Stichworte

zur Lehrveranstaltung **Programmieren 1**
im 1. Semester des Studiengangs **Medien-Informatik Bachelor (B-MI)**
für den **Zug 1**,
im Sommersemester **2016**
an der **Beuth Hochschule für Technik Berlin**.

Wer sollte diese Lehrveranstaltung belegen?

Alle Studierende im 1. Semester des Studiengangs Medieninformatik Bachelor (B-MI),
die eine *ungerade* (und nicht eine *gerade*) Matrikel-Nr. haben (z.B. `s857427`).

Wiederholer können diese LV nur soweit belegen, wie noch Platz ist.

Studierende im 1. Semester mit einer *geraden* Matrikel-Nr (z.B. `s857426`) und Wiederholer, für die hier kein Platz mehr ist, sollten die parallele Lehrveranstaltung **Programmieren 1, Zug 2** belegen (die von den Herren **Michael Steppat** und **Horst-Werner Radners** betreut wird). Für den Zug 2 gibt es **drei** Übungsgruppen (für den Zug 1 nur **zwei** Übungsgruppen).

Wer betreut diese Lehrveranstaltung?

Tobias Klatt (Tutorium)
Elmar Böhler (Übungen)
Ulrich Grude (Übungen, seminaristischer Unterricht)

Was steht in dieser Datei?

Auf den ersten Seiten finden Sie grundsätzliche **Informationen zu dieser LV**
(Termine, Team-Bildungsregeln, Noten-Regeln etc.).

Im Laufe des Semesters wird diese Datei dann (normalerweise wöchentlich)
um **Stichworte** zum jeweils **behandelten Stoff** erweitert.

Was sollte man beim Lesen wissen und beachten?

In diesen Stichworten ist (wenn nicht ausdrückliche anders angegeben)
mit **Buch** das Buch "Java ist eine Sprache" (von Ulrich Grude) gemeint und
Seitenangaben wie z.B. **S. 20** oder **S. 237** etc. beziehen sich auf dieses Buch.
Das Buch kann man sich (für genau 0,- Euro) in der Bibliothek der Beuth Hochschule ausleihen
oder (für etwa 40,- Euro) in einem Buchladen kaufen. Dieses Buch sollten Sie sich am Semesteranfang (oder davor) besorgen.

Dateien, die in diesen Stichworten erwähnt werden
(z.B. **pr1_Aufgaben.odt** oder **pr1_AlleUebProgs.txt** etc.)
sollten Sie auf der Netzseite http://public.beuth-hochschule.de/~grude/ (ziemlich weit oben) suchen
(meistens sind sie auch wirklich dort). Schreiben Sie eine Email an
`grude at beuth-hochschule.de` falls Sie eine benötigte Datei nicht finden können.

Termine: Alle SUs und Üs finden jeweils am **Dienstag** (Block 1 bis 3) und **Donnerstag** (Block 4 und 5) statt. Alle Üs finden im **SWE-Labor** (Haus Bauwesen, Raum DE16) statt.

SW	KW	Datum (Dienstag)	Dienstag			Donnerstag	
			08.00-09.30 SU (B101)	10.00-11.30 Ü1a/Ü1b	12.15-13.45 Ü1b	14.15-15.45 SU (D451)	16.00-17.30 Ü1a
1	14	05.04.16				01	a01
2	15	12.04.16	02	a02/b01	b02	03	a03
3	16	19.04.16	04 (T01)	a04/b03	b04	05	a05
4	17	26.04.16	06 (T02)	a06/b05	b06	07	a07
5	18	03.05.16	08 (T03)	a08/b07	b08	--	--
6	19	10.05.16	09 (T04)	a09/b09	b10	10	a10
7	20	17.05.16	11 (T05)	a11/b11	b12	12	a12
8	21	24.05.16	13 (T06)	a13/b13	b14	14	a14
9	22	31.05.16	15 (T07)	a15/b15	b16	16	a16
10	23	07.06.16	17 (T08)	a17/b17	b18	18	a18
11	24	14.06.16	19 (T09)	a19/b19	b20	20	a20
12	25	21.06.16	21 (T10)	a21/b21	b22	22	a22
13	26	28.06.16	23 (T11)	a23/b23	b24	24	a24
14	27	05.07.16	25 (T12)	a25/b25	b26	26	a26
15	28	12.07.16	27 (T13)	a27/b27	b38	28	a28
16	29	19.07.16	29	a29/b29	b30		Klausur B101
17	30	26.07.16		Rückgabe DE16			

Akü	Bedeutung
SW	Semesterwoche
KW	Kalenderwoche
SU	Seminaristischer Unterricht
Ü	Übung

Abkürzung	Bedeutung
Ü1a, Ü1b	Übungsgruppe 1a, 1b
a01, a02, ...	Übungsgruppe 1a, Termin 01, 02, ...
b01, b02, ...	Übungsgruppe 1b, Termin 01, 02, ...
T01, T02, ...	Test 01, 02, ...

Hauptklausur: am Do **21.07.2016**, **16.00 Uhr**, Raum B101
Rückgabe: am Di 26.07.2016, 08.00 Uhr, Raum DE16

Nachklausur: am Mi 28.09.2016, 10.00 Uhr, Raum DE17
Rückgabe am Fr 30.09.2016, 10.00 Uhr, Raum DE17

In welcher Semesterwoche (SW) sollte welche **Aufgabe abgegeben/vorgeführt werden**?

Aufgabe	A1	A2	A3	A4	A5	A6	A7	A8	A9	A10	A11	A12	A13
SW	03	04	05	06	07	08	09	10	11	12	13	14	15

Belegungszeitraum (für Studierende im 1. Semester) SS16: **05.04.-19.04.2016**

Wichtige Regeln für diese Lehrveranstaltung (Programmieren 1, Zug 1)

1. Übungsgruppen und Teams

Für diese LV sind 2 **Übungsgruppen** (1a und 1b) mit jeweils etwas mehr als 20 TeilnehmerInnen geplant. Welche Gruppe Sie im **Online-Belegsystem** belegt haben, ist unwichtig (wichtig ist nur, dass Sie eine *belegt haben*). Wer zu welcher Übungsgruppe gehört, wird am Do 07.04.2016 im 3. Block (ab 14.15 Uhr) im Raum D451 (d.h. **im ersten SU**) festgelegt. Wenn es für eine der Gruppen mehr Interessierte als Plätze gibt, entscheidet ein Losverfahren. An diesem Losverfahren können nur persönlich Anwesende teilnehmen.

Ein Wechsel der Übungsgruppe, z.B. von 1a nach 1b, ist nur möglich, wenn Sie jemanden finden, der gleichzeitig von 1b nach 1a wechseln möchte. Informieren Sie in einem solchen Fall die Betreuer beider Übungsgruppen (persönlich oder per Email).

Sobald Sie zu einer bestimmten Übungsgruppe gehören, sollten Sie mit einer Person derselben Übungsgruppe ein **Team** bilden. Teams, die aus 3 Personen bestehen, sind nur mit ausdrücklicher Genehmigung des Betreuers Ihrer Übungsgruppe zulässig (und werden möglicherweise abgelehnt). Größere Teams oder "Einzelkämpfer" sind nicht zulässig. Jedes Team sollte sich einen Namen geben (z.B. Die Gummiadler oder Los Olvidados oder ABC oder ...).

Die Mitglieder eines Teams tragen eine wichtige Verantwortung für einander: Sie können sich das Bestehen der Übungen gegenseitig erleichtern oder erschweren (siehe weiter unten). Sie sollten Ihre Namen, Telefon-Nrn und Email-Adressen austauschen und sich gegenseitig immer über alle wichtigen Team-Angelegenheiten, informieren. Während des Semesters plötzlich wegbleiben, ohne seine Team-PartnerIn zu informieren, wird zwar nicht bestraft, ist aber ganz schlechter Stil.

Melden Sie Ihr Team an, indem Sie die **Aufgabe 1** (Seite 2 und 3 der Datei **pr1_Aufgaben.odt**) ausdrucken, lösen und am **Di 17.04.2016** bzw. am **Do 21.04.2016** in der Übung abgeben. Im Kopf der ausgedruckten Blätter sollten die Namen der **Team-Mitglieder** und der Name des **Teams** stehen.

Anmerkung: Erfahrungen beim Arbeiten in einem Team sind für eine InformatikerIn ebenso wichtig wie Erfahrungen beim Programmieren.

2. Wie bekommt man zwei Noten für diese Lehrveranstaltung?

Die LV **Programmieren 1** besteht aus zwei Modulen:
- **Konzepte**: 2 Blöcke seminaristischer Unterricht (SU) pro Woche und
- **Praxis**: 2 Blöcke Übungen (Ü) pro Woche.

Es wird dringend empfohlen, entweder *beide* Module zu belegen oder *keinen*. Nur *einen* der Module zu belegen ist zwar möglich, aber inhaltlich kaum sinnvoll.

