeiten*, z. B. sortieren, durcheinander bringen oder prüfen.
650. Mit der Methode `Collections.shuffle` kann man eine Sammlung durcheinander bringen ("in eine zufällige Reihenfolge bringen" oder "mischen").
651. Mit der Methode `Collections.reverse` kann man die Reihenfolge der Komponenten in einer `List`-Sammlung umkehren.
652. Nein, technisch gesehen sind Reihungen *keine* Sammlungen, weil Reihungstypen nicht die Schnittstelle `Collection` implementieren. Trotzdem haben Reihungen große *Ähnlichkeit* mit Sammlungen.
653. Mit Hilfe der Methode `Arrays.asList` kann man eine Reihung *als Sammlung sehen und bearbeiten*.
654. Mit den zwei Methoden namens `toArray` aus der Schnittstelle `Collection` kann man aus einer Sammlung eine Reihung erzeugen.
655. Eine *Abbildung* besteht aus (oder: enthält) *Einträge*.
656. Ein *Eintrag* besteht aus (oder: enthält) zwei Objekte, ein *Schlüsselobjekt* und ein *Wertobjekt*.
657. Eine *Abbildungsklasse* ist eine Klasse, die die Schnittstelle `Map<S, W>` (oder, wie meistens, eine *Instanz* dieser Schnittstelle) implementiert.
658. Die generische Schnittstelle `Map<S, W>` hat zwei (formale) Typparameter `S` und `W`. Dabei steht `S` für den Typ den *Schlüsselobjekte* und `W` für den Typ der *Wertobjekte* in den Einträgen der Abbildungen.
659. In der Schnittstelle `Map<S, W>` hat die Methode `keySet` den Ergebnistyp `Set<S>`, weil jedes *Schlüsselobjekt* in höchstens *einem* Eintrag einer Abbildung vorkommen darf (bei den Schlüsselobjekten darf es keine Doppelgänger geben). Die Methode `values` hat dagegen den Ergebnistyp `Collection<W>`, weil ein *Wertobjekt* in *mehreren* Einträgen einer Abbildung vorkommen darf.

660. Wenn man z. B. ein `StringBuilder`-Objekt `sb` als Schlüssel in eine Abbildung einfügt und es (während es in der Abbildung eingetragen ist) verändert, kann die Struktur der Abbildung dadurch erheblich gestört werden. Eine mögliche Folge ist, dass Zugriffe auf die Sammlung dann möglicherweise falsche Ergebnisse liefern. Bei auf Sicherheit ausgerichteten Abbildungsklassen werden deshalb nur *Kopien* von (Schlüssel- und Wert-) Objekten in die Sammlung eingefügt, nicht Referenzen auf die Original-Objekte des Abbildungsbenutzers.
661. Ob in einer Abbildung `null` als Schlüssel bzw. als Wert erlaubt ist oder nicht, hängt vom genauen Typ der Abbildung ab. Einige Abbildungsklassen aus der Java-Standardbibliothek erlauben `null`, andere verbieten `null`.
662. 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, wird eine kleine Fehlermeldung zum Bildschirm ausgegeben und die Funktion `EM.liesInt()` liefert den `int`-Wert 0 als Ergebnis.
663. Wenn man mit der Methode `EM.liesDouble()` versucht, einen `double`-Wert von der Tastatur einzulesen, und der Benutzer die Zeichenkette `inf` eingibt, liefert die Funktion `EM.liesDouble` den `double`-Wert `infinity`. Das ist aber nur eine kleine Eingabehilfe, die in die Methode `EM.liesDouble` eingebaut wurde. Andere Einlesemethoden bieten mehr oder weniger solche Hilfen.
664. Wenn man einen Ausgabebefehl wie z. B. `pln("Hallo wie geht es?");` ausführen läßt, werden die Daten häufig nicht wirklich ausgegeben. Statt dessen schreibt der Ausführer die Daten erstmal in einen sogenannten *Ausgabepuffer* und gibt sie erst später aus, wenn sich in diesem Puffer genügend Daten angesammelt haben (oder ein anderes Ereignis die tatsächliche Ausgabe zum Bildschirm verursacht). Diese *gepufferte Ausgabe* ist sinnvoll, weil es (bei heute üblichen, maschinellen Java-Ausführern) viel weniger Zeit kostet, einmal 100 Zeichen auszugeben, statt 100 Mal ein Zeichen.
665. Ströme dienen dazu, Daten ein- bzw. auszugeben. Darüber hinaus kann man mehrere Ströme relativ leicht miteinander *kombinieren* oder "zusammenstecken".
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 --> ???
```

Die Fragezeichen ??? ganz rechts stehen immer für das *Programm*, welches Daten ausgibt bzw. einliest. Die Fragezeichen ??? ganz links stehen z. B. für eine *Datei* und allgemein für die *endgültige Datensenke*, in die Daten geschrie-

ben werden bzw. für die *endgültige Datenquelle*, aus der Daten gelesen werden:

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

Die Ströme zeichnen wir immer (von links nach rechts) in der Reihenfolge, in der man sie auch *vereinbaren* muss.

667. Zu einem Bildschirm kann man eigentlich nur Daten der Typen `char` und `String` ausgeben. Daten aller anderen Typen müssen vor der Ausgabe zu einem Bildschirm in Werte der Typen `char` bzw. `String` umgewandelt werden.
668. Von einer Tastatur kann man eigentlich nur Daten der Typen `String` bzw. `char` einlesen.
669. Wenn man z. B. den `int`-Wert 17 mit einem Ausgabebefehl wie `println(17);` zum Bildschirm ausgibt, wird der `int`-Wert 17 zuerst in ein `String`-Objekt "17" umgewandelt, und dieses `String`-Objekt wird ausgegeben. Mit anderen Ausgabebefehlen könnte man den `int`-Wert 17 auch in ein anderes `String`-Objekt umwandeln lassen, z. B. in "+17" oder " 17" oder "0x11" oder "10001" oder "siebzehn" etc.).
670. Wenn man z. B. den `int`-Wert 17 von der Tastatur einliest, passiert folgendes: Der Benutzer muss eine geeignete Zeichenkette eintippen (z. B. 17 oder +17 oder 00017 oder 0x11 etc.). Diese Zeichenkette wird in ein `String`-Objekt eingelesen. Dann versucht der Ausführer, die Zeichenkette in einen `int`-Wert umzuwandeln. Wenn das gelingt, ist alles in Ordnung. Was passiert, wenn die Umwandlung *nicht gelingt*, hängt von der verwendeten Einlesemethode ab.
