# Stichworte und Wiederholungsfragen

zur Lehrveranstaltung Programmieren 1 im im 1. Semester des Studiengangs Medien-Informatik Bachelor (MB1-PR1) im Sommersemester 2010 für den Zug 1 (und anfangs auch für den Zug 2).

In dieser Datei werden Sie im Laufe des Semester jeweils nach einem Vorlesungstermin Stichworte dazu und Wiederholungsfragen für den nächsten Vorlesungstermin finden. Alle Seiten-Angaben (wie z.B. S. 20 oder S. 237 etc.) in diesem Papier beziehen sich auf das Buch "Java ist eine Sprache".

## Erste Übungstermine

am Fr 09.04.10, 3. Block, Gruppe 1b und
am Di 13.04.10, 3. Block. Gruppe 1a.

1. Vorstellung

2. Klären, wer (vorzugsweise) zu den einzelnen Übungsgruppen gehört

| | | |
|---|---|---|
| Erstsemester mit geraden | Matrikel-Nrn | zu 1a, 1b |
| Erstsemester mit ungeraden | Matrikel-Nrn | zu 2a, 2b |
| Wiederholer | | zu 2c |

Wenn für eine bestimmte Gruppe mehr Interessenten anwesend sind als es Plätze gibt, werden die Plätze verlost. Wer dabei keinen Platz "gewinnt" muss in eine andere Gruppe gehen.

Wenn Sie "unbedingt in die Übungsgruppe xy müssen", aber dort keinen Platz gewonnen haben, können Sie auch versuchen, eine StudentIn, die einen Platz in xy hat, zum Tauschen zu überreden.

Es ist nicht möglich, Termine verschiedener Übungsgruppen (1a, 1b, 2a, 2b, 2c) oder verschiedener Züge (1, 2) zu kombinieren (z.B. einen Block von 1a und einen von 1b oder den seminaristischen Unterricht im Zug 2 bei Herrn Goethe und die Übungen im Zug 1 bei Herrn Grude etc.).

Die Ü-Termine (Übungen):

| 1a (Grude) | 1b (Grude) | 2a (Goethe) | 2b (Goethe) | 2c (Goethe) |
|---|---|---|---|---|
| **Di 3. Block**<br>**Mi 2. Block** | **Mi 1. Block**<br>**Fr 3. Block** | **Di 4. Block**<br>**Do 5. Block** | **Mo 2. Block**<br>**Do 4. Block** | **Mo 3. Block**<br>**Di 5. Block** |

Die SU-Termine (seminaristischer Unterricht, "Vorlesung"):

| 1 (Grude) |
|---|
| Mo 4. Block<br>Di 2. Block |

| 2 (Goethe) |
|---|
| Di 3. Block<br>Do 3. Block |

3. Namensschilder

Bitte zu allen (Ü- und SU-) Terminen mitbringen und unaufgefordert aufstellen. Wer ohne Namensschild in einer MB1-PR1-Veranstaltung sitzt, bekommt sofort Pickel und später bei der Klausur nur halb so viele Punkte wie die anderen.

4. Einloggen

5. Die auf der folgenden Seite beschriebenen Arbeitsschritte ausführen

## Übung 1 MB1-PR1 am Fr 09.04.10 und Di 13.04.10

1. Netzseite von Herrn Grude: public.beuth-hochschule.de/~grude/

2. Dort die Datei für MB1-PR1 TipsZumTextPad.pdf öffnen
Den Tip 5 ausführen

In das erstellte Programm einen Fehler einbauen (z.B. eine Klammer löschen).
Erneut compilieren.
Auf die 1. Zeile der Fehlermeldung doppelklicken.
Den eingebauten Fehler korrigieren (z.B. die Klammer wieder einfügen).
Erneut compilieren und ausführen lassen.

3. Netzseite zum Buch "Java ist eine Sprache":
public.beuth-hochschule.de/~grude/JavaIstEineSprache/

Von dort das Archiv BspJaSp.zip herunterladen
und in ein Verzeichnis namens Z:\BspJaSp entpacken.

4. Ein Verzeichnis namens Z:\Hallo erstellen
und alle Hallo*.java-Dateien aus Z:\BspJaSp nach Z:\Hallo kopieren.

Die Dateien Hallo01.java, Hallo01A.java und Hallo01B.java mit dem TextPad öffnen, genau ansehen, compilieren und ausführen lassen.

5. Auf der Netzseite public.beuth-hochschule.de/~grude/
die Datei für MB1-PR1 Aufgaben.pdf öffnen

Darin die Aufgabe 1 (Seite 5 und 6) ausdrucken lassen

Die Seiten 7 und 8 ansehen und die Aufgabe 1 lösen.

1. SU Mo 12.04.10

1. Begrüßung

2. Wer hat noch kein Namensschild.
Bitte zu allen (SU- und Ü-) Terminen mitbringen und aufstellen.

3. Wie bekommt man (zwei) Noten für die beiden Module
MB1-PR1 (Praxis) und MB1-PR1 (Theorie)? Kurz besprechen. Hinweis auf die Datei
http://public.beuth-hochschule.de/~grude/AufgabenPR1_09SS.pdf
wo man die Regeln (auf Seite 2) genau nachlesen kann.

4. Ich hoffe und bin zuversichtlich, dass wir (Sie und ich) während des Semesters gut miteinander klar kommen werden, offen und locker miteinander reden und freundlich und hilsfbereit zueinander sein können.

Aber: Laßen Sie sich durch den lockeren Ton während des Semesters nicht zu falschen Annahmen verleiten. Am Ende des Semesters geht es um die harte Frage, ob Sie die beiden Module dieses Fachs bestehen oder ob Sie durchfallen. Und ich fühle mich verpflichtet, jede Person durchfallen zu lassen, die nicht aktiv gezeigt hat, dass sie den Stoff dieser Lehrveranstaltung mindestens ausreichend beherrscht. Es ist Ihre Aufgabe zu zeigen, dass Sie mindestens ausreichend viel gelernt und begriffen haben.

Ich werde mich sehr freuen, falls alle bestehen und gute Noten bekommen. Aber die Erfahrung vergangener Semester zeigt, dass in dieser LV (MB1-PR1) bei verschiedenen DozentInnen (nicht nur bei mir) häufig etwa 30 % der TeilnehmerInnen erstmal durchfallen. Helfen Sie mir, dass in diesem Semester das Endergebnis besser wird, indem Sie zu allen (Ü- und SU-) Terminen kommen und sich mit dem behandelten Stoff gründlich vertraut machen (und Ihre Namensschilder aufstellen, damit Sie in der Klausur die volle Punktzahl angerechnet bekommen :-).

Wenn ich Ihnen am Ende des Semesters eine ungenügende Note geben muss, werde ich mich fühlen wie ein Zahnarzt, der Ihnen einen Zahn ziehen muss. Ich weiss, dass meine Handlung Ihnen unangenehm ist und weh tut, aber ich werde den Zahn trotzdem entfernen und nicht drinlassen.

5. Kommen Sie möglichst ein bisschen zu früh oder pünktlich zu allen Terminen, aber nicht zu spät. Wenn man zu einem Ü-Termin ein bisschen zu früh kommt, kann man sich schon mal einloggen, die nächste Aufgabe durchlesen, sich an den Stoff der letzten Übung erinnern etc.
Damit es sich "lohnt", auch zu den SU-Terminen ein bisschen zu früh zu kommen, werde ich in aller Regel auch zu früh kommen und jeweils ein Blatt mit einem kleinen Quiz (Wiederholungsfragen) verteilen. Mit denen können Sie sich dann schon mal beschäftigen und überprüfen, ob Sie gut mitkommen.

6. Hat irgend jemand noch Fragen zur Organisation der LV MB1-PR1?

Bitte anschnallen. Jetzt geht es los mit dem Stoff dieser LV! :-)

## 1.1 Programmieren als ein Rollenspiel

| Rolle | Tätigkeiten (die meisten beziehen sich auf Programme) |
|---|---|
| Programmierer | schreiben, übergeben (an den Ausführer) |
| Ausführer | prüft, lehnt ab/akzeptiert, führt aus |
| Benutzer | läßt ausführen, ist für Ein-/Ausgabedaten zuständig |
| Warter | wartet |
| (Wieder-) Verwender | will wiederverwenden |

Sinn der Rollen *Warter* und (*Wieder-) Verwender*. Gemeinsam bezeichnen wir den *Warter* und den (Wieder-) *Verwender* auch als "die *Kollegen* des Programmierers".

Besetzung der Rollen, verschiedene Möglichkeiten.

Def.: Programm: (S. 3)

Bevor wir uns mit "kleinen Einzelheiten" befassen, ein Versuch, die "großen Ideen" der Programmierung kurz zu beschreiben:

## 1.4 Die vier wichtigsten Grundkonzepte der Programmierung

Variable (Wertebehälter, Inhalt beliebig oft veränderbar)

Typ

Unterprogramm

Modul

Zur Enspannung: Was kostet ein Studium an der BHT?

Haushalt der BHT ca. 50 Millionen [Euro pro Jahr].
An der BHT studieren ca. 10 Tausend StudentInnen.
Also kostet das Studieren an der BHT ca. 5000 [Euro pro StudentIn und Jahr].
Ein Bachelor-Studium (6 Semester, 3 Jahre) kostet also ca. 15 Tausend [Euros pro StudentIn].

Wiederholungsfragen, 2. SU, Di 13.04.10

1. Wie heißen die drei wichtigsten Rollen ( im "Rollenspiel des Programmierens")?

2. Was sollte der Programmierer anstreben und tun, weil es ausser ihm auch noch den *Warter* und den (Wieder-) *Verwender* gibt?

3. Nennen Sie die Tätigkeiten, die wir der Rolle des *Ausführers* zugeordnet haben (die Tätigkeitsworte alleine genügen, etwa so: essen, schlafen, fernsehen oder so ähnlich).

4. Warum hat man das Konzept eines Typs in fast alle Programmiersprachen eingebaut? Tip: Der Grund hat mit Geld und mit Flüchtigkeitsfehlern des Programmierers zu tun.

5. In vielen Programmiersprachen haben Variablen den Charakter von *Wertebehältern*. Durch welche beiden Eigenschaften unterscheiden sich diese Behälter von anderen, alltäglichen Behältern wie z.B. Hutschachteln oder Milchkannen?

6. Ein Modul ist auch ein Behälter, aber nicht für Werte, sondern für andere Dinge. Was für Dinge kann man in einen Modul hineintun?

7. Ein Modul muss aus mindestens 2 Teilen bestehen. Wie bezeichnet man diese beiden Teile (nicht als "Den roten und den grünen Teil", aber so ähnlich).

8. Wodurch müssen sich die beiden Teile eines Moduls unterscheiden?

Rückseite der Wiederholungsfragen, 2. SU, Di 13.04.10

Konkrete Beispiele für Java-Befehle

1. Vereinbarungen (declarations)

```
 1 int otto = 17;
 2 int emil = 5;
 3 char anna = 'A';
 4 String name = "Sarah";
 5 final int fanny = 12;
 6
 7 static void druckeHallo() {
 8    System.out.println("Hallo");
 9    System.out.println("-----");
10 }
11
```

```
 1 class Claudia {
 2    ... // Hier dürfen nur Vereinbarungen stehen
 3 }
```

2. Ausdrücke (expressions)

Hier wird die Vereinbarung von `otto` vorausgesetzt

```
1 ... otto + 3 ...
2 (otto + 98) / (otto * 10) - 1
3 ... otto < 3 ...
4 ... otto ...
5 ... 165 ...
6 ... 0.1 ...
```

3. Anweisungen (statements)

Hier werden die Vereinbarungen von `otto`, `emil`, `sarah` und `anna` vorausgesetzt

```
1 otto = 15;
2 otto = emil + 2;
3 otto = otto + emil;
4 System.out.println("Hallo " + name); // Ausgabe: Hallo Sarah
5 System.out.println(1 + anna);        // Ausgabe: 66
6 System.out.println("" + 1 + anna);   // Ausgabe: 1A
```

Antworten zu den Wiederholungsfragen, 2. SU, Di 13.04.10

1. Wie heißen die drei wichtigsten Rollen ( im "Rollenspiel des Programmierens")?

Programmierer, Ausführer, Benutzer.

2. Was sollte der Programmierer anstreben und tun, weil es ausser ihm auch noch den *Warter* und den (Wieder-) *Verwender* gibt?

Er sollte seine Programme so gestalten, dass seine Kollegen (Warter und Verwender) sie lesen und verstehen können.

3. Nennen Sie die Tätigkeiten, die wir der Rolle des *Ausführers* zugeordnet haben (die Tätigkeitsworte alleine genügen, etwa so: essen, schlafen, fernsehen oder so ähnlich).

Prüfen, ablehnen/akzeptieren, ausführen.

4. Warum hat man das Konzept eines Typs in fast alle Programmiersprachen eingebaut? Tip: Der Grund hat mit Geld und mit Flüchtigkeitsfehlern des Programmierers zu tun.

Um das Auffinden bestimmter Flüchtigkeitsfehler des Programmierers billiger zu machen.

5. In vielen Programmiersprachen haben Variablen den Charakter von *Wertebehältern*. Durch welche beiden Eigenschaften unterscheiden sich diese Behälter von anderen, alltäglichen Behältern wie z.B. Hutschachteln oder Milchkannen?

- Eine Variable enthält immer genau einen Wert (sie kann nicht leer sein!).

- Wenn man einen Wert in eine Variable hineintut, wird der alte Wert zerstört.

6. Eine Modul ist auch ein Behälter, aber nicht für Werte, sondern für andere Dinge. Was für Dinge kann man in einen Modul hineintun?

Variablen, Typen, Unterprogramme, Module (und in einigen Sprachen noch weitere Dinge).

7. Ein Modul muss aus mindestens 2 bestehen. Wie bezeichnet man diese beiden Teile (nicht als "Den roten und den grünen Teil", aber so ähnlich).

öffentlicher Teil / privater Teil oder
sichtbarer Teil / unsichtbarer Teil oder
ungeschützter Teil / geschützer Teil.

8. Wodurch müssen sich die beiden Teile eines Moduls unterscheiden?

Von außerhalb des Moduls kann man nur auf Dinge im öffentlichen Teil zugreifen (aber nicht auf die Dinge im privaten Teil).

2. SU Di 13.04.10

A. Wiederholung

B. Organisatorisches
Den Test 1 (zur Aufgabe 1) schreiben wir an folgenden Terminen:
Gruppe 1a: Di 20.04.10
Gruppe 1b: Fr 16.04.10

Die vier wichtigste Grundkonzepte der Programmierung (Fortsetzung)

Letztes mal haben wir die Konzepte Variable, Typ, Unterprogramm und Modul besprochen, Modul noch nicht vollständig.

Module haben 2 sehr wichtige (positive) Eigenschaften:
Modul-Vorteil-1: Angenommen ein Programm besteht aus 100 Modulen. Im Modul M37 vereinbaren wir eine Ganzzahl-Variablen namens emil von der wir wissen, dass sie eigentlich nie einen negativen Wert enthalten darf. Beim Testen merken wir, dass in gewissen Situationen doch einen negativen Wert enthält. Wo müssen wir die falschen Befehle (die den negativen Wert in emil hineingetan haben), suchen
- wenn wir emil im öffentlichen Teil von M37 vereinbart haben?
- wenn wir emil im privaten Teil von M37 vereinbart haben?

Modul-Vorteil-2: Angenommen, ein Programm wird von 10 Programmierern erstellt von denen jeder 10 Module schreibt. Der Programmierer P3 vereinbart im öffentlichen Teil seines Moduls M3-5 ein Unterprogramm namens `summe` mit 3 Parametern vom Typ `int`. Mehrere der anderen Programmierer rufen (in ihren Modulen) dieses Unterprogramm auf.

Nach einer Weile fällt dem Programmierer P3 eine Verbesserung ein: Eigentlich sollte das Unterprogramm statt summe besser `summe1` heißen (weil es auch noch eine `summe2` gibt), und es sollte 4 Parameter haben (statt 3). Warum hat P3 jetzt ein großes Problem (welches in der Praxis manchmal nicht lösbar ist und deshalb nicht auftreten darf).

Zum Abschluss dieses Abschnitts: Das fünfte der vier wichtigsten Grundkonzepte: Klasse

Def.: Eine *Klasse* ist ein Modul und ... (wird in ein paar Wochen vervollständigt)

Noch zwei Definitionen

Def.: Ein *Programm* ist *eine Folge von Befehlen*, die ein Programmierer aufschreibt in der Hoffnung, dass ein Ausführer sie (ein- oder mehrmals) ausführen wird.

Def.: Ein *Typ* ist ein Bauplan für Variablen

1.5 Drei Arten von Befehlen

| Bezeichnung (deutsch) | Bezeichnung (englisch) | Übersetzung eines solchen Befehls ins Deutsche (oder: Was befiehlt ein solcher Befehl dem Ausführer?) |
|---|---|---|
| Vereinbarung | declaration | "Erzeuge ... !" |
| Ausdruck | expression | "Berechne den Wert des Ausdrucks ... !" |
| Anweisung | statement | "Tue die Werte ... in die Wertebehälter ... !" |

Was ein Ausdruck *macht* und was ein (normaler) Ausdruck *nicht macht*!

### Konkrete Beispiele für Java-Befehle:

Bisher haben wir uns mit sehr abstrakten und allgemeinen Grundkonzepten befaßt. Jetzt sehen wir uns ein paar konkrete Java-Befehle etwas genauer an (auf der Rückseite der Wiederholungsfragen).

Die meisten dieser Beispiel stehen auch im Buch (mit ein paar Erläuterungen drumrum)

- `fanny` ist eine unveränderbare Variable
- Die Vereinbarung der Klasse Claudia ist hier nur angedeutet
- Die einfachsten Ausdrücke bestehen nur aus einem *Variablen-Namen* (z.B. otto) oder aus einem *Literal* (z.B. `165` oder `'A'` oder `"Sarah"`)
- Die Anweisung in Zeile 4 gibt aus: `Hallo Sarah`
- Die Anweisung in Zeile 5 gibt aus: `B`
- Die Anweisung in Zeile 6 gibt aus: `1A`

### Warnung vor Verwechslungen verschiedener Behälter:

Eine *Variable* und ein *Modul* haben etwas gemeinsam: Beide sind *Behälter*.

Eine Variable und ein Modul *unterscheiden* sich durch das, was man *hineintun* kann (etwa so, wie eine Milchkanne und ein Benzinkanister):

Eine Variable enthält immer genau einen *Wert*. Sie kann nicht leer sein und sie kann nichts anderes als einen Wert enthalten.

Ein Modul kann *Variablen, Typen, Unterprogramme,* andere *Module* (und manchmal noch weitere Dinge) enthalten, aber *nie* einen *Wert*. Ein Modul kann auch leer sein (das kommt allerdings sehr selten vor und ist praktisch nicht sehr wichtig).

Ein Modul kann z.B. eine Variable und die kann einen Wert enthalten. Aber in diese Situation ist der Wert in der Variablen und nicht (direkt) im Modul. Nur die Variable ist (direkt) im Modul.

Noch ein guter Rat für den Haushalt: Tun Sie Milch nie in einen (gebrauchten) Benzinkanister und Benzin nie in eine Milchkanne! :-)

Zur Entspannung: Al-Khwarizmi (ca. 780-850)

*Abu Ja'far Muhammad ibn Musa Al-Khwarizmi* war ein islamischer Mathematiker, der in Bagdad lebte, über Indisch-Arabische Zahlen schrieb und zu den ersten gehörte, die die Ziffer 0 benutzten. Aus seinem Namen entstand (durch Übertragung ins Latein und dann in andere Sprachen) das Wort Algorithmus. Eine seiner Abhandlungen heißt *Hisab al-jabr w'al-muqabala* (auf Deutsch etwa: "Über das Hinüberbringen", von einer Seite einer Gleichung auf die andere). Daraus entstand unser Wort Algebra.

Wiederholungsfragen, 3. SU, Mo 19.04.10

1. Was ist ein Programm?
2. Was ist ein Typ?
3. Was ist ein Wertebehälter?
4. Übersetzen Sie die folgende Vereinbarung ins Deutsche:

```
int ilse = 123;
```

5. Übersetzen Sie den folgenden Ausdruck ins Deutsche:

```
... ilse * (ilse + 3) ...
```

6. Übersetzen Sie die folgende Anweisung ins Deutsche:

```
ilse = ilse * (ilse + 3);
```

7. Was für eine Anweisung haben Sie in Frage 6 übersetzt? Oder:
Wie heißt diese spezielle Anweisung?
8. Zu welchem Typ gehört das Literal `1234567`?
9. Zu welchem Typ gehört das Literal `1234567.0`?
10. Was wissen Sie über den Wert des Literals `0.1`?

Antworten zu den Wiederholungsfragen, 3. SU, Mo 19.04.10

1. Was ist ein Programm?

Eine Folge von Befehlen (die ein Programmierer geschrieben hat und
die ein Ausführer ausführen kann).

2. Was ist ein Typ?

Ein Bauplan für Variablen.

3. Was ist ein Wertebehälter?

Eine Variable oder ein E/A-Gerät
(z.B. ein Bildschirm, eine Tastatur, ein Drucker, eine Datei auf einer Festplatte etc.)

4. Übersetzen Sie die folgende Vereinbarung ins Deutsche:

```
int ilse = 123;
```

Erzeuge eine Variable namens ilse vom Typ int mit dem Anfangswert 123.

5. Übersetzen Sie den folgenden Ausdruck ins Deutsche:

```
... ilse * (ilse + 3) ...
```

Berechne den Wert des Ausdrucks ilse * (ilse + 3).

6. Übersetzen Sie die folgende Anweisung ins Deutsche:

```
ilse = ilse * (ilse + 3);
```

Berechne den Wert des Ausdrucks ilse * (ilse + 3) und tue ihn in die Variable ilse (oder: weise ihn der Variablen ilse zu).

7. Was für eine Anweisung haben Sie in Frage 6 übersetzt?
Oder: Wie heißt diese spezielle Anweisung?

Zuweisung (engl: assignment).

8. Zu welchem Typ gehört das Literal `1234567`?

Zum Typ int (nicht zum Typ Integer!).

9. Zu welchem Typ gehört das Literal `1234567.0`?

Zum Typ double (nicht zum Typ Double!).

10. Was wissen Sie über den Wert des Literals `0.1`?

Der Wert des Literals 0.1 ist nicht genau gleich ein Zehntel, sonder
ein ganz klein bißchen größer
(und ein ganz klein bißchen kleiner als der Wert des float-Literals 0.1F).

3. SU Mo 19.04.10

A. Wiederholung
B. Organisation

Tutorium von Frau Nusch:
Mi 5. Block (16.00-17.30 Uhr) im Raum B321 (Beginn: Mi 21.04.10)
Fr 5. Block (16.00-17.30 Uhr) im Raum B325 (Beginn: Fr 23.04.10)

## Verwechslung von Vereinbarungen und Anweisungen

Zu welchen Arten von Befehlen gehören die folgenden Befehle?

```
1 int otto = 17; // Variablen-Vereinbarung mit Initialisierung
2 otto = 25;     // Anweisung (genauer: Zuweisung)
```

Womit beginnt eine Variablen-*Vereinbarung* immer? (Mit einem Typ, d.h. mit einem Bauplan, nach dem Variable gebaut werden soll).

Womit beginnt eine *Zuweisung*? (Mit dem Namen einer Variablen)

## 4 Anweisungen (statements)

Was befiehlt der Programmierer dem Ausführer mit einer Anweisung?
(Bestimmte Werte in bestimmte Wertebehälter zu tun)

Man unterscheidet 2 Arten von Anweisungen:

| deutsch | englisch | Beispiele |
|---|---|---|
| Einfache Anweisungen | simple statement | Zuweisung, `break`, `continue`, `return`, `pln`, ... |
| Zusammengesetzte Anweisungen | compound statement | `if`, `while`, `for`, `switch`, ... |

Eine *zusammengesetzte Anweisung* erkennt man daran,
dass sie andere Anweisungen "als ihre Teile" enthält.

Woraus besteht eine Zuweisung, z.B.: `otto = 2 * (emil + 1);`
(aus einem Variablenamen wie `otto`,
einem Zuweisungszeichen = und
einem Ausdruck wie `2 * (emil + 1)`
)

Man sieht: Diese Zuweisung enthält verschiedene Teile, aber keine Anweisungen.

## Was kann der Programmierer mit zusammengesetzten Anweisungen bewirken?

Angenommen, eine Methode (z.B. die `main`-Methode eines Hallo-Programms) enthält nur *einfache Anweisungen* (z.B. nur Zuweisungen).

Wenn der Ausführer diese Methode *einmal* ausführt, wie oft führt er dazu jede der einfachachen Anweisungen aus? (Natürlich genau *einmal*).

Genau einmal ausgeführt werden ist also der Normalfall.

Mit zusammengesetzten Anweisungen kann der Programmierer bewirken, dass die darin enthaltenen Anweisungen *weniger als einmal* oder *mehr als einmal* ausgeführt werden.

Die if-Anweisung

Angenommen, wir wollen den Wert der Variablen `g1` ausgeben, aber *nicht*, wenn er *negativ* ist. Das geht etwa so:

```
1 if (g1 >= 0) {
2    pln("g1: " + g1);
3 }
```

Diese `if`-Anweisung enthält die Anweisung `pln(...)`. Diese `pln`-Anweisung wird *nicht ausgeführt*, wenn `g1` negativ ist. Ohne die `if`-Anweisung drumrum würde die `pln`-Anweisung *immer* ausgeführt.

Die Befehle in den geschweiften Klammern bezeichnen wir als den *dann-Rumpf* der `if`-Anweisung. So ein dann-Rumpf kann beliebig viele Befehle enthalten. Eine `if`-Anweisung kann zusätzlich auch noch einen *sonst-Rumpf* haben.

S. 61, Beispiel-03

Welche Befehle stehen im dann-Rumpf dieser `if`-Anweisung? In sonst-Rumpf?

Die while-Anweisung

Eine `while`-Anweisung hat nur *einen* Rumpf. Sie bewirkt normalerweise, dass dieser Rumpf *mehr als einmal* (z.B. 3 Mal oder 500 Mal oder 135000 Mal etc.) ausgeführt wird.

S. 69, Beispiel-01

Hier wird eine Methode namens `zufallsZahl` vorausgesetzt und aufgerufen. Stellen Sie sich vor, dass diese Methode bei jedem Aufruf einen *zufällig gewählten* `int`-Wert als Ergebnis liefert. Wie man eine solche Methode programmieren kann besprechen wir später mal.

Die `break`-Anweisung (in Zeile 6) bewirkt, dass die Ausführung der Schleife beendet wird, d.h. der Ausführer springt hinter die Zeile 8.

Anmerkung: In den runden Klammern hinter `while` kann man auch eine andere *Bedingung* angeben als `true`, das sollte man aber nicht: Es hat keine Vorteile und manchmal hat es Nachteile. Beginnen Sie deshalb alle `while`-Schleifen mit `while (true) {` (wie im Beispiel-01 auf S. 69).

Die `if`-Anweisung mit der `break`-Anweisung darin sollte immer so einfach sein wie in diesem Beispiel: Nur ein *dann-Rumpf* (kein `else` mit *sonst-Rumpf* dahinter) und als dann-Rumpf nur eine break-Anweisung (ohne geschweifte Klammern drumrum und ohne weitere Befehle).

Zusammenfassung:

Eine `if`-Anweisung hat *ein* bis *zwei* Rümpfe, eine `while`-Anweisung hat *einen* Rumpf.
Mit der `if`-Anweisung kann man bewirken, dass ihr Rumpf (oder einer der beiden Rümpfe) weniger als einmal (d.h. keinmal) ausgeführt wird.
Mit einer `while`-Anweisung kann man bewirken, dass ihr Rumpf mehr als einmal ausgeführt wird.

Ergänzung: Mit einer `while`-Schleife kann man auch bewirken, dass ihr Rumpf nur einmal oder keinmal ausgeführt wird, aber dass sind nur Sonderfälle. Typischerweise wird der Rumpf einer `while`-Schleife mehr als einmal ausgeführt.

Zur Entspannung: Selbstzutreffende ("autologische") Worte

Es gibt Worte, die offenbar auf sich selbst zutreffen, z. B. *Hauptwort* (ist ein Hauptwort), *deutsch* (ist ein deutsches Wort), *English* (ist ein englisches Wort), *kurz* (ist ein ziemlich kurzes Wort) und *dreisilbig* (ist dreisilbig). Solche Worte bezeichnen wir hier als selbstzutreffend.

Worte, die nicht auf sich selbst zutreffen, bezeichnen wir als selbstunzutreffend, z. B. *zweisilbig* (ist dreisilbig), *Tätigkeitswort* (ist ein Hauptwort) und *englisch* (ist ein deutsches Wort).

Frage: Ist jedes Wort entweder selbstzutreffend oder selbstunzutreffend? Was ist mit dem Wort selbstunzutreffend? Ist es selbstzutreffend oder selbstunzutreffend?

## 2.3 Mehrere Klassen, ein Programm

Was macht die Methode `System.out.println` mit ihrem (einzigen) Parameter?
(Sie gibt ihn zum Bildschirm aus)

Was macht die Methode `pln` in der Klasse `Hallo04`? (siehe S. 30, Zeile 18)
(Sie wendet die Methode `System.out.println` auf ihren Parameter `ob` an und gibt ihn damit zum Bildschirm aus).

Was macht die Methode `main` in der Klasse `Hallo03`? (siehe S. 28, ab Zeile 18)
(Sie gibt insgesamt 5 Zeilen zum Bildschirm aus, 3 davon sind "Sternchen-Zeilen").

Was macht die Methode `main` in der Klasse `Hallo05`? (siehe S. 31, ab Zeile 5)
(Sie ruft die Methode `pln` aus der Klasse `Hallo04` auf mit `"Hallo05: Jetzt geht es los!"` als Parameter.
Dann ruft sie die Methode `main` aus der Klasse `Hallo03` auf mit `null` als Parameter.
... )

Punkt-Notation:

```
Modul  .Element
Hallo04.pln
Hallo03.main
```

Zusammenfassung: Angenommen, wir haben in einer Klasse `Hallo17` ein paar Methoden namens `m01`, `m02`, `m03` etc. vereinbart. Dann können wir diese Methoden auch in anderen Klassen aufrufen, indem wir dort die Namen

```
Hallo17.m01(...)  // Modul Hallo17, Element m01
Hallo17.m02(...)  // Modul Hallo17, Element m02
Hallo17.m03(...)  // Modul Hallo17, Element m03
```

etc.

verwendet.

Wiederholungsfragen, 4. SU, Di 20.04.10

1. Wodurch unterscheiden sich *einfache Anweisungen* von *zusammengesetzten Anweisungen*? Tip: Es geht um die Bestandteile, aus denen die Anweisungen bestehen.

2. Mit welchen zusammengesetzten Anweisungen kann der Programmierer bewirken, dass die darin enthaltenen Anweisungen *weniger als einmal* ausgeführt werden?

3. Mit welchen zusammengesetzten Anweisungen kann der Programmierer bewirken, dass die darin enthaltenen Anweisungen *mehr als einmal* ausgeführt werden?

4. Wie viele Rümpfe kann eine `if`-Anweisung haben und wie bezeichnen wir diese Rümpfe?

5. Betrachten Sie die folgenden Variablen-Vereinbarungen:

```
1 int zahl          = EM.liesInt();
2 int anzahlZiffern = 0;
```

Wie viele Ziffern braucht man, um die eingelesene `zahl` im 10-er-System darzustellen? Geben Sie eine `while`-Schleife an, die die Antwort auf diese Frage in die Variable `anzahlZiffern` bringt. Sie dürfen sich darauf verlassen, dass die eingelesene Zahl nicht negativ ist.

6. Betrachten Sie die folgenden Variablen-Vereinbarungen:

```
1 int zahl          = EM.liesInt();
2 int anzahlZiffern = 0;
```

Wie viele Ziffern braucht man, um die eingelesene `zahl` im 2-er-System darzustellen? Geben Sie eine `while`-Schleife an, die die Antwort auf diese Frage in die Variable `anzahlZiffern` bringt. Sie dürfen sich darauf verlassen, dass die eingelesene Zahl nicht negativ ist.

Die folgenden Beispiele sollen heute im seminaristischen Unterricht ("in der Vorlesungun") behandelt werden und haben nichts mit den Wiederholungsfragen zu tun.

Beispiele für Ausdrücke mit Seiteneffekt:

```
1 int n1=17, n2=17, n3=17, n4=17; // Vier int-Variablen mit Anfangswert 17
2 int m1= 0, m2= 0, m3= 0, m4= 0; // Vier int-Variablen mit Anfangswert  0
3
4            // Werte der Variablen nach der Zuweisung
5 m1 = n1++; // m1:       n1:
6 m2 = ++n2; // m2:       n2:
7 m3 = n3--; // m3:       n3:
8 m4 = --n3; // m4:       n4:
```

Rückseite der Wiederholungsfragen, 4. SU, Di 20.04.10

1.1. Die Vereinbarung einer Methode `druckeZeile01`:

```
 1 static public void druckeZeile01() {
 2    pln( "****************************************" +
 3         "****************************************" );
 4 }
```

1.2. Ein paar Aufrufe der Methode `druckeZeile01`:

```
 5 static public void anwendung01() {
 6    druckeZeile01();
 7    pln("Hallo Alfred!");
 8    druckeZeile01();
 9    pln("Hallo Bianca!");
10    druckeZeile01();
11 }
```

1.3 Ausgabe der Methode `anwendung01` zum Bildschirm:

```
********************************************************************************
Hallo Alfred!
********************************************************************************
Hallo Bianca
********************************************************************************
```

2.1. Die Vereinbarung einer Methode `druckeZeile02`:

```
 1 static public void druckeZeile02(String str, int anz) { // 2 formale Params
 2    while (true) {
 3       if (anz <= 0) break;
 4       p(str);
 5       anz = anz - 1;
 6    }
 7 }
```

2.2. Ein paar Aufrufe der Methode `druckeZeile02`:

```
 8 static public void anwendung02() {
 9    druckeZeile02("*", 80);            // 2 aktuelle Params "*"    und 80
10    pln("Hallo Alfred!");
11    druckeZeile02("*", 40);            // 2 aktuelle Params "*"    und 40
12    pln("Hallo Bianca!");
13    druckeZeile02("\--/", 20);         // 2 aktuelle Params "\--/" und 20
14 }
```

2.3 Ausgabe der Methode `anwendung02` zum Bildschirm:

```
********************************************************************************
Hallo Alfred!
************************************
Hallo Bianca!
\--/\--/\--/\--/\--/\--/\--/\--/\--/\--/\--/\--/\--/\--/\--/\--/\--/\--/\--/\--/
```

Erläuterung:
Die Methode `druckeZeile01` hat 0 Parameter. Wenn man sie mehrmals aufruft macht sie immer dasselbe. Mit ihr kann man (nur) 80 Sternchen ausgeben.