2.1. Wie bekommt man eine Note für den Konzepte-Modul?

Am Ende des SS16 wird eine **Hauptklausur** und kurz vor dem WS16/17 eine **Nachklausur** geschrieben. Zu diesen Klausuren darf man als Unterlagen **5 Blätter** (max. Format DIN A4, beliebig beschriftet) mitbringen. An der Nachklausur dürfen Sie nur teilnehmen, wenn Sie die Hauptklausur nicht bestanden oder nicht mitgeschrieben haben. In einer Klausur können Sie 100 Punkte erreichen. Daraus wird nach der folgenden Tabelle Ihre Note für den Konzepte-Modul ermittelt:

Punkte (ab):	95	90	85	80	75	70	65	60	55	50	0-49
Note:	1,0	1,3	1,7	2,0	2,3	2,7	3,0	3,3	3,7	4,0	5,0

Anmerkung: Es wird stark empfohlen, die **Hauptklausur** mitzuschreiben (und zu bestehen :-). Erfahrungsgemäß ist es sehr schwer, bei der Nachklausur "besser zu sein" als bei der Hauptklausur (in den Semesterferien passiert es leicht, dass man weniger zum Lernen kommt als man geplant hat. Außerdem fehlen die "anregenden Anstöße" durch andere Studierende, die SUs und die Üs).

2.2. Wie bekommt man eine Note für den Praxis-Modul?

Kurze Antwort: Indem Sie (allein) bestimmte *Tests* bestehen und (in Ihrem Team) bestimmte *Aufgaben lösen* (und die Lösungen pünktlich abgeben bzw. vorführen und erklären).

Ausführliche Antwort:

Im Verlauf des Semesters werden **13 kleine Tests** geschrieben (Bearbeitungszeit ca. 10 Minuten). Bei jedem Test können Sie 0 bis 10 Punkte erreichen. *Bestanden* haben Sie einen Test mit 5 oder mehr Punkten. Wenn Sie weniger als 10 der Tests bestehen, bekommen Sie für den Praxisteil die **Note 5,0**.

Die Tests werden jeweils dienstags am Anfang des 1. Blocks (8.00 Uhr) geschrieben.

Anmerkung: Wenn Sie an einem Test *nicht teilnehmen*, dann haben Sie ihn nicht bestanden. Dabei spielt es keine Rolle, *warum* Sie nicht teilgenommen haben (war krank, meine S-Bahn hat sich verspätet, musste einen hohen Lottogewinn persönlich abholen etc.). Das "Nachschreiben" eines verpassten Tests ist nicht möglich. Verwalten Sie *die 3 Tests, an denen Sie nicht unbedingt teilnehmen müssen*, mit Vorsicht und Sparsamkeit und rechnen Sie mit unerwarteten Ereignissen (mit einer Krankheit, einer Störung der S-Bahn, einem hohen Lottogewinn etc.). Besonders empfehlenswert ist der Plan, an **allen 13 Tests** teilzunehmen. Dadurch erhöhen Sie Ihre Chancen, eine gute Praxis-Note zu erreichen (siehe die weiteren Regeln).

Jedes **Team** soll insgesamt **13 Aufgaben** lösen (siehe Datei **pr1_Aufgaben.pdf**), jede Woche eine, und die Lösung in der folgenden Woche an einem der beiden Übungstermine abgeben (bzw. vorführen, je nach Aufgabe). Dabei sollten alle Team-Mitglieder persönlich anwesend und bereit sein, Fragen zu der Aufgabe und der Lösung zu beantworten.

Welche Aufgabe (A1, A2, ..., A13) in welcher Semesterwoche abzugeben ist, steht in einer kleinen Tabelle oben am Ende von Seite 1.

Wenn ein Team eine Lösung nicht pünktlich abgibt, bekommt es dafür *Verspätungspunkte* wie folgt:

Abgabe mit 1 Woche Verspätung	3 Verspätungspunkte
Abgabe mit 2 oder mehr Wochen Verspätung	5 Verspätungspunkte
Keine Abgabe bis **Do 14.07.2016**	8 Verspätungspunkte

Durch pünktliches Abgeben kann ein Team also mit jeder Aufgabe bis zu 8 Verspätungspunkte vermeiden. Nutzen Sie das möglichst oft aus.

Am Ende des Semesters wird für jedes Mitglied **M** eines Teams **T** folgende Punktezahl berechnet:

(Summe der 10 besten Test-Ergebnisse von M) minus (Verspätungspunkte von T).

Aus dieser Zahl (zwischen -104 und +100) wird Ihre Note für den Praxis-Modul berechnet, nach der gleichen Tabelle wie die Klausur-Note (siehe vorigen Abschnitt). Dabei wird eine negative Punktezahl (kommt hoffentlich nie vor) wie die Zahl 0 behandelt.

Noch eine Bitte: Bitte schicken Sie ein **Foto** von sich per Email an **grude AT beuth-hochschule.de** und, wenn Sie zur Übungsgruppe 1b gehören, auch an **eboehler AT beuth-hochschule.de**,. Die Bild-Datei sollte so heißen wie Sie selbst (z.B. Karl_Müller.jpg oder Serkan_Özdemir.jpg etc.), damit die Betreuer dieser LV Ihre Namen lernen können. An die Adresse, von der aus Sie diese Email schicken, werde ich im Laufe des Semesters mehrere Emails schicken (z.B. mit Infos zum nächsten Test).

Diese Regeln sind ziemlich lang und kompliziert. Aber wenn Sie es geschafft haben, sie sorgfältig zu lesen und zu verstehen, dann haben Sie schon mal eine sehr hohe Hürde überwunden. Der Rest dieser Lehrveranstaltung wird Ihnen dann hoffentlich leichter fallen :-).

Viel Spaß und Erfolg in der LV Programmieren 1.

SU 1 Do 07.04.16, Block 4, D451

1. Begrüßung

2. Wenn Sie im 1. Semester sind und eine **gerade** Matrikel-Nr haben, sollten Sie den **Zug 2** belegen!

3. Viele wichtige Regeln für diese Lehrveranstaltungen stehen in der Datei **Stichworte.pdf** auf meiner Netzseite (z.B. wie man für diese LV zwei Noten bekommt). Lesen Sie die Regeln genau durch. Falls Sie dann noch Fragen haben, fragen Sie (z.B. in einem SU).

Die wichtigsten Regeln:

Die Note für den Konzepte-Teil: Wird durch eine Klausur ermittelt.

Die Note für den Praxis-Teil: Insgesamt werden 13 kleine Tests geschrieben (Bearbeitungszeit ca. 10 Minuten), jeweils dienstags ab 8 Uhr. Davon sollten Sie mindestens 10 bestehen.

Wenn Sie einen Test *verpassen*, haben Sie ihn *nicht bestanden*. *Warum* Sie einen Test verpasst haben, spielt *keine Rolle* (die 3 Tests, die Sie nicht unbedingt bestehen müssen, sind für Krankheiten und andere Gründe für ein Verpassen gedacht). Ein Nachschreiben von verpassten Tests ist grundsätzlich nicht möglich. Mit jedem Test können Sie max. 10 Pluspunkte erwerben. Die Punkte für Ihre 10 besten Tests werden addiert.

Außerdem sollen Sie in einem kleinen Team (normalerweise 2 Personen, in Ausnahmefällen 3 Personen) im Laufe des Semester 13 Aufgaben bearbeiten und die Lösungen abgeben bzw. vorführen. Für eine verspätete Abgabe, gibt es Minuspunkte.

Aus den Pluspunkten für Ihre 10 besten Tests und den Minuspunkten für die Aufgaben wird Ihre Note für den Praxisteil berechnet.

4. Wer möchte in welche Übungsgruppe (1a oder 1b)? Ist eine Verlosung nötig?

Erste Übungstermine:
Übungsgruppe 1a: Gleich anschließend im 5. Block
Übungsgruppe 1b: Di 12.04.

Falls nötig, die Plätze verlosen (Für die Übungsgruppe 1b gab es 29 Interessierte. Von denen wurden 5 per Zufall bestimmt, die trotz ihres Wunsches zur Übungsgruppe 1a gehören. Vorläufiges Endergebnis: Zur Gruppe 1a gehören 15 Studierende und zur Gruppe 1b gehören 24 Studierende).

Wichtig: Bitte schicken Sie mir eine Email mit einem Foto von sich als Anhang. Die Datei mit dem Foto sollte ganz ähnlich heißen wie Sie, z.B. `KarlSchmitz.jpg` oder `SerkanÖzyildiz.jpg` etc.

Im Laufe des Semesters werde ich mehrere E-Mails an die Adresse schicken, von der Ihre Email-mit-Bild kam.

In dieser Datei **Stichworte.pdf** werden Sie im Laufe des Semesters jeweils nach einem SU-Termin Stichworte dazu 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".

Wichtige Regel für die SUs: Jedes Team sollte zu jedem SU mindestens einen **Laptop** oder Tablet und ein Exemplar des **Buches** "Java ist eine Sprache" mitbringen.

Abschließendes Wort (nicht zum Sonntag sondern) zum Donnerstag:

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 hilfsbereit 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 dieses Fach **bestehen** oder ob Sie **durchfallen**.

Ich werde mich über jede Person freuen, die besteht, und je besser die Noten sind, mit denen sie bestanden hat, desto mehr werde ich mich freuen.

Auf der anderen Seite fühle ich mich aber 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.

Natürlich ist es unangenehm, im ersten Semester in einer LV durchzufallen. Aber es gibt noch größere Katastrophen, z.B. erst im 5. oder 6. Semester zu merken, dass einem der Studiengang Medieninformatik eigentlich nicht liegt und man besser etwas anderes hätte machen sollen.

Hat noch jemand eine Frage, die wir heute klären sollten?
(Sonst zu Hause die ersten Seiten der Datei **Stichworte.pdf** lesen und nächstes Mal fragen).