671. In der Java-Standardbibliothek gibt es vier *abstrakte Stromklassen*, von denen jede die Wurzel eines kleinen Typgraphen ist. Diese Stromklassen heißen `InputStream`, `OutputStream`, `Reader` und `Writer`.
672. Mit Java-Stromobjekten kann man Daten in folgende *Datensenken* schreiben: In eine Datei, in eine Reihungen vom Typ `byte[]` oder `char[]` und in ein `String`-Objekt.
673. Mit Java-Stromobjekten kann man Daten aus folgenden *Datenquellen* lesen: Aus einer Datei, aus einer Reihungen vom Typ `byte[]` oder `char[]` und aus einem `String`-Objekt. Ausserdem gilt: Ist `url` ein `URL`-Objekt, dann liefert der Funktionsaufruf `url.openStream()` einen Eingabestrom, aus dem man „die Daten der URL“ aus dem Internet lesen kann. Somit ist ein `url`-Objekt auch eine Art *Datenquelle*, von der man Daten einlesen kann.
674. `FileInputStream` und `FileReader` sind Stromklassen, mit deren Objekten man Daten aus einer Datei *lesen* kann.

675. `FileOutputStream`, `FileWriter`, `PrintStream` und `PrintWriter` sind Stromklassen, mit deren Objekten man Daten in eine Datei *schreiben* kann.
676. Objekte verschiedener Stromklassen kann man mit einer *Datei* verbinden (siehe die vorigen beiden Antworten). Dabei kann man die Datei auf eine der folgenden drei Weisen angeben:
1. Durch einen *String*, der den Pfadnamen der Datei enthält, z. B. `"c:/Karl/Java/Hallo01.java"` oder `"c:\\Karl\\Java\\Hallo01.java"` unter Windows, oder `"/usr/karl/java/Hallo01.java"` unter Unix.
 2. Durch ein Objekt der Klasse `File`.
 3. Durch ein Objekt der Klasse `FileDescriptor`.
677. Die Standardströme `out`, `err` und `in` hat man in Java als *Klassenattribute* der Klasse `System` realisiert.
678. Unter Windows und Unix sind die Standardströme `System.out`, und `System.err` normalerweise mit einem *Bildschirm* und `System.in` mit einer *Tastatur* verbunden (und zwar mit *dem* Bildschirm bzw. *der* Tastatur, von denen aus das betreffende Programm gestartet wurde).
679. Mit den Methoden `System.setOut`, `System.setErr` bzw. `System.setIn` kann man die Standardströme `System.out`, `System.err` und `System.in` mit anderen Geräten oder Dateien verbinden.
680. Die Standardströme `System.out` und `System.err` sind Objekte des Typs `PrintStream`.
Der Standardstrom `System.in` ist ein Objekt des Typs `InputStream`.
681. Während der Ausführung eines Java-Programms werden Zeichen (`char`-Werte) grundsätzlich im *Unicode* dargestellt.
682. Ein `OutputStreamWriter` bildet mit Hilfe einer *Endodierung* länderunabhängige `Unicode-char`-Werte auf länderspezifische `byte`-Werte ab.
683. Ein `InputStreamReader` bildet mit Hilfe einer *Enkodierung* länderspezifische `byte`-Werte auf länderunabhängige `Unicode-char`-Werte ab.
684. Ein Objekt zu *serialisieren* bedeutet, aus dem Objekt "eine Serie von Bytes zu erzeugen", die man in eine Datei schreiben kann.
Anmerkung: Ein Objekt kann Attribute enthalten, die auf andere Objekte zeigen. Somit können Objekte beliebig komplizierte Graphen (und nicht nur "serielle" oder "lineare" Strukturen) bilden. Beim Serialisieren eines Objekts wird der gesamte, typischerweise komplizierte, "nicht-serielle" Graph, zu dem das Objekt gehört, in eine einfache Folge ("Serie") von Bytes übersetzt.
685. Angenommen, wir haben in einem Programm ein Objekt `ob` serialisiert und die dabei erzeugte "Serie von Bytes" in eine Datei geschrieben. Das Objekt zu *deserialisieren* bedeutet, diese Serie von Bytes später (möglicherweise mit ei-

- nem anderen Programm) wieder einzulesen und daraus das ursprüngliche Objekt `ob` zu rekonstruieren.
686. Sei `oos` ein Objekt der Klasse `ObjectOutputStream` und sei `ob` irgendein Objekt. Dann bewirkt der Befehl `oos.writeObject(ob)`, dass das Objekt `ob` und alle Objekte, mit denen es durch seine (Referenz-) Attribute direkt oder indirekt verbunden ist, serialisiert und in den Strom `oos` geschrieben werden.
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`. Dann bewirkt der Befehl `bs = ois.readObject()` folgendes:
Aus dem Strom `ois` wird das nächste Objekt gelesen, deserialisiert und als Zielwert von `bs` gespeichert. Ausserdem werden aus dem Strom `ois` alle Objekte gelesen und deserialisiert, mit denen dieses Objekt durch seine (Referenz-) Attribute direkt oder indirekt verbunden ist.
688. Röhren (pipes) eignen sich besonders gut dazu, Fäden (oder: Steuerfäden, engl. threads of control) miteinander zu verbinden. Weil *Fäden* nebenläufig zueinander ("zeitlich unabhängig voneinander") ausgeführt werden, ist eine Pufferung von Daten in einer Röhre zwischen ihnen häufig vorteilhaft.
689. Sei `pw` ein `PipedWriter`-Objekt und `pr` ein `PipedReader`-Objekt und seien `pw` und `pr` zu einer Röhre (pipe) verbunden. Der Datenfluss durch diese Röhre sieht dann (als simple ASCII-Graphik dargestellt) etwa so aus:
- ```

1 pw <-- Programm
2 |
3 pr --> Programm

```
- Auch bei dieser Graphik haben wir die Konvention eingehalten, das Programm immer *rechts* von den Stromobjekten darzustellen.
690. Mit Stromobjekten der folgenden Klassen kann man mehrere Dateien zu einem Archiv zusammenfassen und Daten beim Schreiben komprimieren und beim Lesen dekomprimieren:  
`GZIPInputStream`, `ZIPInputStream`  
`GZIPOutputStream`, `ZIPOutputStream`
691. Mit Stromobjekten der Klasse `SequenceInputStream` kann man *mehrere* `InputStream`-Objekte zu *einem* Eingabestrom zusammenfassen.