Die Methode `druckeZeile02` hat *formale Parameter* (2 Stück). Was sie genau macht hängt von den *aktuellen Parametern* ab, mit denen man sie aufruft. Mit ihr kann man z.B. 80 Sternchen `"*"` ausgeben oder 80 Pluszeichen `"+"` oder 40 Sternchen `"*"` oder 20 Mal den String `"\--/"` oder ... .

Methoden *ohne Parameter* sind schon eine großartige Erfindung, aber Methoden *mit Parametern* sind noch viel flexibler und nützlicher als Methoden *ohne Parameter*.

Antworten zu den Wiederholungsfragen, 4. SU, Di 20.04.10

1. Wodurch unterscheiden sich *einfache Anweisungen* von *zusammengesetzten Anweisungen*? Tip: Es geht um die Bestandteile, aus denen die Anweisungen bestehen.

Zusammengesetzte Anweisungen enthalten Anweisungen als ihre Bestandteile. Einfache Anweisungen können Ausdrücke und Variablen-Namen etc. enthalten, aber keine Anweisungen.

2. Mit welchen zusammengesetzten Anweisungen kann der Programmierer bewirken, dass die darin enthaltenen Anweisungen *weniger als einmal* ausgeführt werden?

Mit den zusammengesetzten Anweisungen if und switch (Fallunterscheidungen).

3. Mit welchen zusammengesetzten Anweisungen kann der Programmierer bewirken, dass die darin enthaltenen Anweisungen *mehr als einmal* ausgeführt werden?

Mit den zusammengesetzten Anweisungen while und for (Schleifenanweisungen).

4. Wie viele Rümpfe kann eine `if`-Anweisung haben und wie bezeichnen wir diese Rümpfe?

Eine if-Anweisung kann einen Rumpf haben (einen dann-Rumpf) oder zwei Rümpfe (einen dann-Rumpf und einen sonst-Rumpf).

5. Betrachten Sie die folgenden Variablen-Vereinbarungen:

```
1 int zahl          = EM.liesInt();
2 int anzahlZiffern = 0;
```

Wie viele Ziffern braucht man, um die eingelesene `zahl` im 10-er-System darzustellen? Geben Sie eine `while`-Schleife an, die die Antwort auf diese Frage in die Variable `anzahlZiffern` bringt. Sie dürfen sich darauf verlassen, dass die eingelesene Zahl nicht negativ ist.

```
3 while (true) {
4    anzahlZiffern = anzahlZiffern + 1;
5    zahl          = zahl / 10;
6    if (zahl == 0) break;
7 }
```

6. Betrachten Sie die folgenden Variablen-Vereinbarungen:

```
1 int zahl          = EM.liesInt();
2 int anzahlZiffern = 0;
```

Wie viele Ziffern braucht man, um die eingelesene `zahl` im 2-er-System darzustellen? Geben Sie eine `while`-Schleife an, die die Antwort auf diese Frage in die Variable `anzahlZiffern` bringt. Sie dürfen sich darauf verlassen, dass die eingelesene Zahl nicht negativ ist.

```
3 while (true) {
4    anzahlZiffern = anzahlZiffern + 1;
5    zahl          = zahl / 2;
6    if (zahl == 0) break;
7 }
```

4. SU Di 20.04.10

A. Wiederholung
B. Organisation

Ausdrücke mit Seiteneffekt

Normale Ausdrücke befehlen dem Ausführer nur, *einen Wert zu berechnen,* aber nicht die Inhalte irgendwelcher Wertebehälter zu verändern (das befiehlt man normalerweise mit Anweisungen).

Als Ausnahme gibt es *Ausdrücke mit Seiteneffekt,* die dem Ausführer befehlen, *einen Wert zu berechnen* und dabei ("als Seiteneffekt") auch *die Inhalte bestimmter Wertebehälter zu verändern.*

Beispiele für Ausdrücke mit Seiteneffekt:

```
1 int n1=17, n2=17, n3=17, n4=17; // Vier int-Variablen mit Anfangswert 17
2 int m1= 0, m2= 0, m3= 0, m4= 0; // Vier int-Variablen mit Anfangswert  0
3
4            // Werte der Variablen nach der Zuweisung
5 m1 = n1++; // m1: 17, n1: 18
6 m2 = ++n2; // m2: 18, n2: 18
7 m3 = n3--; // m3: 17, n3: 16
8 m4 = --n3; // m4: 16, n4: 16
```

Die Ausdrücke `n1++`, `++n2`, `n3--`, `--n4` sind besonders häufig verwendete *Ausdrücke mit Seiteneffekt.* Die Operatoren ++ und -- darf man nur auf *Variablen* anwenden, nicht auf andere Ausdrücke (verboten sind z.B. `(x+1)++` und `--(x+1)` und `++17` und `23--` etc.)

Zusammengesetzte Anweisungen: 4.2.5 Die for-Schleife (S. 73)

S. 73, Beispiel-01
S. 74, Beispiel-02

Welche Zahlenfolge gibt die erste `for`-Schleife (in Zeile 4 bis 6) aus?
Bearbeiten Sie die zweite `for`-Schleife entsprechend zu Hause.

Die 4 Teile einer `for`-Schleife und die Reihenfolge, in der sie ausgeführt werden.

Im Fortschaltungsteil einer for-Schleife macht es keinen Unterschied, ob man `i++` oder `++i` (oder `i--` oder `--i`) schreibt, da die *Werte* der Ausdrücke unbenutzt bleiben und nur die *Seiteneffekte* wichtig sind.

3 Programme selbst ausführen (S. 41)

Die arbeitsökonomische Bedeutung von Schleifen
(wenig Programmierer-Arbeit, viel Ausführer-Arbeit).

Methoden

Def.: Eine *Methode* ist ein Unterprogramm, welches innerhalb einer Klasse vereinbart wurde.

In Java darf man Unterprogramme *nur innerhalb von Klassen* vereinbaren.
Also sind in Java *alle* Unterprogramme Methoden.

Anmerkung: In der Programmiersprache C++ ist das anders. Da kann man Unterprogramme innerhalb und ausserhalb von Klassen vereinbaren.

Zur Entspannung: Christian Morgenstern (1871-1914, München-Meran) Lyriker, Redakteur, Übersetzer (Ibsen, Strindberg).

Der Werwolf

Ein toter Lehrer gibt einem Werwolf Nachhilfeunterricht in Grammatik

Ein Werwolf eines Nachts entwich / von Weib und Kind und sich begab
an eines Dorfschulmeisters Grab / und bat ihn "Bitte, beuge mich!"

Wiederholungsfragen, 5. SU, Mo 26.04.10

1. Betrachten Sie die folgende Vereinbarung und 4 Zuweisungen:

```
1 int a = 5;
2 int b = a++;  // a:      b:
3 int c = ++a;  // a:      c:
4 int d = a--;  // a:      d:
5 int e = --a;  // a:      e:
```

Geben Sie neben jeder Zuweisung die Werte der beteiligten Variablen
nach Ausführung der Zuweisung an.

2. Was gibt die folgende for-Schleife zum Bildschirm aus?

```
6 for (int zahl = 7; zahl < 20; zahl = Zahl + 3) {
7    p(zahl + " ");
8 }
```

3. Geben Sie eine for-Schleife an, die die folgenden Zahlen zum Bildschirm ausgibt:

```
5 3 1 -1 -3 -5
```

4. Wenn jemand schon weiß, was ein *Unterprogramm* ist, wie können Sie ihm dann (ganz kurz) erklären, was eine *Methode* ist?

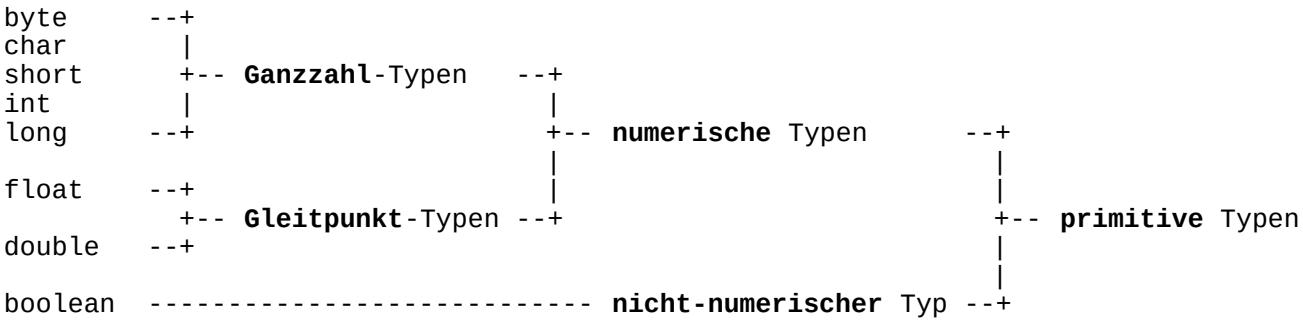
5. Was befiehlt der Programmierer dem Ausführer mit einem *Ausdruck*?
- Einen Modul zu erzeugen?
- Eine Methode auszuführen?
- Ein Ergebnis zu berechnen?
- Werte in Wertebehälter zu tun?
- Einen Wert auszugeben?
- Einen Wert zu berechnen?

5. Was befiehlt der Programmierer dem Ausführer mit einer *Anweisung*?
- Einen Modul zu erzeugen?
- Eine Methode auszuführen?
- Ein Ergebnis zu berechnen?
- Werte in Wertebehälter zu tun?
- Einen Wert auszugeben?
- Einen Wert zu berechnen?

Rückseite der Wiederholungsfragen, 5. SU, Mo 26.04.10

Eine Übersicht über die primtiven Typen und Java

5.1 Primitive Typen (im Buch S. 91)




Wie viele Werte gehören zu den einzelnen primitiven Typen?

| Typ | Länge eines Wertes in Bits | Anzahl der Werte als Formel | Anzahl der Werte (ungefähr) |
|---|---|---|---|
| `byte` | 8 | $2^8$ | 256 |
| `char` | 16 | $2^{16}$ | 65 Tausend |
| `short` | 16 | $2^{16}$ | 65 Tausend |
| `int` | 32 | $2^{32}$ | 4 Milliarden |
| `long` | 64 | $2^{64}$ | 18 Trillionen |
| `float` | 32 | $2^{32}$ | 4 Milliarden |
| `double` | 64 | $2^{64}$ | 18 Trillionen |
| `boolean` | - | - | 2 |

Die Zahlen 256 (für `byte`) und 2 (für `boolean`) sind *exakte* Angaben der Anzahl.
Die anderen Zahlen sind *ungefähre* Angaben ("ca.-Angaben").

Antworten zu den Wiederholungsfragen, 5. SU, Mo 26.04.10

1. Betrachten Sie die folgende Vereinbarung und 4 Zuweisungen:

```
1 int a = 5;
2 int b = a++;  // a: 6    b: 5
3 int c = ++a;  // a: 7    c: 7
4 int d = a--;  // a: 6    d: 7
5 int e = --a;  // a: 5    e: 5
```

Geben Sie neben jeder Zuweisung die Werte der beteiligten Variablen
nach Ausführung der Zuweisung an.

2. Was gibt die folgende for-Schleife zum Bildschirm aus?

```
6 for (int zahl = 7; zahl < 20; zahl = Zahl + 3) {
7    p(zahl + " ");
8 }
```

Ausgabe der for-Schleife: `7 10 13 16 19`

3. Geben Sie eine for-Schleife an, die die folgenden Zahlen zum Bildschirm ausgibt:

```
5 3 1 -1 -3 -5
```

Eine Schleife, die die obigen Zahlen ausgibt:

```
 9 for (int i=5; i>=-5; i=i-2) {
10    p(i + " ");
11 }
```

4. Wenn jemand schon weiß, was ein *Unterprogramm* ist, wie können Sie ihm dann (ganz kurz) erklären, was eine *Methode* ist?

Eine Methode ist ein Unterprogramm, das in einer Klasse vereinbart wurde.

5. Was befiehlt der Programmierer dem Ausführer mit einem *Ausdruck*?
- Einen Modul zu erzeugen?
- Eine Methode auszuführen?
- Ein Ergebnis zu berechnen?
- Werte in Wertebehälter zu tun?
- Einen Wert auszugeben?
- Einen Wert zu berechnen?

5. Was befiehlt der Programmierer dem Ausführer mit einer *Anweisung*?
- Einen Modul zu erzeugen?
- Eine Methode auszuführen?
- Ein Ergebnis zu berechnen?
- Werte in Wertebehälter zu tun?
- Einen Wert auszugeben?
- Einen Wert zu berechnen?

5. SU Mo 26.04.10

A. Wiederholung
B. Organisation

## 1.4.3 Das Konzept eines Unterprogramms / einer Methode

Rückseite der Wiederholungsfragen vom 4. SU: Die Methoden
`druckeZeile01` (ohne Parameter) und
`druckeZeile02` (mit Parametern) besprechen.

### Regel für Methoden:

Der Programmierer muss eine Methode *einmal* vereinbaren.
Dann darf er sie *beliebig oft* aufrufen (0 Mal oder 1 Mal oder 3 Mal oder 17 Mal ...)

Wenn man eine *Methode vereinbart* macht man folgendes:
- man erfindet *einen neuen Befehl*
- legt einen *Namen* dafür fest
- legt die (formalen) *Parameter* fest (Anzahl, Typen und Namen)
- legt die "*alten Befehle*" fest, aus denen der neue Befehl bestehen soll

Die neuen Befehle können *Ausdrücke* oder *Anweisungen* sein.

### Prozeduren und Funktionen (2 Arten von Methoden)

Eine *Prozedur* ist eine Methode die dazu dient, die Inhalte von *Wertbebehältern zu verändern.*
Deshalb ist jeder Aufruf einer Prozedur eine *Anweisung* (und nicht ein Ausdruck).

Kennzeichen: In der Vereinbarung der Methode steht `void` vor dem Namen der Methode.

S. 15, Zeile 4: `druckeWillkommen` ist eine Prozedur. Welcher *Wertebehälter* wird verändert?
S. 15, Zeile 10: Was für eine Methode ist `main`?
S. 27, Zeile 5
S. 29, Zeile 5: Was für eine Methode ist `hoch2`? (Keine Prozedur)

Aufgabe-P: Schreiben Sie eine Prozedur namens `printSum` mit 3 `int`-Parametern,
die die Summe ihrer Parameter mit dem Text `"Summe: "` davor zum Bildschirm ausgibt.

Eine *Funktion* ist eine Methode die dazu dient, *einen Wert zu berechnen* (und als Ergebnis zu liefern).
Deshalb ist jeder Aufruf einer Funktion ein *Ausdruck* (und nicht eine Anweisung).

Kennzeichen: In der Vereinbarung der Methode steht ein richtiger Typ vor dem Namen der Methode (und nicht `void`).

S. 29, Zeile 5: Was für eine Methode ist `hoch2`? (Funktion)

Aufgabe-F1: Schreiben Sie eine Methode namens `sum` mit 3 `int`-Parametern,
die die Summe ihrer Paramter als Ergebnis liefert.

Aufgabe-F2: Schreiben Sie eine Methode entsprechend der folgenden Spezifikation:

```
1 static public int hoch(int b, int h) {
2    // Liefert 0, wenn h kleiner oder gleich 0 ist.
3    // Liefert sonst die h-te Potenz von b (also b hoch h)
4    // Beispiele:
5    // hoch( 3, -2) ist gleich 0
6    // hoch( 3,  0) ist gleich 0
7    // hoch( 3,  2) ist gleich 9
8    // hoch( 3,  4) ist gleich 81
9    // hoch(-2,  3) ist gleich -8
```

| Zur Entspannung: Englische Vokabeln: ambiguous, geek, drag etc. | |
|---|---|
| *ambiguous* | zweideutig, unklar |
| *geek* | Fachmann (z. B. für Computer), mild negativ. Früher war ein *geek* "ein wilder Mann mit Bart" auf einer Kirmes, der z. B. Mäusen den Kopf abbiss. |
| *to execute* | ausführen (z. B. ein Programm oder einen Befehl), hinrichten (z. B. einen Verurteilten, nur in Ländern mit Todesstrafe). |
| *rocket science* | wörtlich: Raketenwissenschaft, sonst: schwierig zu lernen, anspruchsvoll. |
| *to drag* | ziehen, zerren (z. B. eine Maus über eine Tischplatte). |
| *what a drag* | "Was für ne Mühe", Umstand. |
| *in full drag* | aufgebrezelt, aufgetakelt, auffällig zurecht gemacht. |
| *drag queen* | Transvestit. |
| *for the birds* | für die Katz, bringt nichts (wörtlich: für die Vögel), unnütz, z. B. im Wortspiel *nesting (of functions) is for the birds*. |

Wiederholungsfragen, 6. SU, Di 27.04.10

1. Wenn man eine Methodenvereinbarung liest, woran erkennt man, dass die Methode eine *Prozedur* ist?

2. Wenn man eine Methodenvereinbarung liest, woran erkennt man, dass die Methode eine *Funktion* ist?

3. Zu welcher Art von Befehlen (Vereinbarung, Ausdruck, Anweisung)
gehören *Aufrufe von Prozeduren*?

4. Zu welcher Art von Befehlen (Vereinbarung, Ausdruck, Anweisung) gehören
*Aufrufe von Funktionen*?

5. Die Worte *ausgeben* und *zurückgeben* klingen sehr ähnlich, haben aber ganz *verschiedene* Bedeutungen. Mit welchem Befehl werden Werte *zurückgegeben* (oder: geliefert)? Mit welchen Befehlen werden Werte *ausgegeben* (nennen Sie ein oder zwei Beispiele für solche Befehle. Es gibt noch viel mehr).

6. Schreiben Sie eine Methode namens `m1`, die den `int`-Wert `17` zum Bildschirm *ausgibt*.

7. Schreiben Sie eine Methode namens `m2`, die den `int`-Wert `17` *zurückgibt* (oder: als Ergebnis liefert).

Antworten zu den Wiederholungsfragen, 6. SU, Di 27.04.10

1. Wenn man eine Methodenvereinbarung liest, woran erkennt man, dass die Methode eine *Prozedur* ist?

Daran, dass unmittelbar vor dem Namen der Methode void steht.

2. Wenn man eine Methodenvereinbarung liest, woran erkennt man, dass die Methode eine *Funktion* ist?

Daran, dass unmittelbar vor dem Namen der Methode ein richtiger Typ (wie int oder String oder ...) steht.

3. Zu welcher Art von Befehlen (Vereinbarung, Ausdruck, Anweisung) gehören *Aufrufe von Prozeduren*?

Prozedur-Aufrufe sind Anweisungen.

4. Zu welcher Art von Befehlen (Vereinbarung, Ausdruck, Anweisung) gehören *Aufrufe von Funktionen*?

Funktions-Aufrufe sind Ausdrücke.

5. Die Worte *ausgeben* und *zurückgeben* klingen sehr ähnlich, haben aber ganz *verschiedene* Bedeutungen. Mit welchem Befehl werden Werte *zurückgegeben* (oder: geliefert)? Mit welchen Befehlen werden Werte *ausgegeben* (nennen Sie ein oder zwei Beispiele für solche Befehle. Es gibt noch viel mehr).

Zurückgegeben (oder: geliefert) werden Werte mit dem Befehl return.

Ausgeben kann man Werte z.B. mit den Befehlen p und pln (alias System.out.print und System.out.println).

6. Schreiben Sie eine Methode namens `m1`, die den `int`-Wert `17` zum Bildschirm *ausgibt*.

```
1 static public void m1() {pln(17);}
```

7. Schreiben Sie eine Methode namens `m2`, die den `int`-Wert `17` *zurückgibt* (oder: als Ergebnis liefert).

```
2 static public int m2() {return 17;}
```

6. SU Di 27.04.10

A. Wiederholung
B. Organisation Klausur: Di 13.07.2010, 18 Uhr B554

Ähnlichkeiten und Unterschiede zwischen Funktionen und Prozeduren (S. 193)

5 Typen (S. 89)

Was ist ein Typ? (Ein Bauplan für Variablen)

Noch eine andere Definition: Ein Typ besteht aus
einer Menge von Werten und
einer Menge von Operationen, die man (sinnvoll) auf diese Werte anwenden kann.

Beispiel: Der Typ `int` besteht aus ca. 4,3 Milliarden Werten und
zahlreichen Operationen wie `+`, `-`, `*`, `/`, `%`, `pln`, `p` etc.

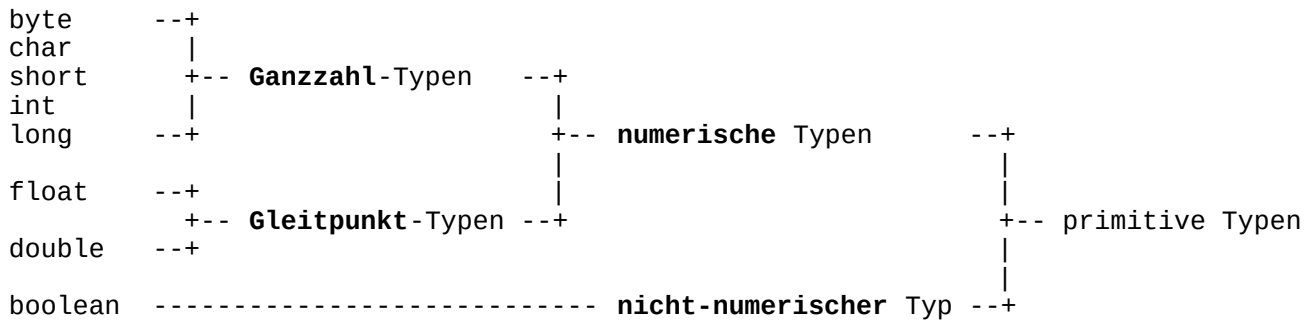
In Java unterscheidet man zuerstmal *zwei Arten* von Typen:

Primitive Typen: Davon gibt es genau 8 Stück
Referenztypen: Unbegrenzt viele, u.a. ca. 4000 in der Java-Standardbibliothek

Jetzt betrachten wir die Rückseite des Blattes mit den Wiederholungsfragen

---

5.1 Primitive Typen (S. 91)




Wie viele Werte gehören zu den einzelnen primitiven Typen?

| Typ | Länge eines Wertes in Bits | Anzahl der Werte als Formel | Anzahl der Werte (ungefähr) |
|---|---|---|---|
| `byte` | 8 | $2^8$ | 256 |
| `char` | 16 | $2^{16}$ | 65 Tausend |
| `short` | 16 | $2^{16}$ | 65 Tausend |
| `int` | 32 | $2^{32}$ | 4 Milliarden |
| `long` | 64 | $2^{64}$ | 18 Trillionen |
| `float` | 32 | $2^{32}$ | 4 Milliarden |
| `double` | 64 | $2^{64}$ | 18 Trillionen |
| `boolean` | - | - | 2 |

Die Zahlen 256 (für `byte`) und 2 (für `boolean`) sind *exakte* Angaben der Anzahl.
Die anderen Zahlen sind *ungefähre* Angaben ("ca.-Angaben").

---

Typen sind ja *Baupläne für Variablen*. Um den Unterschied zwischen *primitiven Typen* und *Referenztypen* zu verstehen, brauchen wir ein genaueres Modell von *Variablen*. Aus wie vielen Teilen und was für Teilen besteht eine Variable? Um solche und ähnliche Fragen anschaulich beantworten zu können, machen wir uns mit der

Bojendarstellung von Variablen

vertraut. An der Tafel das Beispiel-01 von S. 119 anschreiben, entwickeln, erläutern (erstmal *ohne zu verraten*, dass dieses Beispiel auch im Buch steht).

S. 121, Beispiel-02, 2 primitive Variablen und 2 Referenzvariablen.

Wichtige Regeln für die Zuweisung (=) und den Gleichheitsoperator (==):

Eine Zuweisung `x = ...;` weist immer dem Wert-Kästchen von `x` einen neuen Wert zu (nicht dem Zielwert oder der Referenz oder ...).

Eine Zuweisung `x = y;` kopiert den Wert von `y` in das Wert-Kästchen von `x` (nicht den Zielwert oder die Referenz oder ...).

Die Gleichheitsoperation `==` vergleicht immer die Werte ihrer Operanden (nicht die Zielwerte oder die Referenzen oder ...)

S. 125, Beispiel-07:

Mit welchem Wert wird die Variable `b01` (in Zeile 20) initialisiert?
Mit welchem Wert wird die Variable `b02` (in Zeile 21) initialisiert?

Zur Entspannung: Alan Mathison Turing (1912-1954), einer der Begründer der Informatik

Bevor man die ersten elektronischen Computer baute, konzipierte und untersuchte der Mathematiker Alan Turing eine Rechenmaschine, die so einfach war, dass niemand an ihrer prinzipiellen Realisierbarkeit zweifelte. Eine solche Turing-Maschine besteht aus einem unbegrenzt langen Band, welches in kleine Abschnitte eingeteilt ist, von denen jeder genau ein Zeichen eines endlichen Alphabets aufnehmen kann. Ein Schreib-Lese-Kopf über dem Band kann bei jedem Arbeitsschritt der Maschine das Zeichen auf dem aktuellen Abschnitt lesen und in Abhängigkeit davon ein bestimmtes Zeichen auf den aktuellen Abschnitt schreiben und einen Abschnitt nach links oder rechts weiterrücken. Ein Programm für eine solche Maschine besteht aus einer endlichen Menge von Befehlen der folgenden Form:

"Wenn das aktuelle Zeichen gleich X ist, dann schreibe Y und gehe einen Abschnitt nach links bzw. nach rechts bzw. bleib wo du bist" (wobei X und Y beliebige Zeichen des Alphabets sind).

Wichtige Erkenntnis 1: Es gibt viele (präzise definierte, mathematische) Probleme, die man mit Hilfe einer solchen Turing-Maschine lösen kann (z. B. das Multiplizieren von dreidimensionalen Matrizen).

Wichtige Erkenntnis 2: Es gibt aber auch (präzise definierte, mathematische) Probleme, die man nicht mit Hilfe einer solchen Turing-Maschine lösen kann.

Wichtige Vermutung 3: Alle Probleme, die man mit heutigen oder zukünftigen Computern lösen kann, kann man im Prinzip auch mit einer Turing-Maschine lösen.

Im zweiten Weltkrieg arbeitete Turing für die *Government Code and Cypher School* in Bletchley Park (d. h. für den britischen Geheimdienst) und half entscheidend dabei, die Maschine zu durchschauen, mit der die deutsche Marine ihre Funksprüche verschlüsselte (die Enigma), und wichtige Funksprüche zu entschlüsseln. Damit hat er vermutlich einer ganzen Reihe von aliierten Soldaten (Engländern, Amerikanern, Franzosen, Russen) das Leben gerettet.

Weil er homosexuell war, wurde Turing nach dem Krieg zu einer Hormonbehandlung "seiner Krankheit" gezwungen, bekam schwere Depressionen und nahm sich das Leben. Inzwischen wurden die entsprechenden Gesetze in England (und ähnliche Gesetze in anderen Ländern) beseitigt. Im September 2009 entschuldigte sich der britische Premierminister Gordon Brown dafür, wie Turing behandelt worden ist.

Wiederholungsfragen, 7. SU, Mo 03.05.10

1. In Java unterscheidet man (erstmal) zwei Arten von Typen. Wie heißen diese Typ-Arten?

2. Stellen Sie die folgenden beiden Variablen als Bojen dar. Schreiben Sie dabei die Namen der einzelnen Kästchen (Name, Referenz, Wert, Zielwert) gut lesbar neben die Kästchen.

```
1    String sabine  = new String("Hallo!");
2    String sybille = new String("Hallo!");
3    double dieter  = 2.5;
4    double detlev  = 2.5;
```

3. Welche Werte haben die folgenden beiden Ausdrück?

```
5    ... dieter == detlev  ... // Wert?
6    ... sabine == sybille ... // Wert?
```

4. Geben Sie für jeden der folgenden Typen an, wie viele Werte zu ihm gehören (ungefähr!):

```
7    int
8    float
9    long
10   double
```

5. Im folgenden werden die Typen `int` und `float` als *Zahlengeraden* dargestellt. Diese Darstellungen sind sehr "grob und schematisch" und nicht "maßstabsgerecht". Sie machen aber (hoffentlich) deutlich, dass der Bereich zwischen dem kleinsten und dem größten `int`-Wert viel kleiner ist,
als der Bereich zwischen dem kleinsten und dem größten `float`-Wert.

```
<-------------------------|--------|--------|---------------------------------->
                        Integer.    0     Integer.
                        MIN_VALUE         MAX_VALUE

<--|-------------------------------|----------------------------------------|-->
  -Float.                        0.0F                                      +Float.
  MAX_VALUE                                                                MAX_VALUE
```

5.1. Wo ungefähr liegt der Wert `Float.MIN_VALUE` (auf der Zahlengerade des Typs `float`)?

5.2. Gibt es zu jedem `int`-Wert einen genau entsprechenden `float`-Wert? Berücksichtigen Sie beim beantworten dieser Frage auch ihre Antworten zur vorigen Frage 4.

Antworten zu den Wiederholungsfragen, 7. SU, Mo 03.05.10

1. In Java unterscheidet man (erstmal) zwei Arten von Typen. Wie heißen diese Typ-Arten?

Primitive Typen, Referenztypen

2. Stellen Sie die folgenden beiden Variablen als Bojen dar. Schreiben Sie dabei die Namen der einzelnen Kästchen (Name, Referenz, Wert, Zielwert) gut lesbar neben die Kästchen.

```
1    String sabine  = new String("Hallo!");
2    String sybille = new String("Hallo!");
3    double dieter  = 2.5;
4    double detlev  = 2.5;
```

```
1    |sabine|---<100>---[<110>]---["Hallo!"]
2    |sybille|--<120>---[<130>]---["Hallo!"]
3    |dieter|---<140>---[2.5]
4    |detlev|---<150>---[2.5]
5
6    Name     Referenz  Wert      Zielwert
```

3. Welche Werte haben die folgenden beiden Ausdrück?

```
7    ... dieter == detlev  ... // Wert? true  (2.5 ist gleich 2.5)
8    ... sabine == sybille ... // Wert? false (110 ist ungleich 130)
```

4. Geben Sie für jeden der folgenden Typen an, wie viele Werte zu ihm gehören (ungefähr!):

```
9    int    4,3 Milliarden
10   float  4,3 Milliarden
11   long   18 Trillionen
12   double 18 Trillionen
```

5. Im folgenden werden die Typen `int` und `float` als *Zahlengeraden* dargestellt. Diese Darstellungen sind sehr "grob und schematisch" und nicht "maßstabsgerecht". Sie machen aber (hoffentlich) deutlich, dass der Bereich zwischen dem kleinsten und dem größten `int`-Wert viel kleiner ist,
als der Bereich zwischen dem kleinsten und dem größten `float`-Wert.

```
<-------------------------|--------|--------|---------------------------------->
                        Integer.    0     Integer.
                        MIN_VALUE         MAX_VALUE

<--|-------------------------------|----------------------------------------|-->
  -Float.                         0.0F                                     +Float.
  MAX_VALUE                                                                MAX_VALUE
```

5.1. Wo ungefähr liegt der Wert `Float.MIN_VALUE` (auf der Zahlengerade des Typs `float`)?

Dieser Wert liegt ganz nahe beim Wert 0.0, und zwar rechts von der 0.0 (da er positiv ist).

5.2. Gibt es zu jedem `int`-Wert einen genau entsprechenden `float`-Wert? Berücksichtigen Sie beim beantworten dieser Frage auch ihre Antworten zur vorigen Frage 4.

Nein, für viele int-Werte gibt es keinen genau entsprechenden float-Wert, denn die 4,3 Milliarden float-Werte sind über einen viel größeren Bereich der Zahlengeraden vertstreut als die 4,3 Milliarden int-Werte. Deshalb liegen bestimmte float-Werte viel weiter auseinander als die int-Werte.

7. SU Mo 03.05.10

A. Wiederholung
B. Organisation

## Hüllklassen

S. 91: Die Namen der 8 Hüllklassen

In jeder Hüllklasse (mit Ausnahme der Hüllklasse `Boolean`) gibt es zwei unveränderbare Variablen namens `MIN_VALUE` und `MAX_VALUE`. Die gehören zum entsprechenden primitiven Typ.

`Integer.MIN_VALUE` enthälte den kleinsten int-Wert.
`Float.MIN_VALUE` enthält den kleinsten positiven `float`-Wert (nicht den kleinsten `float`-Wert!)

Für `Long.MIN_VALUE` und `Double.MIN_VALUE` gilt entsprechndes.

## Unterschiede zwischen Zahlen in der Mathematik und in Java

S. 103: Die Zahlenmengen `N`, `Z`, `Q`, `R` und `C` in der Mathematik

S. 104: Die Wertemengen der Typen `byte`, `char`, `short`, `int`, `long`, `float`, `double` in Java

## 7 Reihungen

Angenommen, wir brauchen in einem Programm *viele Variablen* eines bestimmten Typs, z.B. 100 `long`-Variablen. Was können wir dann tun?

S. 150: Beispiel-01, Wir können 100 long-Variablen vereinbaren

S. 150: Beispiel-02, Um ihre Werte auszugeben müssen wir dann 100 `pln`-Befehle hinschreiben.

Ein Versuch, viele Variablen mit Hilfe einer *Schleife* zu vereinbaren:

S. 150: Beispiel-03, Das scheitert, weil der Name `v` immer nur eine long-Variable bezeichnet.

Vereinbarung von 100 `long`-Variablen als Reihung:

S.151: Beispiel-04

`v` ist eine Variable vom Typ *Reihung-von-long*. Die Reihung `v` hat 100 *Komponenten* vom Typ `long`.