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

1.1 Programmieren als ein Rollenspiel (Buch S. 1)

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.* (Programme sind nicht nur "für Computer", sondern auch "für Menschen").

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

Besetzung der Rollen, verschiedene Möglichkeiten.

- **1 Person** übernimmt alle Rollen, nur die Rolle Ausführer überträgt man einem Computer
- Jede der Rollen wir von einem **Team** übernommen (nur der Ausführer ist ein Computer)
- Ein **Mensch** übernimmt die Rolle des **Ausführers** und führt ("mit Papier, Bleistift und einem Radiergummi") bestimmte, nicht zu große, Befehlsfolgen aus, z.B. um sie besser zu verstehen. Diese Besetzung ist besonders wichtig, während man eine Programmiersprache lernt.

Def.: **Programm**: (S. 3)

Ein Programm ist eine Folge von Befehlen, die von einem Programmierer geschrieben wurde und von einem Ausführer ausgeführt werden kann

Jetzt kommt ein Versuch, ein paar "große Ideen" der Programmierung ganz allgemein (und unabhängig von einer bestimmten Programmiersprache) zu beschreiben. Im Rest des Semesters werden wir uns dann mit "vielen kleineren Einzelheiten" befassen.

1.4 Die vier wichtigsten Grundkonzepte der Programmierung (Buch, S. 12)

Variable (Wertebehälter, Inhalt beliebig oft veränderbar)

Typ (Bauplan für Variablen, Unterschiede zwischen getypten und ungetypten Variablen)

Unterprogramm (selbst definierte Befehle)

Modul (ein mindestens 2-teiliger Behälter .... Vorteile von Modulen)

(wird im nächsten SU fortgesetzt und vervollständigt).

Zur Entspannung: **Was kostet ein Studium an der BHT?**

Haushalt der BHT ca. 65 Millionen [Euro pro Jahr].
An der BHT studieren ca. 10 Tausend StudentInnen.
Also kostet das Studieren an der BHT ca. 6500 [Euro pro StudentIn und Jahr].
Ein Bachelor-Studium (6 Semester, 3 Jahre) kostet also ca. 20 Tausend [Euros pro StudentIn].

Übungstermin-01

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

2. Dort die Dateien **pr1_Einrichten.pdf** und **pr1_Moebel.zip** finden

3. In der Datei **pr1_Einrichten.pdf** den Abschnitt **1. Einrichten im SWE-Labor**
(ca. 1 1/2 Seiten) lesen und die dort beschriebenen Schritte (0 bis 7) ausführen.

4. 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.

5. 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 **pr1_Aufgaben.odt** öffnen

Darin die **Aufgabe 1: Grundlagen** (Seite 2 und 3) ausdrucken lassen und (als Team) damit beginnen, die Aufgabe zu lösen.

SU 2 Di 12.04.16, Block 1

Kleine Korrektur: In der Datei **Stichworte.pdf** stand eine Bitte, Ihr Team mit einer Email anzumelden, die den Team-Namen und 2 Fotos enthält. Das wurde wie folgt geändert:

1. Jeder Einzelne sollte ein **Foto** an **grude AT beuth-hochschule.de** und, wenn Sie zur Übungsgruppe 1b gehören, auch an **eboehler AT beuth-hochschule.de** schicken. Die Bild-Datei sollte ganz ähnlich heißen wie die abgebildete Person (z.B. **Serkan_Özdemir.jpg** oder **Josephine_Zimmermann.jpg** etc.). An die Adresse, von der aus Sie diese Email geschickt haben, werde ich im Laufe des Semesters mehrmals Emails senden (z.B. mit Infos zum nächsten Test).

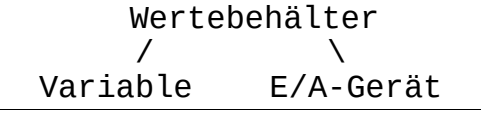
2. Melden Sie Ihr Team an, indem Sie die **Aufgabe 1** lösen und (in der 3. SW, d.h. in der 16. KW) abgeben. Im Kopf dieser Aufgabe sollten die Namen aller Team-Mitglieder und der Name des Teams stehen.

Allgemeiner Hinweis: In dieser LV sind 2 Sprachen besonders wichtig: **Java** und **Deutsch**. Mit "Deutsch" ist nicht die ganze deutsche Sprache gemeint, sondern nur der Teil den man braucht, um sich klar und genau über Programmierung unterhalten zu können. Lernen Sie diesen Teil wie eine Fremdsprache: Schreiben Sie sich alle neuen Begriffe, die erwähnt und behandelt werden, auf einen Zettel, tragen Sie den Zettel möglichst immer bei sich und schauen Sie möglichst oft darauf. 3 mal fünf Minuten Vokabeln lernen bringt meistens mehr als 1 mal fünfundzwanzig Minuten lernen. Benutzen Sie bewusst in Gesprächen (z.B. mit Ihrem Team-Partner) die "neu gelernten Vokabeln".

1.4 Die vier wichtigsten Grundkonzepte der Programmierung (Fortsetzung vom SU 1)

Variable (Wertebehälter, Inhalt beliebig oft veränderbar)

Ein **Wertebehälter** ist entweder eine *Variable* oder ein *E/A-Gerät*
(Tastatur, Bildschirm, Drucker, Datei auf einer Festplatte oder auf einem Stick etc.):



Wichtige **Unterschiede** zwischen Variablen und anderen Behältern (Milchkannen, Hut-Schachteln, ...)

- Eine Variable enthält immer genau **einen Wert**
  (d.h. eine Variable kann nicht leer sein oder 2 Werte enthalten!)
- Wenn man einen Wert in eine Variable tut, geht der alte Wert dadurch kaputt.

Typ (Bauplan für Variablen, Unterschiede zwischen getypten und ungetypten Variablen)

Unterprogramm (selbst definierte Befehle)

Modul

Def: Ein Modul ist ein Behälter für Variablen, Typen, Unterprogrammen, Modulen etc. der aus mindestens 2 Teilen besteht: Einem öffentlichen (ungeschützten, sichtbaren) Teil und einem privaten (geschützten, nicht-sichtbaren) Teil. Auf die Dinge im privaten Teil kann von außerhalb des Moduls *nicht* zugegriffen werden.

Modul-Vorteil-1: Angenommen ein Programm besteht aus 100 Modulen. Im Modul M37 vereinbaren wir eine Ganzzahl-Variablen namens `konto` von der wir wissen, dass sie eigentlich nie einen negativen Wert enthalten darf. Beim Testen merken wir, dass `konto` in gewissen Situationen doch einen negativen Wert enthält. Wo müssen wir die falschen Befehle (die dem `konto` einen negativen Wert zugewiesen haben), suchen
- wenn wir `konto` im *öffentlichen* Teil von M37 vereinbart haben? (in allen 100 Moduln)
- wenn wir `konto` im *privaten* Teil von M37 vereinbart haben? (nur im Modul M37)

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). Wenn P3 diese Verbesserung einbaut, muss er mit Morddrohungen seiner Kollegen P1, P2, P4, ... rechnen (weil die ihre Module jetzt ändern müssen). Hätte P3 das Unterprogramm `summe` in den privaten Teil des Moduls M3-5 geschrieben, könnten seine Kollegen es nicht aufrufen und P3 könnte es problemlos ändern.

Zum Abschluss: Das **fünfte** der vier wichtigsten Grundkonzepte: **Klasse**

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

In Java besteht ein Klasse allerdings nicht nur aus 2 Teilen, sondern aus 4 Teilen:
einem **öffentlichen**, einem "**halb-öffentlichen**", einem "**halb-privaten**" und einem **privaten** Teil.
Wir benutzen aber erst mal nur den öffentlichen und den privaten Teil.

1.5. Drei Arten von Befehlen (Buch S. 19)

Jeder Befehl des Programmierers an den Ausführer gehört zu einer von 3 Arten. Wenn man nur die Art eines Befehls kennt, weiß man schon ziemlich viel über ihn (und muss nur noch "ein paar Einzelheiten" lernen, um ihn vollständig zu kennen"). Das gilt für alle Programmiersprachen (nicht nur für Java):

Befehls-Art	**Übersetzung eines Befehls dieser Art ins Deutsche**
Vereinbarung (engl. declaration)	Erzeuge ....
Ausdruck (engl. expression)	Berechne den Wert des Ausdrucks ...
Anweisung (engl. statement)	Tue die Werte ... in die Wertebehälter ...

Beispiele für Java-Befehle aller drei Arten werden im Buch ab S. 20 beschrieben

Vereinbarungen

```
int otto = 17;
```

Übersetzung ins Deutsche:
Erzeuge eine Variable namens `otto` vom Typ `int` mit dem Anfangswert `17`.

Den "Sinn" von Namen wie `otto` oder `anna` oder `Claudia` etc. erläutern.

```
char anna = 'A';
```

Übersetzung ins Deutsche:
Erzeuge eine Variable namens `anna` vom Typ `char` mit dem Anfangswert `'A'`.

```
static void druckeHallo() {
   System.out.println("Hallo!");
   System.out.println("------");
}
```

Übersetzung ins Deutsche (unvollständig):
Erzeuge ein Unterprogramm namens `druckeHallo` mit dem Ergebnistyp `void` und 0 Parametern, welches folgende Befehle enthält: ...

```
class Claudia {
   ...
}
```

Übersetzung ins Deutsche (unvollständig):
Erzeuge eine Klasse namens `Claudia`, die folgende Elemente enthält: ... .