692. Mit Stromobjekten der Klasse `StreamTokenizer` kann man besonders bequem Zeichenfolgen lesen, die durch bestimmte Trennzeichen (z. B. durch Kommas oder Blanks) voneinander getrennt sind.
693. Stromobjekte der Klasse `LineNumberReader` verfolgen automatisch die Anzahl der bereits eingelesenen Zeilenendemarkierungen (d. h. "die Zeilen-Nummer").

694. Stromobjekte der Klasse `PushBackReader` 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).
695. Beim Entwickeln der Pakete `java.io` und `java.nio` hat man vor allem versucht, die sich widersprechenden Ziele *Plattformunabhängigkeit* und *Schnelligkeit* zu erreichen.
696. Das Paket `java.io` ist stärker auf *Plattformunabhängigkeit* ausgerichtet, und das Paket `java.nio` mehr auf *Schnelligkeit*.
697. Die Klassen im Paket `java.io` unterstützen die Ein-/Ausgabe von Daten mit Hilfe von *Strömen* (Stromobjekten). Die Klassen im Paket `java.nio` unterstützen die Ein-/Ausgabe mit sogenannten `Channel`-Objekten und `Buffer`-Objekten.
698. Bei einer *sequenziellen Datei* muss man die Daten in der Reihenfolge lesen, in der sie in die Datei geschrieben wurden. Bei einer *Direktdatei* kann man zu einer beliebigen Stelle ("Byte-Nr") direkt hinspringen und ab dieser Stelle lesen oder schreiben.
699. Eine Datei muss keinerlei "Strukturdaten" enthalten, damit man sie als *Direktdatei* bearbeiten kann. Im Prinzip kann man also *jede* Datei als Direktdatei bearbeiten (mit Ausnahme von Dateien, die auf ungeeigneten Datenträgern stehen, z. B. auf Magnetbändern).
700. Bei einem sequenziellen Programm werden alle Befehle einer nach dem anderen (sequenziell) ausgeführt und die Reihenfolge dieser Befehlsausführungen ist durch die verwendete Programmiersprache (und durch die Eingabedaten) genau festgelegt.
701. Das Problem, von zwei Tastaturen (an denen zwei Benutzer sitzen) Zahlen einzulesen und zu addieren kann man mit einem sequenziellen Programm nicht wirklich lösen (siehe S. 494/495).
702. Wenn der Programmierer dem Ausführer befiehlt, zwei Blöcke (Block1 und Block2) *nebenläufig* zueinander auszuführen, hat der Ausführer folgende Wahl bei der Ausführung der Blöcke:  
- Erst Block1, dann Block2  
- Erst Block2, dann Block1  
- Stückchenweise abwechselnd (ein Stückchen von Block1, dann ein Stückchen von Block2, dann wieder ein Stückchen von Block1 etc.)  
- Block1 und echt gleichzeitig dazu Block2.
703. Weil die Bezeichnung "nebenläufiges Programm" mehrdeutig ist, haben wir den Begriff *Proma* eingeführt. Ein *Proma* ist ein *Programm*, welches von *mehreren Ausführern* ausgeführt wird. Jeder Ausführer arbeitet sequenziell, aber



- nebenläufig zu den andern, d. h. alle Ausführer arbeiten stückchenweise abwechselnd in irgendeiner Reihenfolge oder gleichzeitig.
704. Nach Einführung des Begriffs *Proma* kann man den Begriff *sequenzielles Programm* so definieren: Ein sequenzielles Programms ist eines, welches von nur *einem* sequenziellen Ausführer ausgeführt wird.
705. Warum ist ein *Proma* in aller Regel erheblich schwerer zu testen als ein sequenzielles Programm?
706. Java-Programme mit einer Grabo (grafischen Benutzeroberfläche, engl. GUI) sind automatisch *Promas* ("Programme mit mehreren Ausführern"), auch wenn der Programmierer darin keine Fäden gestartet hat.
707. In der Praxis unterscheidet man folgende *nebenläufige Einheiten*: *Prozesse* und *Fäden* (engl. threads of control).
708. Sowohl Prozesse als auch Fäden werden *nebenläufig* (zu anderen Prozessen bzw. zu anderen Fäden) *ausgeführt*.
709. Jeder Prozess wird in einem eigenen *Adressraum* ausgeführt. Deshalb kostet es (bei heute üblichen maschinellen Java-Ausführern mit nur einem zentralen Prozessor) relativ viel Zeit, die Ausführung eines Prozesses zu unterbrechen und die Ausführung eines anderen Prozesses fortzusetzen ("Prozesswechsel sind relativ teuer").
- Im Adressraum *eines* Prozesses können *viele* Fäden ausgeführt werden. Wenn mehrere Fäden im selben Adressraum ausgeführt werden, ist es relativ billig, die Ausführung eines Fadens zu unterbrechen und die Ausführung eines anderen Fadens fortzusetzen ("Fadenwechsel sind relativ billig").
710. Fäden, die im selben Adressraum ausgeführt werden, können sich gegenseitig stören (weil sie prinzipiell die Möglichkeit haben, schreibend auf gemeinsam genutzte Variablen zuzugreifen). Prozesse können sich nicht so leicht stören, weil ein Prozess p1 keinen Zugriff auf den Adressraum eines anderen Prozesses p2 hat. Deshalb verwendet man in der Praxis sowohl (sichere) Prozesse als auch (schnelle) Fäden.
711. Wie sich zwei Fäden beim Verändern einer gemeinsam genutzten Variablen stören können, wird im Buch auf S. 499 beschrieben.
712. Prozesse können sich gegenseitig stören, wenn sie z. B. versuchen, eine gemeinsam genutzte Datei (d. h. einen gemeinsam genutzten Wertebehälter) zu verändern.
713. In Java kann man Fäden auf zwei verschiedene Weisen programmieren:
1. Indem man die Klasse `Thread` erweitert.
  2. Indem man die Schnittstelle `Runnable` implementiert.
714. Ein `Thread`-Objekt oder ein `Runnable`-Objekt besteht im Wesentlichen aus einer parameterlosen Prozedur namens `run`. Diese `run`-Methode wird neben-

- läufig zu den `run`-Methoden anderer `Thread`- oder `Runnable`-Objekte ausgeführt.