## Tolle Eigenschaften der Reihung v:

1. Der Programmierer brauchte nur *einen* Namen zu erfinden (`v`), und hat damit 100 Namen für die 100 Komponenten von `v` festgelegt, nämlich die Namen `v[0]`, `v[1]`, `v[2]`, ..., `v[99]`.

2. Außerdem kann er mit noch erstaunlichere Namen auf die 100 `long`-Variablen zugreifen, etwa so:

```
1    int i = 17;
2    pln(v[i]);  // Gibt den Wert von v[17] aus.
3    i = i + 3;
4    pln(v[i]);  // Gibt den Wert von v[20] aus.
5    i = i / 2;
6    pln(v[i]);  // Gibt den Wert von v[10] aus.
7    ...
```

Zwei Anwendungen dieser erstaunlichen Namen:

S. 152: Beispiel-05, Was bewirkt die erste Schleife? Die zweite Schleife?

Aufgabe: Schreiben Sie eine Befehlsfolge, die die Summe aller Komponenten der Reihung `v` berechnet und (mit `pln`) ausgibt.

Hinweis: Reihungen und for-Schleifen sind eine besonders nützliche Kombination von Konstrukten

Aufgabe: Schreiben Sie eine Funktion, die der folgenden Spezifikation entspricht:

```
1 static public boolean enthaeltPaar(long[] r) {
2    // Liefert true, wenn in r zwei benachbarte Komponenten r[i] und r[i+1]
3    // gleiche Werte enthalten, sonst false.
```

## 5.6 Besonderheiten des Ganzzahltyps char

Der Typ `char` ist erstmal ein ganz *normaler Ganzzahltyp*, zu dem Werte von 0 bis 65 535 gehören. `char`-Werte kann man addieren, subtrahieren, multiplizieren, dividieren, "modulieren" (die modulo-Operation anwenden).

Merke: Die Werte in `char`-Variablen sind *Ganzzahlen*, keine "Buchstaben" oder "Zeichen".

Es gibt aber auch ein paar Unterschiede zwischen dem Ganzzahltyp `char` und den übrigen Ganzzahltypen (`byte`, `short`, `int`, `long`).

### 1. Beim Ausgeben von char-Werten

S. 115, Beispiel-01 und -02:
Der `char`-Wert `65` wird nicht als `65` sondern als `A` ausgegeben
(und `66` als `B`, `67` als `C`, ..., `90` als `Z`).
Der `char`-Wert `97` wird nicht als `97` sondern als `a` ausgegeben
(und `98` als `b`, `99` als `c`, ..., `122` als `z`).
Der `char`-Wert `48` wird nicht als `48` sondern als `0` ausgegeben
(und `49` als `1`, `50` als 2 ..., `57` als `9`).

### 2. Es gibt spezielle Literale des Typs char

Das Literal `'A'` bezeichnet den Wert `65` vom Typ `char` (der als `A` ausgegeben wird).
Das Literal `'Z'` bezeichnet den Wert `90` vom Typ `char` (der als `Z` ausgegeben wird).
Das Literal `'0'` bezeichnet den Wert `48` vom Typ `char` (der als `0` ausgegeben wird).
Das Literal `'9'` bezeichnet den Wert `57` vom Typ `char` (der als `9` ausgegeben wird).

### 3. Beim Rechnen mit char-Werten

Die Summe von zwei `char`-Werten ist nicht vom Typ `char`, sondern vom Typ `int`.

```
char c1 = 17;
char c2 =  4;
char c3 = c1 + c2;          // Falsch, Typfehler
char c4 = (char)(c1 + c2);  // Richtig!
```

Ganz entsprechendes gilt auch für die Operatoren `-`, `*`, `/` und `%`.

Zur Entspannung: Hilberts Hotel

Denken Sie sich ein Hotel mit unendlich vielen, nummerierten Zimmern: 1, 2, 3, ... . Alle Zimmer sind belegt. Dieses Hotel wurde nach dem Mathematiker David Hilbert (1862-1943) benannt.

1. Wie kann man 1 weiteren Gast unterbringen? (`ZrNr := ZrNr + 1`)

2. Wie kann man 100 weitere Gäste unterbringen? (`ZrNr := ZrNr + 100`)

3. Wie kann man unendlich viele weitere Gäste unterbringen? (ZrNr := ZrNr * 2, danach sind alle Zimmer mit ungeraden Nrn. (1, 3, 5, ...) frei.

Wiederholungsfragen, 8. SU, Di 04.05.10

1. Vereinbaren Sie 800 Variablen vom Typ `int` und initialisieren Sie sie alle mit dem Wert `17`.

2. Schreiben Sie eine `for`-Schleife, die in jede der 800 Variablen einen Wert einliest (mit dem Befehl `EM.liesInt()`).

3. Schreiben Sie eine `for`-Schleife, die den Wert von jeder der 800 Variablen verdoppelt.

4. Zu welchen Typen gehören die folgenden Literale:

| Literal | Typ? |
|---|---|
| `123` | |
| `2.5` | |
| `2.5F` | |
| `"Hallo"` | |
| `'H'` | |

5. Was geben die folgenden Anweisungen (zum Bildschirm) aus?

| Anweisung | Ausgabe? |
|---|---|
| `pln(65);` | |
| `pln('A');` | |
| `pln((char) 65);` | |
| `pln('A' + 1);` | |
| `pln((char)('A' + 1));` | |

Rückseite der Wiederholungsfragen, 8. SU, Di 04.05.10

Nobel-Preisträger unter den Angehörigen der Humboldt-Universität in Berlin

| | | |
|---|---|---|
| 1901-1909: | 8 | (1902 Theodor Mommsen, Literatur, 1905 Robert Koch, Medizin) |
| 1910-1919: | 6 | (1918 Max Plank, Physik, 1914 Max von Laue, Physik) |
| 1920-1929: | 4 | (1921 Albert Einstein, Physik, 1925 Gustav Hertz und James Franck, Physik) |
| 1930-1939: | 6 | (1932 Werner Heisenberg, Physik) |
| 1940-1949: | 1 | (1944 Otto Hahn, Chemie) |
| 1950-1956: | 4 | (1954 Max Born, Physik) |
| Summe | 29 | |

(Quelle: The Economist, September 10th-16th, 2005).

Antworten zu den Wiederholungsfragen, 8. SU, Di 04.05.10

1. Vereinbaren Sie 800 Variablen vom Typ `int` und initialisieren Sie sie alle mit dem Wert `17`.

```
1    int[] otto = new int[800];
2    for (int i=0; i<otto.length; i++);
```

2. Schreiben Sie eine `for`-Schleife, die in jede der 800 Variablen einen Wert einliest (mit dem Befehl `EM.liesInt()`).

```
3    for(int i=0; i<otto.length; i++) otto[i] = EM.liesInt();
```

3. Schreiben Sie eine `for`-Schleife, die den Wert von jeder der 800 Variablen verdoppelt.

```
4    for(int i=0; i<otto.length; i++) otto[i] = otto[i] * 2;
```

4. Zu welchen Typen gehören die folgenden Literale:

| Literal | Typ? |
|---|---|
| `123` | int |
| `2.5` | double |
| `2.5F` | float |
| `"Hallo"` | String |
| `'H'` | char |

5. Was geben die folgenden Anweisungen (zum Bildschirm) aus?

| Anweisung | Ausgabe? |
|---|---|
| `pln(65);` | 65 |
| `pln('A');` | A |
| `pln((char) 65);` | A |
| `pln('A' + 1);` | 66 |
| `pln((char)('A' + 1));` | B |

8. SU Di 04.05.10

A. Wiederholung
B. Organisation

7.1 Reihungen vereinbaren und als Bojen darstellen (S. 155)

S. 155: Beispiel-01: Eine Reihung vereinbaren und dabei gleich die Werte der Komponenten festlegen

Das geht nur, wenn die Reihung nicht "sehr lang" ist.

Von welchem *Typ* ist die Variable `lr01`?
Von welchem *Typ* sind die *Komponenten* der Reihung `lr01`?

Die Reihung `lr01` in ausführlicher Bojendarstellung (S. 156, Bild 7.1)

Bisher gehörten die Komponenten unserer Reihungen immer zu einem primitiven Typ (`long`, `int`, ...). Reihungen mit Komponenten eines Referenztyps sehen etwas komplizierter aus:

S. 157: Beispiel-02: Eine Reihung mit `StringBuilder`-Komponenten

Von welchem *Typ* ist die Variable `sr01`?
Von welchem *Typ* sind die *Komponenten* der Reihung `sr01`?

Die Reihung `sr01` in ausführlicher Bojendarstellung (S. 157, Bild 7.3)
Die Reihung `sr01` in vereinfachter Bojendarstellung (S. 158, Bild 7.4)

Der Unterschied zwischen einer leeren Reihung und keiner Reihung

Ein Kaffeetrinker unterscheiden (bewußt oder intuitiv) vor allem die folgenden drei Fälle:

1. Ich habe keine Kaffee-Tasse
2. Ich habe eine Kaffee-Tasse, aber sie ist leer.
3. Ich habe eine Kaffee-Tasse in der Kaffee drin ist.

Eine entsprechende Unterscheidung gibt es auch bei Reihungen:

```
1    double[] r1 = null;
2    double[] r2 = {};
3    double[] r3 = {1.5, 2.8, 3.2};
```

Die Variable `r1` hat den Wert `null` und zeigt somit nicht auf eine Reihung.
Die Variable `r2` zeigt auf eine Reihung, aber die Reihung ist leer (sie enthält 0 Komponenten)
Die Variable `r3` zeigt auf eine Reihung der Länge 3.

7.2 Die Erzeugung einer Reihung in 3 Schritten (und in einem Schritt)

Besondere for-Schleifen (for-each-Schleifen) zur Bearbeitung von Reihunen (und Sammlungen)

`for`-Schleifen zur Bearbeitung von allen Komponenten einer Reihung (*for-i-Schleifen*) wie etwa:

```
4    for (int i=0; i<r3.length; i++) {
5       pln(r3[i]; ...
6    }
```

braucht man so häufig, dass man (mit Java 5) dafür eine besonders einfache Sonderform eingeführt hat (sog. *for-each-Schleifen*):

```
7    for (double d : r3) {
8       pln(d); ...
9    }
```

S. 153: Beispiel-06: Eine `for-each`-Schleife

Machen Sie sich heute (spästestens morgen) mit `for-each`-Schleifen vertraut. Man kann damit viele Fehler vermeiden (weil `for-each`-Schleifen so viel einfacher sind als `for-i`-Schleifen)

## Richtig über geschachtelte Schleifen sprechen

Eine (`while`- oder `for`-) Schleife S1 kann in ihrem Rumpf beliebige Anweisungen enthalten, unter anderem auch eine Schleife S2 (oder sogar mehrere solche Schleifen). Und für die Schleife S2 gilt natürlich Entsprechendes. Man bezeichnet S2 dann auch als eine geschachtelte Schleife (wei sie in S1 "eingschachtelt ist").

Beispiel:

```
1    for (int i=1; i<=5; i++) {
2       pln("----------");
3       for (int j=10; j>=1; j--) { // Eine geschachtelte Schleife
4          p(i*j);
5       }
6       pln();
7    }
```

Wie oft wird der Befehl `p(i*j);` ausgeführt? (5 mal 10 gleich 50 Mal).

Beim Sprechen über Schleifen wird häufig nicht genau zwischen einer *Schleife* (z.B. der in Zeile 1 bis 7) und ihrem *Rumpf* (in Zeile 2 bis 6) unterschieden. Aber in bestimmten Fällen ist es wichtig, diesen Unterschied genau zu verstehen und zu beachten.

S. 86, Beispiel-02:

Die Methode `geschachtelt01` enthält zwei `for`-Schleifen

Geben Sie Zeilen-Nrn an (von wo bis wo) für

| | |
|---|---|
| die erste Schleife | (Zeilen 44-51) |
| den Rumpf der ersten Schleife | (Zeilen 45-50) |
| die zweite Schleife | (Zeilen 46-48) |
| den Rumpf der zweiten Schleife | (Zeile 47) |

Wenn die Methode geschachtelt01 einmal ausgeführt wird, wie oft wird dann ausgeführt

| | |
|---|---|
| die erste Schleife? | (1 Mal) |
| der Rumpf der ersten Schleife | (10 Mal) |
| die zweite Schleife | (10 Mal) |
| der Rumpf der zweiten Schleife | (200 Mal) |

Wenn man die zweite Schleife 1 Mal ausführt, wird ihr Rumpf 20 Mal ausgeführt.

Mit einem Satz wie
"Diese *Schleife* wird 17 Mal ausgeführt" ist sehr häufig gemeint:
"Der *Rumpf* dieser Schleife wird 17 Mal ausgeführt".

Zur Entspannung: Nobel-Preisträger unter den Angehörigen der Humboldt-Universität in Berlin

| | | |
|---|---|---|
| 1901-1909: | 8 | (1902 Theodor Mommsen, Literatur, 1905 Robert Koch, Medizin) |
| 1910-1919: | 6 | (1918 Max Plank, Physik, 1914 Max von Laue, Physik) |
| 1920-1929: | 4 | (1921 Albert Einstein, Physik, 1925 Gustav Hertz und James Franck, Physik) |
| 1930-1939: | 6 | (1932 Werner Heisenberg, Physik) |
| 1940-1949: | 1 | (1944 Otto Hahn, Chemie) |
| 1950-1956: | 4 | (1954 Max Born, Physik) |
| Summe | 29 | |

In den etwa 50 Jahren seit 1957 hat kein Angehöriger der Humboldt-Universität einen Nobelpreis gewonnen (Quelle: The Economist, September 10th-16th, 2005).

Wiederholungsfragen, 9. SU, Mo 10.05.10

1. Stellen Sie die folgende Reihungsvariable `r77` als *Boje* dar:

```
1    int[] r77 = {300, 100, 200};
```

2. Stellen Sie die folgende Reihungsvariable `r88` als *Boje* dar:

```
2    String[] r88 = {
3       new String("ABC"),
4       null,
5       new String("DE");
6    };
```

3. Von welchem Typ ist die Variable `r77`?

4. Von welchem Typ sind die *Komponenten* der Reihung `r77`?

5. Von welchem Typ ist die Variable `r88`?

6. Von welchem Typ sind die *Komponenten* der Reihung `r88`?

7. Stellen Sie sich vor, dass die folgende Methode einmal ausgeführt wird:

```
7 static public void m99() {
8    for (int i=1; i<=3; i++) {
9       for (int j=1; j<=10; j=j+2) {
10         pln("X");
11      }
12   }
13 }
```

7.1. Wie oft wird die *äußere* `for`-Schleife (Zeile 8 bis 12) ausgeführt?

7.2. Wie oft wird die *innere* `for`-Schleife (Zeile 9 bis 11) ausgeführt?

7.3. Wie oft wird der Rumpf der *äußeren* `for`-Schleife ausgeführt?

7.4. Wie oft wird der Rumpf der *inneren* `for`-Schleife ausgeführt?

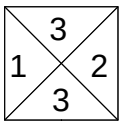
Rückseite der Wiederholungsfragen, 9. SU, Mo 10.05.10

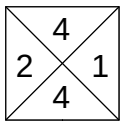
Exakt definierte Probleme von denen (Mathematiker und Informatiker) bewiesen haben, dass man sie mit keinem Computer (egal wie groß und schnell) lösen kann. Eine Lösung für eines dieser Probleme müsste ein Programm sein, das aus bestimmten Eingaben bestimmte Ausgaben berechnet. Im Folgenden werden diese Ein- und Ausgaben kurz beschrieben

Beispiel 1: Das Kachel-Katalog-Problem

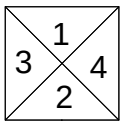
Eingabe: Ein Katalog von "Kacheln mit vier farbigen Kanten" und der Anschlussbedingung, dass nur gleichfarbige Kanten aneinander stoßen dürfen.
Ausgabe: "Ja", wenn man mit Kacheln aus dem Katalog eine beliebig große Fläche kacheln kann, "Nein" wenn das nicht möglich ist.

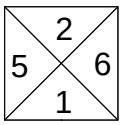
Katalog 1:



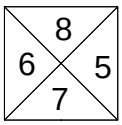




Katalog 2:

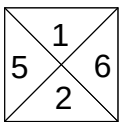


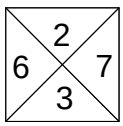



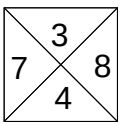


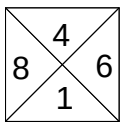



Katalog 3:




Kann mit Kacheln aus dem Katalog 1 eine beliebig große Fläche kacheln?

Ebenso mit Katalog 2?

Ebenso mit Katalog 3?

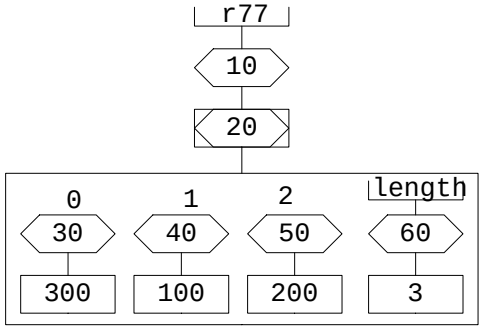
Beispiel 2: Das Halte-Problem

Eingabe: Ein Java-Programm P und ein Satz Eingaben E für P.
Ausgabe: "Hält", wenn das Programm P mit der Eingabe E nach endlich vielen Rechenschritten von allein anhält. "Hält nicht", wenn P mit E in eine Endlosschleife oder in eine Endlos-Rekursion gerät.

Antworten zu den Wiederholungsfragen, 9. SU, Mo 10.05.10

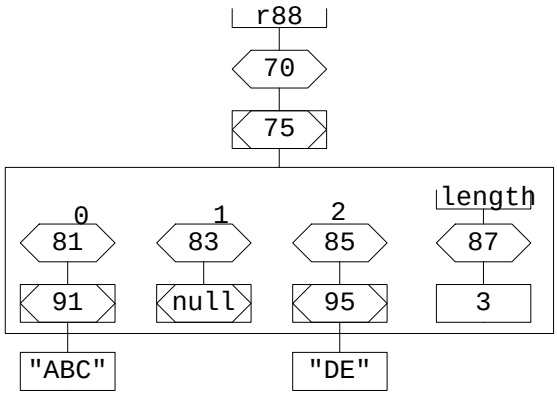
1. Stellen Sie die folgende Reihungsvariable `r77` als *Boje* dar:

```
1    int[] r77 = {300, 100, 200};
```




2. Stellen Sie die folgende Reihungsvariable `r88` als *Boje* dar:

```
2    String[] r88 = {
3       new String("ABC"),
4       null,
5       new String("DE");
6    };
```




3. Von welchem Typ ist die Variable `r77`? Reihung von int

4. Von welchem Typ sind die *Komponenten* der Reihung `r77`? int

5. Von welchem Typ ist die Variable `r88`? Reihung von String

6. Von welchem Typ sind die *Komponenten* der Reihung `r88`? String

7. Stellen Sie sich vor, dass die folgende Methode einmal ausgeführt wird:

```
 7 static public void m99() {
 8    for (int i=1; i<=3; i++) {
 9       for (int j=1; j<=10; j=j+2) {
10          pln("X");
11       }
12    }
13 }
```

7.1. Wie oft wird die *äußere* `for`-Schleife (Zeile 8 bis 12) ausgeführt? 1 Mal

7.2. Wie oft wird die *innere* `for`-Schleife (Zeile 9 bis 11) ausgeführt? 3 Mal

7.3. Wie oft wird der Rumpf der *äußeren* `for`-Schleife ausgeführt? 3 Mal

7.4. Wie oft wird der Rumpf der *inneren* `for`-Schleife ausgeführt? 15 Mal

9. SU Mo 10.05.10

A. Wiederholung
B. Organisation

Vereinfachte Bojendarstellung von Reihungen

S. 155, Beispiel-01: Vereinbarung einer Reihung namens `lr01`
S. 156, Bild 7.1: Die Reihung `lr01` in ausführlicher Bojendarstellung
S. 157, Bild 7.2: Die Reihung `lr01` in vereinfachter Bojendarstellung

Was darf man bei der vereinfachten Darstellung weglassen?

Die *Referenzen der Komponenten* und und die `length`-Variable.
Die Referenz der Reihungsvariable (`lr01`) darf man *nicht* weglassen!

S. 157, Beispiel-02: Vereinbarung einer Reihung namens `sr01`
S. 157, Bild 7.3: Die Reihung `sr01` in ausführlicher Bojendarstellung
S. 158, Bild 7.4: Die Reihung `sr01` in vereinfachter Bojendarstellung

Was kann der Java-Programmierer mit dem Werten einer Referenzvariablen machen?

Durch Zuweisungen = *verändern* (z.B. `r=null;` oder `r1=r2;`)
Mit den Operatoren == bzw. != mit einem anderen (Referenz-) Wert *vergleichen*
(auf *gleich* bzw. *ungleich*, z.B. `r==0` oder `r!=0` oder `r1==r2` oder `r1!=r2`)

Was kann der Java-Programmierer mit dem Werten einer Referenzvariablen nicht machen?

Mit den Operatoren < bzw > auf *kleiner* bzw. *größer vergleichen*
Mit arithmetischen Operationen *verändern* (z.B. `r+1` oder `2*r` etc.)
Mit `pln` oder anderen Ausgabebefehlen *ausgeben*.

7.4 Mehrstufige Reihungen (Reihungen von Reihungen)

S. 165, Beispiel-01: Eine Reihung von Reihung von `int`-Variablen

```
1 int[][] irrA = new int[7][22];
```

`irrA` ist eine Reihung der Länge 7.

Sie enthält 7 *Komponenten* vom Typ `int[]`.
Jede dieser Komponenten ist eine Reihung der Länge 24.
`irrA` enthält somit (7 mal 24 gleich) 168 *elementare Komponenten* vom Typ `int`.

Welchen Wert hat der Ausdruck `irrA.length`? (7)
Welchen Wert hat der Ausdruck `irrA[0].length`? (24)
Welchen Wert hat der Ausdruck `irrA[1].length`? (24)
Welchen Wert hat der Ausdruck `irrA[6].length`? (24)

Namen für die elementaren Komponten von irrA:

```
irrA[0][0], irrA[0][1], ..., irrA[0][23]
irrA[1][0], irrA[1][1], ..., irrA[1][23]
...
irrA[6][0], irrA[6][1], ..., irrA[6][23]
```

Von welchen Typen sind die folgenden Variablen?

`irrA`? (Reihung von Reihungen von `int`)
`irrA[3]`? (Reihung von `int`)
`irrA[3][17]`? (`int`)

Mit was für einem Befehl kann man alle elementaren Komponenten einer zweistufigen Reihungen wie irrA bearbeiten (z.B. ausgeben oder addieren oder ...)? (Mit geschachtelten `for`-Schleifen)

S. 166, Beispiel-02: Eine Methode zum Bearbeiten von Variablen des Typs `int[][]`.

Was für eine Methode ist `statistikIRR` (Prozedur oder Funktion)? (Prozedur)
Wie viele Parameter hat die Methode `statistikIRR`? (einen)
Von welchem Typ ist der (eine) Parameter der Methode? (vom Typ Reihung von Reihungen von `int`)

Der Reihungsparameter irr wird mit zwei geschachtelten for-each-Schleifen bearbeitet.

Was gibt der Prozeduraufruf

```
2    statistikIRR(irrA);
```

zum Bildschirm aus (wobei `irrA` die auf S. 165 vereinbarte und dargestellte Reihung ist)?

Ausgabe: `Anzahl Zahlen: 168, Summe: 0`

Es gibt 1-stufige, 2-stufige, 3-stufige, ... Reihungen (z.B. der Typen
`int[]`, `int[][]`, `int[][][]` ... oder `String[]`, `String[][]`, `String[][][]`, ...)

S. 167: Zwei Regeln über Reihungen ("Reihungen mit Flatterrand")

S. 168, Beispiel-04: Eine dreistufige Reihung und ihre Komponenten

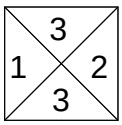
Zur Entspannung: Unlösbare Probleme?

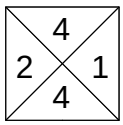
Gibt es exakt definierte Probleme, die man mit Computern nicht lösen kann? Ja.

Beispiel 1: Das Kachel-Katalog-Problem

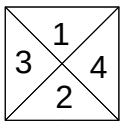
Eingabe: Ein Katalog von "Kacheln mit vier farbigen Kanten" und der Anschlussbedingung, dass nur gleichfarbige Kanten aneinander stoßen dürfen.
Ausgabe: "Ja", wenn man mit Kacheln aus dem Katalog eine beliebig große Fläche kacheln kann, "Nein" wenn das nicht möglich ist.

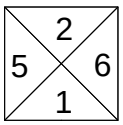
Katalog 1:

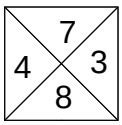


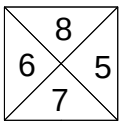




Katalog 2:

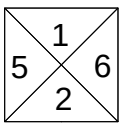


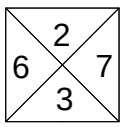



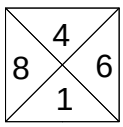




Katalog 3:








Kann mit Kacheln aus dem Katalog 1 eine beliebig große Fläche kacheln? (Ja)
Ebenso mit Katalog 2? (Ja) Ebenso mit Katalog 3? (Nein)

Beispiel 2: Das Halte-Problem

Eingabe: Ein Java-Programm P und ein Satz Eingaben E für P.
Ausgabe: "Hält", wenn das Programm P mit der Eingabe E nach endlich vielen Rechenschritten von allein anhält. "Hält nicht", wenn P mit E in eine Endlosschleife oder in eine Endlos-Rekursion gerät.

## 7.5 Mehrdimensionale Reihungen

Eine mehrstufige Reihung enthält nicht nur ihre elementaren Komponenten (z.B. `int-Variablen` oder `String`-Variablen), sondern auch noch Referenzen, die auf die Komponenten-Reihungen zeigen und `length`-Variablen.

S. 165, Die Reihung `irrA`. Angenommen, eine Referenz belegt gleich viel Speicherplatz wie ein int-Wert. Platz für wie viele Werte/Referenzen belegt irrA dann?

| | |
|---|---|
| Elementare Komponenten: 7 x 24 gleich | 168 |
| `length`-Variablen der Komponenten | 7 |
| Referenzen der Komponenten | 7 |
| `length`-Variabe von `irrA` | 1 |
| Summe | 183 |

Eine mehrdimensionale Reihung ist so organisiert, dass sie praktisch nur ihre elementaren Komponenten enthält.

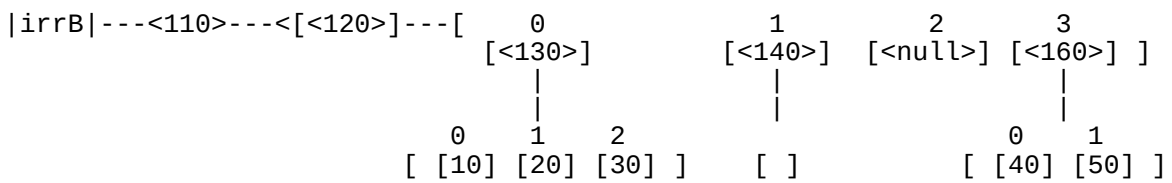
Mehrstufige Reihungen sind für den Programmierer bequemer, brauchen aber etwas mehr Speicherplatz und Bearbeitungszeit als mehrdimensioale Reihungen.

Anfangs gab es in Java keine mehrdimensionalen Reihungen (nur mehrstufige Reihungen). Dann hat IBM in Java mehrdimensionale Reihungen für Java programmiert. Ein entsprechendes Paket (ca. 3 MB) kann man sich (kostenlos) aus dem Internet herunterladen.

Aufgabe: Stellen Sie die folgende Reihung als Boje dar:

```
3    int[][] irrB = {{10, 20, 30}, {}, null, {40, 50}};
```

Lösung (als ASCII-Boje):



Wiederholungsfragen, 10. SU, Di 11.05.10

1. Stellen Sie die folgende Reihung zweimal als Boje dar, einmal in *ausführlicher* Darstellung und einemal in *vereinfachter* Darstellung:

```
1    int[] irX = {100, 200, 300};
```

2. Welche Operationen kann der Programmierer (in Java) auf auf das Wert-Kästchen einer Referenzvariablen anwenden?

3. Geben Sie zwei Operationen an, die Programmierer (in Java) *nicht* auf das Wert-Kästchen einer Referenzvariablen anwenden kann.

4. Aus welchem Material sind Reihungen? Aus Holz? Aus Glas? Aus Beton? Aus Stahl? Aus Sand?

5. Betrachten Sie die folgende Vereinbarung einer Reihungsvariablen:

```
2    String[][][][][] sigurd = new String[2][5][3][2][3];
```

5.1. Von welchem Typ ist die Variable `sigurd`?

5.2. Von welchem Typ sind die Komponenten von `sigurd`?

5.3. Von welchem Typ sind die elementaren Komponenten von `sigurd`?

5.4. Welche Stufigkeit hat die Reihung `sigurd`? (Geben Sie eine Zahl wie 2 oder 17 etc. an)

5.5. Welche Stufigkeit haben die Komponenten von `sigurd`?

5.6. Wie lang ist `sigurd`?

5.7. Wie viele elementare Komponenten enthält `sigurd`?

Rückseite der Wiederholungsfragen, 10. SU, Di 11.05.10

Ein Programm, in dem Reihungen (zum Bildschirm) ausgegeben werden:

```
 1 // Datei ReihungenToString01.java
 2 /* ------------------------------------------------------------------------
 3 Demonstriert die Methoden Arrays.toString und Arrays.deepToString
 4 ------------------------------------------------------------------------ */
 5 import java.util.Arrays;
 6
 7 class ReihungenToString01 {
 8    static public void main(String[] _) {
 9       pln("ReihungenToString01: Jetzt geht es los!");
10       pln("-----------------------------------");
11
12       int[]   r11 = {5, 2, 7};       // 1-stufig
13       int[]   r12 = {5, 2, 7};       // 1-stufig
14       int[][] r21 = {r11, r11, r11}; // 2-stufig
15
16                                      // Ausgabe:
17       pln(r11);                      // [I@3e25a5
18       pln(r12);                      // [I@19821f
19       pln(Arrays.    toString(r11)); // [5, 2, 7]
20       pln(Arrays.    toString(r11)); // [5, 2, 7]
21       pln(Arrays.    toString(r21)); // [[I@3e25a5, [I@3e25a5, [I@3e25a5]
22       pln(Arrays.deepToString(r21)); // [[5, 2, 7], [5, 2, 7], [5, 2, 7]]
23
24       pln("-----------------------------------");
25       pln("ReihungenToString01: Das war's erstmal!");
26    } // main
27    // ------------------------------------------------------------------------
28    // Eine Methode mit einem kurzen Namen:
29    static void pln(Object ob) {System.out.println(ob);}
30 } // class ReihungenToString01
31 /* ------------------------------------------------------------------------
32 Ausgabe des Programms:
33
34 ReihungenToString01: Jetzt geht es los!
35 -----------------------------------
36 [I@3e25a5
37 [I@19821f
38 [5, 2, 7]
39 [5, 2, 7]
40 [[I@3e25a5, [I@3e25a5, [I@3e25a5]
41 [[5, 2, 7], [5, 2, 7], [5, 2, 7]]
42 -----------------------------------
43 ReihungenToString01: Das war's erstmal!
44 ------------------------------------------------------------------------ */
```

Der `import`-Befehl (in Zeile 5) importiert nichts, sondern legt fest, dass der Name `Arrays` eine Abkürzung für den vollen Namen `java.util.Arrays` sein soll.

Wenn man eine 1-stufige Reihung wie in Zeile 17 oder 18 ausgibt, ist das Ergebnis wohl eher enttäuschend. Die komplizierteren Befehle in den Zeile 19 oder 20 produzieren das, was man möchte.

Wenn man eine 2-stufige Reihung wie in Zeile 21 ausgibt, ist das Ergebnis wohl eher enttäuschend. Der kompliziertere Befehl in Zeile 22 produziert das, was man möchte.

Die Zeichenkette `[I` ist ein Code für den Typnamen *Reihung von int*.
Die Zeichenkette `[[I` ist ein Code für den Typnamen *Reihung von Reihung von int*.
In den Ausgaben steht nach dem @-Zeichen ein sogenannter *Hash-Code*.
Man beachte, dass die Hash-Codes der Reihungen `r11` und `r12` völlig verschieden sind, obwohl die Reihungen sich sehr ähneln.

Antworten zu den Wiederholungsfragen, 10. SU, Di 11.05.10

1. Stellen Sie die folgende Reihung zweimal als Boje dar, einmal in *ausführlicher* Darstellung und einemal in *vereinfachter* Darstellung:

```
1    int[] irX = {100, 200, 300};
```

```
                          0            1            2
|irX|--<10>--[<20>]--[ <30>--[100]  <40>--[200]  <50>--[300] |length|--<60>--[3] ]

                          0     1      2
|irX|--<10>--[<20>]--[ [100] [200]  [300] ]
```

2. Welche Operationen kann der Programmierer (in Java) auf auf das Wert-Kästchen einer Referenzvariablen anwenden?

Zuweisung, Vergleich (auf gleich/ungleich)

3. Geben Sie zwei Operationen an, die Programmierer (in Java) *nicht* auf das Wert-Kästchen einer Referenzvariablen anwenden kann.

Addition, Multiplikation, ..., pln, ...

4. Aus welchem Material sind Reihungen? Aus Holz? Aus Glas? Aus Beton? Aus Stahl? Aus Sand?

Aus Beton!

5. Betrachten Sie die folgende Vereinbarung einer Reihungsvariablen:

```
2    String[][][][][] sigurd = new String[2][5][3][2][3];
```

5.1. Von welchem Typ ist die Variable `sigurd`?

Vom Typ Reihung von Reihungen von Reihungen von Reihungen von Reihungen von String.

5.2. Von welchem Typ sind die Komponenten von `sigurd`?

Vom Typ Reihung von Reihungen von Reihungen von Reihungen von String.

5.3. Von welchem Typ sind die elementaren Komponenten von `sigurd`?

Vom Typ String.

5.4. Welche Stufigkeit hat die Reihung `sigurd`? (Geben Sie eine natürliche Zahl wie 2 oder 17 etc. an)

Stufigkeit 5 (oder: sigurd ist eine 5-stufige Reihung)

5.5. Welche Stufigkeit haben die Komponenten von `sigurd`?

Stufigkeit 4 (oder: Die Komponenten von sigurd sind 4-stufige Reihungen)

5.6. Wie lang ist `sigurd`?

sigurd hat die Länge 2.