Ausdrücke

```
otto + 3                                    Typ int
```

Übersetzung ins Deutsche:
Berechne den Wert des Ausdrucks otto + 3

```
(otto + 98) / (otto * 10) - 1               Typ int
```

Übersetzung ins Deutsche:
Berechne den Wert des Ausdrucks (otto + 98) / (otto * 10) - 1

```
otto < 3                                    Typ boolean
otto                                        Typ int
3                                           Typ int
0.1                                         Typ double, Wert etwas größer als 0.1
```

Anweisungen

```
otto = 2 * otto;
```

Übersetzung ins Deutsche:
Berechne den Wert des Ausdrucks 2* otto und weise ihn (den Wert) der Variablen otto zu.

```
otto = 15;
```

Übersetzung ins Deutsche:
Berechne den Wert des Ausdrucks 15 und weise ihn der Variablen otto zu.

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

Übersetzung ins Deutsche:
Berechne den Wert des Ausdrucks "Hallo!" und weise ihn dem Wertebehälter Bildschirm zu.

Achtung: Die Befehle `int otto = 17;` und `otto = 15;` sehen *ähnlich* aus, sind aber ganz *verschiedene* Befehle:
Der erste ist eine Variablen-**Vereinbarung** mit Initialisierung, der zweite ist eine **Anweisung**.
Der erste ist nur erlaubt, wenn es noch keine Variable namens `otto` gibt.
Der zweite ist nur erlaubt, wenn es schon eine `int`-Variable namens `otto` gibt.

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**.

Übungstermin-02

1. Falls Sie die folgenden Punkte beim **Übungstermin-01** nicht vollständig erledigt haben, dann jetzt erledigen:

3. In der Datei **pr1_Einrichten.pdf** den Abschnitt **1. Einrichten im SWE-Labor**
(ca. 1 1/2 Seiten) lesen und die dort beschriebenen Schritte (0 bis 7) ausführen.

4. 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.

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

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

2. Einen Ordner **Z:\Aufgabe2** (oder so ähnlich) anlegen und
die Dateien `EM.java` und `EMTst.java` hineinkopieren
(die Dateien stehen in Ihrem Ordner **Z:\BspJaSp** und im Anhang der Datei **pr1_Aufgaben.odt**).

Öffnen Sie die Datei `EMTst.java` mit dem TextPad und übergeben Sie sie dem Java-Ausführer (z.B. mit **Strg-1**).

Anmerkung: Das Programm `EMTst` besteht aus den beiden Klassen `EMTst` und `EM`. Wenn Sie die Datei `EMTst.java` übergeben, sucht der Ausführer automatisch auch nach nach der Datei `EM.java`. Da diese Datei im selben Ordner steht wie die Datei `EMTst.java`, findet er sie dort. Dann prüft (und compiliert) er beide Dateien und akzeptiert sie (hoffentlich) beide.

Starten Sie das Programm `EMTst` (z.B. mit **Strg-2**, wenn die Datei `EMTst.java` im TextPad den Fokus hat).

Beantworten Sie die Frage `Was soll eingelesen werden (b, c, s, i ..., h, q)?` mit `i` (wie `int`, d.h. Sie wollen dem Programm ein paar `int`-Werte eingeben).

Beantworten Sie dann die Frage `Ein int-Wert (999 zum Beenden)?` ein- oder mehrmals mit je einer Ganzzahl (wie z.B. mit 17 oder mit -123 etc.).

Beantworten Sie dann die Frage `Ein int-Wert (999 zum Beenden)?` der Reihe nach mit folgenden vier Eingaben: `max`, `-max`, `min`, `-min`.

Lesen Sie die Ausgaben des Programms genau und kritisch durch. Welche der Ausgaben ist völlig falsch? Wie hätte Sie eigentlich lauten müssen?

3. Schauen Sie sich (in der Datei **pr1_Aufgaben.odt**) die Aufgabe 2 an und schreiben Sie das dort beschriebene Programm `GanzRech` . Ein "Grundgerüst" für dieses Programm sollten Sie nicht mühsam "eintippen", sondern bequem "zusammenklicken", wie in der vorigen Übungen behandelt (falls Sie Mühe haben, sich zu erinnern, dann schauen Sie sich in der Datei **Einrichten.pdf** noch mal den Abschnitt **2.5. Ein kleines Java-Programm zusammen klicken** an).

SU 3 Do 14.94.16, Block 4

Wiederholungsfragen:

Was ist eine Variable (in der Informatik)? Was ist ein Modul? Was ist ein Programm?

Wenn Sie Mühe hatten, diese Fragen zu beantworten, dann sollten Sie in Zukunft verstärkt "Vokabeln lernen" (alle neuen Begriffe auf einen Zettel schreiben und den immer mit sich herumtragen und öfters mal draufsehen).

Warnung vor Verwechslungen verschiedener Behälter:

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

Eine Variable und ein Modul *unterscheiden* sich aber ganz wesentlich 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).

Wichtige Unterschiede zwischen Variablen und anderen Behältern:

1. Eine Variable enthält immer genau 1 Wert (d.h. eine Variable kann nicht leer sein!).
2. Wenn man einen Wert in eine Variable tut, geht dadurch der "alte Wert" vollständig kaputt

Ein Modul kann z.B. eine Variable und die kann einen Wert enthalten. Aber in dieser 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! :-)

4 Anweisungen (statements) (Buch S. 46)

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

Man unterscheidet 2 Arten von Anweisungen:

Einfache Anweisungen (engl. simple statements), die keine Anweisungen enthalten,
z.B. die Zuweisung und die Anweisungen `return, break, continue, println, ...`

Zusammengesetzte Anweisungen (engl. compound statements), die Anweisungen enthalten,
z.B. `if, while, for, switch` und die Block-Anweisung `{...}`

Erläuterung anhand von Beispielen:

Eine Zuweisung wie z.B. `otto = 3*emil;`
besteht im Wesentlichen aus 2 Teilen
dem Variablen-Namen `otto` und dem Ausdruck `3*emil` (der Rest ist "syntaktischer Zucker")
Keines dieser Teile ist eine *Anweisung*. Also ist diese Zuweisung eine *einfache* Anweisung (und Gleiches gilt für *alle* Zuweisungen).

Eine `if`-Anweisung wie z.B. `if (a < 3) otto = 3*emil;`
besteht im Wesentlichen aus 2 Teilen
dem Ausdruck `a < 3` und der Anweisung `otto = 3*emil` (der Rest ist "syntaktischer Zucker") .
Also ist diese `if`-Anweisung eine *zusammengesetzte* Anweisung (und Gleiches gilt für *alle* `if`-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 und `println`-Befehle).

Wenn der Ausführer diese Methode *einmal* ausführt, wie oft führt er dazu jede der einfachen 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.

Von dieser Regel gibt es eine Ausnahme: Die **Block-Anweisung**

Buch S. 58, **Beispiel-01**

Mit einer **Blockanweisung** kann man 0 oder *mehr* Befehle (Vereinbarungen und Anweisungen) zu *einer* Anweisung zusammenfassen.

Weitere Beispiele für zusammengesetzte Anweisungen:

if-Anweisung (S. 60)

Buch S. 60, **Beispiel-01**: (statt "einfach" sollte es "simple" oder "kurz" heißen)

```
if (a < b) pln(a);
```

Lesen Sie sich die Übersetzung ins Deutsche durch.

Wichtig:
Diese `if`-Anweisung bewirkt, dass die Anweisung `pln(a);` manchmal *nicht ausgeführt* wird (denn ohne das `if (a < b)` würde Sie ja immer ausgeführt).

Fachbegriffe:

Eine **Bedingung** ist ein Ausdruck, der dem Ausführer befiehlt, einen `boolean`-Wert zu berechnen.
Kurz: Eine Bedingung ist ein Ausdruck vom Typ `boolean`.

Eine `if`-Anweisung besteht aus 2 Teilen, einem Ausdruck und einer Anweisung.
Der Ausdruck muss vom Typ `boolean` sein (d.h. er muss eine *Bedingung* sein).
Die Anweisung bezeichnen wir als dann-Rumpf (engl. then-body).

(S. 60, Beispiel-01, S.61, Beispiel-02).

Eine `if-else`-Anweisung besteht aus 3 Teilen, einem Ausdruck und zwei Anweisungen.
Der Ausdruck muss vom Typ `boolean` sein (d.h. er muss eine *Bedingung* sein).
Die 1. Anweisung bezeichnen wir als dann-Rumpf (engl. then-body).
Die 2. Anweisung bezeichnen wir als sonst-Rumpf (engl. else-body).

(Besipiel-03).

while-Anweisung (oder: while-Schleife)

Eine `while`-Schleife besteht aus 2 Teilen: Einem Ausdruck und einer Anweisung.
Der Ausdruck muss vom Typ `boolean` sein (d.h. er muss eine *Bedingung* sein).
Die Anweisung bezeichnet man auch als den *Rumpf* der Schleife (engl. body of the loop).

Ein Beispiel einer while-Schleife für Aufgabe 2

```
 1 int g1;
 2 int g2;
 3 while (true) {           // Der folgende Block wird "immer wieder" ausgefuehrt
 4    p("Zwei Ganzzahlen (999 zum Beenden): ");
 5    g1 = EM.liesInt();
 6    if (g1==999) break;   // Hier springt der Ausfuehrer eventuell zur Zeile 12
 7    g1 = EM.liesInt();
 8    pln("g1    ist gleich " + g1);
 9    pln("g2    ist gleich " + g2);
10    pln("g1+g2 ist gleich " + (g1+g2));
11 }
12 pln("Fertig!");
```

`true` ist ein Ausdruck vom Typ `boolean`, d.h. eine Bedingung. Diese Bedingung ist immer erfüllt.