715. Einem Java-Ausführer kann man befehlen, sich in zwei Ausführer aufzuspalten (oder: einen neuen, zusätzlichen Ausführer zu erzeugen), indem man ihm befiehlt, ein `Thread`-Objekt zu erzeugen und es zu starten (siehe im Buch S. 501 und 503).
716. Wenn der Java-Ausführer ein Programm ausführt, besteht er aus mindestens einem Faden, dem sogenannten *Hauptfaden* (engl. main thread). Dieser Hauptfaden hat folgende Aufgaben:
1. Die `.class`-Datei zu finden und zu laden, die die Hauptklasse des Programms enthält.
  2. In dieser Hauptklasse eine `main`-Methode zu finden.
  3. Die `main`-Methode auszuführen.
717. Viele (z. B. 500) Fäden eines bestimmten Typs kann man in Form einer *Reihung* (mit z. B. 500 Komponenten) vereinbaren. Um alle Fäden in einer solchen Reihung zu starten muss man eine kleine `for`-Schleife programmieren (ca. 3 Zeilen). Diese Vorgehensweise ist besonders in Deutschland empfehlenswert (wegen des relativ hohen Lohnniveaus).
718. Der Sinn und Zweck von Fadengruppen (`ThreadGroup`-Objekten) ist es, *Ausnahmen*, die in jedem Faden einer Gruppe auftreten können, an *einer* gemeinsamen Stelle im Programm zu fangen und zu behandeln (statt in jedem einzelnen Fadenobjekt).
719. Wenn der Ausführer sich bei der Ausführung eines Java-Programms in mehrere Fäden aufgespalten hat, gilt folgendes:
1. *Klassenattribute* und *Objektattribute* werden in der Klasse bzw. in jedem Objekt nur *einmal* erzeugt und alle Fäden, die auf die Klasse bzw. das Objekt zugreifen können, können diese Variablen gemeinsam nutzen (und sich dabei stören).
  2. *Lokale Variablen* (die in einer Methode, in einem Konstruktor oder in einem Initialisierer vereinbart wurden) werden von jedem Faden erneut erzeugt. Auf die von einem Faden F erzeugten lokalen Variablen hat nur F Zugriff (jeder andere Faden hat "seine eigenen Exemplare dieser Variablen").
720. Eine Methode ist *fadensicher*, wenn dadurch, dass mehrere Fäden versuchen, sie auszuführen, keine Fehler entstehen können (die nicht auch ein einzelner Faden erzeugen könnte).
- Anmerkung:** Die im Buch auf S. 510 angegebene Definition sollte durch die hier stehende ersetzt werden.

721. In Java kann man mit dem `synchronized`-Befehl bewirken, dass eine Methode nur von einem Faden auf einmal ausgeführt werden kann (und damit automatisch fadensicher ist).

722. Ein Beispiel für eine *nicht* fadensichere Methode:

```
1 int zaehler = 0;
2 void plus1() {
3 // Erhoeht die Variable zaehler um 1
4 zaehler++;
5 }
```

723. Ein Beispiel für eine *fadensichere* Methode:

```
1 int zaehler = 0;
2 int getZaehler() {
3 // Liefert den momentanen Wert von zaehler
4 return zaehler;
5 }
```

724. *Verklemmungen* (engl. deadlocks) sind Programmfehler, die man mit Hilfe von `synchronized`-Befehlen erzeugen kann.

725. Um eine Verklemmung zu programmieren, braucht man mindestens zwei Fäden und zwei reservierbare Objekte (zwei `synchronized`-Befehle).

726. Verklemmungen kann man sicher verhindern, indem man alle zu reservierenden Objekte irgendwie durchnummeriert und eine Reservierung mehrerer Objekte nur in aufsteigender Reihenfolge dieser Nummern erlaubt. Zum Beispiel ist dann eine Reservierung der Objekte Nr. 12, 15 und 27 (in dieser Reihenfolge) erlaubt. Eine Reservierung derselben Objekte in der Reihenfolge 15, 12, 27 ist dagegen verboten.

727. Zur sogenannten *Reflektionsschnittstelle* von Java gehören die etwa 20 Klassen und Schnittstellen im Paket `java.lang.reflect` und ("als Kernstück oder Filet") die Klasse `java.lang.Class`.

728. Alle Programme, Klassen, Methoden etc., die irgendwie mit Reflexion zu tun haben, bezeichnen wir als *reflektiv*.

729. Jedes Objekt `c` der Klasse `Class` reflektiert eine Klasse `K`. Damit ist gemeint: Das `Class`-Objekt `c` enthält alle wichtigen Informationen über die Klasse `K`, darunter die Antworten auf folgende Fragen:

- Wieviele *Konstruktoren* wurden in dieser Klasse vereinbart?
- Wieviele *Parameter* haben die einzelnen Konstruktoren?
- Von welchen *Typen* sind diese Parameter?
- Wieviele *Methoden* wurden in dieser Klasse vereinbart?
- Wieviele *Parameter* haben die einzelnen Methoden?
- Von welchen Typen sind diese Parameter?
- Welche *Ergebnistypen* haben die einzelnen Methoden?

- Welche Methoden sind *Klassenmethoden* und welche sind *Objektmethoden*?
- Wieviele *Attribute* wurden in dieser Klasse vereinbart?
- Von welchen *Typen* sind die einzelnen Attribute?
- Welche Attribute sind *Klassenattribute* und welche sind *Objektattribute*?
- etc.

730. Während der Ausführung eines Java-Programms existieren seine Klassen als Objekte der Klasse `Class` (oder "in Form von `Class`-Objekten").

731. Der Programmierer kann sich auf die folgenden drei verschiedene Weisen Zugriff auf das `Class`-Objekt verschaffen, welches die Klasse `StringBuilder` der reflektiert:

1. Mit Hilfe des Ausdrucks `StringBuilder.class` !
2. Mit Hilfe des Ausdrucks `sb.getClass()` !  
(wobei `sb` irgendein `StringBuilder`-Objekt sein muss)!
3. Mit Hilfe des Ausdrucks

`Class.forName("java.lang.Stringbuilder")` !

Von den in der Frage erwähnten Möglichkeiten ist die erste also ganz *falsch* (der Programmierer kann keine neuen `Class`-Objekte erzeugen lassen),

die zweite und dritte Möglichkeit sind *richtig* und

die vierte Möglichkeit ist *fast richtig* (aber man muss der Methode `forName` den *vollen Namen* der Klasse übergeben, nicht nur ihren *einfachen Namen*).