5.7. Wie viele elementare Komponenten enthält `sigurd`?

sigurd enthält (2 mal 5 mal 3 mal 2 mal 3 gleich) 180 elementare Komponenten

10. SU Di 11.05.10

A. Wiederholung
B. Organisation

Eine Reihung in einen String umwandeln und ausgeben etc.

Die Rückseite des Blattes mit den Wiederholungsfragen besprechen.

7.7 Die Klasse Arrays (S. 172)

Diese Klasse (dieser Modul) enthält noch weitere interessante Methoden
zur Bearbeitung von Reihungen.

Verwechseln Sie die Klasse `Arrays` möglichst selten mit der Klasse `Array` (ohne s)!

Was bedeutet oder bezeichnet der Name einer Rerferenzvariablen?

Angenommen, wir haben folgende Referenzvariable:

```
1 StringBuilder stb = new StringBuilder("ABC");
```

Die vier Teile der Variablen `stb` als Boje dargestellt:

```
 |stb|      Name
   |
 <100>      Referenz
   |
[<110>]     Wert
   |
["ABC"]     Zielwert
```

In jedem der folgenden Befehle kommt der Variablenname `stb` vor:

```
1 if (stb == ...) ...
2 stb.append("ZZ");
3 stb = ... ;
4 pln(stb);
```

Welchen Teil der Variablen bezeichnet der Name `stb` in den einzelnen Befehlen?

In Zeile 1: Den *Wert* `[<110>]`, denn der wird durch den Operator `==` verglichen.
In Zeile 2: Den *Zielwert* `["ABC"]`. Dieser Zielwert ist ein Objekt des Typs `StringBuilder`, d.h. ein *Modul*, in dem sich u.a. eine Methode namens `append` (mit einem `String`-Parameter) befindet. Diese Methode `append` im Zielwert-Modul von `sb` wird aufgerufen.
In Zeile 3: Den *Wert* `[<110>]`, denn dieser Wert wird durch einen anderen Wert ersetzt.
In Zeile 4: Ein Teil des *Zielwertes* d.h. Moduls `["ABC"]`, falls ein Zielwert vorhanden ist, sonst der *Wert* der Variablen `sb` (der dann gleich `null` ist).

Man muss also immer aus dem Zusammenhang erkennen, ob mit dem Namen einer Referenzvariablen gerade der *Wert* oder der *Zielwert* (bzw. welcher Teil des Zielwertes) gemeint ist.

6. Ausdrücke

Man unterscheidet (in Java und anderen Sprachen)
2 Arten von Ausdrücken: *einfache* und *zusammengesetzte*
(*simple* and *compound expressions*).

Haben Sie einen ganz ähnlichen Satz schon mal gehört oder gelesen?

6.1. Einfache Ausdrücke

Jeder Variablenname und jedes Literal ist ein (einfacher) Ausdruck.

S. 133, Beispiel-01: Typische Literale verschiedener Typen

S. 133, Beispiel-02: `int`-Literale

Def.: Ein Literal ist ein *Name für einen bestimmten Wert.*

Ein Java-Programmierer kann im Prinzip von jedem Literal wissen,
zu wlechem Typ es gehört und welchen Wert es bezeichnet!

Die Literale `10`, `0xA`, `0Xa` und `012` bezeichnen alle den `int`-Wert zehn.

Das Literal `0.1` bezeichnet einen `double`-Wert, der ein bisschen größer ist als ein Zehntel (diesen Wert braucht man nur "im Prinzip" zu kennen, ohne seine exakte Darstellung auswendig zu wissen).

Das Literal `0.1F` bezeichnet einen `float`-Wert, der ein bisschen größer ist als ein Zehntel und ein bisschen größer als der Wert des `double`-Literals `0.1`.

Ein Variablenname ist auch eine Art *Name für einen Wert* (für den Wert, den die Variable gerade enthält). Aber so ein Name sagt einem nicht, welcher Wert gemeint ist: Eine `int`-Variable namens `drei` muss nicht unbedingt den Wert `3`, sondern kann ohne weiteres einen anderen Wert enthalten.

Eine `double`-Variable namens `pi` muss nicht den Wert der berühmten Kreiszahl enthalten, sondern kann auch den Wert `5.0` oder `-7.5` etc. enthalten.

Dagegen bezeichnet das Literal `17` immer den `int`-Wert siebzehn (sonst dürfen Sie ihren Java-Ausführer umtauschen und einen neuen verlangen :-).

Literale sind *syntaktische* Größen (d.h. sie können in einem Quellprogramm vorkommen).

Werte sind *semantische* Größen (d.h. sie können in Quellprogrammen nur *benannt* werden, dort aber selbst *nicht vorkommen*). Sie werden erst bei der Ausführung eines Programms vom Ausführer berechnet, miteinander verglichen, umgewandelt, in Variablen gespeichert etc.

Vergleich: Menschen können in Romanen nur *benannt* werden, dort aber selbst (als Wesen aus Fleisch und Blut) *nicht vorkommen.*

## 6.2 Zusammengesetzte Ausdrücke

Sie bestehen aus Ausdrücken, Operatoren wie `+`, `-`, `*`, `/`, `%`, `<`, `<=`, `&&`, `||`, `!` ... etc. und runden Klammern.

Beispiele: Aus welchen Teilen bestehen die Ausdrücke in der nachfolgenden Tabelle?

```
// Als "Rohmaterial" vereinbaren wir ein paar Variablen:
int     otto, emil, anna, bert, carl, dora;
boolean fany, gerd, heinz;
```

| Ausdruck | Teil 1 | Teil 2 | Teil 3 |
|---|---|---|---|
| `otto + emil` | `otto` | `+` | `emil` |
| `anna + bert + carl + dora` | `anna + bert + carl` | `+` | `dora` |
| `anna + bert * carl` | `anna` | `+` | `bert * carl` |
| `anna * bert + carl` | `anna * bert` | `+` | `carl` |
| `-otto` | `-` | `otto` | |
| `fany \|\| gerd && heinz` | `fany` | `\|\|` | `gerd && heinz` |
| `fany && gerd \|\| heinz` | `fanny && gerd` | `\|\|` | `heinz` |
| `!heinz` | `!` | `heinz` | |

Zur Entspannung: Christian Morgenstern (1871-1914)

Tertius Gaudens

Da freut sich der Dritte (eigentlich das 4. Perlhuhn nach 3 Schweinen)

Vor vielen Jahren sozusagen, hat folgendes sich zugetragen:
Drei Säue taten um ein Huhn / in einem Korb zusammen ruhn.

Wiederholungsfragen, 11. SU, Mo 17.05.10

1. Stellen Sie die folgende Variable als Boje dar:

```
1    StringBuilder sb = new StringBuilder("Sonja");
```

2. In jedem der folgenden Befehle kommt der Name `sb` vor. Geben Sie für jeden Befehl an, welchen Teil der Variablen `sb` der Name in diesem Zusammenhang bezeichnet:

```
2    if (sb == null) pln("Hallo!");    //
3    pln(sb);                          //
4    sb.append(" Meier");              //
5    sb = null;                        //
6    pln(sb);                          //
```

3. Geben Sie von jedem der folgenden Literale an, zu welchem Typ es gehört:

```
7    1234.5                            //
8    'A'                               //
9    999999999999999L                  //
10   5432.123F                         //
11   "ABC"                             //
12   '\n'                              //
13   ""                                //
```

4. Füllen Sie die folgende Tabelle aus indem Sie angeben, aus welchen Teilen der angegebene Ausdruck besteht:

| Ausdruck | Teil 1 | Teil 2 | Teil 3 |
|---|---|---|---|
| `a + b - c` | | | |
| `a / b * c` | | | |
| `a + b * c` | | | |
| `a - b - c` | | | |
| `-a` | | | |
| `d \|\| e \|\| f` | | | |
| `d \|\| e && f` | | | |

5. Was ist ein Modul?

6. Was ist ein Typ?

Rückseite Wiederholungsfragen, 11. SU, Mo 17.05.10

1. Eine Schleife mit einer `break`-Anweisung:

```
 1    int summeA = 0;
 2    while (true) {
 3       p("Eine Ganzzahl (0 zum Beenden)? ");
 4       int ein = EM.liesInt();
 5       if (ein == 0) break;
 6       summeA += ein;
 7    }
 8    pln("Summe: " + summeA);
```

2. Eine Schleife mit einer `continue`-Anweisung (und einer `break`-Anweisung):

```
 9    int summeB = 0;
10    while (true) {
11       p("Eine positive Ganzzahl (0 zum Beenden)? ");
12       int ein = EM.liesInt();
13       // Negative Zahlen werden einfach ignoriert:
14       if (ein <  0) continue;
15       if (ein == 0) break;
16       summeB += ein;
17    }
18    pln("Summe: " + summeB);
```

3. Eine `break`-Anweisung-*mit-Marke* (in einer geschachtelten Schleife in einer Funktion):

```
19    // Welches Ergebnis im kleinen Ein-Mal-Eins ist das erste, welches
20    // groesser als 50 ist? Die folgenden Befehle finden die Antwort
21    // (naemlich: 6 * 9 ist gleich 54).
22    int na, nb;
23    anna: for (na=1; na<=9; na++) {
24       bert: for (nb=1; nb<=9; nb++) {
25          if (na*nb > 50) break anna;
26       }
27    }
28    pln(na + " * " + nb " ist gleich " + (na*nb));
```

4. Eine `continue`-Anweisung-*mit-Marke* (in einer geschachtelten Schleife in einer Funktion):

```
29 static boolean enthaelt(int[] rA, int[] rB) {
30    // Liefert true genau dann wenn rB in rA enthalten ist, d.h.
31    // wenn jede Zahl, die in rB vorkommt, auch in rA vorkommt.
32
33    bert: for (int iB=0; iB<rB.length; iB++) {
34       int rbib = rB[iB];
35
36       // Kommt rbi in der Reihung rA vor?
37       anna: for (int iA=0; iA<rA.length; iA++) {
38          int raia = rA[iA];
39          if (rbib == raia) continue bert    ;
40       }
41       return false;
42    }
43    return true;
44 }
```

Weitere Beispiele für `break`- und `continue`-Anweisungen mit Marken findet man im Buch auf S. 87 (Beispiel-03 und -04) und S. 88 (Beispiel-05).

Antworten zu den Wiederholungsfragen, 11. SU, Mo 17.05.10

1. Stellen Sie die folgende Variable als Boje dar:

```
  1    StringBuilder sb = new StringBuilder("Sonja");
```

**|sb|--<110>--[<120>]--["Sonja]**

2. In jedem der folgenden Befehle kommt der Name sb vor. Geben Sie für jeden Befehl an, welchen Teil der Variablen sb der Name in diesem Zusammenhang bezeichnet:

```
  2    if (sb == null) pln("Hallo!");    // Wert (120)
  3    pln(sb);                          // Zielwert ("Sonja")
  4    sb.append(" Meier");              // Zielwert ("Sonja")
  5    sb = null;                        // Wert (120)
  6    pln(sb);                          // Wert (null)
```

3. Geben Sie von jedem der folgenden Literale an, zu welchem Typ es gehört:

```
  7    1234.5                            // double
  8    'A'                               // char
  9    999999999999999L                  // long
 10    5432.123F                         // float
 11    "ABC"                             // String
 12    '\n'                              // char
 13    ""                                // String
```

4. Füllen Sie die folgende Tabelle aus indem Sie angeben, aus welchen Teilen der angegebene Ausdruck besteht:

| Ausdruck | Teil 1 | Teil 2 | Teil 3 |
|---|---|---|---|
| a + b - c | a + b | - | c |
| a / b * c | a / b | * | c |
| a + b * c | a | + | b * c |
| a - b - c | a - b | - | c |
| -a | - | a | |
| d \|\| e \|\| f | d \|\| e | \|\| | f |
| d \|\| e && f | d | \|\| | e & f |

5. Was ist ein Modul?

Ein Behälter für Variablen, Methoden, Typen, Module etc., der aus mindestens 2 Teilen besteht, einem öffentlichen und einem privaten Teil. Von ausserhalb des Moduls kann man nur auf die Größen im öffentlichen Teil des Moduls zugreifen.

6. Was ist ein Typ?

Ein Bauplan für Variablen.

11. SU Mo 17.05.10

A. Wiederholung
B. Organisation

Die Tabelle aller Java-Operatoren (S. 143-144)

Wo steht in dieser Tabelle die Regel *Punktrechnung geht vor Strichrechnung*?

(*Punktrechnung*: multiplikative Operatoren, *Strichrechnung*: additive Operatoren,
siehe S. 143, 4. Zeile von oben).

Wo steht in dieser Tabelle, dass der Und-Operator && *stärker bindet* als der Oder-Operator || ?

(sieh S. 144, 5. und 6. Zeile von oben)

Wie viele Zuweisungsoperatoren gibt es? (12, einen einfachen und 11 zusammengesetzte)

Die Zuweisungsoperatoren als Abkürzungen:

| Abkürzung | bedeutet dasselbe wie: |
|---|---|
| `otto += 3;` | `otto = otto + 3;` |
| `otto -= 3;` | `otto = otto - 3;` |
| `otto *= 3;` | `otto = otto * 3;` |
| `otto /= 3;` | `otto = otto / 3;` |
| ... | ... |

Jeder Funktionsaufruf gilt als zusammengesetzter Ausdruck

S. 149, Beispiel-06

4.1.4 Die Anweisungungen break und continue (S. 51)

Wo (in einem Java-Programm) darf man die Anweisungen break und continue anwenden?

| Anweisung | Anwendungsorte |
|---|---|
| `break` | nur in Schleifen und in `switch`-Anweisungen |
| `continue` | nur in Schleifen |

Die Beispiele auf der Rückseite des Blattes mit den Wiederholungsfragen besprechen:

- Eine `while`-Schleife mit einer `break`-Anweisung (ohne Marke)
- Eine `while`-Schleife mit einer `continue`-Anweisung (ohne Marke)
- Eine geschachtelte `for`-Schleife mit einer `break`-Anweisung-*mit-Marke*
- Eine geschachtelte `for`-Schleife mit einer `continue`-Anweisung-*mit-Marke*

Verschiedene Beenden-Befehle:

| Befehl | Was wird beendet? |
|---|---|
| `continue` | Eine Ausführung eines Schleifenrumpfes |
| `break` | Eine Schleife oder `switch`-Anweisung |
| `return` | Eine Methode |
| `System.exit(0);` | Das ganze umgebende Programm |

Endlich: Die volle Wahrheit über Klassen (zumindest der Anfang davon :-)

Zur Wiederholung:

Was ist ein Modul? (Ein Behälter für Unterprogramme, Variablen, Typen, Module etc., der aus mindestens zwei Teilen besteht, einem privaten und einem öffentlichen Teil)

Was ist ein Typ? (Ein Bauplan für Variablen)

Def.: Eine Klasse ist ein *Modul* und ein *Bauplan* für Module.

Die Module, die nach einem solchen Bauplan gebaut wurden, werden als *Objekte* (oder: *Instanzen*) der Klasse bezeichnet.

S. 203, Beispiel-01: Die Klasse `Zaehler01` ist ein *Modul* und ein *Bauplan*

In Zeile 1 bis 22 steht *ein* einziger Befehl: Eine Klassen-Vereinbarung.
Auf Deutsch etwa: Erzeuge eine Klasse namens `Zaehler01`, die Folgendes enthält: ... .

Wie viele Vereinbarungen sind in der Klassen-Vereinbarung enthalten? (6 Stück)

Was für Dinge vereinbaren sie?

1. Eine Variable namens `anzahl`
2. Eine Methode namens `getAnzahl`
3. (Neu!) Ein Konstruktor (heisst wie die Klasse, sieht aus wie eine Methode ohne Ergebnistyp)
4. Eine Variable namens `punkte`
5. Eine Methode namens `add`
6. Eine Methode namens `getPunkte`

Allgemein: Innerhalb einer Klassenvereinbarung kann man *Elemente* und *Konstruktoren* vereinbaren (d.h. in Java zählen Konstruktoren *nicht* zu den Elementen einer Klasse).

Alle mit `static` gekennzeichneten Elemente und alle Konstruktoren
gehören zum *Modulaspekt* der Klasse.

Alle nicht mit `static` gekennzeichneten Elemente
gehören zum *Bauplanaspekt* der Klasse

Die Klasse `Zaehler01`:
Modulaspekt: `anzahl`, `getAnzahl`, 1 Konstruktor
Bauplanaspekt: `punkte`, `add`, `getPunkte`

S. 204: Bild 9.1 Der Modulaspekt der Klasse `Zaehler01`

Die Elemente `punkte`, `add` und `getPunkte` gehören *nicht* zum Modulaspket!

S. 205, Beispiel-02: Wie man die Klasse `Zaehler01` als Bauplan benutzen kann:

S. 205, Bild 9.2 Zwei Objekte der Klasse `Zaehler01`

Jedes enthält nur *die* Elemente, die zum *Bauplanaspekt* der Klasse gehören (`punkte`, `add`, `getPunkte`), aber nicht die, die zum Modulaspekt gehören (`anzahl`, `getAnzahl`, 1 Konstruktor).

Ausführung der *Variablen-Vereinbarung* in Zeile 26 (zelia) im Detail:

Schritt 1: Der `new`-Befehl erzeugt ein neues Objekt der Klasse `Zaehler01`
Schritt 2: Der Konstruktor `Zaehler01` wird aufgerufen (mit dem aktuellen Parameter `10`)
Schritt 3: Der `new`-Befehl ist fertig und liefert eine Referenz auf das neue Objekt, z.B. `<50>`
Schritt 4: Der Ausführer erzeugt eine Referenz-Variable mit dem Wert `<50>`

Aufgabe: Führen Sie die Vereinbarung der Variablen `zelma` (S. 205, Zeile 27) aus und stellen Sie die Variable als Boje dar.

Zur Entspannung: Niklaus Wirth (geb. 1934 in Winterthur, Schweiz)
Diplom 1959 an der ETH Zürich, M.Sc. 1960 an der Laval University, Canada und 1963 Ph.D. an der UCL Berkley. Wirth erhielt 1984 den Turing-Award. Wichtige Programmiersprachen:

| | | | |
|---|---|---|---|
| 1955 | Fortran, Cobol | Erste höhere Programmiersprachen | |
| 1960 | Algol60 | | |
| 1965 | Algol-W | Besseres Algol | (von Wirth) |
| 1968 | Algol68 | 2-Stufen-Grammatiken., Bojen | |
| 1970 | C | Maschinennahe höhere Sprache | |
| 1972 | Pascal | Strukturierte Programmierung | (von Wirth) |
| 1975 | Modula | Module | (von Wirth) |
| 1980 | Modula-2 | Module, Nebenläufigkeit | (von Wirth) |
| 1980 | Ada | Module, Nebenläufigkeit, Schablonen | |
| 1984 | C++ | C mit Klassen, Schablonen | |
| 1990 | Oberon | Klassen | (von Wirth) |
| 1994 | Java | Maschinenunabhängigkeit | |
| 1995 | Ada | Klassen | |

Sehr gut: Er hat immer wieder von vorn angefangen.
Schlecht: Er hat immer wieder von vorn angefangen.

Wiederholungsfragen, 12. SU, Di 18.05.10

Der Ausdruck `a * b + c / d + e` ohne Klammern ist äquivalent zu dem voll geklammerten Ausdruck `(((a * b) + (c / d)) + e)`.
Geben Sie für jeden der folgenden Ausdrücke ohne Klammern den äquivalenten voll geklammerten Ausdruck an. Es ist ausdrücklich *nicht* verboten, dabei das Buch auf S. 143-144 aufzuschlagen.

1. `q % r == s`
2. `a >>> b + c`
3. `a << b < c << d`
4. `e || f && g || h`
5. `m | n ^ o & p`

Betrachten Sie die folgende Klassenvereinbarung:

```
1 class Carl {
2    static String s1 = "Hallo!";
3    String s2;
4    String machWas() {pln("Hallo!");}
5    Carl() {s2 = "Hallo!";}
6    static String nochWas() {pln("Hello!");}
7    Carl(String s) {s2 = s;}
8 }
```

6. Wie viele *Konstruktoren* werden in dieser Klassenvereinbarung vereinbart?
7. Wie viele *Elemente* werden in dieser Klassenvereinbarung vereinbart?
8. Welche Elemente gehören zum *Modulaspekt* der Klasse? Geben Sie nur die Namen dieser Elemente an.
9. Welche Elemente gehören zum *Bauplanaspekt* der Klasse? Geben Sie nur die Namen dieser Elemente an.

Rückseite der Wiederholungsfragen, 12. SU, Di 18.05.10

Was passiert im Einzelnen, wenn der Ausführer die folgende Variablen-Vereinbarung ausführt?

```
Zaehler01 zorro = new Zaehler01(8);
```

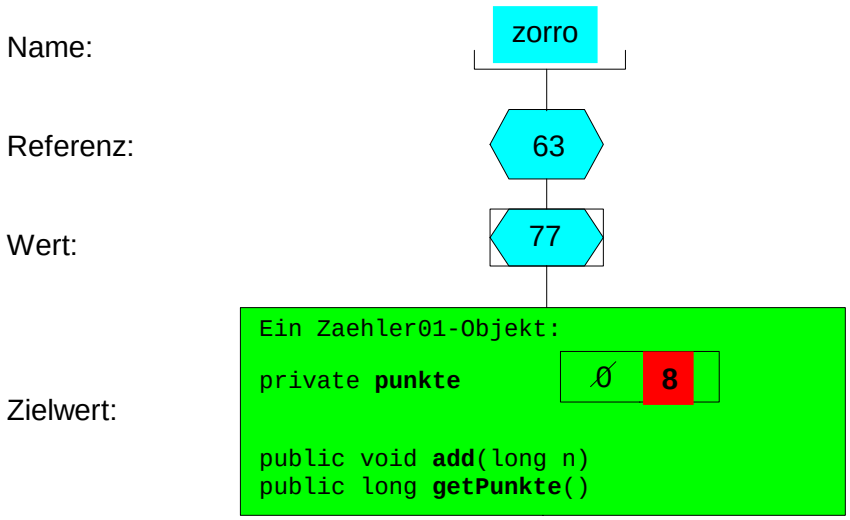
Der Ausführer führt die folgenden 4 Schritte aus:

1. Aufgrund des `new`-Befehls erzeugt er ein neues `Zaehler01`-Objekt. Darin ist die Variable `punkte` *standardmäßig* initialisiert (d.h. sie hat den Wert 0).

2. Dann führt er den Konstruktor-Aufruf `Zaehler01(8)` aus. Dadurch wird der Wert der `punkte`-Variable im neuen `Zaehler01`-Objekt (von 0) zu 8 verändert.

3. Damit ist der Befehl `new Zaehler01(8)` fertig ausgeführt und *liefert* als Ergebnis eine Referenz, die auf das neu erzeugte Objekt zeigt, z.B. die Referenz **<77>**.

4. Dann erzeugt der Ausführer eine Referenz-Variable mit dem *Namen* `zorro` und dem *Wert* [<77>]. Diese Variable zeigt mit ihrem Wert auf das neu erzeugte Objekt.



Antworten zu den Wiederholungsfragen, 12. SU, Di 18.05.10

Der Ausdruck `a * b + c / d + e` ohne Klammern ist äquivalent zu dem
voll geklammerten Ausdruck `(((a * b) + (c / d)) + e)`.
Geben Sie für jeden der folgenden Ausdrücke ohne Klammern den äquivalenten voll geklammerten Ausdruck an. Es ist ausdrücklich *nicht* verboten, dabei das Buch auf S. 143-144 aufzuschlagen.

1. `q % r == s` **`((q % r) == s)`**
2. `a >>> b + c` **`(a >>> (b + c))`**
3. `a << b < c << d` **`((a << b) < (c << d))`**
4. `e || f && g || h` **`((e || (f && g)) || h)`**
5. `m | n ^ o & p` **`(m | (n ^ (o & p)))`**

Betrachten Sie die folgende Klassenvereinbarung:

```
1 class Carl {
2    static String s1 = "Hallo!";
3    String s2;
4    String machWas() {pln("Hallo!");}
5    Carl() {s2 = "Hallo!";}
6    static String nochWas() {pln("Hello!");}
7    Carl(String s) {s2 = s;}
8 }
```

6. Wie viele *Konstruktoren* werden in dieser Klassenvereinbarung vereinbart?

Zwei (in den Zeilen 5 und 7)

7. Wie viele *Elemente* werden in dieser Klassenvereinbarung vereinbart?

Vier (in den Zeilen 2, 3, 4 und 6)

8. Welche Elemente gehören zum *Modulaspekt* der Klasse? Geben Sie nur die Namen dieser Elemente an.

Die Elemente s1 und nochWas.

9. Welche Elemente gehören zum *Bauplanaspekt* der Klasse? Geben Sie nur die Namen dieser Elemente an.

Die Elemente s2 und machWas.

12. SU Di 18.05.10

A. Wiederholung
B. Organisation

Was bewirkt der new Befehl im Einzelnen?

Die Rückseite der Wiederholungsfragen genau besprechen.

Die Elemente einer Klasse nach verschiedenen Kriterien in Gruppen einteilen (S. 216)

| Einteilungskriterium | Gruppen<br>die sich bei diesem Einteilungskriterium ergeben | Anzahl Gruppen |
|---|---|---|
| nach Art | Attribut (d.h. Variable), Methode, Klasse, Schnittstelle | 4 |
| nach Aspektzugehörigkeit | Klassenelemente (gehören zum Modulaspket)<br>Objektelemente (gehören zum Bauplanaspekt) | 2 |
| nach Erreichbarkeit | öffentliche (`public`), geschützte (`protected`),<br>paketweit errreichbare, private (`private`) Elemente. | 4 |

Achtung: Wenn man vor die Vereinbarung eines Elements `protected` schreibt,
dann ist es "weniger geschützt" als wenn man *keinen* Erreichbarkeitsmodifizierer hinschreibt!

Zur Entspannung: G-vorn oder K-vorn (big endian or little endian)

Jonathan Swift (1667-1745, Irland, damals unter englischer Herrschaft) veröffentlichte 1726 einen Roman mit dem Titel "Travels Into Several Remote Nations of The World", im Wesentlichen eine Satire gegen die politischen Verhältnisse in Irland und England. Der Roman erschien unter dem Titel "Gullivers Reisen" auch in Deutschland und wurde lange als Kinderbuch (miss-) verstanden.

Damals waren die wichtigsten politischen Parteien die Tories und die Whigs. In seinem Roman beschrieb Swift zwei Parteien ("in einem fernen Land"), die sich nur dadurch unterschieden, auf welcher Seite sie ihre Frühstückseier aufschlugen: Die Big Endians an der stumpferen Seite und die Little Endians an der spitzeren Seite.

Allgemein kann man bei vielen Daten zwei Formen unterscheiden: G-vorn (das Große vorn, big endian) und K-vorn (das Kleine vorn, little endian). Beispiele:

| | |
|---|---|
| Ein Datum 17.12.2005 | K-vorn |
| Neues Datum 2005.12.17 | G-vorn |
| Ein Pfadname `d:\bsp\java\Hallo.java` | G-vorn |
| Eine Netzadresse `bht-berlin.de` | K-vorn |
| Eine arabische Zahl im Deutschen 12345 | G-vorn |
| Eine arabische Zahl im Arabischen 12345 | K-vorn |

Man unterscheidet auch zwischen K-vorn Prozessoren (little endian processors) und G-vorn-Prozessoren (big endian p.), die Zahlen mit den niedrigwertigen (bzw. hochwertigen) Ziffern vorn darstellen. Keine der beiden Formen ist prinzipiell überlegen, beide haben Vor- und Nachteile.

| | |
|---|---|
| x86-Prozessoren von AMD oder Intel sind | K-vorn (little endian) |
| m68-Prozessoren von Motorola | G-vorn (big endian) |
| PowerPCs von IBM sind | G-vorn (big endian) |

Wiederholungsfragen, 13. SU, 25.05.10

Zur Erinnerung: Innerhalb einer Klassenvereinbarung kann man *Konstruktoren* und *Elemente* vereinbaren. Die Elemente kann man nach verschiedenen *Kriterien* in *Gruppen* einteilen. Auf diese Kriterien und Gruppen beziehen sich die Fragen 1. bis 3.

1. Wenn man die Elemente nach ihrer *Art* in Gruppen einteilt,
wie viele Gruppen erhält man dann und wie heißen diese Gruppen?

2. Wenn man die Elemente nach ihrer *Aspektzugehörigkeit* in Gruppen einteilt,
wie viele Gruppen erhält man dann und wie heißen diese Gruppen?

3. Wenn man die Elemente nach ihrer *Erreichbarkeit* in Gruppen einteilt,
wie viele Gruppen erhält man dann und wie heißen diese Gruppen?

Betrachten Sie die folgende Klassenvereinbarung:

```
 1 class Carola {
 2    static int anz;
 3    int zahl1;
 4    int zahl2;
 5
 6    static public void main(String[] sonja) {
 7       System.out.println("Hallo");
 8       Carola c1 = new Carola();
 9       Carola c2 = new Carola();
10       Carola c3 = new Carola();
11       ...
12    }
13 }
```

4. Angenommen, der Ausführer hat das Programm `Carola` *bis zur Zeile 7* (einschließlich) ausgeführt. Wie viele int-Variablen existieren in dem Moment und wie heißen sie?

5. Ebenso, aber mit "*bis zur Zeile 8*".

6. Ebenso, aber mit "*bis zur Zeile 9*".

7. Ebenso, aber mit "*bis zur Zeile 10*".

8. Zur Erinnerung: Die Klasse `Zaehler01` wird auf S. 203 des Buches vereinbart.
Befehlen Sie dem Java-Ausführer 5800 Objekte der Klasse `Zaehler01` zu erzeugen. Jedes dieser Objekte soll ein Konto mit 3 Punkten repräsentieren.

9. Befehlen Sie dem Java-Ausführer, jedes der 5800 Punkte-Konten um 5 zu erhöhen.

Antworten zu den Wiederholungsfragen, 13. SU, 25.05.10

Zur Erinnerung: Innerhalb einer Klassenvereinbarung kann man *Konstruktoren* und *Elemente* vereinbaren. Die Elemente kann man nach verschiedenen *Kriterien* in *Gruppen* einteilen. Auf diese Kriterien und Gruppen beziehen sich die Fragen 1. bis 3.

1. Wenn man die Elemente nach ihrer *Art* in Gruppen einteilt,
wie viele Gruppen erhält man dann und wie heißen diese Gruppen?

4 Gruppen: Attribute, Methoden, Klassen, Schnittstellen

2. Wenn man die Elemente nach ihrer *Aspektzugehörigkeit* in Gruppen einteilt,
wie viele Gruppen erhält man dann und wie heißen diese Gruppen?

2 Gruppen: Modulaspekt, Bauplanaspekt

3. Wenn man die Elemente nach ihrer *Erreichbarkeit* in Gruppen einteilt,
wie viele Gruppen erhält man dann und wie heißen diese Gruppen?

4 Gruppen: Öffentlich, geschützt, paketweit erreichbar, privat.

Betrachten Sie die folgende Klassenvereinbarung:

```
 1 class Carola {
 2    static int anz;
 3    int zahl1;
 4    int zahl2;
 5
 6    static public void main(String[] sonja) {
 7       System.out.println("Hallo");
 8       Carola c1 = new Carola();
 9       Carola c2 = new Carola();
10       Carola c3 = new Carola();
11       ...
12    }
13 }
```

4. Angenommen, der Ausführer hat das Programm `Carola` *bis zur Zeile 7* ausgeführt. Wie viele int-Variablen existieren in dem Moment und wie heißen sie?

Eine: Carola.anz

5. Ebenso, aber mit "*bis zur Zeile 8*".

Drei: Carola.anz, c1.zahl1, c1.zahl2

6. Ebenso, aber mit "*bis zur Zeile 9*".

Fünf: Alle aus 5. und c2.zahl1, c2.zahl2

7. Ebenso, aber mit "*bis zur Zeile 10*".

Sieben: Alle aus 6. und c3.zahl1, c3.zahl2

8. Zur Erinnerung: Die Klasse `Zaehler01` wird auf S. 203 des Buches vereinbart.
Befehlen Sie dem Java-Ausführer 5800 Objekte der Klasse `Zaehler01` zu erzeugen. Jedes dieser Objekte soll ein Konto mit 3 Punkten repräsentieren.

```
14    Zaehler01[] otto = new Zaehler01[5800];
15    for (int i=0; i<otto.length; i++) otto[i] = new Zaehler01(3);
```

9. Befehlen Sie dem Java-Ausführer, jedes der 5800 Punkte-Konten um 5 zu erhöhen.

```
16    for (Zaehler01 z : otto) z.add(5);
```

13. SU Di 25.05.10 ("Pfingstdienstag")

A. Wiederholung
B. Organisation: Tests: Es gilt weiterhin: Gruppe 1a schreibt jeweils mittwochs, 1b freitags.

Achtung: Am Mi 09.06.10 schreibt die Gruppe 1a *zwei* Tests: Test08 und Test09!

S. 218, Beispiel-01:

Was für ein Element ist `n1`? (Eine öffentliches Klassenattribut)
`n2`? (Ein privates Klassenattribut), `inkrementA` (öffentliche Klassenmethode) , ...
`m1`? (öffentliches Objektattribut), ... `dekerementB`? (pvivate Objektmethode).

10 Ein paar Standardklassen und Methoden (S. 221)

Die Java Standardbibliothek enthielt gestern 3777 Klassen und Schnittstellen (heute sind es wahrscheinlich schon wieder ein paar mehr :-)

Die Standardbibliothek ist (im Vergleich zu anderen Bibliotheken) *sehr gut dokumentiert.* Ein Java-Programmierer muss mit der Dokumentation der Standardbibliothek bestens vertraut sein und schnell darin nach bestimmten Informationen suchen können (etwa so wie ein Schaffner in einem Kursbuch). Das üben wir

10.3 Die Klasse StringBuilder (S. 233)

Die Kapazität (capacity) und die Länge (size) eines StringBuilder-Objekts.

12 Klassen erweitern (beerben)

Motivation: Angenommen, wir haben eine alte, bewährte Klasse, brauchen jetzt aber eine Variante davon (d.h. eine Klasse, die der alten sehr ähnelt, aber ein bisschen anders ist).