Empfehlung: Lassen Sie erst mal alle `while`-Schleifen mit `while (true)` beginnen, und beenden Sie die Schleife mit einem Befehl der Form `if (...) break;` irgendwo im Rumpf der Schleife (wie im obigen Beispiel in Zeile 6). Es gibt noch andere Formen von `while`-Schleifen und `do-while`-Schleifen, aber die sind eigentlich überflüssig, man kann mit ihnen nicht "mehr machen" als mit `while-true`-Schleifen.

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 Sonderfälle. Typischerweise wird der Rumpf einer `while`-Schleife **mehr als einmal** ausgeführt.

Zur Entspannung: **Christian Morgenstern (1871-1914)**

Tertius Gaudens ("Da freut sich der Dritte")

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

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

S. 73, Beispiel-01

S. 75: Die vier Teile einer for-Schleife und die Reihenfolge ihrer Ausführung

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.

Übungstermin-03

Die **Aufgabe 1** sollte jetzt fertig sein und in der nächsten Übung abgegeben werden. Falls es noch Fragen dazu gibt, möglichst in dieser Übung erledigen.

Die **Aufgabe 2** sollte bearbeitet und möglichst weit fertig gestellt werden. Alle Fragen zu **Aufgabe 2** sollten möglichst in dieser Übung behandelt werden.

Wer mit **Aufgabe 2** schon fertig ist, sollte mit **Aufgabe 3** anfangen. Die Methode `BruchRech` (die Sie in Aufgabe 3 schreiben sollen) hat viel Ähnlichkeit mit der Methode `GanzRech` (die Sie in **Aufgabe 2** geschrieben haben).

Wichtig: Alle sollten möglichst bald am Editor **TextPad** (auf den Rechnern des SWE-Labors und auf dem eigenen privaten Laptop oder Computer) die Einstellungen vornehmen, die

im Papier **pr1_Einrichten.pdf** im Abschnitt **4. TextPad für die erste Woche** (S. 9 und 10) beschrieben werden!

SU 4 Di 19.04.16, Block 1

Heute schreiben wir den Test-01

Organisation: In Emails können Sie mich anreden, wie Sie möchten, aber besonders gut finde ich kurze, informelle Anreden (z.B. *Hallo Herr Grude* oder *Guten Tag* etc.). Ich werde Sie in Emails ganz entsprechend anreden, wenn Sie nicht ausdrücklich etwas anderes wünschen. Gestalten Sie die **Betreff-Zeile** aber bitte immer bewusst und **aussagekräftig**. Eine Betreffzeile "*Re: Zu allen Tests ...*" in einer Email, in der es um *while-Schleifen* geht ist gar nicht gut. Ich habe mir vorgenommen, die Betreff-Zeile für diese LV immer mit "**PR1**" beginnen zu lassen (habe mich bisher aber selbst noch nicht an diese Regel gehalten).

Wichtige Wiederholungsfragen:

Was befiehlt der Programmierer dem Ausführer mit einer *Anweisung*?
(dass er bestimmte Werte in bestimmte Wertebehälter tun soll)
Woran erkennt man, das eine Anweisung eine *zusammengesetzte Anweisung* ist?
(daran, dass die Anweisung andere Anweisung als ihre Teile enthält)
Was kann der Programmierer mit einer *zusammengesetzten Anweisungen* erreichen?
(dass die darin enthaltenen Anweisungen weniger als einmal bzw. mehr als einmal ausgeführt werden)
Beispiele für zusammengesetzte Anweisungen (die wir schon besprochen haben)?
`if`-Anweisung, `while`-Anweisung (oder: `while`-Schleife), `for`-Anweisung (oder: `for`-Schleife)

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

Wir führen eine kleine Befehlsfolge selbst aus ("mit Papier und Bleistift"):

Im Buch auf S. 41 unten steht eine Befehls-Folge (16 Zeilen)
Die wollen wir jetzt zusammen "mit Tafel und Kreide" ausführen.

Übung 3: if A (ausführen)

Öffnen Sie die Datei **pr1_Übungen.odt** und gehen Sie zur **Seite 6**.
Dort finden Sie ein Java-Programm namens **UebIfA** (56 Zeilen lang) und dahinter ein paar Übungsaufgaben. Führen Sie (als Team) die Übungsaufgabe 1. aus (und danach 2.. 3., ... soweit Sie kommen).

Lösung 3: if A (ausführen)

1. Die Ausgabe des Programms `UebIfA` für die Eingaben 6:

```
 1   ------------------------------
 2  A UebIfA_Loes: Eine Ganzzahl n? 6
 3  B n ist positiv!
 4  D n war einstellig und wurde
 5  E verdoppelt zu 12
 6  F n ist eine gerade Zahl!
 7  H n ist durch 3 teilbar!
 8  L Ist n einstellig?  false
 9  M Ist n zweistellig? true
10  N UebIfA_Loes: Das war's erstmal!
11 ------------------------------
```

2. Die Ausgabe des Programms `UebIfA` für die Eingaben -9:

```
12  ------------------------------
13 A UebIfA_Loes: Eine Ganzzahl n? -9
14 C n ist negativ!
15 D n war einstellig und wurde
16 E verdoppelt zu -18
17 F n ist eine gerade Zahl!
18 H n ist durch 3 teilbar!
```

```
19 L Ist n einstellig?  false
20 M Ist n zweistellig? true
21 N UebIfA_Loes: Das war's erstmal!
22 ------------------------------
```

3. Die Ausgabe des Programms `UebIfA` für die Eingaben 10:

```
23  ------------------------------
24 A UebIfA_Loes: Eine Ganzzahl n? 10
25 B n ist positiv!
26 F n ist eine gerade Zahl!
27 J n ist durch 5 teilbar!
28 L Ist n einstellig?  false
29 M Ist n zweistellig? true
30 N UebIfA_Loes: Das war's erstmal!
31 ------------------------------
```

4. Genau dann wenn der Benutzer eine Zahl zwischen `-4` und `4` (einschließlich) eingibt, gibt das Programm `UebIfA` (unter anderem) die Meldung `Ist n einstellig?  true` aus.

5. Antworten auf Fragen:
5.1. Eine *Variable* ist ein *Behälter* für *Werte* (ein Wertebehälter).
5.2. Ein *Typ* ist ein *Bauplan* für *Variablen* (für Wertebehälter).
5.3. Ein *Modul* ist ein *Behälter* für Variablen, Unterprogramme, Typen und andere Dinge, der aus einem *sichtbaren* und einem *unsichtbaren* Teil besteht.

6. Befehle klassifizieren und übersetzen:
6.1. *Vereinbarung*: Erzeuge eine Variable namens `mirko` vom Typ `int` mit dem Anfangswert `3`.
6.2. *Vereinbarung*: Erzeuge eine Variable namens `sascha` vom Typ `String` und gib ihr den Anfangswert (eigentlich: Anfangs-Zielwert) `"Pickelheringe"`.
6.3. *Ausdruck*: Berechne den Wert des Ausdrucks `mirko + sascha`.
6.4. *Ausdruck*: Berechne den Wert des Ausdrucks `mirko + 14`.
6.5. *Anweisung*: Berechne den Wert des Ausdrucks `mirko + sascha` und tue ihn in den Wertebehälter `mirko` (oder: weise ihn der Variablen `mirko` zu).
6.5. *Anweisung*: Berechne den Wert des Ausdrucks `mirko + sascha.length()` und weise ihn der Variablen `sascha` zu.
6.5. *Anweisung*: Berechne den Wert des Ausdrucks `mirko > 16`. Wenn dieser Wert gleich `true` ist, dann berechne den Wert des Ausdrucks `mirko - 1` und weise ihn der Variablen `mirko` zu.

7. Der Variablen `n` werden nacheinander die folgenden Werte zugewiesen:
`13, 40, 20, 10, 5, 16, 8, 4, 2, 1.`
In der Variablen `zaehler` steht nach Ausführung aller Befehle der Wert `9`.

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. Im Dezember 2013 wurde Turing von Queen Elizabeth "begnadigt" (he got a royal pardon). 2014 kam die "dramatisch zugespitzte Filmbiografie Turing's "The Imitation Game" heraus.

Übungstermin-04

1. In dieser Woche soll die **Aufgabe 1** abgegeben werden..

2. Falls Sie es noch nicht gemacht haben

Wichtig: Alle sollten möglichst bald am Editor **TextPad** (auf den Rechnern des SWE-Labors und auf dem eigenen privaten Laptop oder Computer) die Einstellungen vornehmen, die

im Papier **pr1_Einrichten.pdf** im Abschnitt **4. TextPad für die erste Woche** (S. 9 und 10) beschrieben werden!

Das sollte sich ziemlich schnell erledigen lassen.

3. Programmier-Übung zum Thema `if`-Anweisungen

3.1. Öffnen Sie die Datei **pr1_AlleUebProgs.txt** (mit dem TextPad).