732. Mit Hilfe der folgenden drei Ausdrücke kann man auf das `Class`-Objekt zugreifen, welches den Reihungstyp `long[][]` reflektiert (dabei muss die Variable `lr1` auf eine Reihung des Typs `long[][]` zeigen):

1. `... long[][].class ...`
2. `... lr1.getClass() ...`
3. `... Class.forName("[[J") ...`

733. Mit Hilfe der folgenden drei Ausdrücke kann man auf das `Class`-Objekt zugreifen, welches den Reihungstyp `Short[][][]` reflektiert (dabei muss die Variable `sr1` auf eine Reihung des Typs `Short[][][]` zeigen):

1. `... Short[][][].class ...`
2. `... sr1.getClass() ...`
3. `... Class.forName("[[[Ljava.lang.Short;") ...`

734. Mit Hilfe der folgenden beiden Ausdrücke kann man auf das `Class`-Objekt des primitiven Typs `double` bzw. der Hüllklasse `Double` zugreifen:

1. `... Double.TYPE ...` // `Class`-Ob. des primitiven Typs
2. `... Double.class ...` // `Class`-Ob. der Hüllklasse

735. Für die im Abschnitt 21.2 behandelte Aufgabe ("Methoden einer beliebigen Klasse aufrufen") wird eine Lösung entwickelt, die aus drei Methoden namens `main`, `pruefeUndRufeAuf` und `rufeAuf` besteht.

736. Um *reflektiv* auf die Oberklassen, Konstruktoren, Methoden oder Attribute einer Klasse `AbRechnung` zuzugreifen muss man sich zuerst einmal Zugriff auf das `Class`-Objekt `AbRechnung.class` verschaffen, welches die Klasse `AbRechnung` reflektiert.
737. Auch eine generische Klasse wie `Vector<K>` wird nur durch ein einziges `Class`-Objekt reflektiert. Dieses eine `Class`-Objekt reflektiert alle Typen, die die Klasse `Vector` repräsentiert: Den rohen Typ `Vector` und alle parametrisierten Typen wie `Vector<String>`, `Vector<Integer>` und `Vector<Double>` etc. Mit anderen Worten: Ein `Class`-Objekt reflektiert eine *Klasse* ("mit allen ihren Typen"), und nicht nur einen *Typ*.
738. Sei `kob` das `Class`-Objekt, welches die Klasse `AbRechnung` reflektiert. Dann liefert die Funktion `kob.getDeclaredMethods` alle in der Klasse `AbRechnung` selbst vereinbarten Methoden, auch die nicht-öffentlichen (aber keine *geerbten* Methoden). Die Funktion `kob.getMethods` liefert dagegen alle öffentlichen (`public`) Methoden der Klasse `AbRechnung`, unabhängig davon, ob dieses Methoden in der Klasse `AbRechnung` selbst vereinbart oder geerbt wurden.
739. Sei `kob` das `Class`-Objekt, welches die Klasse `AbRechnung` reflektiert. Dann liefert die Funktion `kob.getSuperclass` das `Class`-Objekt, welches die *direkte Oberklasse* der Klasse `AbRechnung` reflektiert.
740. Unterschiede zwischen *Grabo*-Programmen und *Konsolen*-Programmen:  
**Grabo-1:** *Grabo*-Programme haben eine *grafische Benutzeroberfläche* (die aus Fenstern, Knöpfen, Menüs etc. besteht).  
**Konso-1:** *Konsolen*-Programme werden über eine *Kommandozeile* (oder ein Kommando-Eingabefenster) bedient.  
**Grabo-2:** Der Ablauf eines *Grabo*-Programms wird *vom Benutzer gesteuert* (das heißt, der Ablauf hängt ganz wesentlich von den Aktionen des Benutzers, von seinen Mausbewegungen und Klicks, von seinen Tastatureingaben etc. ab).  
**Konso-2:** Der Ablauf eines *Konsolen*-Programms wird hauptsächlich *vom Programm gesteuert* (und kann vom Benutzer nur relativ wenig durch seine Eingaben beeinflusst werden).  
**Grabo-3:** Ein *Grabo*-Programm ist immer ein *Proma* (ein Programm mit mehreren Ausführern).  
**Konso-3:** Ein *Konsolen*-Programm ist in aller Regel ein *sequenzielles Programm*.
741. *Transaktionsorientierte Dialog-Programme* (IBM, Cics, 3270) kann man als eine Zwischenstufe zwischen *Konsolen*-Programmen und *Grabo*-Programmen verstehen.

742. Ein Objekt der Klasse `javax.swing.JFrame` wird auf dem Bildschirm als ein *Fenster* (mit Titelleiste und drei kleinen Knöpfen) dargestellt. Die Darstellung ist weitgehend unabhängig vom dem Betriebssystem, unter dem man das betreffende Java-Programm ausführen läßt.
743. Nachdem man (in einem Java-Programm) ein `JFrame`-Objekt `job` hat erzeugen lassen, ist das Objekt noch nicht auf dem Bildschirm zu sehen. Erst nachdem man den Befehl `job.setVisible(true);` gegeben hat, wird das Objekt sichtbar.
744. Die anfänglich Größe und Position (der Bildschirmdarstellung) eines `JFrame`-Objekts `job` kann man mit dem Befehl `job.setBounds` festlegen. Diese Antwort enthält das Wort "anfänglich", weil der Benutzer die Größe und Position (der Bildschirmdarstellung) eines `JFrame`-Objekts `job` jederzeit (mit Hilfe seiner Maus) verändern kann.
745. Wenn der Java-Ausführer ein *Grabo*-Programm ausführt, spaltet er sich in mindestens zwei Fäden (Ausführer) auf, in einen *Hauptfaden* (engl. *main thread*) und einen *Ereignisfaden* (engl. *event thread*).
746. Für das Zeichnen von Fenstern, Knöpfen, Menüs etc. ist normalerweise der *Ereignisfaden* (engl. *event thread*) verantwortlich. Die *main*-Methode wird vom *Hauptfaden* (engl. *main thread*) ausgeführt.
747. Die Klasse `paint`, die man z. B. in einer Erweiterung der Klasse `JFrame` programmieren kann, hat die bemerkenswerte Eigenschaft, dass man sie zwar *vereinbart*, aber *nirgends* (direkt und sichtbar) *aufruft*.