Warum sollte man die alte Klasse nicht verändern? Warum sollte man keine Kopie der alten Klasse anfertigen, umbenennen und verändern? Wie sollte man vorgehen?

S. 278, Beispiel-01: Die Klasse `Person01`

S. 278, Bild 12.1 Der Modulaspekt der Klasse `Person01`

S. 279, Bild 12.2 Ein `Person01`-Objekt

S. 280, Beispiel02: Die Klasse `Person02`, eine Erweiterung von `Person01`

S. 281, Bild 12.3 Der Modulaspekt der Klasse `Person02`

S. 281 Bild 12.4 Ein `Person02`-Objekt

Die volle Wahrheit: Jede Klasse erweitert (oder: beerbt) *genau eine* andere Klasse.

Einzige Ausnahme: Die Klasse `Object` (erweitert *keine* andere Klasse).

Die Klasse `Person01` erweitert also die Klasse `Object`.

S. 282, Bild 12.5 Der Modulaspekt der Klasse `Person02` (vollständig)

S. 283, Bild 13.6 Ein `Person02`-Objekt vollständig

12.2 Allgemeine Regeln zum Beerben und Vererben (S. 283)

Zur Entspannung: Wie lernt man? Ein simples Modell.

Angenommen, Sie sollen 1024 kleine Einzelteile ("Perlen der Länge 1 Millimeter") zu einer ungefähr einen Meter langen Kette zusammenfügen. Dann können Sie folgendermaßen vorgehen:

1. Sie verbinden je zwei Einzelteile zu einem 2-er-Teil. Das sind 500 Arbeitsschritte. Sie sehen kaum einen Fortschritt, weil ein 2-er-Teil nicht viel länger aussieht als ein Einzelteil.

2. Sie verbinden je zwei 2-er-Teile zu einem 4-er-Teil. Das sind 250 Arbeitsschritte. Immer noch ist kaum ein Fortschritt zu sehen, denn die 4-er-Teile sind noch sehr kurz im Vergleich zum ein Meter langen Endergebnis.

3. Sie verbinden je zwei 4-er-Teile zu einem 8-er-Teil. 125 Arbeitsschritte.

...

9. Sie verbinden je zwei 256er-Teile zu einem 512er-Teil (2 Arbeitsschritte). Die Ergebnisse sind immer noch viel kürzer als das Endergebnis.

10. Sie verbinden zwei 512er-Teile zu einem 1024er-Teil. Das ist nur ein einziger Arbeitsschritt, aber der Fortschritt ist beeindruckend: Aus 50 Zentimeter langen Teilen wird ein 100 Zentimeter langes Ganzes.

Falls beim Lernen in unseren Gehirnen etwas (entfernt) Ähnliches abläuft, dann muss man sehr geduldig sein: Erst ganz am Ende eines komplizierten Lernvorgangs sieht man einen "großen Fortschritt". Der letzte Schritt eines Lernvorgangs ist manchmal ein so genanntes Aha-Erlebnis.

Wiederholungsfragen, 14. SU, Mo 31.05.10

Schema zum Beschreiben der Elemente einer Klasse:

| Erreichbarkeit | Aspekt | Art |
|---|---|---|
| öffentlich<br>geschützt<br>paketweit erreichbar<br>privat | Klassen-<br>Objekt- | Attribut<br>Methode<br>Klasse<br>Schnittstelle |

1. Betrachten Sie die folgende Klassen-Vereinbarung:

```
 1 class Charlotte {
 2    static private int e01;
 3    protected void e02(int n) {...}
 4    static public class e03 {...}
 5    interface e04 {...}
 6    static public String e05() {...}
 7    class e06 {...}
 8    static interface e07 {...}
 9    double e08(String s) {...}
10    double e09 = 1.5;
11    static private int e10() {...}
12 }
```

Beschreiben Sie die Elemente `e01` bis `e10` dieser Klasse entsprechend dem obigen Schema.

2. Angenommen, Sie vereinbaren eine Klasse `K1.`
Wie viele anderen Klassen kann Ihre Klasse `K1` *beerben*?
Your class `K1` can *inherit from* how many other classes?
Null? Eine? Beliebig viele?

3. An wie viele andere Klassen kann Ihre Klasse `K1` ihre Elemente *vererben*?
Your class `K1` can *bequeathe* its elements to how many other classes?
Null? Eine? Beliebig viele?

4. Wie heißt die (einzige) Klasse, die keine andere Klasse beerbt (oder: erweitert)?

Antworten zu den Wiederholungsfragen, 14. SU, Mo 31.05.10

Schema zum Beschreiben der Elemente einer Klasse:

| Erreichbarkeit | Aspekt | Art |
|---|---|---|
| öffentlich<br>geschützt<br>paketweit erreichbar<br>privat | Klassen-<br>Objekt- | Attribut<br>Methode<br>Klasse<br>Schnittstelle |

1. Betrachten Sie die folgende Klassen-Vereinbarung:

```
 1 class Charlotte {
 2    static private int e01;
 3    protected void e02(int n) {...}
 4    static public class e03 {...}
 5    interface e04 {...}
 6    static public String e05() {...}
 7    class e06 {...}
 8    static interface e07 {...}
 9    double e08(String s) {...}
10    double e09 = 1.5;
11    static private int e10() {...}
12 }
```

Beschreiben Sie die Elemente `e01` bis `e10` dieser Klasse entsprechend dem obigen Schema.

e01: Privates Klassen-Attribut
e02: Geschützte Objekt-Methode
e03: Öffentliche Klassen-Klasse (statische Klasse)
e04: Paketweit erreichbare Objekt-Schnittstelle (nicht-statische Schnittstelle)
e05: Öffentliche Klassen-Methode
e06: Paketweit erreichbare Objekt-Klasse (nicht-statische Klasse)
e07: Paketweit erreichbare Klassen-Schnittstelle (statische Schnittstelle)
e08: Paketweit erreichbare Objekt-Methode
e09: Paketweit erreichbares Objekt-Attribut
e10: Private Klassen Methode

2. Angenommen, Sie vereinbaren eine Klasse `K1`.
Wie viele andere Klassen kann Ihre Klasse `K1` *beerben*?
Your class `K1` can *inherit* from how many other classes?
Null? Eine! Beliebig viele?

3. An wie viele andere Klassen kann Ihre Klasse `K1` ihre Elemente *vererben*?
Your class `K1` can *bequeathe* its elements to how many other classes?
Null? Eine? Beliebig viele!

4. Wie heißt die (einzige) Klasse, die keine andere Klasse beerbt (oder: erweitert)?

java.lang.Object

14. SU Mo 31.05.10

A. Wiederholung
B. Organisation

### Ein Typgraf

Aus den Regeln für das Beerben von Klassen folgt, dass alle Klassen zusammen mit der Relation beerbt (oder: erweitert) einen *Baum* bilden.

S. 285, Bild 12.7 Ein Typgraf mit vielen Typen (und noch mehr Auslassungen).

Die Begriffe *direkte Oberklasse*, *direkte Unterklasse*, *Oberklasse* und *Unterklasse*.

Wichtige Regel: Wenn irgendwo in einem Java-Programm ein Objekt einer Klasse K benötigt wird (z.B. als Parameter einer Methode) kann man ("als Ersatz") auch ein Objekt einer Unterklasse von K angeben.

Konvention (für die Begriffe *Untertyp* und *Obertyp*):
Jeder Typ (z.B. `int` oder `String` oder `E01Punkt` etc.)
gilt als *Untertyp* und als *Obertyp* von sich selbst.

Der Typ `Exception` hat somit 3 Obertypen (`Exception`, `Throwable` und `Object`) und der Typ `Person01` hat 2 Untertypen (`Person01` und `Person02`).

### 12.3 Ein größeres Beispiel für Beerbung

S. 286, Bild 12.8 Ein Typgraf mit 6 Typen

### Die Klasse E01Punkt

Wie viele Konstruktoren und wie viele Elemente werden in dieser Klasse vereinbart?
(1 Konstruktor, 6 Elemente)

Wie viele Elemente werden in jedes Objekt dieser Klasse eingebaut?
(6 Elemente)

Wie viele Attribute und wie viele Methoden werden in jedes `E01Punkt`-Objekt eingebaut?
(2 Attribute und 4 Methoden)

```
1    E01Punkt paul = new Punkt(2.5, 1.7);
```

Wie sieht der String `paul.toString()` aus?
(`"Punkt: (2.5, 1.7)"`)

Ein kleines Benennungsproblem im Konstruktor: Der Name x bezeichnet einen Parameter des Konstruktors und ein Attribut der Klasse (und für y gilt das Gleiche).

Jedes Objekt enthält eine (Referenz-) Variable namens `this`, die auf dieses Objekt selbst zeigt.

### Die Klasse E01Rechteck

Wie viele Konstruktoren und Elemente werden in dieser Klasse vereinbart?
(1 Konstruktor, 7 Elemente)

Der Befehl `super` im Konstruktor (S. 288, Zeile 39) ruft einen Konstruktor der direkten Oberklasse (E01Punkt) auf.

Allgemein: Der erste Befehl in *jedem* Konstruktor ruft immer einen Konstruktor der direkte Oberklasse auf. Wenn der Programmierer keinen `super`-Befehl hinschreibt, baut der Ausführer den Befehl

```
2    super();
```

als ersten Befehl ein.

Wie viele Elemente werden in jedes Objekt dieser Klasse eingebaut?

(13, die 7 in `E01Rechteck` vereinbarten und die 6 von `E01Punkt` geerbten)

Die in `E01Rechteck` vereinbarte Methode `toString` überschreibt die geerbte Methode `toString`.

```
3    E01Rechteck rolf = new E01RechtEck(1.0, 2.0, 3.0, 4.0)
```

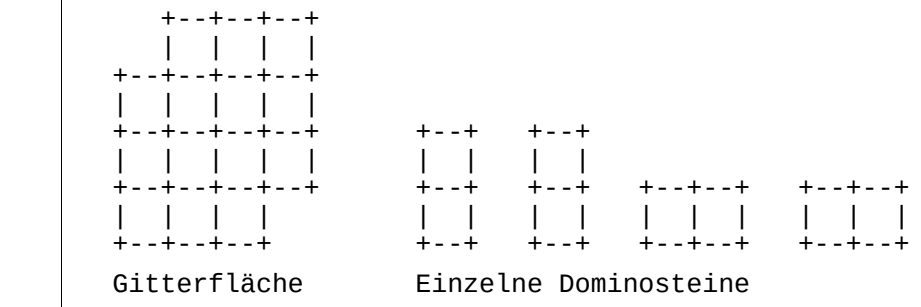
Wie sieht der String `rolf.toString()` aus?
`("Rechteck, Mittelpunkt: (1.0, 2.0), Seiten: 3.0|4.0")`

Aufgabe: Vereinbaren Sie eine Klasse namens `E01Quadrat` als Erweiterung der Klasse `E01Rechteck`.

Eine Lösung dieser Aufgabe findet man im Buch auf S. 290.

Zur Entspannung: Eine Gitterfläche mit Dominosteinen bedecken

Kann man die folgende Gitterfläche (4x4, mit 2 fehlenden Eckfeldern) mit Dominosteinen *genau bedecken* (d.h. jedes Feld der Gitterfläche muss bedeckt sein, aber kein Dominostein darf "überstehen"):




Kann man entsprechend ein 8x8-Schachbrett mit 2 (einander diagonal gegenüberliegenden) fehlenden Eckfeldern mit Dominosteinen genau bedecken ? Was ist mit einem 9x9-Gitter? Und 100x100?

Objekte als Zwiebeln darstellen

S. 291, Bild 12.9: Ein `E01Quadrat`-Objekt (als "Zwiebel dargestellt")

10. 4 Die Klasse ArrayList (in der Standard-Bibliothek) (S. 237)

Die Klasse `ArrayList` ist eine *Sammlungsklasse* und ihre Objekte sind *Sammlungen*. Das bedeutet Folgendes:

Sei `sam` ein `ArrayList`-Objekt. Dann kann man andere Objekte
- in das Objekt `sam` hineintun,
- darin suchen oder
- wieder entfernen,

kurz: im Objekt `sam` kann man (andere) Objekte sammeln.

Eine *Sammlung* hat Ähnlichkeit mit einer *Reihung*, bietet dem Programmierer aber mehr "Möglichkeiten und Komfort" (d.h. mehr Methoden).

Kurzvergleich Reihungen mit ArrayList-Sammlungen:

Reihungen sind aus Beton, aber man kann primitive Werte oder Objekte darin aufbewahren.

ArrayList-Sammlungen sind aus Gummi, aber man kann nur Objekte darin aufbewahren.

`ArrayList` ist keine "gewöhnliche Klasse" (wie `String`, `StringBuilder`, `Integer` etc.), sondern eine *generische Klasse*! Das bedeutet:

Eine gewöhnliche Klasse ist *ein* Typ.
Eine generische Klasse repräsentiert *viele* Typen.

Die Klasse `ArrayList` repräsentiert die Typen
`ArrayList<String>`,

```
ArrayList<Integer>,
...
ArrayList<ArrayList<String>>,
ArrayList<ArrayList<ArrayList<Integer>>>
...
```

In einer Sammlung des Typs `ArrayList<String>` kann man nur `String`-Objekte sammeln. Versucht man aus Versehen, z.B. ein Integer-Objekt in eine solche Sammlung einzufügen, merkt der Ausführer diesen Fehler und lehnt das Programm ab.

Allgemeines Prinzip:

Eine gute Programmiersprache enthält "starke Befehle" und "starke Einschränkungsmöglichkeiten".

Typen (und insbesondere parametrisierte Typen wie `ArrayList<String>` etc.) gehören zu den "starken Einschränkungsmöglichkeiten" der Sprache Java.

S. 238, Beispiel-01: `String`-Objekte in einem Sammlungsobjekt sammeln

Druckfehler in Zeile 10 (`"cccc"` statt `"ccccc"`)

Wiederholungsfragen, 15. SU, Di 01.06.10

Sehen Sie sich auf S. 285 des Buches noch einmal das Bild 12.7 an.

1. Wie heißt die direkte Oberklasse der Klasse `Exception`?
2. Wie viele Oberklassen hat die Klasse `Exception` und wie heißen diese Klassen?
3. Wie viele Unterklassen hat die Klasse `Person01` und wie heißen diese Klassen?
4. Wie viele direkte Unterklassen hat die Klasse `Throwable` und wie heißen diese Klassen?
5. Wie viele Unterklassen hat die Klasse `Object` (ungefähr)?

Die Vereinbarung der Klasse `E01Punkt` steht im Buch auf S. 287.

6. Betrachten Sie die folgenden Befehle:

```
1    E01Punkt peter = new E01Punkt(-3.0, 5.0);
2    peter.urSpiegeln();
3    pln(peter.toString());
```

Was gibt der `pln`-Befehl in Zeile 3 aus?

Die Vereinbarung der Klasse `E01Rechteck` steht im Buch auf S. 288.

7. Betrachten Sie die folgenden Befehle:

```
4    E01Rechteck rolf = new E01Rechteck(3.0, -4.0, 1.0, 1.0);
5    pln(rolf.urAbstand());
```

Was gibt der `pln`-Befehl in Zeile 5 aus?

Antworten zu den Wiederholungsfragen, 15. SU, Di 01.06.10

Sehen Sie sich auf S. 285 des Buches noch einmal das Bild 12.7 an.

1. Wie heißt die direkte Oberklasse der Klasse `Exception`?

Throwable

2. Wie viele Oberklassen hat die Klasse `Exception` und wie heißen diese Klassen?

Drei (Exception, Throwable und Object)

3. Wie viele Unterklassen hat die Klasse `Person01` und wie heißen diese Klassen?

Zwei (Person01 und Person02)

4. Wie viele direkte Unterklassen hat die Klasse `Throwable` und wie heißen diese Klassen?

Zwei (Error und Exception)

5. Wie viele Unterklassen hat die Klasse `Object` (ungefähr)?

Mehr als 3000.

Die Vereinbarung der Klasse `E01Punkt` steht im Buch auf S. 287.

6. Betrachten Sie die folgenden Befehle:

```
1    E01Punkt peter = new E01Punkt(-3.0, 5.0);
2    peter.urSpiegeln();
3    pln(peter.toString());
```

Was gibt der `pln`-Befehl in Zeile 3 aus?

Ausgabe: `Punkt: (3.0, -5.0)`

7. Betrachten Sie die folgenden Befehle:

```
4    E01Rechteck rolf = new E01Rechteck(3.0, -4.0, 1.0, 1.0);
5    pln(rolf.urAbstand());
```

Was gibt der `pln`-Befehl in Zeile 5 aus?

Ausgabe: `5.0`

15. SU Di 01.06.10

A. Wiederholung
B. Organisation

12.4 Wozu Untertypen gut sind

Angenommen, die 5 Typen `E01Punkt`, `E01Rechteck`, `E01Quadrat`, `E01Ellipse`, `E01Kreis` (siehe Typgrafen auf S. 286, Bild 12.8) wären nur "5 einzelne Typen",
aber keiner davon wäre ein *Untertyp* von einem anderen.

Sei weiter angenommen: In einem Programm wollen wir von jedem dieser 5 Typen je 100 Objekte erzeugen und bearbeiten. Dann müssten wir wohl 5 Reihungen vereinbaren:

```
1    E01Punkt   [] pr = new E01Punkt   [100];
2    E01Rechteck[] rr = new E01Rechteck[100];
3    E01Quadrat [] qr = new E01Quadrat [100];
4    ...
```

und die 500 Objekte in diesen 5 Reihungen erzeugen und bearbeiten.
Kurz: Wir müssten "alles 5 Mal machen".

Tatsächlich geht es in Java einfacher: Weil unsere 5 Typen alle Untertypen von `E01Punkt` sind, gilt:

Eine `E01Punkt`-Variable darf nicht nur auf ein `E01Punkt`-Objekt zeigen,
sondern auch auf ein `E01Rechteck`-Objekt, oder auf ein `E01Quadrat`-Objekt oder ...
(weil jedes `E01Rechteck` auch ein `E01Punkt`-Objekt ist und
das Gleiche für *alle* Untertypen von `E01Punkt` gilt).

Deshalb darf man in eine Reihung des Typs `E01Punkt[]` nicht nur `E01Punkt`-Objekt hineintun, sondern auch `E01Rechteck`-Objekte und `E01Quadrat`-Objekte und ... .

S. 295, Beispiel-01: Eine Reihung `tab`, die eine "wilde Mischung von Objekten" enthält

Druckfehler in den Zeilen 3 bis 8: Statt `pr` muss es immer `tab` heißen.

Wie man eine Reihung wie `tab` auf keinen Fall bearbeiten sollte:

S. 296, Aufgabe-01 und Beispiel-02

Die Methode `reihungAusgeben02` macht genau das, was sie machen soll,
sie ist aber viel zu kompliziert (und damit praktisch falsch).

S. 298, Beispiel-03: Die Methode `reihungAusgeben` macht das Gleiche wie `reihungAusgeben02`, ist aber auch praktisch richtig.

Zur Entspannung: Englische Vokabeln: boot, bootstrap

Ein *boot* ist ein hoher Schuh oder Schnürstiefel, ein *bootstrap* ein Schnürsenkel für einen Schnürstiefel. Ein *bootstrap* gilt als simples Werkzeug, welches immer zur Hand ist und mit dem man sich andere Werkzeuge "heranziehen kann", z. B. so: Eine Gruppe von Pfadfindern (natürlich alle in Schnürstiefeln) will ein schweres Stahlkabel über einer Felsspalte anbringen. Dazu knüpfen sie zuerst ihre *bootstraps* zusammen und ziehen damit ein dünnes Seil über die Felsspalte. Mit dem dünnen Seil ziehen sie ein dickeres Seil über die Felsspalte und mit dem dickeren Seil schliesslich das Stahlkabel.

Beim Booten eines Rechners liest der Prozessor zuerst von einem bestimmten Gerät einen einzigen Datenblock (z. B. 512 Byte) und springt dann zum ersten Byte dieses Blocks. Der Block sollte ein kleines Ladeprogramm, enthalten, welches ein größeres Ladeprogramm (z. B. ein paar Tausend Bytes) von einer bestimmten Platte liest und zum Anfang dieses Ladeprogramms springt. Das größere Ladeprogramm lädt dann wichtige Teile des Betriebssystems (ein paar Megabyte) und springt an eine Stelle in diesem Betriebssystem.

Wiederholungsfragen, 16 SU, Mo 07.06.10

Angenommen, wir haben die 5 Klassen `E01Punkt`, `E01Rechteck`, `E01Quader`, `E10Ellipse`, `E10Kreis` (die im Typgrafen auf S. 286 abgebildet sind).

1. Auf Objekte welcher dieser Typen darf eine Variable des Typs `E01Rechteck` zeigen?

2. Auf Objekte welcher dieser Typen darf eine Variable des Typs `E01Ellipse` zeigen?

3. Auf Objekte welcher dieser Typen darf eine Variable des Typs `E01Punkt` zeigen?

Angenommen, Sie haben die folgenden beiden Klassen:

```
 1 class K1 {
 2    int n1;
 3    void   machWas(double d) {...}
 4    double machMehr(double d) {...}
 5    K1() {
 6       n1 = 17;
 7    }
 8 }
 9
10 class K2 extends K1 {
11    double  d1;
12    boolean b1;
13    String nochMehr(int n1, int n2) {...}
14
15    static K2 carl = new K2();
16 }
```

4. Wie viele Konstruktoren gehören zur Klasse `K1`? Und zur Klasse `K2`?

5. Welchen Wert hat die Variable `carl.n1`? Und die Variable `carl.d1`?

6. Wie sieht ein `K2`-Objekt (z.B. das `K2`-Objekt `carl`) als *Zwiebel* dargestellt aus? Ein Beispiel für eine Zwiebeldarstellung finden Sie im Buch auf S. 291.

Rückseite der Wiederholungsfragen, 16 SU, Mo 07.06.10

Zum Unterschied zwischen überschreiben (engl. override) und verdecken (engl. hide)

```
 1 // ------------------------------------------------------------
 2 class Ober {
 3    String att = new String("   Ober-Attribut");
 4    String met() {return    "   Ober-Methode ";}
 5 } // class Ober
 6
 7 class Mittel extends Ober {
 8    String att = new String(" Mittel-Attribut");
 9    String met() {return    " Mittel-Methode ";}
10 } // class Mittel
11
12 class Unter  extends Mittel{
13    String att = new String("  Unter-Attribut");
14    String met() {return    "  Unter-Methode ";}
15 } // class Unter
16 // ------------------------------------------------------------
```

Für die Klassen `Ober`, `Mittel` und `Unter` gilt:

Das Attribut `att` in `Mittel` *verdeckt* das von `Ober` geerbte Attribut namens `att`
Das Attribut `att` in `Unter` *verdeckt* die von `Mittel` geerbten Attribute namens `att`

Die Methode `met` in `Mittel` *überschreibt* die von `Ober` geerbte Methode namens `met`
Die Methode `met` in `Unter` *überschreibt* die von `Mittel` geerbten Methoden namens `met`

Sei `v` eine Variable des Typs `Ober` oder `Mittel` oder `Unter.`

überschreiben (engl. to override) bedeutet:
Welche *Objektmethode* der Name `v.met` bezeichnet,
hängt nur vom *Zieltyp* der Variablen `v` ab (nicht von ihrem *Typ*).

verdecken (engl. to hide) bedeutet:
Welches *Objektattribut* der Name `v.att` bezeichnet,
hängt nur vom *Typ* der Variablen `v` ab (nicht von ihrem *Zieltyp*).

```
17 ... Befehle in iregendeiner Methode in irgendeiner Klasse:
18
19                                //      Typ     Zieltyp
20    Ober   oOb = new Unter(); // oOb: Ober,   Unter
21    Mittel mOb = oOb;         // mOb: Mittel, Unter
22    Unter  uOb = mOb;         // uOb: Unter,  Unter
23
24                                // Ausgabe:
25    pln(oOb.att);             //   Ober-Attribut
26    pln(mOb.att);             // Mittel-Attribut
27    pln(uOb.att);             //  Unter-Attribut
28
29    pln(         oOb.att));   //   Ober-Attribut
30    pln((Mittel) oOb.att));   // Mittel-Attribut
31    pln((Unter)  oOb.att));   //  Unter-Attribut
32
33    pln(oOb.met());           //  Unter-Methode
34    pln(mOb.met());           //  Unter-Methode
35    pln(uOb.met());           //  Unter-Methode
```

Der Cast-Befehl
`(Mittel)` (in Zeile 30) verändert den Typ des Ausdrucks `oOb` von `Ober` zu `Mittel`.
Der Cast-Befehl
`(Unter)` (in Zeile 31) verändert den Typ des Ausdrucks `oOb` von `Ober` zu `Unter`.

Antworten zu den Wiederholungsfragen, 16 SU, Mo 07.06.10

Angenommen, wir haben die 5 Klassen `E01Punkt`, `E01Rechteck`, `E01Quader`, `E10Ellipse`, `E10Kreis` (die im Typgrafen auf S. 286 abgebildet sind).

1. Auf Objekte welcher dieser Typen darf eine Variable des Typs `E01Rechteck` zeigen?

Auf Objekt der Typen E01Rechteck und E01Quadrat.

2. Auf Objekte welcher dieser Typen darf eine Variable des Typs `E01Ellipse` zeigen?

Auf Objekte der Typen E01Ellipse und E01Kreis.

3. Auf Objekte welcher dieser Typen darf eine Variable des Typs `E01Punkt` zeigen?

Auf Objekte der Typen E01Punkt, E01Rechteck, E01Quader, E10Ellipse, E10Kreis.

Angenommen, Sie haben die folgenden beiden Klassen:

```
 1 class K1 {
 2    int n1;
 3    void   machWas(double d) {...}
 4    double machMehr(double d) {...}
 5    K1() {
 6       n1 = 17;
 7    }
 8 }
 9
10 class K2 extends K1 {
11    double  d1;
12    boolean b1;
13    String nochMehr(int n1, int n2) {...}
14
15    static K2 carl = new K2();
16 }
```

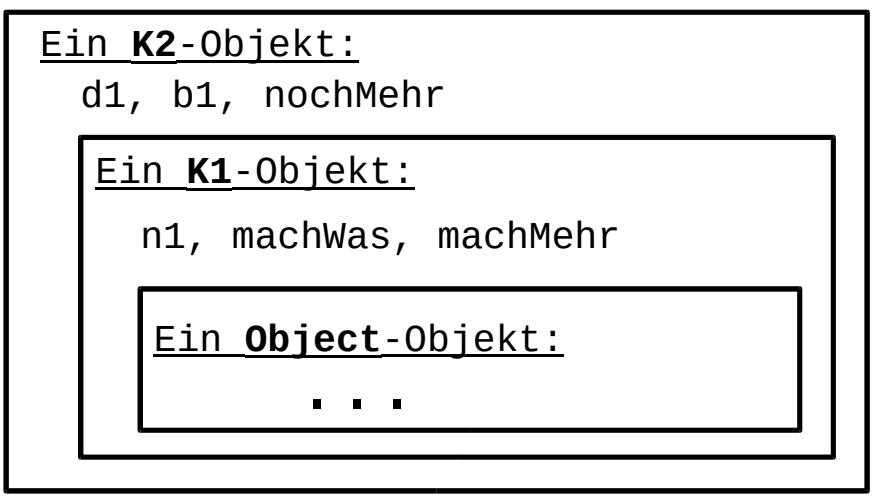
4. Wie viele Konstruktoren gehören zur Klasse `K1`? Und zur Klasse `K2`?

Zu K1: Ein (Standard-) Konstruktor (vereinbart in Zeile 5 bis 7)
Zu K2: Ein (Standard-) Konstruktor (geschenkt vom Ausführer).

5. Welchen Wert hat die Variable `carl.n1`? Und die Variable `carl.d1`?

carl.n1 hat den Wert 17, carl.d1 hat den Wert 0.0

6. Wie sieht ein K2-Objekt (z.B. das K2-Objekt `carl`) als *Zwiebel* dargestellt aus? Ein Beispiel für eine Zwiebeldarstellung finden Sie im Buch auf S. 291.



16. SU Mo 07.05.10

A. Wiederholung
B. Organisation: Morgen schreibt die Übungsgruppe 1a *zwei* Tests!

Wie viele Objekte haben wir?

Angenommen, wir haben folgende Objekte erzeugen lassen:

```
1    E01Punkt    p01 = new E01Punkt   (2.5, 1.5);
2    E01Punkt    p02 = new E01Punkt   (3.5, 2.5);
3    E01Rechteck r01 = new E01Rechteck(4.5, 3.5, 2.5, 1.5);
4    E01Rechteck r02 = new E01Rechteck(5.5, 4.5, 3.5, 2.5);
5    E01Quadrat  q01 = new E01Quadrat (6.5, 5.5, 4.5);
6    E01Quadrat  q02 = new E01Quadrat (7.5, 6.5, 5.5);
```

Wie viele `E01Rechteck`-Objekte haben wir dann?
(Nicht nur 2, sondern 4, weil jedes `E01Quadrat`-Objekt auch ein `E01Rechteck`-Objekt enthält).

Und wie viele `E01Punkt`-Objekte haben wir?
(6, weil jedes `E01Quadrat`-Objekt und jedes `E01Rechteck`-Objekt ein `E01Punkt`-Objekt enthält).

Wer ist größer, eine Oberklasse oder eine Unterklasse? Antwort: Tja!

S. 301, Bild 12.10 Noch mal unser Typgraf

Einen Weg durch den Grafen, der ganz oben (bei `Object`) beginnt und immer weiter nach unten führt (entgegen der Pfeilrichtung) bezeichnet man auch als einen *Ast* des Grafen.

Beispiel: Die Klassen `Object`, `E01Punkt`, `E01Rechteck`, `E01Quadrat` liegen auf einem Ast.

Angenommen, wir folgen (oben bei `Object` beginnend) einem Ast. Was gilt dann
für die *Anzahl der Elemente* der Klassen, an denen wir vorbeikommen?

Und was gilt für die *Anzahl der Objekte* der Klassen (in jedem Moment einer Programmausführung)?

Anmerkung: Im Folgenden steht manchmal einfach "größer" anstelle des genaueren, aber umständlicheren Ausdrucks "größer oder gleich groß".

Zusammenfassung: Es gibt zwei (wichtige) Kriterien für "die Größe einer Klasse":
Nach dem Kriterium "Anzahl der Elemente" ist eine Unterklasse größer als ihre Oberklassen,
nach dem Kriterium "Anzahl der Objekte" ist eine Oberklasse größer als ihre Unterklassen.

Das Kriterium "Anzahl der Objekte" ist in Zweifelsfällen das wichtigere. Nach diesem Kriterium ist `Object` die größte Klasse und steht deshalb in einem Typgrafen auch (meistens) ganz oben.

12.7 Der Typ und der Zieltyp einer Referenzvariablen

S. 303, Definition Zieltyp

S. 204, Beispiel-01: Der Typ und der Zieltyp der Variablen `p02`

Der Typ ist immer `E01Punkt` (siehe Zeile 5),
der Zieltyp ist anfänglich `void` und wird mehrfach verändert.

12.9 Geerbte Elemente ersetzen (S. 308)

Begriffe: Signatur, Profil, homonyme Attribute, homonyme Methoden,

Regel: In einer Klasse K kann man ein *geerbtes* Element E *ersetzen*, indem man in K ein zu E *homonymes* Element vereinbart

Die "Wirkung" des Ersetzens ist bei *Objektemethoden* anders als bei *allen anderen Elementen* (Objektattributen, Klassenmethoden, Klassenattributen, ...). Dieser Unterschied wird auch so beschrieben:

Ersetzt man eine Objektmethode, so wird (die "alte Methode") *überschrieben* (engl. to override an object method).

Ersetzt man ein anderes Element, so wird (das "alte Element") *verdeckt* (engl. to *hide* an element).

Faustregel:
Objektmethoden zu *überschreiben* ist in vielen Fällen *sehr sinnvoll*.
Andere Elemente zu *verdecken* ist *fast nie sinnvoll*.

Trotzdem sollte man auch wissen, was "verdecken" (engl. to hide) bedeutet, um es beim Lesen von Programmen zu erkennen und zu verstehen.

Der Unterschied zwischen überschreiben und verdecken

Die Rückseite der Wiederholungsfragen besprechen.

Zur Entspannung: Ein Gedicht von Joseph von Eichendorff (1788-1857)

Um und nach 1800 entstand in Deutschland eine Bewegung zahlreicher Dichter und Philosophen, die man heute als *Romantik* bezeichnet. Die Anhänger dieser Bewegung mussten sich unter anderem mit den Auswirkungen der französischen Revolution und mit Napoleon auseinandersetzen. J. v. E. war ein Adliger aus Oberschlesien, der seine Güter verlor und preussischer Beamter wurde.

Im Abendrot

Wir sind durch Not und Freude / gegangen Hand in Hand,
vom Wandern ruhen wir beide / nun überm stillen Land.
...

Wiederholungsfragen, 17. SU, Di 08.06.10

1. Angenommen, der Ausführer hat gerade die folgenden Befehle ausgeführt:

```
1    E01Punkt anna = new E01Punkt   (1.0, 2.0);
2    E01Punkt bert = new E01Rechteck(2.0, 3.0, 4.0, 5.0);
3    E01Punkt carl = new E01Quadrat (3.0, 4.0, 5.0);
```

Wieviele `E01Punkt`-Objekte existieren dann?
Und `E01Rechteck`-Objekte? Und `E01Quadrat`-Objekte?