Diese Text-Datei enthält die Inhalte von vielen `.java`-Dateien. Die Inhalte verschiedener Dateien sind durch Sternchen-Zeilen voneinander getrennt. Eine Sternchen-Zeile sieht so aus:

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

Der Inhalt der Datei `UebIfB.java` beginnt mit folgender Zeile:

```
// Datei UebIfB.java
```

und für andere Dateien gilt ganz entsprechendes.

3.2. Legen Sie einen Ordner **Z:\Uebungen** (oder so ähnlich) an.

3.3. Legen Sie in diesem Ordner zwei Dateien namens **UebIfB.java** und **UebIfB_Jut.java** an.
Die Inhalte dieser Dateien stehen in der Text-Datei **pr1_AlleUebProgs.txt** unmittelbar hintereinander (nur durch eine Sternchen-Zeile voneinander getrennt).

3.4. Übergeben Sie beide Dateien dem Ausführer (d.h. compilieren Sie sie) und lassen Sie **UebIfB_Jut** ausführen. Zwei Fenster sollten aufgehen und das zweite sollte unter anderem einen "dicken roten Balken" anzeigen. Was ist passiert?

Das Test-Programm **UebIfB_Jut** hat die Klasse **UebIfB** getestet und noch viele Fehler gefunden (darum ist der Balken rot und nicht grün). Das ist nicht erstaunlich, weil die Datei **UebIfB** noch sehr unvollständig ist und von Ihnen vervollständigt werden soll.

3.5. Schauen Sie sich jetzt die Datei **UebIfB.java** genauer an und beseitigen Sie darin einen Fehler nach dem anderen, indem Sie an den mit

```
// HIER SOLLTEN SIE ETWAS EINFUEGEN
```

gekennzeichneten Stellen eine geeignete if-Anweisung einfügen.

Immer, wenn Sie eine neue if-Anweisung eingefügt haben, sollten Sie das Testprogramm erneut ausführen lassen und prüfen, ob es jetzt weniger Fehler anzeigt als vorher. Wenn nein, dann müssen Sie die zuletzt eingefügte if-Anweisung wahrscheinlich noch ein bisschen verbessern. *Fehler* werden im Test-Programm als **failures** bzw. als **errors** bezeichnet.

4. Falls danach noch Zeit ist können Sie z.B. die **Aufgabe 2** und die **Aufgabe 3** bearbeiten.

SU 5 Do 21.04.16, Block 4

Rechnen mit Ganzzahlen und Struktur eines Ganzzahl-Typs

Buch S. 12, Bild 5.3

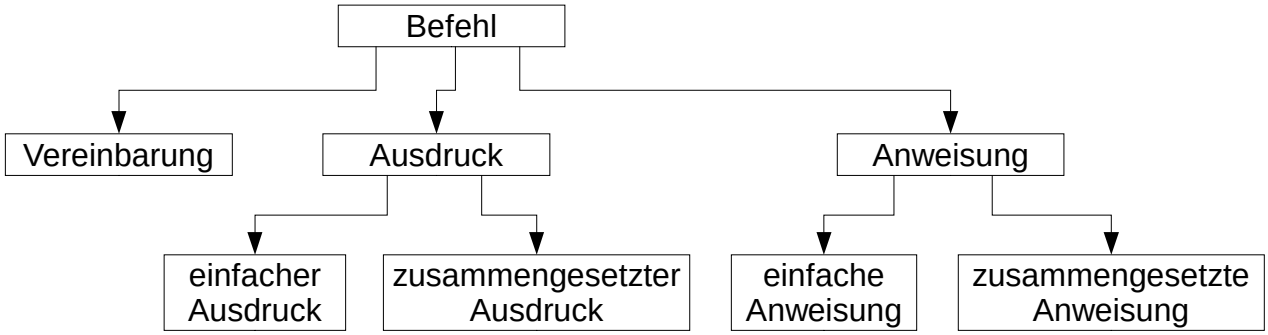
Ungefähr wie groß ist der größte `int`-Wert? (2.15 Milliarden)
Ungefähr wie viele `int`-Werte gibt es insgesamt? (4.3 Milliarden)

Zwei `int`-Variablen:

```
int min = Integer.MIN_VALUE; // ungefähr -2.15 Milliarden
int max = Integer.MAX_VALUE; // ungefähr +2.15 Milliarden
int mrd = 1_000_000_000;     // genau 1 Milliarde
```

Ausdruck	ungefährer Wert	genauer Wert
`7 / 3`		`2`
`-7 / 3`		`-2`
`2 * max`	`0`	`-2`
`3 * max`	`max`	`max-2`
`4 * max`	`0`	`-4`
`5 * max`	`max`	`max-4`
`6 * max`	`0`	`-6`
`max + 1*mrd`	`-1.15 Mrd`	
`max + 2*mrd`	`-0.15 Mrd`	
`max + 3*mrd`	`+0.85 Mrd`	
`max + 4*mrd`	`+1.85 Mrd`	

Zwei weitere Begriffe unterhalb von "Befehl"



6. Ausdrücke

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

Unterschied zwischen Literalen und unveränderbaren Variablen

Der Name einer unveränderbaren Variablen ist auch eine Art *Name für einen Wert* (für den unveränderbaren Wert der Variablen). Diesen Wert legt der Programmierer (mit seinen Befehlen) fest. Die Werte von Literalen sind durch die Sprache Java (oder: durch den Ausführer) festgelegt. Das Literal `17` bezeichnet garantiert den `int`-Wert siebzehn. Eine unveränderbare `int`-Variable namens `siebzehn` kann z.B. den Wert `18` (oder irgend einen anderen `int`-Wert) enthalten.

Unterschied zwischen Literalen und Werten

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: Ein Kriminalroman enthält keine "Menschen aus Fleisch und Blut", nur Namen und Bezeichner für Menschen (z.B. "Frau Meier" oder "der Kommissar" etc.). Diese Namen sind syntaktische Größen. Die entsprechenden Menschen sind semantische Größen.

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

Übungstermin-05

1. **Aufgabe 1** abgeben!

2. **Aufgabe 2** möglichst fertig machen. Haben alle die letzte Teilaufgabe (Formel die erklärt, warum 100_000 * 100_000 nicht 10 Milliarden ergibt sondern nur etwa 1.4 Milliarden) gut gelöst?

3. Gibt es Fragen zur **Übung 4: if B (programmieren)** ?

4. **Aufgabe 3** bearbeiten (sie hat viel Ähnlichkeit mit der Aufgabe 2)

SU 6 Di 26.04.16, Block 1

Heute schreiben wir den Test-02

Wiederholungsfragen

Wir haben 2 Arten von Ausdrücken unterschieden. Welche?
(einfache / zusammengesetzte Ausdrücke, engl: simple / compound Ausdrücke)
Woran erkennt man einen zusammengesetzten Ausdruck?
(daran, dass er andere Ausdrücke, einen oder mehr, als seine Teile enthält)
Es gibt 2 Arten von einfachen Ausdrücken. Welche?
(Literale, Namen von Variablen)
Woraus besteht ein zusammengesetzter Ausdruck? Aus was für Teilen?
(aus Ausdrücken, Operatoren und runden Klammern. Diese Liste ist noch nicht vollständig!)

Zum Thema Ausdrücke (Fortsetzung)

Viele *Operatoren* haben 2 Operanden und wir schreiben den Operator üblicherweise *zwischen* seine beiden Operanden, z.B. so: `3 + 4`, `x + y`, `14 - 5`, `10 * 20`, `20 / 5` etc. Diese Notation wird als **infix-Notation** bezeichnet (weil der Operator "**in** der Mitte zwischen seinen Operanden" steht. Die infix-Notation kam im 14. Jahrhundert in Europa auf und wurde seitdem leider nicht durch etwas besseres ersetzt (ähnlich wie die Maßeinheiten *Fuß* und *Meile* in der Luftfahrt).

Nachteil der infix-Notation: Sie ist (ohne zusätzliche Regeln) **nicht eindeutig**. Beispiele:

Ist `8 - 4 - 2` gleich 2 oder gleich 6?
Ist `8 / 4 / 2` gleich 1 oder gleich 4?
Ist `8 + 4 * 2` gleich 24 oder gleich 64?

Deshalb musste man sie durch durch komplizierte Regeln ergänzen (denen man auch noch abstoßende Namen gegeben hat) und diese Regeln allen Schulkindern eintrichtern (statt den Kindern in dieser Zeit interessante Mathematik beizubringen).

Assoziativität

Ein Operator `op` mit 2 Operanden ist
links-assoziativ, wenn man `a op b op c` wie `(a op b) op c` lesen soll, und
rechts-assoziativ, wenn man `a op b op c` wie `a op (b op c)` lesen soll.
Im ersten Fall gehört `b` zum *linken* `op` und im zweiten Fall zum *rechten* `op`.

Üblicherweise (auch in Java) gelten die Operatoren +, -, *, /, % als *links-assoziativ.*
Wenn es einen Potenzierungs-Operator gibt (ist in Java nicht der Fall), z.B. **, dann gilt der meistens (aber nicht immer) als rechts-assoziativ (`a ** b ** c` soll als `a ** (b ** c)` gelesen werden).
In Java ist die Zuweisung = (merkwürdigerweise) ein Operator und rechts-assoziativ (`a = b = c` bedeutet dasselbe we `a = (b = c)`).

Priorität von Operatoren (oder: Bindungsstärke)

Viele Menschen kennen die Regel "Punktrechnung geht vor Strichrechnung". Mit anderen Worten: Die Operatoren * und / haben eine höhere Priorität (oder: Bindungsstärke) als die Operatoren + und −.