748. Die `paint`-Methode wird *vom Ausführer* aufgerufen (und ausgeführt), wenn der den Eindruck hat, dass es Zeit dazu ist. Der Ausführer ruft die `paint`-Methode z. B. dann auf, wenn das Fenster (zu dem sie gehört) vom Benutzer wieder sichtbar gemacht wird, nachdem er es mit anderen Fenstern vorübergehend verdeckt hatte.
749. Wenn man eine `paint`-Methode programmiert, sollte man als erstes immer die Methode `super.paint` aufrufen.
750. Wenn der Benutzer mit seiner Maus auf den Fenster-Schließen-Knopf eines `JFrame`-Fensters klickt, wird normalerweise nur dieses Fenster *geschlossen* und vom Bildschirm gelöscht. Das zugehörige Java-Programm wird aber nicht beendet. Außerdem wird die Aktivität des nächstgelegenen Vulkans durch solch einen Mausklick in aller Regel *nicht angeregt* (aber das hätte wahrscheinlich auch kaum jemand vermutet :-).
751. Sei `job` ein `JFrame`-Objekt. Mit dem Befehl `job.setDefaultCloseOperation` kann der Programmierer die normale Wirkung eines Klicks auf den Fenster-Schließen-Knopfs von `job` modifizieren.

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. Das kann der Programmierer wie folgt erreichen:
1. Er vereinbart eine Erweiterung der Klasse `WindowAdapter` und nennt sie zum Beispiel `MWA` (wie "MeinWindowAdapter").
 2. In der Klasse `MWA` vereinbart er eine Methode mit dem Profil
- ```
public void windowClosing(WindowEvent we)
```
- (damit überschreibt er eine Methode, die
- `MWA`
- von
- `WindowAdapter`
- geerbt hat)
3. In diese Methode `windowClosing` schreibt er den Befehl, der bei einem Klick auf den Fenster-Schließen-Knopf ausgeführt werden soll, z. B.
- ```
pln("Der Fenster-Schliessen-Knopf wurde angeklickt!");
```
4. Er lässt ein Objekt der Klasse `MWA` erzeugen und nennt es z. B. `mwaOb`.
5. Mit der Methode `addWindowListener` meldet er das Objekt `mwaOb` beim `JFrame`-Objekt `job` an, etwa so:
- ```
job.addWindowListener(mwaOb);
```
753. Ein *Grabo-Objekt* ist ein Objekt, welches nach seiner Erzeugung (mehr oder weniger automatisch) auf dem Bildschirm *grafisch dargestellt* wird (das war eine *Java-unabhängige, inhaltliche* Definition, keine *Java-abhängige, formale* Definition).
754. Eine *Grabo-Klasse* ist (wenn man ganz einfach auf der vorigen Definition aufbaut) eine Klasse, deren Instanzen *Grabo-Objekte* sind.
755. Eine *Grabo-Klasse* ist eine Unterklasse der Standardklasse `java.awt.Component` (das war eine *Java-abhängige, formale* Definition, keine *Java-unabhängige, inhaltliche* Definition).
756. Ein *Grabo-Objekt* ist (wenn man ganz einfach auf der vorigen Definition aufbaut) ein Objekt einer *Grabo-Klasse*.
757. Die oberste Klasse im Typgraphen aller Java-Grabo-Klassen heißt also `java.awt.Component` (und kam schon in der vorvorigen Antwort vor).
758. Eine (Grabo-) *Behälterklasse* ist eine Unterklasse der Java-Standardklasse `java.awt.Container`.
759. Von den folgenden drei Sätzen
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.
- trifft nur 2. zu.
760. In ein *Behälterobjekt* darf man nur *Grabo-Objekte* hineintun!
761. Das hängt vom Typ des Sammlungsobjekts ab. In ein Sammlungsobjekt des Typs `ArrayList<Object>` darf man beliebige Objekte hineintun. In ein

- Sammlungsobjekt des Typs `HashSet<String>` darf nur `String`-Objekte hineintun etc.
762. Aus den beiden Tatsachen
1. Jedes `Container`-Objekt ist auch ein `Component`-Objekt.
  2. Ein `Container`-Objekt darf beliebige `Grabo`-Objekte enthalten.
- folgt:
3. Ein `Container`-Objekt darf auch `Container`-Objekte enthalten, d. h. man darf Behälter (`Container`-Objekte) *schachteln*, etwa so wie Dateiverzeichnisse (Ordner) unter Unix oder Windows.
763. Ein Objekt darf gleichzeitig zu *beliebig vielen Sammlungen* (engl. *collection objects*) gehören. Ein (Grabo-) Objekt darf in jedem Moment zu *höchstens einem Behälter* (engl. *container object*) gehören. (S. 542)
764. Die beiden obersten Grabo-Pakete `awt` und `swing` heißen mit vollem Namen `java.awt` bzw. `javax.swing`. Diese beiden Pakete sind also *keine* Top-Pakete, nur oberste Grabo-Pakete! Zur Erinnerung: Ein Top-Paket ist Paket, welches in keinem anderen Paket enthalten ist, z. B. `java`, `javax` und `org`.
765. Zwei unterschiedliche Strategien, nach denen man die Grabo-Klassen und -Objekten einer Programmiersprache PS unter einem bestimmten Betriebssystem BS implementieren kann:
- Strategie 1:** Man bildet die Grabo-Objekt der Sprache PS auf die "eingeborenen" Grabo-Objekte des Betriebssystems BS ab.
- Strategie 2:** Man greift möglichst wenig auf die eingeborenen Routinen des Betriebssystems BS zurück und programmiert "alles neu" in der Sprache PS.
766. Das Paket `java.awt` wurde nach der **Strategie 1**, das Paket `javax.swing` dagegen nach der **Strategie 2** entwickelt.
767. Die oberste Klasse im Typgraphen aller `awt`-Klassen (und damit auch aller Java-Grabo-Klassen) heißt (mit vollem Namen) `java.awt.Component`.
768. Die oberste Klasse im Typgraphen aller `swing`-Klassen heißt (mit vollem Namen) `javax.swing.JComponent`.
769. Aus der Tatsache, dass die Klasse `javax.swing.JComponent` eine Erweiterung der Klasse `java.awt.Container` ist, folgt: Jede `swing`-Komponente (jedes `JComponent`-Objekt) ist auch ein Behälter, in den man andere Grabo-Komponenten (`Component`-Objekte) hineintun kann.