2. Füllen Sie die folgende Tabelle aus:

| Variable | Typ | Zieltyp |
|---|---|---|
| `anna` | | |
| `bert` | | |
| `carl` | | |

3. Betrachten Sie die folgenden Methoden-Vereinbarungen:

```
4    int berechne(long ludwig, double dora) { ... }
5    String kombiniere(String sonja, StringBuilder siggy) { ... }
6    static public void machWas() { ... }
```

Füllen Sie die folgende Tabelle aus:

| Methode | Signatur | Profil |
|---|---|---|
| `berechne` | | |
| `kombiniere` | | |
| `machWas` | | |

4. Geben Sie von jedem der folgenden Elemente einer Klasse an, ob man es überschreiben (engl. override) oder überdecken (engl: hide) kann:

- eine öffentliche Klassenmethode
- ein privates Objektattribut
- eine geschütze Objektmethode
- eine paketweit erreichbare Klassenmethode
- eine paketweit erreichbare Objektmethode
- ein öffentliches Klassenattribut

5. Betrachten Sie die folgenden Befehle:

```
7 Ober   ob = new Ober();
8 Unter  un = new Unter();
```

Was geben die folgenden Befehle aus (die Vereinbarungen der Klassen `Ober`, `Mittel` und `Unter` stehen im Buch auf S. 310 und auf der Rückseite der Wiederholungsfragen zum 16. SU):

```
 9                  // Ausgabe:
10 pln(ob.att);     //
11 pln(un.att);     //
12 pln(ob.met());   //
13 pln(un.met());   //
```

6. Auf Objekte welcher Klassen darf eine `E01Punkt`-Variable zeigen (laut Typgraf auf S. 286)?

Antworten zu den Wiederholungsfragen, 17. SU, Di 08.06.10

1. Angenommen, der Ausführer hat gerade die folgenden Befehle ausgeführt:

```
1    E01Punkt anna = new E01Punkt   (1.0, 2.0);
2    E01Punkt bert = new E01Rechteck(2.0, 3.0, 4.0, 5.0);
3    E01Punkt carl = new E01Quadrat (3.0, 4.0, 5.0);
```

Wieviele `E01Punkt`-Objekte existieren dann?
Und `E01Rechteck`-Objekte? Und `E01Quadrat`-Objekte?

2. Füllen Sie die folgende Tabelle aus:

| Variable | Typ | Zieltyp |
|---|---|---|
| `anna` | E01Punkt | E01Punkt |
| `bert` | E01Punkt | E01Rechteck |
| `carl` | E01Punkt | E01Quadrat |

3. Betrachten Sie die folgenden Methoden-Vereinbarungen:

```
4    int berechne(long ludwig, double dora) { ... }
5    String kombiniere(String sonja, StringBuilder siggy) { ... }
6    static public void machWas() { ... }
```

Füllen Sie die folgende Tabelle aus:

| Methode | Signatur | Profil |
|---|---|---|
| `berechne` | berechne long double | int berechne long double |
| `kombiniere` | kombiniere String StringBuilder | String kombiniere String StringBuilder |
| `machWas` | machWas | void machWas |

4. Geben Sie von jedem der folgenden Elemente einer Klasse an, ob man es überschreiben (engl. override) oder verdecken (engl: hide) kann:

- eine öffentliche Klassenmethode verdecken
- ein privates Objektattribut verdecken
- eine geschütze Objektmethode überschreiben
- eine paketweit erreichbare Klassenmethode verdecken
- eine paketweit erreichbare Objektmethode überschreiben
- ein öffentliches Klassenattribut verdecken

5. Betrachten Sie die folgenden Befehle:

```
7 Ober   ob = new Ober();
8 Unter  un = new Unter();
```

Was geben die folgenden Befehle aus (die Vereinbarungen der Klassen `Ober`, `Mittel` und `Unter` stehen im Buch auf S. 310 und auf der Rückseite der Wiederholungsfragen zum 16. SU):

```
9                  // Ausgabe:
10 pln(ob.att);    //   Ober-Attribut
11 pln(un.att);    //  Unter-Attribut
12 pln(ob.met());  //   Ober-Methode
13 pln(un.met());  //  Unter-Methode
```

6. Auf Objekte welcher Klassen darf eine `E01Punkt`-Variable zeigen (laut Typgraf auf S. 286)?

Auf Objekte der Klassen E01Punkt, E01Rechteck, E01Quadrat, E01Ellipse, E01Kreis.

17. SU Di 08.06.10

A. Wiederholung
B. Organisation

Die Klasse E01Punkt ist keine gute Wurzelklasse unserer Hierarchie

Eine `E01Punkte`-Variable darf auf Objekte vieler verschiedener Klassen zeigen (siehe letzte Wiederholungsfrage). Aber die Sache hat folgenden Haken:

```
1    E01Punkt p01 = new E01Rechteck(1.0, 2.0, 3.0, 4.0);
2    pln(p01.urAbstand()); // erlaubt
3    pln(p01.getUmfang()); // nicht erlaubt
```

Der Methodendaufruf `p01.urAbstand()` ist erlaubt, weil die Methode `urAbstand` zur Klasse `E01Punkt` gehört.

Der Methodenaufruf `p01.getUmfang()` ist *nicht erlaubt*, weil die Methode `getUmfang` nicht zur Klasse `E01Punkt` gehört.

Über die `E01Punkt`-Variable `p01` können wir nur auf die `E01Punkte`-Elemente eines Objekts zugreifen, auch wenn das Objekt ein `E01Rechteck`-Objekt ist und noch weitere Elemente enthält (z.B. Methoden namens `getUmfang` und `getFlaeche` und Attribut namens `seiteX` und `seiteY` etc.).

S. 299, Aufgabe-05: Die Methode `flaechenSummeAusgeben03`

Für diese Aufgabe gibt es keine wirklich gute Lösung, und daran ist die Klasse `E01Punkt` schuld, weil in ihr noch keine Methoden `getFlaeche` und `getUmfang` etc. vereinbart wurden.

Andererseits: In einem bestimmten (intuitiven) Sinn haben Punkte keine Fläche und keinen Umfang, Methoden `getFlaeche` und `getUmfang` passen deshalb nicht sehr gut in eine Klasse `E01Punkt`.

Bevor wir unsere kleine Klassenhierarchie (`E01Punkt`, `E01Rechteck`, ... etc.) verbessern, soll noch ein neues Konzept vorgestellt werden:

14.1 Abstrakte Klassen

Motivation: Wir wollen mehrere Klassen vereinbaren, die vieles gemeinsam haben. Wie können wir es vermeiden, diesen "gemeinsamen Kern" mehrmals hinzuschreiben?

S. 335, eine abstrakte Klasse G und

mehrere Klassen K1, K2, ... die G beerben (erweitern).

Regel: Ist G eine abstrakte Klasse, dann ist ein Befehl wie `new G ...` verboten.

Befehle wie `class K1 extends G {...}` sind aber durchaus erlaubt.

Angenommen, wir haben die abstakte Klasse `G` und die konkrete Unterklasse `K1` von `G`. Wie kann man dann ein Objekt der abstrakten Klasse `G` erzeugen?

(Man erzeugt ein Objekt der konkreten Klasse `K1`. Dieses Objekt enthält ja ein Objekt der abstrakten Klasse `G`, wie man mit der Zwiebeldarstellung deutlich machen kann).

Anmerkung: In vielen (fast allen) Java-Büchern steht sinngemäß: "Man kann keine Objekte einer abstrakten Klasse erzeugen lassen". Das ist höchstens bei oberflächlicher Betrachtung richtig. Außerdem ist es dann schwer zu verstehen, dass abstrakte Klassen auch Konstruktoren enthalten dürfen (die ja zum Initialisieren von Objekten dieser Klasse dienen).

Regel: In einer abstrakten Klasse darf man alles vereinbaren, was in einer konkreten Klasse erlaubt ist. Außerdem darf man in einer abstrakten Klasse auch noch *abstrakte Objektmethoden* vereinbaren (was in einer konkreten Klasse *nicht* erlaubt ist).

Def.: Eine abstrakte Klasse legt das *Profil* einer Methode fest, läßt den *Rumpf* aber noch offen.

S. 336, Beispiel-01: Die abstrakte Klasse `E02GeoFigur`

Erinnert Sie die (abstrakte) Klasse `E02GeoFigur` an eine (konkrete) Klasse, die wir schon mal besprochen haben)? (Nein, nicht die Klasse `StringBuilder` und auch nicht die Klasse `Person02`, sondern die Klasse `E01Punkt` :-).

Die abstrakte Klasse `E02GeoFigur` ist eine bessere Wurzelklasse für eine Hierarchie als `E01Punkt`, weil sie auch Methoden `getUmfang` und `getFlaeche` enthält. Die anderen Klassen der `E02`-Hierarchie sehen dann ganz ähnlich aus wie die entsprechenden Klassen der `E01`-Hierarchie.

S. 337, die Klasse `E02Rechteck`

Regel: Wenn eine konkrete Klasse eine abstrakte Methode erbt, muss sie sie mit einer konkreten Methode überschreiben.

Eine abstrakte Klasse darf abstrakte Methoden erben ohne sie zu überschreiben.

S. 338, die Klasse `E02GeoFigurMitFarbe`

## 14.2 Schnittstellen

Angenommen: Ein Programmierer will eine Sammlung programmieren, die ihre Komponenten immer in (z.B. aufsteigend) sortierter Reihenfolge enthält. Er kann fast beliebige Objekte als Komponenten seiner Sammlung zulassen, aber was muss er mit den Objekten (z.B. wenn sie in die Sammlung eingefügt werden) machen können? (Er muss sie vergleichen können, und zwar nicht nur auf gleich / ungleich sondern auf kleiner / nicht kleiner oder auf größer / nicht größer).

Wie kann man ganz allgemein ausdrücken und festlegen, dass die Objekte einer Klasse *vergleichbar* sein sollen? Mit einer entsprechenden *Schnittstelle* (engl. interface).

Eine Schnittstelle ist (normalerweise) eine Zusammenfassung von *abstrakten Objektmethoden.*

S. 341, Beispiel-01: Die Schnittstelle `Vergrößerbar`

Was kann man mit so einer Schnittstelle machen?
Man kann Klassen vereinbaren, die die Schnittstelle implementieren!

S. 342, Beispiel-02: Die Klasse `GanzZahl01` implementiert die Schnittstelle `Vergrößerbar`

Mit dem Hinweis `impelements Vergrößerbar` (in Zeile 8) verspricht der Programmierer, in der Klasse `GanzZahl01` die abstrakten Methoden der Schnittstelle `Vergrößerbar` mit konkreten Methoden zu überschreiben (siehe Zeile 15 und 16). Ähnlich wie eine Klasse ist auch eine Schnittstelle ein *Typ* (ein "Bauplan für Variable"). Die Objekte der Klasse `GanzZahl01` gehören jetzt zu *zwei* Typen:

Zum Typ `GanzZahl01` und zum Typ `Vergrößerbar`.

Zur Entspannung: Richard Strauss (1864-1949)

Deutscher Musiker, *nicht* verwandt mit den beiden Musikern namens Johann Strauss, begann mit 6 Jahren zu komponieren, komponierte zahlreiche *Tondichtungen* (Don Juan, Till Eulenspiegels lustige Streiche, Also sprach Zarathustra, Ein Heldenleben, Sinfonia domestica, Eine Alpensinfonie, ...) und Opern (Salome, Elektra, Der Rosenkavalier, ...). Teile der Tondichtung *Also sprach Zarathustra* wurden im Film 2001 und von Elvis Presley verwendet (und werden heute in einer Bierreklame zitiert). War weltberühmt, galt bei vielen als "der größte lebende deutsche Komponist". Setzte sich intensiv für die Rechte von Musikern und Komponisten ein (Genossenschaft deutscher Tonsetzer, GEMA).

Sein Verhältnis zu den Nazis sieht heute alles andere als vorbildlich aus. Als 1933 die Nazis einen Auftritt des jüdischen Dirigenten *Bruno Walter* verboten, sprang Strauss als Ersetzt ein. Als *Toscanini* seine Teilnahme an den Bayreuther Wagner-Festspielen (aus Protest gegen die Nazis) absagte, sprang er ebenfalls ein. Er war von 1933 bis 1935 Präsident der Reichsmusikkammer, was für die Nazis ein wichtiger Propaganda-Erfolg war.

Komponierte auch viele Lieder ("Lieder" ist eine eigene Musikgattung, die auch im Englischen und in anderen Sprachen als "Lieder" bezeichnet wird), unter anderem "Vier letzte Lieder" (für Sopran und großes Orchester), zu denen auch das Lied "Im Abendrot" (Text von Joseph von Eichendorff) gehört.

Wiederholungsfragen, 18. SU, Mo 14.06.10

1. Geben Sie von jeder der folgenden Methoden die *Signatur* und das *Profil* an:

```
1  public static void tuDieses(String s, double d) { ... }
2  String tuJenes()                                 { ... }
3  String tuDieses(String vorName, double Gewicht) { ... }
4  void tuDieses(String xxx, double yyy)            { ... }
```

| Nr | Signatur | Profil |
|---|---|---|
| 1 | | |
| 2 | | |
| 3 | | |
| 4 | | |

1. Woran erkennt man eine *abstrakte Klasse*?

2. Was für Elemente darf man *nicht* in *konkreten Klassen*, wohl aber in *abstrakten Klassen* vereinbaren?

3. Welcher Befehl ist im Zusammenhang mit einer *konkreten* Klasse *erlaubt*, im Zusammenhang mit einer *abstrakten* Klasse aber *verboten*?

4. Vereinbaren Sie eine Methode, die der folgenden Spezifikation entspricht:

```
void verZwoelffache(Vergroesserbar vbar) {
    // Vergroessert vbar um den Faktor 12.




}
```

Hinweis: Die Vereinbarung der Schnittstelle `Vergroesserbar` steht im Buch auf S. 341.

5. Vereinbaren Sie eine Klasse namens `Bruchzahl02`, die die Schnittstelle `Veraenderbar` implementiert. Jedes `Bruchzahl02`-Objekt soll ein privates Attribut namens `groesse` vom Typ `double` enthalten. Die Vereinbarung dieser Schnittstelle `Veraenderbar` steht im Buch auf S. 343. Die Schnittstelle `Verkleinerbar` enthält nur *eine* abstrakte Methode, und zwar eine parameterlose Prozedur namens `halbiere`.

6. Wie viele Klassen darf eine Klasse *beerben*?

7. Wie viele Schnittstellen darf eine Schnittstelle *beerben*?

8. Wie viele *Schnittstellen* darf eine Klasse *implementieren*?

Rückseite der Wiederholungsfragen, 18. SU, Mo 14.06.10

Pakete und Verzeichnisse (Zitat aus dem Kapitel 27 des Buches, welches nur online existiert)

Angenommen, ein Beispiel-Programm besteht aus 6 Klassen, verteilt auf 3 Pakete, etwa so:

Übersicht-1: Zu welchem Paket gehören welche Klassen bzw. Quelldateien

| *zum Paket* | *gehören die Klassen* | *bzw. die Quelldateien* |
|---|---|---|
| `p00` | `K10Tst` | `K10Tst.java` |
| `p00.p10` | `K12, K11, K10` | `K12.java, K11.java, K10.java` |
| `p00.p20` | `K22, K21` | `K22.java, K21.java` |

Um die Quelldateien dieses Programms abzulegen sollte man zuerst einmal irgendwo ein oberstes *Quellcode-Verzeichnis* anlegen, z.B. das Verzeichnis `d:\Qcd`.

In diesem Verzeichnis sollte man für jedes Paket ein entsprechendes Unterverzeichnis anlegen und die Quelldateien in dem Unterverzeichnis ablegen, welches ihrem Paket entspricht.

Übersicht-2: In welches Verzeichnis gehören welche Quelldateien

| *Dem Paket* | *entspricht das Verzeichnis* | *in das folgende Quelldateien gehören* |
|---|---|---|
| | `d:\Qcd` | |
| `p00` | `d:\Qcd\`**`p00`** | `K10Tst.java` |
| `p00.p10` | `d:\Qcd\`**`p00\p10`** | `K12.java, K11.java, K10.java` |
| `p00.p20` | `d:\Qcd\`**`p00\p20`** | `K22.java, K21.java` |

Ein Java-Compiler erzeugt aus jeder Klasse namens `Otto` eine sogenannte *Bytecode-Datei* namens `Otto.class`. Für das Ablegen dieser Bytecode-Dateien gibt es zwei Möglichkeiten oder Schemata:

Schema 1: Die *Bytecode-Datei* `Otto.class` wird *im selben Verzeichnis* abgelegt, in dem auch die *Quell-Datei* `Otto.java` steht.

Schema 2: Man erzeugt irgendwo ein oberstes Bytecode-Verzeichnis, z.B. das Verzeichnis `c:\Bcd`. Der Java-Compiler kann dann unterhalb von diesem Verzeichnis ganz analog zu den Quellcode-Verzeichnissen (`d:\Qcd\p00`, `d:\Qcd\p00\p10`, ...) entsprechende Bytecode-Verzeichnisse (`c:\Bcd\p00`, `c:\Bcd\p00\p10`, ...) anlegen und die Bytecode-Dateien in diesen Verzeichnissen ablegen, etwa so:

Übersicht-3: In welchem Verzeichnis legt der Compiler welche Bytecodedatei ab

| *Dem Paket* | *entspricht das Verzeichnis* | *in dem der Compiler folgende Bytecodedatei ablegt* |
|---|---|---|
| | `c:\Bcd` | |
| `p00` | `c:\Bcd\`**`p00`** | `K10Tst.class` |
| `p00.p10` | `c:\Bcd\`**`p00\p10`** | `K12.class, K11.class, K10.class` |
| `p00.p20` | `c:\Bcd\`**`p00\p20`** | `K22.class, K21.class` |

Compilationsbefehle

für Schema 1: `> javac K10Tst.java`
für Schema 2: `> javac -d c:\Bcd K10Tst.java`

Antworten zu den Wiederholungsfragen, 18. SU, Mo 14.06.10

1. Geben Sie von jeder der folgenden Methoden die *Signatur* und das *Profil* an:

```
1  public static void tuDieses(String s, double d) { ... }
2  String tuJenes()                                 { ... }
3  String tuDieses(String vorName, double Gewicht) { ... }
4  void tuDieses(String xxx, double yyy)            { ... }
```

| Nr | Signatur | Profil |
|---|---|---|
| 1 | `tuDieses String double` | `void tuDieses String double` |
| 2 | `tuJenes` | `String tuJenes` |
| 3 | `tuDieses String double` | `String tuDieses String double` |
| 4 | `tuDieses String double` | `void tuDieses String double` |

1. Woran erkennt man eine *abstrakte Klasse*?

Am Wort abstract vor dem Wort class.

2. Was für Elemente darf man *nicht* in *konkreten Klassen*, wohl aber in *abstrakten Klassen* vereinbaren?

Abstrakte Objektmethoden.

3. Welcher Befehl ist im Zusammenhang mit einer *konkreten* Klasse *erlaubt*, im Zusammenhang mit einer *abstrakten* Klasse aber *verboten*?

Der Befehl new (new String(...) ist erlaubt, new E02GeoFigur(...) ist verboten).

4. Vereinbaren Sie eine Methode, die der folgenden Spezifikation entspricht:

```
void verZwoelffache(Vergroesserbar vbar) {
    // Vergroessert vbar um den Faktor 12.
    vbar.verdopple();
    vbar.verdopple();
    vbar.verdreifache();
}
```

Hinweis: Die Vereinbarung der Schnittstelle `Vergroesserbar` steht im Buch auf S. 341.

5. Vereinbaren Sie eine Klasse namens `Bruchzahl02`, die die Schnittstelle `Veraenderbar` implementiert. Jedes `Bruchzahl02`-Objekt soll ein privates Attribut namens `groesse` vom Typ `double` enthalten. Die Vereinbarung der Schnittstelle `Veraenderbar` steht im Buch auf S. 343. Die Schnittstelle `Verkleinerbar` enthält nur *eine* abstrakte Methode, und zwar eine parameterlose Prozedur namens `halbiere`.

```
 5 class Bruchzahl02 implements Veraenderbar {
 6    double groesse;
 7    public void verdopple()          {groesse*=2;}
 8    public void verdreifache()       {groesse*=3;}
 9    public void halbiere()           {groesse/=2;}
10    public void aendere(int prozent) {groesse=groesse*prozent/100;}
11    ...
12 }
```

6. Wie viele Klassen darf eine Klasse *beerben*? Genau eine

7. Wie viele Schnittstellen darf eine Schnittstelle *beerben*? Beliebig viele (0, 1, 2, ..)

8. Wie viele *Schnittstellen* darf eine Klasse *implementieren*? Beliebig viele (0, 1, 2, ...)

18. SU Mo 14.06.10

A. Wiederholung
B. Organisation

Schnittstellen (Fortsetzung)

Regel: Ein Klasse darf beliebig viele Schnittstellen implementieren (0 oder 1 oder 2 oder ...).

S. 343, Beispiel-03: Die Klasse `GanzZahl02` implementiert *zwei* Schnittstellen

Regel: Eine Schnittstelle darf beliebig viele Schnittstellen erweitern (oder: beerben)

S. 343, Beispiel-04: Die Schnittstelle `Veraenderbar` erweitert *zwei* Schnittstellen.

*Mehrfache Beerbung* (die es z.B. in C++ gibt), ist ein mächtiges, aber auch sehr gefährliches und schwer zu handhabendes Konstrukt. Es wurde in Java deshalb bewußt nicht aufgenommen. Man kann Schnittstellen auch sehen als eine Art *Trostpflaster* für die fehlende mehrfache Beerbung.

17 Pakete (S. 424)

Motivation: Angenommen, eine Firma A will ein Programm entwickeln, das aus 300 Klassen besteht. 100 dieser Klassen will die Firma selbst schreiben, weitere 100 will sie von einer Firma B und die restlichen 100 von einer Firma C kaufen. Dann besteht eine erhebliche Gefahr, dass einige dieser Klassen gleiche Namen haben (z.B. `Test` oder `Ausgabe` oder `MainClass`).

Die Namen dieser Klassen zu ändern kann sehr aufwendig und fehlerträchtig sein (weil "alte Klassen sich ja auf die alten Namen verlassen haben", und ebenfalls geändert werden müssen.

Um dieses Problem zu vermeiden, hat man Pakete erfunden.

Ein Paket hat Ähnlichkeit mit einem Ordner (oder: Verzeichnis, directory) eines Dateisystems:
Ein Ordner kann normale Dateien und (weitere Ordner) enthalten.

S. 424, Def.: Ein Paket kann Klassen, Schnittstellen und (weitere) Pakete enthalten.

Ein Top-Paket ist ein Paket, welches in keinem anderen Paket enthalten ist (z.B. `java`, `javax`, `org` oder `de`).

Pakete wie `java.lang` oder `java.util` sind im Paket `java` und somit *keine* Top-Pakete.

Begriffe:
Name eines Pakets, vollständiger Name eines Pakets,
Name einer Klasse, vollständiger Name einer Klasse

Beispiel: `java.util.concurrent.DelayQueue`

| | |
|---|---|
| `concurrent` | ist der *Name* eines Pakets |
| `java.util.concurrent` | ist der *volle Name* des Paketes `concurrent` |
| `DelayQueue` | ist der *Name* einer Klasse |
| `java.util.concurrent.DelayQueue` | ist der *volle Name* der Klasse `DelayQueue` |

Ein Top-Paket ist die Wurzel eines *Baums* (ähnlich wie ein Laufwerk unter Windows)
Alles Top-Pakete zusammen bilden einen *Wald* (d.h. eine Menge von Bäumen)

Lösung des eingangs erwähnten Problems (mit den 300 Klassen):

1. Heute hat jede Firma eine weltweit eindeutige Internetadresse, z.B. `www.beuth-hochschule.de`
2. Das `www` läßt man weg, den Rest der Adresse dreht man herum: `de.beuth-hochschule`

3. Alle an der Beuth-Hochschule entwickelten Klassen (die nach draußen verkauft oder verschenkt werden sollen), sollten zu Unterpaketen des Paketes `de.beuth-hochschule` gehören (z.B. zum Paket `de.beuth-hochschule.fb6.ss10.schmidt.frank`).

4. Wenn alle Firmen sich an diese einfache Regel halten, kann man Klassen beliebiger Firmen kombinieren: Auch wenn die Namen der Klassen zum Teil gleich sind, sind die vollen Namen garantiert verschieden.

Zur Entspannung: Charles Babbage (1791-1871)

Forschte auf verschiedenen Gebieten. Unternahm mehrere Versuche, mechanische Rechenmaschinen zu bauen. Keine davon wurde funktionstüchtig, aber seine Pläne und Überlegungen dazu waren wichtige Stationen auf dem Weg zu Computern.

1823 Difference Engine no. 1 (in die Entwicklung floßen 17000 Pfund, was etwa dem Preis von zwei Schlachtschiffen entsprach)
1833 Analytical Engine (die sollte sogar schon programmierbar werden)
1847 Difference Engine no. 2 (wurde ab 1985 im Science Museum in London genau nach den Plänen von Babbage gebaut und funktioniert)

Ada Byron, Lady Lovelace (1815-1852), Tochter des berühmten Dichters Lord Byron, arbeitete mit an der Analytical Engine und gilt seitdem als erste Programmiererin. Nach ihr ist die Sprache Ada benannt (ANSI/MIL-STD-1815, nach ihrem Geburtsjahr).

Weitere Erfindungen von Charles Babbage: Kuhfänger (für Lokomotiven, z. B. im Wilden Westen). Er knackte als erster eine Vignére-Verschlüsselung (die vorher als sicher galt). Schrieb das Buch *Economy of machinery and manufactures*, eine Analyse des Frühkapitalismus und wichtige Quelle für Karl Marx.

## 15 Ausnahmen fangen und werfen

Was sind *Ausnahmesituationen*? S. 362/363, 5 typische Beispiele.

In älteren Programmiersprachen typische Muster zur Behandlung von Ausnahmen:

```
int status = 0;
status = UPRO01(p11, p12, ...)
if (status != 0) {
   // Ausnahmebehandlung 1
}
status = UPRO02(p21, p22, ...)
if (status != 0) {
   // Ausnahmebehandlung 2
}
status = UPRO03(p31, p32, ...)
if (status != 0) {
   // Ausnahmebehandlung 3
}
... etc. etc.
```

Problem: Befehle für "den Normalfall" und "Befehle für Ausnahmefälle" sind "eng vermischt" und deshalb beide schwer zu lesen und zu verstehen.

In neueren Sprachen (C++, Ada, Java, C#, ...) gibt es spezielle Konstrukte zur Behandlung von Ausnahmen. Sie sollen dazu dienen, die Befehle für den Normalfall und die Befehle für Ausnahmefälle voneinander zu trennen und dadurch beide leichter lesbar und verstehbar zu machen.

S. 363, Def. *Ausnahmeklasse* und *Ausnahmeobjekte*

Grundprinzip der Ausnahmebehandlung: Wenn an einer bestimmten Stelle eines Programms ein Ausnahmefall erkannt wird, muss er nicht gleich dort behandelt werden. Statt dessen werden Informationen über den Ausnahmefall in eine Ausnahmeobjekt gepackt und das Ausnahmeobjekt wird geworfen. Es "fliegt dann durch das Programm" und kann an einer anderen Stelle des Programms gefangen und behandelt werden. Auf diese Weise kann man die Befehle für den Normalfall und die Befehle für Ausnahmefälle voneinander trennen.

Im Einzelnen:

Bestimmte Java-Befehle sind gefährlich: In gewissen Fällen funktionieren sie nicht normal sondern werfen ein Ausnahmeobjekt (kurz: eine Ausnahme).

S. 363, Beispiel-01: Division durch 0

S. 364, Beispiel-02: Strings in `int`-Werte umwandeln

Der Programmierer kann selbst "gefährliche Befehle" erfinden und selbst Ausnahmeobjekte (Ausnahmen) werfen.

S.364, Beispiel-03: Ein Ausnahmeobjekt erzeugen und werfen

Was passiert, wenn eine Ausnahme geworfen, aber nirgends gefangen wird? Das betreffende Programm wird mit einer Fehlermeldung beendet.

S. 364, Beispiel-04:

Wiederholungsfragen, 19. SU, Di 15.06.10

1. Ein Paket (in Java) ist ein Behälter. Was ein socher Behälter enthalten?

2. Welche Eigenschaft muss ein Paket haben, damit es ein Top-Paket ist?

3. Es gibt ein Top-Paket namens java. Wie lautet sein voller Name?

4. Woraus besteht der volle Name eines Paketes p1, welches sich in einem Paket p2 befindet?

5. Woraus besteht der volle Name einer Klasse k, welche sich in einem Paket p befindet?

6. Die Abteilung 6 einer Firma mit der Internetadresse `www.meier-und-sohn.de` entwickelt eine Klasse namens `Abrechnung`. Wie sollte der volle Name dieser Klasse lauten?

7. Viele moderne Sprachen enthalten spezielle Befehle und Konstrukte zum Behandeln von Ausnahmen. Was sollen diese speziellen Befehle und Konstrukte dem Programmierer ermöglichen?

Antworten zu den Wiederholungsfragen, 19. SU, Di 15.06.10

1. Ein Paket (in Java) ist ein Behälter. Was ein socher Behälter enthalten?

Klassen, Schnittstellen, Pakete.

2. Welche Eigenschaft muss ein Paket haben, damit es ein Top-Paket ist?

Es darf in keinem anderen Paket enthalten sein.

3. Es gibt ein Top-Paket namens java. Wie lautet sein voller Name?

java

4. Woraus besteht der volle Name eines Paketes p1, welches sich in einem Paket p2 befindet?

Aus dem vollen Namen von p2 gefolgt von einem Punkt . und dem Namen von p1.

5. Woraus besteht der volle Name einer Klasse k, welche sich in einem Paket p1 befindet?

Aus dem vollen Namen von p gefolgt von einem Punkt . und dem Namen der Klasse k.

6. Die Abteilung 6 einer Firma mit der Internetadresse `www.meier-und-sohn.de` entwickelt eine Klasse namens `Abrechnung`. Wie sollte der volle Name dieser Klasse lauten?

de.meier-und-sohn.abteilung6.Abrechnung

7. Viele moderne Sprachen enthalten spezielle Befehle und Konstrukte zum Behandeln von Ausnahmen. Was sollen diese speziellen Befehle und Konstrukte dem Programmierer ermöglichen?

Die Befehle für den Normalfall und die Befehle für die Ausnahmefälle zu trennen (und nicht zu vermischen wie die Streifen eines Zebramusters)

19. SU Di 15.06.10

A. Wiederholung
B. Organisation

Pakete (Fortsetzung von gestern)

Die Rückseite des Blattes mit den Wiederholungsfragen von gestern (18. SU) besprechen

Ausnahmen fangen und behandeln (Fortsetzung von gestern)

S. 365, Beispiel-01: Eine gefährliche Methode `dividiere`

Was kann schiefgehen, wenn man die Funktion `dividiere` aufruft?
S. 366, Beispiel-02: Die Methode `dividiere` mit "guten" und "schlechten" Parametern aufrufen

Falls die Methode `dividiere` eine Ausnahme wirft, wird die gefangen und behandelt.

Wenn im try-Block eine Ausnahme geworfen wird, wird ein dazu passender catch-Block gesucht und ausgeführt.

Was passiert in Zeile15? (Ausgabe z.B. `9 / 4`)

Was passiert, wenn im try-Block eine ArithmeticException geworfen wird? (Zeile 22)

Was passiert, wenn im try-Block eine NumberFormatException geworfen wird? (Zeile 19)

In den Zeilen 27 bis 34 sieht man die Ausgabe des Programms

Def.: Ein `try-catch`-Befehl besteht aus
*einem* `try`-Block gefolgt von *einem oder mehreren* `catch`-Blöcken.

15.2 Flugregeln für Ausnahmen (S. 369)

Was ist ein Stapel?

Wozu der Java-Ausführer einen Stapel benützt (S. 370, Beispiel-01)

Zwei Darstellungen des Stapels: S. 370 unten und S. 371 oben

Was passiert, wenn der Ausführer gerade im Zustand z5 ist (d.h. er führt die Methode `metB` aus) und jetzt tritt eine Ausnahme auf?

Allgemeines Schema auf S. 372: Was fragt sich der Ausführer, wenn er eine Ausnahme geworfen hat? Und was macht er dann (je nach der Antwort ja oder nein auf die Frage)?

Die Frage (im "Nappo" auf S. 372) auswendig lernen!

15.3 Der try-catch-finally-Befehl (S. 373)

Zur Entspannung: Christian Morgenstern (1871-1914):

Anto-Logie Die Evolution großer Tiere und Zahlen:

Im Anfang lebte, wie bekannt, als größter Säuger der Gig-ant.

Wiederholungsfragen, 20. SU, Mo 21.06.10

1. Skizzieren Sie (d.h. beschreiben Sie ganz kurz) mind. zwei *Ausnahmefälle*.

2. Zahlreiche ältere Programme bestehen aus einem Zebramuster von Befehlen (
ein Streifen für den Normallfall,
ein Streifen für Ausnahmefälle,
ein Streifen für den Normalfall,
ein Streifen für Ausnahmefälle,
...
Was ist der Nachteil eines solchen Zebramusters?

3. Wozu hat man spezielle Befehle zum Bearbeiten von Ausnahmefällen erfunden (und u.a. in die Sprache Java aufgenommen)? Was wollte man damit dem Programmierer ermöglichen?

4. Angenommen, Sie haben (in einem Java-Programm) in zwei `int`-Variablen namens `erg1` und `erg2` zwei Ergebnisse berechnet, die eigentlich gleich sein müssten. Geben Sie einen Befehl an, der eine geeignete Ausnahme (z.B. des Typs `ArithmeticException`) wirft, wenn die beiden Ergebnisse verschieden sind.