In Java gibt es 40 Operatoren und 13 verschiedene Bindungsstärken.

Buch, S. 143-144: Tabelle der Operatoren, ihrer Stelligkeit, Assoziativität und Bindungsstärke. Da diese Tabelle im Buch unglücklicherweise "auf zwei Seiten verteilt" ist, wird sie hier noch mal (etwas übersichtlicher) auf einer einzigen Seite wiedergegeben. Diese Tabelle fasst die wichtigsten Informationen über alle Java Operatoren zusammen:

Operatoren	Stelligkeit	Kurzbeschreibung	Fix	Assoziativität	Bindungsstärke
`+ -`	1	Vorzeichen	prä	-	13
`++ --`	1	In-/Dekrement	prä, post	-	13
`~`	1	bitweise Negation	prä	-	13
`!`	1	logische Negation	prä	-	13
`()`	2	Cast-Operator	mix	R	13
`* / %`	2	multiplikative Operatoren	in	L	12
`+ −`	2	additive Operatoren	in	L	11
`<< >> >>>`	2	Shift-Operatoren	in	L	10
`< <= >= > instanceof`	2	relationale Operatoren	in	L	9
`== !=`	2	Gleichheitsoperatoren	in	L	8
`&`	2	Und (schwach)	in	L	7
`^`	2	Exklusives Oder	in	L	6
`|`	2	Oder (schwach)	in	L	5
`&&`	2	Und (nicht-schwach)	in	L	4
`||`	2	Oder (nicht-schwach)	in	L	3
`... ? ... : ...`	3	Bedingungsoperator	mix	R	2
`= += -= *= /= <<= >>= >>>= %= &= ^= |=`	2	Zuweisungsoperatoren	in	R	1

Die Abkürzungen und ihre Bedeutungen:

R rechtsassoziativ **prä** Präfixnotaion **in** Infixnotation
L linksassozitativ **post** Postfixnotation **mix** Mixfixnotation

Es gibt zwei eindeutige **Alternativen** zur mehrdeutigen infix-Notation:
Die **postfix-Notation** und die **praefix-Notation**.
Die Firma Hewlett-Packard hat in den 1980-er Jahren versucht, mit Taschenrechnern die postfix-Notation zu verbreiten, ist aber an der Trägheit der Menschheit gescheitert. Schade.

Praktische Regel: In Zweifelsfällen sollten man sich nicht auf die Assoziativität und Priorität von Operatoren verlassen, sondern die Mehrdeutigkeit der infix-Notation durch runde Klammern beseitigen.

Alternative Bezeichnung für den 3-stelligen Bedingungsoperator: **Ausdrucks-if-then-else**

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 (in der Programmiersprache C++ kann man Unterprogramme innerhalb und außerhalb von Klassen vereinbaren).

Grundregel für alle Methoden:

Der Programmierer muss eine Methode *einmal* vereinbaren.
Dann darf er sie *beliebig oft* aufrufen (0 mal oder 1 mal oder 3 mal oder 5 Millionen mal ...)

2 Arten von Methoden: Prozeduren und Funktionen

Eigenschaft	**Prozedur**	**Funktion**
Sinn und Zweck:	**verändert** den Inhalt von Wertebehältern (Variablen, Bildschirm, Datei, ...)	berechnet und **liefert** einen Wert
Jeder Aufruf ist ...	eine **Anweisung**	ein **Ausdruck**
Rückgabetype	`void`	ein richtiger Typ wie `int` oder `String` oder ...
Beendigung	*darf* mit `return` erfolgen	*muss* mit `return` erfolgen
return	**ohne Ausdruck** dahinter	**mit Ausdruck** dahinter

Anmerkung: Offiziell ist `void` ein Typ, allerdings einer, zu dem 0 Werte gehören (ziemlich wenig).

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

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

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

Parameter und Argumente

Wenn man eine Methode *vereinbart*, beschreibt man (in den runden Klammern nach dem Namen der methode) 0 oder mehr **Parameter** (indem man einen Typ und einen Namen angibt).

Wenn man eine Methode *aufruft*, muss man für jeden Parameter ein entsprechendes **Argument** angeben.
Etwas vereinfacht gilt: Ein Argument ist ein *Ausdruck* vom Typ des entsprechenden Parameters.

Beispiel:

```
 1 // Vereinbarung:
 2 static String kombiniere(String s, int n) {
 3    return s + ": " + n;
 4 }
 5 ...
 6    // Aufrufe:
 7    String sm     = kombiniere("Mai", 150);
 8    String monat  = "Juni"
 9    int    betrag = 400;
10    int    rabatt = betrag*3/100;
11    pln(kombiniere("Summe " + monat, betrag-rabatt);
```

Zur Entspannung: **Selbst-zutreffende ("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 selbst-zutreffend.

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

Frage: Ist jedes Wort entweder selbst-zutreffend oder selbst-unzutreffend? Was ist mit dem Wort selbst-unzutreffend? Ist es selbst-zutreffend oder selbst unzutreffend?

Übungstermin-06

1. **Aufgabe 2** abgeben

2. In der Datei **pr1_Übungen.odt** die **Übung 5: Schleifen A (ausführen)** mit Papier und Bleistift ausführen.

3. **Aufgabe 3** bearbeiten (evtl. auch Aufgabe 4)

SU 7 Do 28.04.16, Block 4

Ausdrücke (Fortsetzung und Ergänzungen)

Was befiehlt der Programmierer dem Ausführer mit einem *Ausdruck*?
(Einen Wert zu berechnen)

Was bewirkt ein "normaler Ausdruck" nicht?
(Dass die Inhalte irgendwelcher Wertebehälter verändert werden, denn das ist die Aufgabe von *Anweisungen*).

Zu dieser letztgenannten Eigenschaft gibt es **Ausnahmen**:

Def.: Ein **Ausdruck mit Seiteneffekt** befiehlt dem Ausführer, einen Wert zu berechnen und ("nebenbei", als Seiteneffekt) die Inhalte bestimmter Wertebehälter zu verändern.

Beispiel-01: Ausdrücke mit den Inkrement- und Dekrement-Operatoren `++` und `--`

```
1 int n1=17, n2=17, n3=17, n4=17;
2
3                  // Nachher:
4 int m1 = n1++;  // m1:17, n1:18
5 int m2 = ++n2;  // m2:18, n2:18
6 int m3 = n3--;  // m3:17, n3:16
7 int m4 = --n4;  // m4:16, n4:16
```

Regel: Die Operatoren `++` und `--` darf man nur auf **Variablen** anwenden, nicht auf andere Ausdrücke.

Nicht erlaubte Anwendungen: `++13, ++(n1+n2), (2*n1)--` etc.

Erlaubte Anwendungen, die man unbedingt **vermeiden** sollte (mehrere Inkrement-/Dekrement-Operatoren in einem Ausdruck): `++n1 * ++n1, n1++ + --n1`

Beispiel-02: Eine Funktion kann einen Seiteneffekt haben. Trotzdem gilt jeder Aufruf der Funktion als ein Ausdruck.

```
static int plus3MS(int n) { // plus 3 mit Seiteneffekt
   int erg = n + 3;
   pln("Ergebnis: " + erg); // Seiteneffekt auf Bildschirm
   return erg;              // Das Ergebnis erg wird geliefert
}
```

Ausdrücke mit Seiteneffekt können sehr nützlich, aber auch gefährlich sein. Man sollte sie nur anwenden, wenn es keine gute andere Lösung gibt.

Schleifen: Die Kehrseite der arbeitsökonomischen Bedeutung

Schleifen machen den Programmierer sehr mächtig: Mit wenigen Zeilen "Schreib-Arbeit", etwa so:

```
for (int i=1; i<=100_000; i++) {
   ...
}
```

kann er dem Ausführer sehr viel "Ausführ-Arbeit" befehlen.

Die Kehrseite: Um eine Schleife zu verstehen, muss man sich die gesamte "Ausführ-Arbeit", die sie dem Ausführer befiehlt, vorstellen können. Das kann ziemlich schwierig sein.

Empfehlung: Üben Sie oft, Schleifen zu programmieren. Und üben Sie oft, Schleifen selbst auszuführen. Nur so wird man eine guter "Schleifen-Programmierer".

Typen (Buch S. 89)

Was ist ein Typ?
(Ein Bauplan für Variablen).
Wenn man dem Ausführer befehlen will, eine Variable zu erzeugen, muss man ihm den Bauplan nennen, nach dem er die Variable "bauen" oder erzeugen soll.

In Java unterscheidet man 2 Arten von Typen:
Primitive Typen und **Referenztypen**.

Es gibt **genau 8** primitive Typen (die in 4 Untergruppen eingeteilt sind):

numerische Typen	**Ganzzahltypen**	`byte`
		`char`
		`short`
		`int`
		`long`

	Bruchzahltypen	`float` `double`
nicht-numerischer Typ		`boolean`

In der Java-Standardbibliothek gibt es (zur Zeit, in Java 8) **ungefähr 4_200** Referenztypen.

Der Programmierer kann weitere Typen *vereinbaren* (d.h. vom Ausführer erzeugen lassen),
aber nur *Referenztypen*.
Die Anzahl der primitiven Typen (8) ist seit der ersten Version von Java *unverändert* geblieben.

Die **Namen aller primitiven Typen** bestehen nur aus Kleinbuchstaben.