770. Die Namen vieler `swing`-Klassen fangen mit dem Buchstaben `J` an (z. B. `JButton`, `JLabel`, `JPanel`, `JFrame` etc.).
771. *Aktionen* werden vom *Benutzer* ausgeführt (indem er mit einer Grabo interagiert).
772. Beispiele für typische *Aktionen*: Der Benutzer klickt mit der linken Taste seiner Maus einen Knopf (ein `Button`- oder `JButton`-Objekt) an. Der Benutzer

bewegt den Mauszeiger ein Stückchen weit über eine Zeichenfläche (z. B. ein Panel- oder JPanel-Objekt). Ebenso, aber zusätzlich hält der Benutzer die linke Taste seiner Maus gedrückt. Der Benutzer gibt über die Tastatur eine Zeichenkette ein etc.

773. Ein *Ereignis* findet zu einem bestimmten Zeitpunkt an einem bestimmten Ort auf dem Bildschirm des Benutzers statt und ist grundsätzlich nicht wiederholbar (mit *Ereignissen* sind bei der Grabo-Programmierung also immer *historische Ereignisse* gemeint). Ein Beispiel: Am 17. März 2005, etwa 35 Nanosekunden nach 8 Uhr hat der Benutzer Karl Mayer den mit "Beenden" beschrifteten Knopf links oben auf seinem Bildschirm (Koordinaten des linken-oberen Eckpunktes des Knopfes: Pixelzeile 30, Pixelspalte 50) mit seiner Maus angeklickt.

774. Beispiele für *Oberarten* von Ereignissen: Fensterereignisse, Mausereignisse, Mausbewegungsereignisse, Mausevents, Aktionsereignisse.

775. Zur Oberart *Fensterereignis* gehören die folgenden 7 Arten von Ereignissen: `windowOpened`, `windowClosing`, `windowClosed`, `windowIconified`, `windowDeiconified`, `windowActivated` und `windowDeactivated`.

776. Die Schnittstelle `WindowListener` enthält die folgenden 7 Methoden: `windowOpened`, `windowClosing`, `windowClosed`, `windowIconified`, `windowDeiconified`, `windowActivated` und `windowDeactivated`.

777. Objekte der Klasse `java.awt.Window` können Quellen von Fensterereignissen sein. Anmerkung: Die Klasse `javax.swing.JFrame` ist eine Erweiterung der Klasse `java.awt.Window`. Somit ist jedes `JFrame`-Objekt auch ein `Window`-Objekt und `JFrame`-Objekte können auch Quellen von Fensterereignissen sein.

778. Die ausgefüllte Tabelle zeigt, welche *Schnittstelle* und welche *Adapterklasse* zusammengehören (und zu welchen Schnittstellen es keine Adapterklasse gibt):

| Schnittstelle                    | Anz. Methoden | Adapterklasse                          |
|----------------------------------|---------------|----------------------------------------|
| <b>WindowListener</b>            | <b>7</b>      | <code>WindowAdapter</code>             |
| <b>MouseListener</b>             | <b>5</b>      | <code>MouseAdapter</code>              |
| <code>MouseMotionListener</code> | <b>2</b>      | <b><code>MouseMotionAdapter</code></b> |
| <code>MouseWheelListener</code>  | <b>1</b>      | --                                     |
| <code>ActionListener</code>      | <b>1</b>      | --                                     |

779. Zu Schnittstellen, die nur *eine* einzige Methode enthalten (z. B. `MouseWheelListener` und `ActionListener`) gibt es *keine* Adapterklasse. Adapterklassen sollen es einem ja nur ersparen, *mehrere* Methoden zu überschrei-

ben, wenn man nur *eine* überschreiben möchte. Bei Schnittstellen mit nur einer Methode ist eine solche "Ersparnis" nicht möglich.

780. Im Zusammenhang mit Adapterklassen verstehen wir unter einer *Stroh puppe* eine Methode mit einem leeren Rumpf (ein Aufruf einer solchen Methode ist nicht schädlich und die Zeit, die er kostet, ist in aller Regel vernachlässigbar).

781. Objekte der Klasse `java.awt.Component` (und all ihrer Unterklassen) können Quellen von *Mausereignissen* sein. Mit anderen Worten: Alle Grabo-Objekte können Quellen von *Mausereignissen* sein.

782. Objekte der Klasse `java.awt.Component` (und all ihrer Unterklassen) können Quellen von *Mausbewegungsereignissen* sein. Mit anderen Worten: Alle Grabo-Objekte können Quellen von *Mausbewegungsereignissen* sein.

783. Objekte der Klasse `java.awt.Component` (und all ihrer Unterklassen) können Quellen von *Mausradereignissen* sein. Mit anderen Worten: Alle Grabo-Objekte können Quellen von *Mausradereignissen* sein.

784. Objekte, die Quellen von *Aktionsereignissen* sein können, bezeichnen wir als *aktionsfähig*. Aktionsfähig sind z. B. die Objekte der Klassen `Button`, `JTextField`, `JButton`, `JMenuItem` und `JToggleButton`.

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*. Beide Fäden können auf den Wertebehälter *Bildschirm* zugreifen. Wenn der Programmierer sich ein bisschen Mühe gibt (dem Autor ist das so gar mehrmals mühelos und aus Versehen gelungen :-)) stören sich die beiden Fäden und der eine zerstört das, was der andere gerade zum Bildschirm ausgegeben hat.

786. Nebenläufigkeitsfehler (z. B. Störungen zwischen dem Haupt- und dem Ereignisfaden eines Java-Grabo-Programms) werden im Englischen als *race conditions* bezeichnet. Dabei ist mit *race* keine *group of people of common ancestry, distinguished from others by physical characteristics* gemeint, sondern a *contest of speed, as in running, swimming, driving, riding, etc.* (Zitate aus "Collins Dictionary of the English Language", 1985).

787. In die *main*-Methode eines Grabo-Programms sollte man keine Befehle schreiben, die direkt etwas zum selben Bildschirm schreiben, auf dem der Ereignisfaden die Grabo (grafische Benutzeroberfläche) zeichnet.

788. Ein *Zeichencode* ist eine Abbildung zwischen einer *Menge von Zeichen* und einer *Menge von Codezahlen*.

789. Der 7-Bit-ASCII-Code ordnet 128 Zeichen Codezahlen zwischen 0 und 127 zu.

790. Der 7-Bit-ASCII-Code ist sehr weit verbreitet und hat sich als einfaches und gut handhabbares Werkzeug durchgesetzt.