5. Woraus besteht ein `try-catch`-Befehl?

6. Woraus besteht ein `try-catch-finally`-Befehl?

7. Welche Frage stellt sich der Java-Ausführer jedes Mal, wenn er eine Ausnahme a geworfen hat?

8. Was macht der Ausführer, wenn die Antwort auf diese Frage (siehe 7.) "Ja" ist?

Rückseite der Wiederholungsfragen, 20. SU, Mo 21.06.10

```
 1 // Vereinbarung einer geprüften Ausnahmeklasse:
 2 class ZahlIstUngerade extends Exception {
 3    public  ZahlIstUngerade(String s, int n) {super(s); zahl = n;}
 4    public  int getZahl() {return zahl;}
 5    private int zahl = 0;
 6 } // class ZahlIstUngerade
 7 // ---------------------------------------------------------------------
 8 public class Ausnahmen03 {  // Die Hauptklasse dieses Programms
 9    // ---------------------------------------------------------------------
10    static public int liesEineGeradeZahl() throws
11       java.io.IOException, // Wenn die Tastatur Feuer faengt
12       ZahlIstUngerade      // Wenn die Eingabe eine ungerade Zahl ist
13       // Liest eine gerade Ganzzahl von der Standardeingabe ein und
14       // liefert sie als Ergebnis.
15    {
16
17       String einGabeS = EM.liesString();
18       // Die Methode Integer.decode wirft evtl. eine NumberFormatException:
19       int    einGabeI = Integer.decode(einGabeS);
20       if (einGabeI % 2 != 0) throw                          //   <--- throw
21          new ZahlIstUngerade("Verflixt!!!", einGabeI);
22       return einGabeI;
23    } // liesEineGeradeZahl
24    // ---------------------------------------------------------------------
25    static public void main(String[] _) {
26       p("Ausnahmen20: Eine gerade Zahl? ");
27       int zahl = 0;
28
29       try {                                                 //     <--- try
30          zahl = liesEineGeradeZahl();
31          pln(zahl + " ist eine sehr gute Eingabe!");
32       }
33       catch (ZahlIstUngerade ziu) {                         //   <--- catch
34          pln("Ihre Eingabe " + ziu.getZahl() + " ist ungerade!");
35          pln(ziu.getMessage());
36       }
37       catch (java.io.IOException ioex) {                    //   <--- catch
38          pln("Eine Ausnahme des Typs java.io.IOException trat auf!");
39          pln(ioex.getMessage());
40       }
41       catch (java.lang.NumberFormatException nef) {         //   <--- catch
42          pln("NumberFormatException, Meldung: " + nef.getMessage());
43          pln("Diese Ausnahme wird propagiert!");
44          throw nef; // Die Ausnahme nef wird propagiert     //   <--- throw
45       }
46       finally {                                             // <--- finally
47          pln("Diese Meldung erscheint auf jeden Fall!");
48       }
49
50       // Die folgende Meldung wird nicht ausgegeben wenn das Programm mit
51       // einer NumberFormatException abgebrochen wird:
52       pln("Ausnahmen20: Das war's erstmal!");
53    } // main
54 } // class Ausnahmen20
```

1. Welche Zeilen werden zum Bildschirm ausgegeben, wenn der Benutzer (für den Lesebefehl in Zeile 30) die Zahl `6` eingibt?
2. Ebenso für die Eingabe `7`.
3. Ebenso für die Eingabe `abc`.

Hinweis: Die Meldung in einem `NumberFormatException`-Objekt sieht z.B. so aus:
`For input string: "xyz"`
wobei `"xyz"` der String ist, der nicht umgewandelt werden konnte.

Antworten zu den Wiederholungsfragen, 20. SU, Mo 21.06.10

1. Skizzieren Sie (d.h. beschreiben Sie ganz kurz) mind. zwei *Ausnahmefälle*.

Ein Programm versucht, eine Zahl einzulesen, aber der Benutzer gibt ungeeignete Daten ein.

Ein Programm versucht, aus einer Datei zu lesen, aber die Datei existiert nicht.

2. Zahlreiche ältere Programme bestehen aus einem Zebramuster von Befehlen (
ein Streifen für den Normallfall,
ein Streifen für Ausnahmefälle,
ein Streifen für den Normalfall,
ein Streifen für Ausnahmefälle,
...

Was ist der Nachteil eines solchen Zebramusters?

Die Befehle für den Normalfall sind schwer zu verstehen, weil sie dauernd durch Befehle für Ausnahmefälle unterbrochen werden.

Die Befehle für die Ausnahmefälle sind schwer zu verstehen, weil sie dauernd durch Befehle für den Normalfall unterbrochen werden.

3. Wozu hat man spezielle Befehle zum Bearbeiten von Ausnahmefällen erfunden (und u.a. in die Sprache Java aufgenommen)? Was wollte man damit dem Programmierer ermöglichen?

Die Befehle für den Normalfall von den Befehlen für die Ausnahmefälle zu trennen.

4. Angenommen, Sie haben (in einem Java-Programm) in zwei `int`-Variablen namens `erg1` und `erg2` zwei Ergebnisse berechnet, die eigentlich gleich sein müssten. Geben Sie einen Befehl an, der eine geeignete Ausnahme (z.B. des Typs `ArithmeticException`) wirft, wenn die beiden Ergebnisse verschieden sind.

if (er1 != erg2) throw new ArithmeticException(erg1 + " und " + erg2 + " sind nicht gleich!");

5. Woraus besteht ein `try-catch`-Befehl?

Aus einem try-Block und einem oder mehreren catch-Blöcken.

6. Woraus besteht ein `try-catch-finally`-Befehl?

Aus einem try-Block, null oder mehr catch-Blöcken und einem finally-Block

7. Welche Frage stellt sich der Java-Ausführer jedes Mal, wenn er eine Ausnahme a geworfen hat)?

Trat die Ausnahme a in einem try-Block auf, dem ein zum Fangen von a geeigneter catch-Block folgt?

8. Was macht der Ausführer, wenn die Antwort auf diese Frage (siehe 7.) "Ja" ist?

Er führt den geeigneten catch-Block aus. Damit ist dann der gesamte try-catch-Befehl fertig ausgeführt.

Lösung zu der Übung auf der Rückseite der Wiederholungsfragen, 20. SU, Mo 21.06.10

Die Zeilen-Nummern gehören nicht zu den Ausgaben

1. Ein Dialog mit dem Programm `Ausnahmen20` (Eingabe des Benutzers: 6)

```
 1 Ausnahmen20: Eine gerade Zahl? 6
 2 6 ist eine sehr gute Eingabe!
 3 Diese Meldung erscheint auf jeden Fall!
 4 Ausnahmen20: Das war's erstmal!
```

2. Noch ein Dialog mit dem Programm `Ausnahmen20` (Eingabe des Benutzers: 7)

```
 5 Ausnahmen20: Eine gerade Zahl? 7
 6 Ihre Eingabe 7 ist ungerade!
 7 Verflixt!!!
 8 Diese Meldung erscheint auf jeden Fall!
 9 Ausnahmen20: Das war's erstmal!
```

3. Noch ein Dialog mit dem Programm `Ausnahmen20` (Eingabe des Benutzers: abc)

```
10 Ausnahmen20: Eine gerade Zahl? abc
11 NumberFormatException, Meldung: For input string: "abc"
12 Diese Ausnahme wird propagiert!
13 Diese Meldung erscheint auf jeden Fall!
```

20. SU Mo 21.06.10

A. Wiederholung
B. Organisation

15.5 Geprüfte und ungeprüfte Ausnahmem (S. 377)

Wenn man eine Methode aufruft (z.B. eine aus der Java-Standardbibliothek) sollte man wissen, welche Typen von Ausnahmen dadurch an die Aufrufstelle geflogen kommen können.

Beispiel: Wenn man die Methode `dividiere` aufruft, können Ausnahmen der Typen `NumberFormatException` und `ArtithmetikException` an die Aufrufstelle geflogen kommen (siehe S. 365/366).

Problem: Viele Programmierer dokumentieren nicht gern.

Die Entwickler von Java wollten erzwingen, dass der Programmierer am Anfang jeder Methode wenigstens die *Typen der Ausnahmen*, die durch die Methode ausgelöst werden können, dokumentiert werden müssen.

Problem: Es gibt einige Ausnahmen, die in *jeder* Java-Methode auftreten können, z.B. Ausnahmen des Typs `OutOfMemory` ("Speicher alle"). Die müsste man dann eigentlich am Anfang *jeder* Methode angeben. Das wäre lästig und wenig nützlich.

Also werden in Java *zwei Arten von Ausnahmen* unterschieden:

*Geprüfte Ausnahmen* (die *muss* man am Anfang jeder Methode dokumentieren) und
*ungeprüfte Ausnahmen* (die darf und sollte man, muss man aber nicht dokumentieren).

Beispiele:
Ausnahmen des Typs `NumberFormatException` sind *ungeprüft*
Ausnahmen des Typs `IOException` sind *geprüft*.

S. 378, Beispiel-01: Eine Methode mit `throws`-Klausel

Die `throws`-Klausel sollte man möglichst selten mit dem `throw`-Befehl verwechseln!

Fehlermeldung des Sun-Compilers,
wenn man eine geprüfte Ausnahme weder fängt noch (in der `throws`-Klausuel) dokumentiert:

```
D:\meineDateien\Ausnahme01.java:11:
unreported exception java.lang.Exception; must be caught or declared
to be thrown
      throw new Exception("Diese Ausnahme ist geprueft!");
      ^
```

Die Hierarchie der Ausnahmeklassen (S. 382)

Def.: Eine Ausnahmeklasse ist eine Unterklasse der Klasse `Throwable`.

Eine Klasse ist *geprüft/ungeprüft*, wenn sie eine *geprüfte/ungeprüfte* Klasse erweitert.

Ausnahmen: Die Klassen `Error` und `Runtime`

Solche Klassen (die ungeprüft sind, obwohl ihre direkte Oberklasse geprüft ist) kann ein normaler Programmierer nicht schreiben (nur die Entwickler der Sprache Java konnten das).

Ausnahme in einem catch-Block werfen

S. 376, Beispiel-01: Eine Ausnahme fangen und erneut werfen

S, 376, Beispiel-02: Eine Ausnahme "umwandeln" (eine fangen, eine andere werfen)

Die Übung auf der Rückseite der Wiederholungsfragen bearbeiten

Zur Entspannung: Grundlagen für das Verständnis englischer Fachbücher

Um angloamerikanische Fachbücher verstehen zu können, sollte man sich als Grundlage (unter anderem) mit folgenden Büchern vertraut machen:

1. "Winnie-the-Pooh" von A. A. Milne, illustrated von E. H. Shepard.

Z. B. geht der Fachbegriff "a Pooh problem" auf Piglets Frage "Do bees sting?" und Poohs Antwort: "Some do, and some don't!" zurück.

2. "Alice's Adventures in Wonderland and Through the Looking Glass" von Lewis Carroll.

Als Beispiel ein wichtiges Zitat: "When *I* use a word," Humpty Dumpty said, in a rather scornful tone, "it means just what I choose it to mean - neither more nor less.". "The question is," said Alice, "whether you *can* make words mean so many different things."

Siehe auch: Martin Gardner: "The Annotated Alic", W. W. Norton & Company

3. "The Hitch Hikers Guide to the Galaxy" von Douglas Adams (und die übrigen vier Bände dieser Trilogie).

Zahlreiche englische Artikel und Vorträge beginnen mit der (oder erwähnen die) Antwort 42 und erklären dann die dazugehörige Frage.

Die Reihenfolge der catch-Blöcke ist wichtig!

S. 368, Beispiel-03

Welche Ausnahmen werden vom zweiten catch-Block gefangen?

Was passiert, wenn man die beiden catch-Blöcke vertauscht?

Fehlermeldung des Sun-Compilers, wenn man eine geprüfte Ausnahme weder fängt noch dokumentiert:

```
D:\meineDateien\Ausnahme01.java:11:
unreported exception java.lang.Exception; must be caught or declared
to be thrown
      throw new Exception("Diese Ausnahme ist geprueft!");
      ^
```

15.6 Eigene Ausnahmeklassen vereinbaren

S. 379, Beispiel-01: Eigene Ausnahmeklassen vereinbaren (4 Stück)

Diese Ausnahmeklassen definieren *keine neuen Elemente*, bilden aber eine kleine Hierarchie.

S. 380, Beispiel-02: Ausnahmen selbst vereinbarter Klassen werfen

S. 380, Beispiel-03: Ausnahmen selbst vereinbarter Klassen fangen und behandeln

S. 381, Beispiel-04: Ausnahmeklassen mit zusätzlichen Elementen vereinbaren

Wiederholungsfragen, 21. SU, Di 22.06.10

1. In Java unterscheidet man *geprüfte* und *ungeprüfte* Ausnahmen. Angenommen, der Programmierer schreibt eine Methode, in der eventuell eine *geprüfte Ausnahme* geworfen wird. Was muss er dann mit dieser Ausnahme machen?

2. Beantworten Sie die vorige Frage auch auf *Englisch*.

3. Wie heißen die beiden ungeprüften Ausnahmeklassen, deren direkten Oberklassen geprüft sind? Falls Sie die Seiten 382/383 des Buches immer noch nicht auswendig gelernt haben, dürfen Sie sie jetzt aufschlagen.

4. Was ist unsinnig an dem folgenden `try-catch`-Befehl?

```
1    try {
2       ...
3    } catch (RuntimeException ex) {
4       pln("Eine RuntimeException trat auf!");
5    } catch (ArithmeticException ex) {
6       pln("Eine ArithmeticException trat auf!");
7    )
```

5. Schreiben Sie einen `try-catch`-Block, der Ausnahmen des Typs `Exception` in Ausnahmen des Typs `Error` "umwandelt". Deuten Sie dabei die Befehle im `try-Block` durch drei Pünktchen `...` an (wie bei der vorigen Frage in Zeile 2).

6. In Java gibt es unter anderem die beiden Schlüsselworte `throw` und `throws`.

Welches leitet eine spezielle Art von Kommentar ein?

Welches bezeichnet eine Anweisung?

Rückseite der Wiederholungsfragen, 21. SU, Di 22.06.10

Übung Reihungen 3:

*Mehrstufige Reihungen* werden im Abschnitt 7.4 des Buches behandelt.

Betrachten Sie die folgenden fünf Vereinbarungen von *zweistufigen* Reihungsvariablen:

```
 1 int[][] irrA = null;                      // Keine Reihung
 2 int[][] irrB = new int[3][];              // 2-stufige Reihung
 3 int[][] irrC = new int[3][2];             // 2-stufige Reihung
 4 int[][] irrD =               { {11, 12,}, // 2-stufige Reihung
 5                                {21, 22,},
 6                                {31, 32,},
 7                              };
 8 int[][] irrE = new int[][]   { {41, 42,}, // 2-stufige Reihung
 9                                {51, 52, 53},
10                                {},
11                                null,
12                              };
```

Stellen Sie die fünf Variablen `irrA` bis `irrE` als *Bojen* dar. Die *vereinfachte Bojendarstellung* (ohne die Referenzen von Komponentenvariablen und ohne das `length`-Attribut) ist hier ausdrücklich *erlaubt*.

Antworten zu den Wiederholungsfragen, 21. SU, Di 22.06.10

1. In Java unterscheidet man *geprüfte* und *ungeprüfte* Ausnahmen. Angenommen, der Programmierer schreibt eine Methode, in der eventuell eine *geprüfte Ausnahme* geworfen wird. Was muss er dann mit dieser Ausnahme machen?

Er muss sie entweder fangen oder (in einer throws-Klausel) erklären, dass sie geworfen wird.

2. Beantworten Sie die vorige Frage auch auf Englisch.

He must catch the exception or declare it (in a throws clause) to be thrown.

3. Wie heißen die beiden ungeprüften Ausnahmeklassen, deren direkten Oberklassen geprüft sind? Falls Sie die Seiten 382/383 des Buches immer noch nicht auswendig gelernt haben, dürfen Sie sie jetzt aufschlagen.

Error und RuntimeException

4. Was ist unsinnig an dem folgenden `try-catch`-Befehl?

```
1    try {
2       ...
3    } catch (RuntimeException ex) {
4       pln("Eine RuntimeException trat auf!");
5    } catch (ArithmeticException ex) {
6       pln("Eine ArithmeticException trat auf!");
7    )
```

Der zweite catch-Block wird nie benutzt, weil ArtihneticException einer Unterklasse von RuntimeException ist.

5. Schreiben Sie einen `try-catch`-Block, der Ausnahmen des Typs `Exception` in Ausnahmen des Typs `Error` "umwandelt". Deuten Sie dabei die Befehle im `try`-Block durch drei Pünktchen `...` an (wie bei der vorigen Frage in Zeile 2).

```
 8    try {
 9       ...
10    } catch (Exception ex) {
11       throw new Error(ex); // ex wird als Ursache (cause) in die neue
12                            // Ausnahme eingebaut.
13    }
```

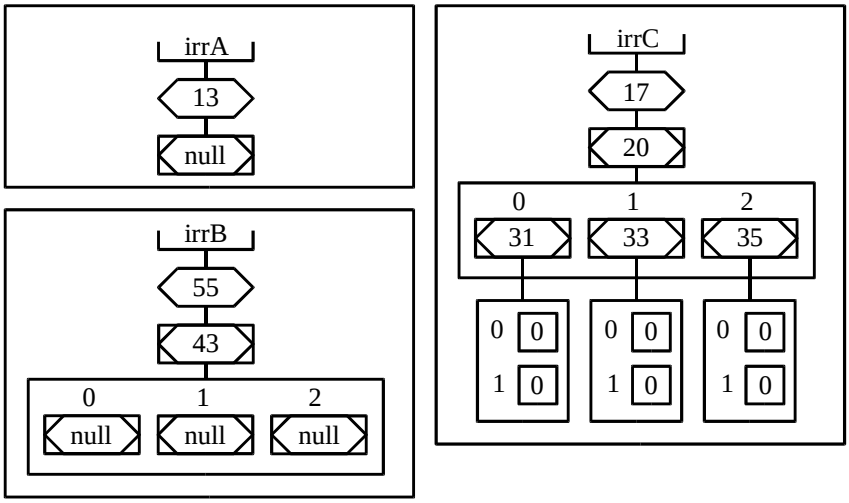
6. In Java gibt es unter anderem die beiden Schlüsselworte `throw` und `throws`.

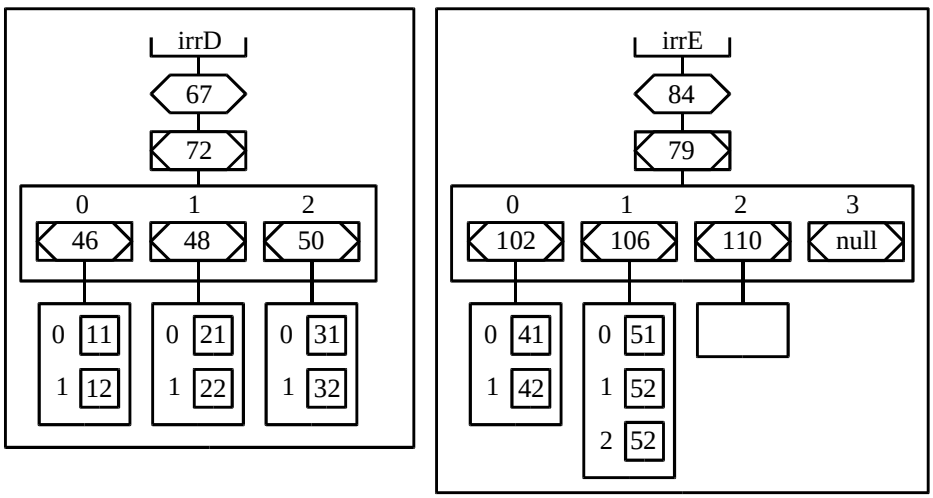
Welches leitet eine spezielle Art von Kommentar ein? throws

Welches bezeichnet eine Anweisung? throw

Lösung zur Übung Reihungen 3:

Die fünf zweistufigen Reihungsvariablen `irrA` bis `irrE` als (vereinfachte) Bojen dargestellt:



21. SU Di 22.06.10

A. Wiederholung
B. Organisation

1. Die in der gestrigen Vorlesung begonnene Übung vervollständigen

Ein Programm, in dem Ausnahmen geworfen und gefangen werden, mehrmals ausführen.

2. Die Übung auf der Rückseite der heutigen Wiederholungsfragen bearbeiten

5 Reihungsvariablen (mehrstufge) als Bojen darstellen.

Zur Entspannung: Die Regel von Moore und Hardware-Tendenzen

1965 formulierte Gordon Moore (einer der Gründer der Firma Intel) eine einfache Regel: "Etwa alle 2 Jahre verdoppelt sich die Anzahl der Transistoren, die auf einen Chip passen."

Diese einfache Regel ist weder ein juristisches noch ein physikalisches *Gesetz*, aber als "Daumenregel" sehr nützlich. Allerdings hat man sie inzwischen neueren Entwicklungen angepasst: Anstelle von "2 Jahren" nimmt man heute etwa 18 Monate an. Offenbar gilt heute (2010):

1. Prozessoren werden kaum noch schneller.
2. Prozessoren (in einem Rechner) werden aber zahlreicher.
3. Man hofft, dass die Rechner dadurch schneller werden.

Damit wird das Konzept der *Nebenläufigkeit* noch wichtiger als es schon war.

Die nächste einschneidend wichtige Hardware-Entwicklung: Nicht-flüchtige Hauptspeicher. Hoffnung: Kein Hochfahren (booten) mehr notwendig. Wenn man einen Rechner anschaltet, ist er nach weniger als einer Sekunde betriebsbereit (und macht genau da weiter, wo er beim Ausschalten war).

Wiederholungsfragen, 22. SU, Mo 28.06.10

1. Welche Frage stellt sich der Ausführer jedes Mal, wenn er gerade eine Ausnahme a geworfen hat?
2. Was macht er, wenn die Antwort auf diese Frage (siehe 1.) "ja" ist?
3. Was macht er, wenn die Antwort auf diese Frage (siehe 1.) "nein" ist?
4. Stellen Sie die folgenden Variablen als Bojen dar:

```
1    int   []     r1 = null;
2    int   [][][] r2 = null;
3    String[]     r3 = null;
4    String[][][] r4 = null;
```

Antworten zu den Wiederholungsfragen, 22. SU, Mo 28.06.10

1. Welche Frage stellt sich der Ausführer jedes Mal, wenn er gerade eine Ausnahme a geworfen hat?

Trat die Ausnahme a in einem try-Block auf, dem ein zum Fangen von a geeigneter catch-Block folgt?

2. Was macht er, wenn die Antwort auf diese Frage (siehe 1.) "ja" ist?

Er führt den betreffenden catch-Block aus und macht dann hinter dem betreffenden try-catch-Befehl weiter.

3. Was macht er, wenn die Antwort auf diese Frage (siehe 1.) "nein" ist?

Er bricht die aktuelle Methode ab, kehrt zurück an die Stelle, an der sie aufgerufen wurde und wirft dort die Ausnahme erneut.

4. Stellen Sie die folgenden Variablen als Bojen dar:

```
1    int   []     r1 = null;
2    int   [][][] r2 = null;
3    String[]     r3 = null;
4    String[][][] r4 = null;
```

Die Bojen:

```
1 |r1|--<10>--[<null>]
1 |r2|--<20>--[<null>]
1 |r3|--<30>--[<null>]
1 |r4|--<40>--[<null>]
```

22. SU Mo 28.06.10

A. Wiederholung
B. Organisation

Ströme (S. 470)

In Java wird das Einlesen und Ausgeben von Daten meistens mit Hilfe von sogenannten *Strömen* (engl. streams) durchgeführt. Das sind Objekte bestimmter Klassen, die man als *Stromklassen* (engl. stream classes) bezeichnet.

Ein Strom-Objekt kann man sich als eine *Röhre* vorstellen, durch die Daten fließen können. Dabei werden die Daten vom Strom-Objekt "ein bisschen bearbeitet". Typische Bearbeitungsschritte:

- Umwandlungen zwischen verschiedenen Typen (z.B. von `int` nach `Reihung von char` oder von `Reihung von char` in `Reihung von byte`)
- Umwandlungen zwischen verschiedenen Zeichencodes (z.B. von einem 8-Bit-ASCII-Code in den Unicode oder umgekehrt)
- Pufferung (Daten werden in einem Puffer gesammelt und erst dann weitertransportiert,
- wenn der Puffer voll ist)
- ... etc.

Außerdem bilden Strom-Objekte eine Art *Baukasten*: Bestimmte Stromobjekte kann man miteinander kombinieren ("aneinanderschrauben") und so
aus *mehreren* Röhren, die einfache Bearbeitungsschritte durchführen,
*eine* Röhre erstellen, die insgesamt eine komplexe Bearbeitung durchführt.

Noch zwei Fachbegriffe:

Eine *Datenquelle* ist irgendein Ding, von dem man Daten einlesen kann
(z.B. eine Datei oder ein String oder eine Reihung von byte oder eine Adresse im Internet)

Eine *Datensenke* ist irgendein Ding, zu dem man Daten ausgeben kann
(z.B. eine Datei oder ein StringBuilder oder eine Reihung von byte oder eine Adresse im Internet)

S. 472, Beispiel-02 (sollte eigentlich -01 heißen)

Textdaten (`String`-Objekte und `char`-Werte) in eine Datei ausgeben

Ströme, die aus "mehreren Stromobjekten zusammengeschraubt sind", kann man durch sehr einfache (Datenfluß-) Grafen darstellen, den Strom aus dem Beispiel-02 etwa so:

```
Datei Stroeme01.tmp <-- felix <-- oskar <-- bruno <-- Programm
```

Konvention: Die Datenquelle oder Datensenke steht immer ganz links, das Programm ganz rechts!

Das gilt auch für Eingabströme (bei denen die Pfeile dann von links nach rechts zeigen)!

S. 473, Beispiel-03: Daten zeilenweise aus einer Datei lesen

Das `InputStreamReader`-Objekt `ilse` enthält (etwas vereinfacht gesagt) nur eine Methode `read`, mit der man einzelne Zeichen lesen kann.

Das `BufferedReader`-Objekt `britta` enthält auch eine komfortablere Methode `readLine`, mit der man eine ganze *Zeile* auf einmal lesen kann (ohne dass man sich selbst um das lästige Zeilen-Ende-Problem kümmern muss).
Darum wurde im Beispiel-03 der Strom `britta` an den Strom `ilse` "drangeschraubt".

Zur Entspannung: Einen selbstbezüglichen Satz wahr machen

Einfache Beispiele für (wahre bzw. falsche) *selbstbezügliche Sätze*:

```
"Dies ist ein Satz der deutschen Sprache!"     wahr
"This is a sentence of the Russian language!"  falsch
"Dieser Satz besteht aus 35 Zeichen!"          wahr (oder hab ich mich verzählt?)
"This sentence consists of 100 characters!"    falsch
```

Bei den letzten beiden Sätzen sollen auch die Blanks zwischen den Worten als je ein Zeichen gezählt werden (sonst sollte man im vorletzten Satz die 35 durch 30 ersetzen). Betrachten Sie den folgenden (selbstbezüglichen) Satz:

```
"Dieser Satz enthaelt
 genau n0 mal die Ziffer 0,
 genau n1 mal die Ziffer 1,
 genau n2 mal die Ziffer 2,
 genau n3 mal die Ziffer 3,
 genau n4 mal die Ziffer 4,
 genau n5 mal die Ziffer 5,
 genau n6 mal die Ziffer 6,
 genau n7 mal die Ziffer 7,
 genau n8 mal die Ziffer 8,
 genau n9 mal die Ziffer 9!"
```

Können Sie die Namen `n0` bis `n1` so durch Ziffern zwischen 0 ind 9 ersetzen, dass der Satz wahr wird? Ein Programm schreiben, welches Lösungen findet? (Eine Lösung: 1, 7, 3, 2, 1, 1, 1, 2, 1, 1).

## 22. Grafische Benutzeroberflächen (Grabos)

Konsole: Ein *zeichenorientierter Bildschirm* (z.B. 25x80 Zeichen) oder ein entsprechendes Fenster auf einem grafischen Bildschirm, plus eine *Tastatur*.

Ein *sequentieller Ausführer* ist ein Ausführer, der Befehle eines Programms einen nach dem anderen, in einer (von der betreffenden Programmiersprache) genau *festgelegten Reihenfolge* ("sequentiell") ausführt.

Ein *Konsolen-Programm* ist typischerweise ein *sequentielles Programm* (alle Befehle werden von *einem* sequentiellen Ausführer ausgeführt), Interaktionen zwischen Programm und Benutzer (z.B. die Eingabe von Daten) werden *vom Programm aus gesteuert.*

Ein *Grabo-Programm* (ein Programm mit Grabo) ist ein *nebenläufiges Programm* (die Befehle des Programms werden von *mehreren* sequentiellen Ausführern ausgeführt, die *nebenläufig* zueinander arbeiten), Interaktionen zwischen Programm und Benutzer werden *vom Benutzer gesteuert.*

Anmerkung: In den 1970-er Jahren verbreitete IBM sogenannte *transaktionsorientierte Dialog-Programme* (mit einem System namens CICS: Customer Information Control System). Diese CICS-Programme stellen eine Art "Zwischenform" zwischen Konsolen-Programmen und Grabo-Programmen dar. Sie hatten noch keine Grabo, aber der Benutzer kontrollierte schon die Interaktionen zwischen sich und dem Programm.

Wichtig: Grabo-Programme sind *ganz andere Tiere* als Konsolen-Programme. Sie werden nach ganz anderen Regeln ausgeführt und müssen entsprechend ganz anders programmiert werden.

Def.: Ein *Grabo-Objekt* ist Objekt, welches nach seiner Erzeugung (mehr oder weniger automatisch) auf dem Bildschirm grafisch dargestellt wird.

Beispiele:

Ein `JFrame`-Objekt wird als ein *Fenster* dargestellt (mit einem Rahmen, einem Titel und drei Knöpfchen im Rahmen zum Maximieren, Minimieren bzw. Schließen des Fensters).

Ein `JButton`-Objekt wird als ein *Knopf* (engl. button) dargestellt.

## 22.1 Ein paar kleine Grabo-Programme

S. 531, Beispiel-01: Zwei `JFrame`-Objekte erzeugen und sichtbar machen

Zeile 1: `mr1` ohne Titel
Zeile 2: `mr2` mit Titel
Zeile 4-5: `setBounds`
Zeile 7-8: `setDefaultCloseOperation`
Zeile 10-11: `setVisible`

S. 531, Bild 22.1: Grafische Darstellung der beiden `JFrame`-Objekte auf dem Bildschirm

Wiederholungsfragen, 23.SU, Di 29.06.10

1. Betrachten Sie die folgenden Vereinbarungen von Strom-Objekten:

```
1    String           pfad = "D:/tmp/Datei17.txt";
2    FileReader       fr   = new FileReader(pfad);
3    LineNumberReader lr   = new LineNumberReader(br);
```

Stellen Sie den Gesamtstrom ("der hier zusammengeschraubt wurde") als einfache (Datenfluß-) Grafik dar.

2. Erläutern Sie kurz, was (im obigen Gesamtstrom) der Strom `fr` leistet.

3. Erläutern Sie kurz, was (im obigen Gesamtstrom) der Strom `lr` leistet.

4. Beschreiben Sie charakteristische Eigenschaften eines Konsolen-Programms.

5. Beschreiben Sie charakteristische Eigenschaften eines Grabo-Programms (engl. of a GUI program).

6. Was ist ein Grabo-Objekt (engl.: a GUI object)?

Rückseite der Wiederholungsfragen, 23.SU, Di 29.06.10

Anonyme Objekte und anonyme Klassen

Beispiele für *anonyme Objekte*:

```
 1    pln(new String("Hallo"));
 2    pln(new Punkt3D(1.0, 2.0, 3.0));
 3    String s = Arrays.toString(new int[]{10, 20, 30});
 4    this.addWindowListener(new ProgTerminator()); // siehe S. 536, Z. 63
```

Beispiele für benannte Objekte *anonymer Klassen*:

```
 5    Punkt3D pEn = new Punkt3D(1.0, 2.0, 3.0){
 6       public String toString() {
 7          return super.toString().replace("Punkt", "Point");
 8       }
 9    };
10
11    static WindowAdapter arnold = new WindowAdapter() {
12       public void windowClosing(WindowEvent we) {
13          pln( ... )
14          System.exit(0);
15       }
16    };
```

Beispiel für *ein anonymes Objekt einer anonymen Klasse*:

```
17    pln(new Punkt3D(1.0, 2.0, 3.0){
18       public String toString() {
19          return super.toString().replace("Punkt", "Point");
20       }
21    });
```

Fragen zu einigen Zeilen:

1 Was wird ausgegeben?
2 Was wird ausgegeben?
3 Welchen Zielwert bekommt die Variable s?
4 Ein Objekt von welchem Typ wird hier angemeldet? Bei welchem Objekt wird es angemeldet?

5 Nach welchem Bauplan wird das Objekt, welches hier nit `new` erzeugt wird, gebaut?
Wodurch unterscheidet sich dieses Objekt von einem Punkt3D-Objekt?

11 Nach welchem Bauplan wird das Objekt, welches hier nit `new` erzeugt wird, gebaut?
Wodurch unterscheidet sich dieses Objekt von einem `WindowAdapter`-Objekt?

17 Was wird ausgegeben? Was hat diese Ausgabe mit der Variablen `pEn` (siehe Zeile 5) zu tun?

Anmerkung:

In einem Java-Quellprogramm ist die Zeichenfolge `});` ist ein typisches Kennzeichen für die Benutzung eines *anonymen Objekts* einer *anonyomen Klasse*.

Antworten zu den Wiederholungsfragen, 23.SU, Di 29.06.10

1. Betrachten Sie die folgenden Vereinbarungen von Strom-Objekten:

```
1    String           pfad = "D:/tmp/Datei17.txt";
2    FileReader       fr   = new FileReader(pfad);
3    BufferedReader   br   = new BufferedReader(fr);
4    LineNumberReader lr   = new LineNumberReader(br);
```

Stellen Sie den Gesamtstrom ("der hier zusammengeschraubt wurde") als einfache (Datenfluß-) Grafik dar.

**`Datei17.txt --> fr --> lr -> Programm`**

2. Erläutern Sie kurz, was (im obigen Gesamtstrom) der Strom `fr` leistet.

Er wird an die Datei17.txt "angeschraubt" und kann aus ihr Daten lesen.

43. Erläutern Sie kurz, was (im obigen Gesamtstrom) der Strom `lr` leistet.

Er enthält eine komfortabele readLine-Methode, mit der man eine ganze Zeile auf einmal lesen kann (ohne sich um das lästige Zeilen-Ende-Problem kümmern zu müssen). Ausserdem enthält das Objekt lr eine Funktion getLineNr, die (bei jedem Aufruf) die aktuelle Zeilen-Nummer liefert.

4. Beschreiben Sie charakteristische Eigenschaften eines Konsolen-Programms.

Es ist sequentiell. Interaktionen mit dem Benutzer werden vom Programm gesteuert.

5. Beschreiben Sie charakteristische Eigenschaften eines Grabo-Programms (engl. of a GUI program).

Es ist nebenläufig (d.h. es wird von mehreren sequentiellen Ausführern ausgeführt). Interaktionen mit dem Benutzer werden vom Benutzer gesteuert.

6. Was ist ein Grabo-Objekt (engl.: a GUI object)?

Ein Objekt, welches eine grafische Darstellung besitzt, die beim Erzeugen des Objekts (mehr oder weniger) automatisch auf dem Bidlschirm erscheint.

Erläuterungen zur Rückseite der Wiederholungsfragen, 23.SU, Di 29.06.10

1 Ausgegeben wird `Hallo`

2 Ausgegeben wird `Punkt: (1,0, 2,0, 3,0)`

3 Die Variable `s` hat den Zielwert `"[10, 20, 30]"`

4 Ein neues `ProgTerminator`-Objekt wird erzeugt und beim aktuellen Objekt (`this`) angemeldet

5 `pEn` ist ein Objekt einer Erweiterung der Klasse `Punkt3D`. Diese Erweiterung überschreibt die geerbte (deutsche) `toString`-Methode mit einer englischen `toString`-Methode.

11 `arnold` ist ein Objekt einer Erweiterung der Klasse `WindowAdapter`. Diese Erweiterung überschreibt die geerbte Methode `windowClosing` (die einen leeren Rumpf hat) mit einer neuen Methode (die einen nicht-leeren Rumpf hat, siehe Zeile 13-14).

17 Ausgegeben wird `Point: (1,0, 2.0, 3.0)`.
Der Befehl `pln(pEn);` würde den gleichen Text ausgeben.

23. SU Di 29.06.10

A. Wiederholung
B. Organisation

S. 533, Beispiel-02: Eine selbst vereinbarte `JFrame`-Klasse `Grabo01`

Was wird in Zeile 14-17 vereinbart? (Ein Konstruktor)
Was macht der Befehl in Zeile 15? (Ruft einen Konstruktor der Klasse `JFrame` auf).
Die Objektmethode `getContentPane`

Die `paint`-Methode wird vom *Programmierer* vereinbart, aber nur vom *Ausführer* aufgerufen und dann ausgeführt (wenn er meint, dass es wieder mal Zeit dafür ist)

Der Parameter der `paint`-Methode (ein `Graphics`-Objekt) repräsentiert ein Stück Bildschirm.

Erklären, wann der Ausführer die `paint`-Methode aufruft.

Wichtige Regel: In der `paint`-Methode muss man als erstes die geerbte ( in der Klasse `JFrame` vereinbarte) `paint`-Methode aufrufen! Die zeichnet das `JFrame`-Objekt (das Fenster mit Rahmen, Titel und drei Knöpfchen).

Achtung: Die Klasse `Grabo01` ist nur ein Bauplan (für Objekte), aber kein Programm.

S. 534, Beispiel-03: Ein Beispielprogramm mit zwei `Grabo01`-Fenstern

S. 535, Bild 22.2 Die grafische Darstellung der beiden `Grabo01`-Objekte auf dem Bildschirm

Mit einer Aktion des Benutzers eine bestimmte Methode verbinden

Ziel: Wir wollen eine Erweiterung der Klasse `JFrame` namens `Grabo02` vereinbaren. Wenn der Benutzer auf den Fenster-Schließen-Knopf eines `Grabo02`-Objekts klickt, soll eine *von uns* programmierte Methode ausgeführt werden.

S. 536. Beispiel-04: Die Klasse `Grabo02`

Das Profil der Methode ist schon festgelegt:

```
public void windowClosing(WindowEvent we)
```

Wir müssen folgendes machen:

1. Eine *Klasse* als Erweiterung der Klasse `WindowAdapter` vereinbaren (z.B. `ProgTerminator`)
2. Darin die Methode `windowClosing` vereinbaren (z.B. `arnold`)
3. Ein *Objekt* dieser Klasse erzeugen.
4. Dieses Objekt bei dem betreffenden `Grobo02`-Objekt *anmelden* (mit `addWindowListener`)

Problem: Um eine Methode mit einer Benutzer-Aktion zu verbinden, mussten wir
einen Klassen-Namen (`ProgTerminator`) und
einen Variablen-Namen (`arnold`)
festlegen.
Den Klassen-Namen (`ProgTerminator`) benutzen wir nur einmal (um das Objekt zu erzeugen).
Den Objekt-Namen (`arnold`) benutzen wir nur einmal (um das Objekt mit der Methode `addWindowListener` beim aktuellen `Grabo02`-Objekt anzumelden).

In großen Grabo-Programmen will man -zig oder hunderte von Benutzer-Aktionen mit entsprechend vielen Methoden verbinden. Dafür müsste man sich -zig oder hunderte von Klassen- und Variablen-*Namen* ausdenken.

## Anonyme Objekte und anonyme Klassen

Die Rückseite der Wiederholungsfragen bearbeiten. Die dort stehenden Fragen stellen und gemeinsam beantworten.

S.538, Beispiel-05, ab Zeile 77

Noch eine Aktion des Benutzers (auf einen bestimmten Knopf klicken oder `Alt-k` eintippen) mit einer Methode verbinden:
S. 539, Beispiel-07

Zur Entspannung: Mega-, giga-, ... und wie geht es dann weiter?
Dazu ein Blatt verteiln und besprechen.

Wiederholungsfragen, 24. SU, Mo 05.07.10

1. Was ist ein *Grabo-Objekt*? Geben Sie eine *inhaltliche* Definition an (die "auf weiche Weise" die wichtigsten Eigenschaften eines Grabo-Objekts beschreibt).

2. Was ist eine *Grabo-Klasse*? Geben Sie eine *formale* Definition an (die ganz kurz und "auf harte Weise" Grabo-Klassen von anderen Klassen unterscheidet).

3. Beschreiben Sie kurz zwei Beispiele für *Aktionen*.

4. Was ist der wichtigste Unterschied zwischen der `paint`-Methode in einem Grabo-Objekt und einer anderen, "normalen" Methode?

5. Die `paint`-Methode hat einen Parameter. Von welchem Typ ist der?

6. Was ist besonders an dem Parameter, mit dem die `paint`-Methode normalerweise aufgerufen wird? Warum kann der Programmierer einen solchen Parameter nicht "nachmachen" oder erzeugen lassen?

7. Geben Sie (mit der Methode `pln`) ein *anonymes Objekt* vom Typ `E01Punkt` aus (die Vereinbarung der Klasse steht im Buch auf S. 287).

8. Betrachten Sie im Buch auf S. 536, Beispiel-04, die Zeile 63. Durch diesen Befehl wird ein *Behandlerobjekt* bei einem *Grabo-Objekt* angemeldet.

8.1. Von welchem *Typ* ist das *Grabo-Objekt*, bei dem das Behandler-Objekt angemeldet wird?
8.2. Von welchem *Typ* ist *Behandler-Objekt*, welches beim Grabo-Objekt angemeldet wird?

9. Betrachten Sie im Buch auf S. 538, Beispiel-05, die Zeilen 77 bis 82. Durch diesen Befehl wird ein *Behandlerobjekt* bei einem *Grabo-Objekt* angemeldet.

9.1. Von welchem *Typ* ist das *Grabo-Objekt*, bei dem das Behandler-Objekt angemeldet wird?
9.2. Von welchem *Typ* ist *Behandler-Objekt*, welches beim Grabo-Objekt angemeldet wird?

Rückseite der Wiederholungsfragen, 24. SU, Mo 05.07.10

Die Standardbibliothek von Java 6 enthält zur Zeit (Juli 2010) 72 Unterschnittstellen (subinterfaces) der Schnittstelle `EventListener`. Diese (Unter-) Schnittstellen spielen eine zentrale Rolle bei der *Ereignisbehandlung* (vor allem in Grabo-Programmen). Im Buch werden 5 dieser Schnittstellen beispielhaft behandelt. Ihre Namen sind in der folgenden Liste fett hervorgehoben.

```
 1 Action                                LineListener
 2 ActionListener                        ListDataListener
 3 AdjustmentListener                    ListSelectionListener
 4 AncestorListener                      MenuDragMouseListener
 5 AWTEventListener                      MenuKeyListener
 6 BeanContextMembershipListener         MenuListener
 7 BeanContextServiceRevokedListener     MetaEventListener
 8 BeanContextServices                   MouseInputListener
 9 BeanContextServicesListener           MouseListener
10 CaretListener                         MouseMotionListener
11 CellEditorListener                    MouseWheelListener
12 ChangeListener                        NamespaceChangeListener
13 ComponentListener                     NamingListener
14 ConnectionEventListener               NodeChangeListener
15 ContainerListener                     NotificationListener
16 ControllerEventListener               ObjectChangeListener
17 DocumentListener                      PopupMenuListener
18 DragGestureListener                   PreferenceChangeListener
19 DragSourceListener                    PropertyChangeListener
20 DragSourceMotionListener              RowSetListener
21 DropTargetListener                    RowSorterListener
22 FlavorListener                        SSLSessionBindingListener
23 FocusListener                         StatementEventListener
24 HandshakeCompletedListener            TableColumnModelListener
25 HierarchyBoundsListener               TableModelListener
26 HierarchyListener                     TextListener
27 HyperlinkListener                     TreeExpansionListener
28 IIOReadProgressListener               TreeModelListener
29 IIOReadUpdateListener                 TreeSelectionListener
30 IIOReadWarningListener                TreeWillExpandListener
31 IIOWriteProgressListener              UndoableEditListener
32 IIOWriteWarningListener               UnsolicitedNotificationListener
33 InputMethodListener                   VetoableChangeListener
34 InternalFrameListener                 WindowFocusListener
35 ItemListener                          WindowListener
36 KeyListener                           WindowStateListener
```

Antworten zu den Wiederholungsfragen, 24. SU, Mo 05.07.10

1. Was ist ein *Grabo-Objekt*? Geben Sie eine *inhaltliche* Definition an (die "auf weiche Weise" die wichtigsten Eigenschaften eines Grabo-Objekts beschreibt).

Ein Grabo-Objekt ist ein Objekt, für das eine grafische Darstellung festgelegt ist und welches, wenn man es (mit new) erzeugt, mehr oder weniger automatisch auf dem Bildschirm dargestellt wird.

2. Was ist eine *Grabo-Klasse*? Geben Sie eine *formale* Definition an (die ganz kurz und "auf harte Weise" Grabo-Klassen von anderen Klassen unterscheidet).

Eine Grabo-Klasse ist eine Unterklasse der Klasse java.awt.Component.

3. Beschreiben Sie kurz zwei Beispiele für *Aktionen*.

- Der Benutzer bewegt den Mauszeiger quer über ein Fenster
- Der Benutzer klickt einen Knopf an
- Der Benutzer drückt auf eine Taste der Tastatur
- ...

4. Was ist der wichtigste Unterschied zwischen der `paint`-Methode in einem Grabo-Objekt und einer anderen, "normalen" Methode?

Die paint-Methode wird (wie andere Methoden auch) vom Programmierer vereinbart, aber (anders als andere Methoden) nur vom Ausführer aufgerufen, nicht vom Programmierer.

5. Die `paint`-Methode hat einen Parameter. Von welchem Typ ist der?

java.awt.Graphics

6. Was ist besonders an dem Parameter, mit dem die `paint`-Methode normalerweise aufgerufen wird? Warum kann der Programmierer einen solchen Parameter nicht "nachmachen" oder erzeugen lassen?

Der Parameter repräsentiert einen Teil des Bildschirms. Solche Graphics-Objekte kann nur der Ausführer (genauer: das Betriebssystem) erzeugen.

7. Geben Sie (mit der Methode `pln`) ein *anonymes Objekt* vom Typ `E01Punkt` aus (die Vereinbarung der Klasse steht im Buch auf S. 287).

pln(new E01Punkt(1.0, 2.0));

8. Betrachten Sie im Buch auf S. 536, Beispiel-04, die Zeile 63. Durch diesen Befehl wird ein *Behandlerobjekt* bei einem *Grabo-Objekt* angemeldet.

8.1. Von welchem *Typ* ist das *Grabo-Objekt*, bei dem das Behandler-Objekt angemeldet wird?

Vom Typ Grabo02

8.2. Von welchem *Typ* ist *Behandler-Objekt*, welches beim Grabo-Objekt angemeldet wird?

Vom Typ ProgTerminator

9. Betrachten Sie im Buch auf S. 538, Beispiel-05, die Zeilen 77 bis 82. Durch diesen Befehl wird ein *Behandlerobjekt* bei einem *Grabo-Objekt* angemeldet.

9.1. Von welchem *Typ* ist das *Grabo-Objekt*, bei dem das Behandler-Objekt angemeldet wird?

Vom Typ Grabo03

9.2. Von welchem *Typ* ist *Behandler-Objekt*, welches beim Grabo-Objekt angemeldet wird?

Von einem (anonymen) Untertyp des Typs WindowAdapter.

24. SU Mo 05.07.10

A. Wiederholung
B. Organisation:
Wir haben noch 3 SU-Termine vor uns, und am Di 13.07.10 (18 Uhr, B554) schreiben wir die Klausur.
Übung und Vorlesung am Di 13.07.10: Finden nur statt, wenn mind. 5 TeilnehmerInnen das möchten (und mir das per Email mit mind. 5 Namen darauf mitteilen).

Aktionen, Ereignisse, Behandlermethoden

1. In einem Programm wird ein `JFrame-Objekt` namens `mr1` erzeugt.
Auf dem Bildschirm wird ein entsprechendes Fenster dargestellt
u.a. mit einem Fenster-Schließen-Knopf (oben rechts) im Rahmen.

2. Der Benutzer kann jetzt u.a. eine Fenster-Schließen-Aktion ausführen
(indem er mit dem Mauszeigen auf den Fenster-Schließen-Knopf von klickt)

3. Dadurch wird im im `JFrame`-Objekt `mr1` ein Ereignis der Art windowClosing erzeugt.

4. Ereignisse der *Art* `windowClosing` gehören zur Oberart *Fensterereignis*. Zu dieser Oberart gehören insgesamt 7 Arten von Ereignissen (ausser `windowClosing` auch noch `windowOpened`, `windowIconified` etc.)

5. Wenn ein Ereignis einer dieser 7 Arten erzeugt wird, sucht der Ausführer nach einem Behandlerobjekt für Fensterereignisse. Ein solches Behandlerobjekt muss 7 Methoden namens `windowClosing`, `windowOpened`, `windowIconified`, ... enthalten.

6. Der Ausführer ruft in jedem angemeldeten *Behandlerobjekt für Fensterereignisse* die Methode `windowClosing` auf und führt sie aus.

7. Die Methode namens `windowClosing` wird auch als Behandlermethode (für Ereignisse der Art `windowClosing`) bezeichnet.

8. Ähnlich wie die Methode `paint` werden Behandlermethoden *vom Programmierer vereinbart*, aber *vom Ausführer aufgerufen* (wenn ein entsprechendes Ereignis erzeugt wurde).

9. Eigentlich möchte der Programmierer *Behandlermethoden* bei einem Grabo-Objekt (z.B. bei `mr1`) anmelden und abmelden können. Aber in Java kann man Methoden nur *vereinbaren* und *aufrufen*, aber nicht anmelden oder abmelden. Man kann in Java aber *Objekte anmelden* und *abmelden.* Deshalb muss man Behandlermethoden wie `windowClosing`, `windowOpened`, `windowIconified`, ... etc. in Objekte "verpacken" und diese Objekte anmelden bzw. abmelden.

S. 546, Def.: Ereignis

*Ereignisse* werden zu *Arten* (von Ereignissen) zusammengefasst
(z.B. alle `windowClosing`-Ereignisse)

*Ereignisarten* werden zu (Ereignis-) *Oberarten* zusammengefaßt
(z.B. die Arten `windowClosing`, `windowOpened`, `windowIconified`, ...
zur Oberart Fensterereignis)

S. 547, Tabelle mit Oberarten und Arten von Ereignissen

Zu jeder Oberart von Ereignissen gehören ein oder mehrere Arten von Ereignissen

Zu jeder Oberart gibt es eine Listener-Schnittstelle.

Die enthält für jede Art von Ereignissen eine Behandlermethode. Wir bezeichnen die *Ereignisarten* mit dem selben Namen wie ihre *Behandlermethode* (z.B. ist `windowClosing` eine Art von Ereignissen und eine Methode zur Behandlung von Ereignissen dieser Art).

Zu einigen Lister-Schnittstellen gibt es eine Adapter-Klasse (die die Schnittstelle mit "Strohpuppen" implementiert).

Warum gibt es zu den Schnittstellen `MousWheelListener` und `ActionListener` keine `Adapter`-Klassen? (Weil diese Schnittstellen nur je eine Methode enthalten und da lohnt sich eine Adapter-Klasse nicht).

Nochmal Grabo02 und Grabo03

Für Grabo02 ( S. 536) gilt:

Die Klasse `WindowAdapter` implementiert die Schnittstelle `WindowListener`
(aber nur mit "Strohpuppen")
Die Klasse `ProgTerminator` erweitert die Klasse `WindowAdapter`
Zu welchen Typen gehört das Objekt arnold?
(`ProgTerminator`, `WindowAdapter`, `WindowListener`)

Das Objekt `arno` ist nur eine Art "Verpackung" für die 7 Methoden, die in der Schnittstelle `WindowListener` beschrieben werden (Methoden ohne eine solche Verpackung kann man in Java nur aufrufen, aber nicht "anmelden" oder "abmelden". Das Objekt arno kann man auch in Java "anmelden" und "abmelden").

Immer wenn ein neues `Grabo02`-Objekt erzeugt wird, wird das Objekt `arnold` bei ihm "angemeldet" (mit der Methode `addWindowListener`)

Für Grabo03 (S. 538) gilt:

Immer wenn ein neues Grabo03-Objekt erzeugt wird, wird ein anonymes Objekt einer anonymen Klasse bei ihm angemeldet. Dieses Objekt leistet genau das Gleiche wie das Objekt arnold in Grabo02-Objekten.

Die Klassen `Grabo02` und `Grabo03` leisten "genau das Gleiche" (aber auf etwas unterschiedliche Weisen).

Zur Entspannung: Kurt Gödel (1906-1978, Österreich-Ungarn, Princeton, USA)

Studierte Physik und Mathe in Wien, frühe Begabung für Mathe.

1931: "Über formal unentscheidbare Sätze der Principia Mathematica und verwandte Sätze". Die Principia Mathematika (3 Bände, erschienen 1910-1913) war ein philosophisch-mathematisch wichtiges Werk von Bertrand Russell (1872-1970) und Alfred North Whitehead (1861-1947).

Mit diesem Papier zerstöre Gödel die Hoffnung des Mathematikers David Hilbert (1862-1943), alle mathematischen Sätze rein formal aus einer Basis von Axiomen abzuleiten.

1932: Habilitation in Wien.
1933: Hitler kam an die Macht.
1934: Vorlesungen in Princeton.
1938: "Anschluss" Österreichs an Deutschland.
1940: Auswanderung in die USA, bis 1978 in Princeton, Freund von Einstein. "Consistency of the Axiom of Choice and the Generalized Continuum Hypothesis with the Axioms of Set Theory".

Einer der bedeutendsten Mathematiker des 20. Jahrhunderts. Starb in einer Nervenheilanstalt an Unterernährung, weil er Angst vor einer Vergiftung hatte.

Wiederholungsfragen, 25. SU, Di 06.07.10

1. Was sind (im Zusammenhang mit Java-Grabo-Programmen)
die wichtigsten Eigenschaften eines Ereignisses?

2. Geben Sie zwei Beispiele für *Arten* von Ereignissen an.

3. Geben Sie zwei Beispiele für *Oberarten* von Ereignissen.

4. *Wie viele Arten* von Ereignissen gehören zur Oberart *Fensterereignis*?

5. Ebenso für die Oberart *Mausereignis*.

6. Ebenso für die Oberart *Aktionsereignis*.

7. Welcher Oberart von Ereignissen ist die Schnittstelle `WindowListener` zugeordnet?

8. Ebenso für die Schnittstelle `MouseWheelListener`.

9. Wie heißt die *Adapter-Klasse* zur Schnittstelle `FocusListener`?

10. Wie heißen die Methoden, mit denen man ein `FocusListener`-Objekt (bei bestimmten Grabo-Objekten) *anmelden* bzw. *abmelden* kann?

Rückseite der Antworten zu den Wiederholungsfragen, 25. SU, Di 06.07.10

Ein paar Programmieraufgaben (zum Trainieren vor der Klausur):

```
 1    static public char[] Statt1A1B_R(char[] orig) {                    // *
 2       // Liefert eine Kopie von orig, in der jedes Vorkommen von 'A'
 3       // durch ein 'B' ersetzt wurde.
 4       ...
 5    }
 6
 7    static public char[] Statt1A2B_R(char[] orig) {                    // **
 8       // Liefert eine Kopie von orig, in der jedes Vorkommen einer
 9       // Komponenten 'A' durch zwei Komponenten 'B' ersetzt wurde.
10       ...
11    }
12
13    static public char[] Statt2A3B_R(char[] orig) {                    // ***
14       // Liefert eine Kopie von orig, in der jedes Vorkommen von zwei
15       // unmittelbar nacheinander liegenden 'A' durch drei 'B' ersetzt
16       // wurden. Nur nicht-ueberlappende Vorkommen von Doppel-'A' werden
17       // ersetzt.
18       // Beispiele:
19       // char[] r1 = {'A'};
20       // char[] r2 = {'A', 'A'};
21       // char[] r3 = {'A', 'A', 'A'};
22       // char[] r4 = {'X', 'A', 'A', 'A', 'X'};
23       // char[] r5 = {'X', 'A', 'A', 'X'};
24       // char[] r6 = {'X', 'Y', 'Z', '1', '2', '3'}:
25       // char[] r7 = {};
26       //
27       // Statt2A3B(r1) ist gleich {'A'};
28       // Statt2A3B(r2) ist gleich {'B', 'B', 'B'};
29       // Statt2A3B(r3) ist gleich {'B', 'B', 'B', 'A'};
30       // Statt2A3B(r4) ist gleich {'X', 'B', 'B', 'B', 'A', 'X'};
31       // Statt2A3B(r5) ist gleich {'X', 'B', 'B', 'B', 'X'};
32       // Statt2A3B(r6) ist gleich {'X', 'Y', 'Z', '1', '2', '3'}:
33       // Statt2A3B(r7) ist gleich {};
34       ...
35    }
```

Antworten zu den Wiederholungsfragen, 25. SU, Di 06.07.10

1. Was sind (im Zusammenhang mit Java-Grabo-Programmen)
die wichtigsten Eigenschaften eines Ereignisses?

Ein Ereignis findet zu einem bestimmten Zeitpunkt und an einem bestimmten Ort statt und ist grundsätzlich nicht wiederholbar.

2. Geben Sie zwei Beispiele für *Arten* von Ereignissen an.

windowOpened, windowClosed, ..., mouseClicked, mousePressed, ..., mouseMoved, ...

3. Geben Sie zwei Beispiele für *Oberarten* von Ereignissen.

Fensterereignis, Mausereignis, Mausbewegungsereignis, ...

4. *Wie viele Arten* von Ereignissen gehören zur Oberart *Fensterereignis*?

7 Arten

5. Ebenso für die Oberart *Mausereignis*.

5 Arten

6. Ebenso für die Oberart *Aktionsereignis*.

1 Art

7. Welcher Oberart von Ereignissen ist die Schnittstelle `WindowListener` zugeordnet?

Der Oberart Fensterereignis.

8. Ebenso für die Schnittstelle `MouseWheelListener`.

Der Oberart Mausradereignis.

9. Wie heißt die *Adapter-Klasse* zur Schnittstelle `FocusListener`?

FocusAdapter

10. Wie heißen die Methoden, mit denen man ein `FocusListener`-Objekt (bei bestimmten Grabo-Objekten) *anmelden* bzw. *abmelden* kann?

addFocusListener, removeFocusListener

25. SU Di 06.07.10

A. Wiederholung
B. Organisation: Am kommenden Montag (12.07.10) findet die letzte Vorlesung statt (die besonders viel Platz zum Stellen von Fragen haben wird).

Zur Klausur: Es schadet wahrscheinlich nicht, folgenden Stoff zu beherrschen:

1. Kleine Methoden (5 bis 15 Zeilen) schreiben können
2. Reihungen und Sammlungen bearbeiten können
3. Bojen zeichnen
4. Ausnahmen werfen und fangen
5. Kleine Programmstücke und Programme "mit Papier und Bleistift" ausführen
6. Einfache Grabo-Programme lesen und schreiben können
7. Die Wiederholungsfragen gut beantworten können

Programmieraufgaben zum Üben

Im Archiv `UebungenMitJut.zip` finden Sie ca. 40 Aufgaben der Art: "Schreiben Sie eine Methode die das und das macht". Zu jeder Aufgabe gibt es ein JUnit-Testprogramm ("Jut-Programm"), mit dem Sie Ihre Lösung testen können. Die Aufgaben sind mit *einem Sternchen* (leicht), *zwei Sternchen* (etwa so schwer wie eine Klausuraufgabe) und *drei Sternchen* (schwierig) gekennzeichnet. Einige der Aufgaben setzten Dinge voraus, die wir in dieser Lehrveranstaltung nicht behandelt haben (z.B. die "Bitfummel-Aufgaben").

Wie findet man heraus, welche Grabo-Objekte z.B. Mausereignisse produzieren können?

Was haben solche Objekte gemeinsam?

Welches Schnittstelle ist der Oberart Mausereignis zugeordnet? (`MouseListener`, S. 547, Tabelle)

Tip: Objekte, die Mausereignisse produzieren können, müssen auch die Behandlung solcher Ereignisse erlauben (d.h. das Anmelden und Abmelden von entsprechenden Listener-Objekten).

Antwort: Wir suchen (im Methoden-Index, nicht in der API-Dokumentation!) die Klassen, in denen eine Methode namens `addMousListener` vereinbart wird. Objekte dieser Klassen (und Objekte all ihrer Unterklassen) können Mausereignisse produzieren.

Ensprechendes gilt auch für jede andere Oberart von Ereignissen, z.B. für Fensterereignisse, Mausradereignisse, ... etc.

Auf der Rückseite der Wiederholungsfragen sind die Namen von 72 Schnittstellen aufgeführt. Jede ist einer Oberart von Ereignissen zugeordnet.

23.3 Die Pakete awt und swing, schwere und leichte Komponenten

Moderne *Betriebssystem* (wie das Mac OS, Linux, Windows) bieten von sich aus spezielle Routinen an, mit denen Fenster, Menüs, Knöpfe etc. (windows, menus, buttons etc.) erzeugen und verwalten kann.

Beim Implementieren einer Sprache wie Java gibt es deshalb 2 Strategien: (S. 543)

Das Grabo-Paket `java.awt` wurde nach Strategie-1 (*schwere Komponenten*) und unter extrem hohem Zeitdruck entwickelt.

Das Grabo-Paket `javax.swing` wurde nach Strategie-2 (*leichte Komponenten*), aufbauend auf `java.awt` und unter normalem Zeitdruck entwickelt.

Anmerkung: Für Eclipse hat IBM noch ein drittes Grabo-Paket namens `swt` (Standard Widget Toolkit, "Standard" weil sie *nicht* zum Java-Standard von Sun gehören) entwickelt, nach Strategie-1 (*schwere Komponenten*)

S. 544, Bild 22.4: Ein Typgraf von wichtigen Grabo-Klassen

Zur Entspannung: Christian Morgenstern (1871-1914): Das Butterbrotpapier

Ein Butterbrotpapier im Wald / da es beschneit wird, fühlt sich kalt.

Wiederholungsfragen, 26. SU, Mo 12.07.10

1. In der Klasse `java.awt.Component` wird eine Methode namens `addFocusListener` vereinbart. Was können Sie aus dieser Tatsache schließen?

2. In der Java-Standardbibliothek gibt es eine Schnittstelle namens `TextListener`, aber keine Klasse namens `TextAdapter`. Was können Sie aus dieser Tatsache schließen?

3. In der Java-Standardbibliothek gibt es eine Klasse namens `ComponentAdapter`. Was können Sie aus dieser Tatsache schließen?

Antworten zu den Wiederholungsfragen, 26. SU, Mo 12.07.10

1. In der Klasse `java.awt.Component` wird eine Methode namens `addFocusListener` vereinbart. Was können Sie aus dieser Tatsache schließen?

Jedes Grabo-Objekt enthält eine solche Methode und ist eine Quelle für Focus-Ereignisse.

2. In der Java-Standardbibliothek gibt es eine Schnittstelle namens `TextListener,` aber keine Klasse namens `TextAdapter.` Was können Sie aus dieser Tatsache schließen?

Vermutlich enthält die Schnittstelle TextListener nur eine Methode.

3. In der Java-Standardbibliothek gibt es eine Klasse namens `ComponentAdapter`. Was können Sie aus dieser Tatsache schließen?

Vermutlich gibt es eine Schnittstelle namens ComponentListener, die mehr als eine Methode enthält.

26. SU Mo 12.07.10

A, Wiederholung
B. Organisation

Klausurvorbereitungen:

2. Es gibt 6 Klausuraufgaben. Für 2 Programmieraufgaben gibt es je 20 Punkte, für die restlichen 4 Aufgaben je 15 Punkte (2*20 + 4*15 gleich 100 Punkte insgesamt).

3. Lesen Sie zuerst alle Aufgaben "oberflächlich" durch und wählen Sie dann die Aufgabe, die Sie am besten / einfachsten / schnellsten lösen können. Lösen Sie dann die nächst-einfache Aufgabe etc.

4. Bevor Sie mit dem Lösen einer Aufgabe beginnen, sollten Sie den Text genau und vollständig lesen. Prüfen Sie auch bewußt, ob der Aufgabentext eventuell *auf der folgenden Seite fortgesetzt* wird.

5. Wenn Sie den Aufgabentext nicht genau verstehen, dann melden Sie sich und besprechen den Text mit Ihrem Betreuer. Stellen Sie dabei möglichst Fragen der Form: "Ist das so und so gemeint?" oder "Ist es richtig wenn ich annehme dass ..." oder so ähnlich.

6. Stellen Sie sicher, dass Sie genau verstanden haben, wozu *Beispiele* (im Text einer Aufgabe) dienen.

7. Schreiben Sie jede Ihrer Lösungen auf die Vorderseite eines neuen Blattes. Lassen Sie die Rückseiten der Blätter, die Sie abgeben, leer und schreiben Sie nicht mehrere Lösungen auf ein Blatt (auch wenn dadurch einige Blätter teilweise leer bleiben).

8. Wie kann man kleine Sammlungen besonders einfach initialisieren? Etwa so:

```
1    String[]          sr  = {"ABC", "DE", "FGHI", "J"};
2    ArrayList<String> als = new ArrayList<String>(Arrays.asList(sr));
```

Jetzt enthält die Sammlung `als` die 4 String-Objekte `"ABC"`, `"DE"`, `"FGHI"`, `"J"`.

9. Audrücke, die Reihungen bezeichnen:

```
3    ... new int[]{10, 20, 30} ...
```

Dieser Ausdruck bezeichnet eine Reihung von 3 `int`-Variablen mit den Werten `10`, `20` und `30`. Er kann überall verwendet werden, wo man eine solche Reihungen braucht, z.B. als Parameter einer Methode.

10. Sie dürfen überall die Methodennamen `p` und `pln` (als Abkürzungen für System.out.print bzw. System.out.println) verwenden.

Unterschied zwischen überladen und überschreiben:

| | überladen | überschreiben |
|---|---|---|
| Was kann man ... | Einen Methodennamen | Eine (geerbte) Methode |
| Wie viele Klassen sind beteiligt? | 1 | 2 (Klasse / direkte Oberklasse) |
| Wie viele Methoden sind beteiligt? | 2 oder mehr | Genau 2 |
| Zentrale Bedingung? | Unterschiedliche Signaturen | Gleiche Profile |

Die equals-Methoden

In jedem Objekt (Zielwert einer Referenzvariablen) gibt es eine Methode namens `equals`, mit der man das betreffende Objekt mit einem anderen Objekt vergleichen kann. Diese Methode hat den Rückgabetyp `boolean`.

Was diese `equals`-Methode genau mach ("wie sie vergleicht") hängt vom *Typ* des Objekts ab: Die `equals`-Methode in einem `String`-Objekt funktioniert ganz anders als die `equals`-Methode in einem `StringBuilder`-Objekt, und die in einem `Date`-Objekt funktioniert noch anders etc.

Dazu zwei wichtige Beispiele:

Wenn man zwei `String`-Variablen mit der Methode `equals` vergleicht, werden die *Zielwerte* der Variablen verglichen (nicht die Werte):

```
 1 String st1 = new String("Hallo!");
 2 String st2 = new String("Hallo!");
 3
 4 |st1|--<110>--[<120>]--["Hallo!"];
 5 |st2|--<130>--[<140>]--["Hallo!"];
```

Der Ausdruck `st1.equals(st2)` hat den Wert `true`, weil `"Hallo!"` gleich `"Hallo!"` ist.

Wenn man zwei `StringBuilder`-Variablen mit der Methode `equals` vergleicht, werden die *Werte* der Variablen veglichen (nicht die Zielwerte):

```
 6 StringBuilder sb1 = new StringBuilder("Hallo!");
 7 StringBuilder sb2 = new StringBuilder("Hallo!");
 8
 9 |sb1|--<210>--[<220>]--["Hallo!"];
10 |sb2|--<230>--[<240>]--["Hallo!"];
```

Der Ausdruck `sb1.equals(sb2)` hat den Wert `false`, weil `<220>` ungleich `<240>` ist.

D.h. die `equals`-Methode für `StringBuilder`-Variablen vergleicht wie der Operator `==`, aber die `equals`-Method für `String`-Variablen macht etwas ganz anderes (und mit Bojen kann man sich den Unterschied klar machen).

Wie kann man die *Zielwerte* von zwei `StringBuilder`-Variablen vergleichen?

Indem man aus den `StringBuilder`-Objekten `String`-Objekte erzeugt und vergleicht, etwa so:

```
sbA.toString().equals(sbB.toString())
```

Bevor man Variablen einer Klasse `XXX` mit `equals` vergleicht, sollte man vorher in der Dokumentation der Klasse `XXX` nachsehen, was diese `equals`-Methode genau macht.

Das war's für dieses Semester. Viel Erfolg bei der Klausur am Di 13.07.10 um 18 Uhr im B554.

Anmerkung zur Nachklausur:

Die Nachklausur findet statt am Do 23.09.10 um 14 Uhr im Beuth-Saal.

Wenn mir mindestens drei Teilnehmer etwa eine bis zwei Wochen vor der Nachklausur per Email ihr Interesse an einem Treffen (zum Besprechen von Fragen, die beim Wiederholen des Stoffs aufgetaucht sind) mitteilen, bin ich gern bereit zu solch einem Treffen.