791. Es gibt nicht nur einen, sondern sehr viele verschiedene 8-Bit-ASCII-Codes. Jeder dieser Codes ist eine Erweiterung des bewährten 7-Bit-ASCII-Codes und ordnet zusätzlichen 128 Zeichen Codezahlen zwischen 128 und 255 zu. Jeder 8-Bit-ASCII-Code ordnet also insgesamt 256 Zeichen Codezahlen zwischen 0 und 255 zu.
792. Auch 8-Bit-ASCII-Codes sind sehr weit verbreitet, aber in jedem Land (und manchmal in jeder Stadt oder sogar in jedem Stadtviertel) ein anderer. Diese vielen verschiedenen 8-Bit-ASCII-Codes haben sich zwar durchgesetzt, sind aber eine Quelle zahlreicher und manchmal sehr lästiger Probleme. Beispiel: Den Zeichen ä, ö, ü, Ä, Ö und Ü werden zwar in vielen 8-Bit-ASCII-Codes Codezahlen zugeordnet, aber in jedem solchen Code andere Codezahlen.
793. Der Unicode ordnet etwa 60 Tausend häufig verwendeten Zeichen je eine 16-Bit-Codezahl zu und kann einer weiteren Million (seltener verwendeter) Zeichen je eine 32-Bit-Codezahl zuordnen (viele dieser 32-Bit-Codezahlen sind zur Zeit noch frei).
794. Unter **CJK**-Zeichen versteht man chinesische Schriftzeichen, die vor allem in China, Japan, Korea, Taiwan und Vietnam benutzt werden.
795. **Vorteil 1** des Unicode: Jedes Zeichen hat eine feste Codezahl, fehlerträchtige Umcodierungen entfallen.  
**Vorteil 2** des Unicode: Alle weltweit häufig verwendeten Zeichen haben *gleich lange* Codezahlen (16-Bit-Codezahlen).
796. **Nachteil** des Unicode: Die in westlichen Ländern besonders häufig verwendeten Zeichen (denen auch der 7-Bit-ASCII-Code Codezahlen zuordnet), belegen im Unicode *doppelt soviele Speicherplatz* wie im ASCII-Code (16 Bits statt 8 Bits).
797. Der UTF-8-Code ordnet *genau denselben Zeichen* wie der Unicode Codezahlen zu, er ordnet ihnen aber *andere Codezahlen* zu als der Unicode.
798. Der wichtigste Unterschied zwischen dem *Unicode* und dem *UTF-8-Code* ist der folgende: Der Unicode ordnet allen (häufig verwendeten) Zeichen Codezahlen der gleichen Länge (16 Bit) zu. Der UTF-8-Code ordnet denselben Zeichen Codezahlen unterschiedlicher Längen zu, einigen Zeichen eine 8-Bit-Codezahl, anderen Zeichen eine 16-Bit-Codezahl und anderen eine 24- bzw. eine 32-Bit-Codezahl.
799. Der UTF-8-Code ordnet den "in westlichen Ländern besonders häufig verwendeten Zeichen" besonders kurze Codezahlen (d. h. 8-Bit-Codezahlen) zu. In westlichen Ländern werden besonders häufig die 7-Bit-ASCII-Zeichen verwendet. Für diese Zeichen stimmt der UTF-8-Code mit dem ASCII-Code überein (z. B. ordnen beide Zeichencodes dem Buchstaben A die Codezahl 65 und der Ziffer 0 die Codezahl 48 zu).

800. Den *Unicode* und den *UTF-8-Code* verbindet vor allem die Möglichkeit, sehr einfach aus Unicode-Codezahlen UTF-8-Codezahlen und umgekehrt (aus UTF-8-Codezahlen Unicode-Codezahlen) zu berechnen. Beide Codes sind extra so gestaltet worden, dass diese Umrechnungen möglichst einfach sind.
801. Bei Dateien, die zum größten Teil nur die 7-Bit-ASCII-Zeichen und nur sehr wenige chinesische Schriftzeichen enthalten, ist es günstiger, sie im UTF-8-Code abzuspeichern statt im Unicode (weil man dadurch bis zu 50% des Speicherplatzes einspart).
802. Ein Unicode-Editor sollte die folgenden drei Dinge können:
1. Er sollte die *Eingabe* von beliebigen Unicode-Zeichen ermöglichen.
  2. Er sollte beliebige Unicode-Zeichen auf dem Bildschirm *darstellen* können.
  3. Er sollte Dateien in einem Code *abspeichern* können, bei dem alle Unicode-Zeichen erhalten bleiben (der Unicode und der UTF-8-Code sind zwei Beispiele für geeignete Codes, 8-Bit-ASCII-Codes sind *nicht* geeignet).
803. Die Eingabe beliebiger Unicode-Zeichen erfordert spezielle Verfahren, weil eine Tastatur mit etwa 60 Tausend Tasten schwer herzustellen und vermutlich auch schwer zu handhaben wäre. Zwei Verfahren, die nur eine übliche Tastatur (mit ca. 100 Tasten) voraussetzen, haben sich bewährt:
1. Für die am häufigsten benötigten Zeichen gibt es spezielle Tasten. Die übrigen Zeichen gibt man in Form ihrer *Codezahlen* ein.
  2. Für die am häufigsten benötigten Zeichen gibt es spezielle Tasten. Die übrigen Zeichen sucht man in umfangreichen Menüs oder Tabellen und gibt sie *durch Anklicken* ein.
- Insbesondere für die Eingabe chinesischer Schriftzeichen sind weitere effiziente Eingabeverfahren entwickelt worden (z. B. die sogenannte Pinyin-Eingabe).
804. Für die Namen seiner Klassen, Methoden, Attribute etc. darf der Java-Programmierer nur solche Unicode-Zeichen verwenden, für die die Funktionen `Character.isJavaIdentifierStart` (für das erste Zeichen eines Namens) bzw. `Character.isJavaIdentifierPart` (für alle übrigen Zeichen eines Namens) den Wert `true` liefern. Das sind jeweils etwa 45 bzw. 46 Tausend Zeichen. Andere Zeichen sind in Namen nicht zulässig!
805. Wenn einem kein Unicode-Editor zur Verfügung steht, kann man chinesische Schriftzeichen in Form von Unicode-Literalen in ein Java-Programm eingeben, z. B. so: `'\u5000'`. Solche Literale bezeichnen Werte des Typs `char`.



<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