Die **Namen aller Referenztypen** in der Standard-Bibliothek beginnen mit einem Großbuchstaben. Wenn der Programmierer eigene Typen vereinbart, kann er gegen diese Regel verstoßen, sollte das aber unbedingt vermeiden.

Was ein Java-Programmierer über die primitiven Typen wissen sollte

Typ T	Wie viele Bits belegt ein Wert des Typs T?	Wie viele Werte gehören zum Typ T?
`byte`	8	genau 256
`char`	16	ungefähr 65 Tausend
`short`	16	ungefähr 65 Tausend
`int`	32	ungefähr 4.3 Milliarden
`long`	64	ungefähr 19 Trillionen
`float`	32	ungefähr 4.3 Milliarden
`double`	64	ungefähr 19 Trillionen
`boolean`	8 oder 1 oder ...	genau 2

Zahlen in der Mathematik und Werte numerischer Typen (Buch S. 103)

Mathematiker stellen sich verschiedene Mengen von Zahlen vor, die ineinander *geschachtelt* sind. Jede dieser Mengen enthält *unendlich* viele Zahlen. Nach dieser Vorstellung gibt es z.B. *eine* Zahl `3`, die gleichzeitig eine *natürliche* Zahl, eine *ganze* Zahl, eine *rationale* Zahl (`3/1`), eine *reelle* Zahl (`3.0`) und eine *komplexe* Zahl (`3.0 + 0.0i`) ist. Die Zahl `2.5` ist eine *rationale* Zahl, eine *reelle* Zahl und eine *komplexe* Zahl etc.

Zur Erinnerung: 2 Mengen sind disjunkt, wenn kein Element in beiden Mengen enthalten ist (die Mengen {1, 3, 5} und {2, 4} sind disjunkt, die Mengen {1, 3, 5} und {2, 3} sind nicht disjunkt).

Java-Ausführer gehen von einer ganz anderen Grundsituation aus. Für sie gibt es
sieben *disjunkte*, *endliche* Mengen von Zahlen (S. 104, Bild 5.2).

In jeder dieser sieben Mengen gibt es z.B. einen Wert 3, und diese sieben Werte (byte-3, char-3, short-3, int-3, long-3, float-3, double-3) unterscheiden sich in wichtigen Eigenschaften voneinander. Z.B. gilt:

```
int-3   / int-2     ist gleich int-1.
float-3 / float-2   ist gleich float-1.5
char-3  / char-2    ist gleich int-1
int-3   / float-2   ist gleich float-1.5
```

2 wichtige Rechenregeln

Die drei Typen `byte`, `char`, `short` bezeichnen wir als "die kleinen Ganzzahltypen".

Rechenregel-01:
Der Programmierer darf dem Ausführer befehlen, mit Werten der kleinen Ganzzahltypen zu rechnen.
Der Java-Ausführer kann nicht mit Werten der kleinen Ganzzahltypen rechnen.
Deshalb wandelt er solche Werte in `int`-Werte um und rechnet mit den `int`-Werten (das kann er!).

Rechenregel-02:
Der Programmierer darf dem Ausführer befehlen, mit Werten *unterschiedlicher* Typen zu rechnen (z.B. einen `int`-Wert und einen `long`-Wert zu addieren oder einen `double`-Wert mit einem `long`-Wert zu multiplizieren etc.)
Der Java-Ausführer kann solche "gemischten Berechnungen" nicht durchführen.
Deshalb wandelt er "den Wert des kleineren Typs" in einen "Wert des größeren Typs" um und rechnet mit den beiden Werten des größeren Typs.

Beispiele:

```
int-5    + long-3 ist gleich long-8    // long   ist größer als int
double-7 * long-2 ist gleich double-14 // double ist größer als long
```

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 (z.B. als "Rückgrat" einer Hängebrücke). 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 (Platte, CD/DVD, Stick, ...) 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) vom selben Gerät 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.

Übungstermin-07

1. **Aufgabe 2** abgeben (letzter pünktlicher Termin)

2. In der Datei **pr1_Übungen.odt** die **Übung 6: Schleifen B (programmieren)** mit Papier und Bleistift ausführen.

3. **Aufgabe 3** bearbeiten (evtl. auch Aufgabe 4)

SU 8 Di 03.05.16, Block 1

Heute schreiben wir den Test-03

Organisation: Es ist sehr schwer bis unmöglich, für den **Konzepte-Teil** dieser Lehrveranstaltung eine ausreichende Note zu bekommen (durch das Bestehen einer Klausur), wenn man nicht während des ganzen Semesters auch den **Praxis-Teil** so intensiv mitmacht, wie es für das Bestehen des Praxis-Teils nötig ist.

Korrektur: Was ich letztes Mal **Zur Entspannung** erzählt habe, war falsch. Im Englischen sind *bootstraps* keine Schnürsenkel für einen Schnürschuh, sondern Schlaufen an einem Stiefel (eine hinten oder zwei an den Seiten), die das Anziehen erleichtern. Sorry, und Dank an Frau Schenk für die Aufklärung meines Fehlers.

Wiederholungsfragen: Wie viele *primitive* Typen gibt es in Java (8).
Wie viele *numerische* Typen? (7)
Wie viele *nicht-numerischen* Typen? (1)
Wie heißen die numerischen Typen? (`byte`, `char`, `short`, `int`, `long`, `float`, `double`).

Der cast-Operator (...)...

Seien T1 und T2 zwei *numerische Typen*. Mit dem cast-Operator kann man einen T1-Wert in einen (mehr oder weniger) entsprechenden T2-Wert umwandeln lassen (z.B. einen `int`-Wert in einen `double`-Wert oder einen `double`-Wert in einen `short`-Wert etc.).

Der cast-Operator ist ein ganz besonderer Operator: Er ist zweistellig und wird **mixfix** notiert (oder "in mixfix-Notation geschrieben").
Als 1. Operanden muss man (**in** den runden Klammern) den *Namen eines Typs* angeben.
Als 2. Operanden muss man (**nach** den Klammern) einen *Ausdruck* angeben.

Beispiel-01: Zur Erinnerung: `17` ist ein Literal vom Typ `int`, `17.0` ist ein Literal vom Typ `double`.

Der Ausdruck `(double) 17`	bezeichnet den Wert `17.0`	(siebzehn vom Typ `double`)
Der Ausdruck `(int) 17.0`	bezeichnet den Wert `17`	(siebzehn vom Typ `int`)
Der Ausdruck `(int) (1.3 + 1.0)`	bezeichnet den Wert `2`	(zwei vom Typ `int`)
Der Ausdruck `(int)  1.3 + 1.0`	bezeichnet den Wert `2.0`	(vom Typ `double`)

Was man mit dem cast-Operator und *Referenztypen* machen kann, besprechen wir später.

Besonderheiten des Ganzzahltyps char (Buch S. 114)

Ausgangspunkt: Der Typ `char` ist ein weitgehend normaler **Ganzzahltyp** (wie `byte`, `short`, `int` und `long`), denn die Werte des Typs `char` sind Ganzzahlen.

Ähnlich wie für die Typen `byte` und `short` gilt auch für den Typ `char`: Der Java-Ausführer kann **nicht** mit `char`-Werten rechnen (addieren, multiplizieren, ...). Wenn man es ihm trotzdem befiehlt (was erlaubt ist), wandelt er die `char`-Werte automatisch in `int`-Werte um und rechnet mit denen.

Beispiel-01:

```
1 char c1 = 5;
2 char c2 = 3;
3 int  n1 = c1 + c2;           // c1 + c2 ist vom Typ int (nicht char!)
4 char c3 = (char) (c1 + c2);  // (char) (c1 + c2) ist vom Typ char
5 char c4 =         c1 + c2    // Fehler: c1 + c2 ist vom Typ int!
```

Besonderheit 1: Zum Typ `char` gehören *keine negative Zahlen*, sondern folgende Ganzzahlen: ungefähr von `0` bis `65` Tausend (genau von `0` bis `65_535`).

Besonderheit 2: Wenn man einen `char`-Wert (z.B. zum Bildschirm) *ausgibt*, dann erscheint draußen keine *Zahl*, sondern ein bestimmtes *Zeichen*.

Merke: Nur beim *Ausgeben* von `char`-Werten "kommen Zeichen mit ins Spiel".

Buch S. 115, Beispiel-01

Wie wird der Ganzzahl-Wert *fünfundsechzig* ausgegeben? (das kommt auf seinen Typ an!)

Besonderheit 2.5: Es gibt spezielle *Literale* vom Typ `char`.
Da das auch für die Typen `int` und `long` gilt, zählt das hier nur als eine *halbe* Besonderheit :-).

Beispiele für char-Literale:

Das Literal ...	ist ein Name für den Wert ...	der ausgegeben wird als ...
`'A'`	`char-65`	`A`
`'Z'`	`char-90`	`Z`
`'a'`	`char-97`	`a`
`'z'`	`char-122`	`z`
`'0'`	`char-48`	`0`
`'9'`	`char-57`	`9`

Variablen als Bojen darstellen

Das Konzept einer Variablen, deren Wert man beliebig oft ändern kann, ist wohl das wichtigste Grundkonzept der meisten Programmiersprachen.

Beim Ausführen von Programmen "mit Papier und Bleistift" haben wir eine Variable bisher einfach als ein "nach rechts offenes Rechteck" dargestellt. Diese Darstellung ist sehr simpel und reicht in vielen Fällen aus.

Aber in einigen Fällen ist eine genauere Darstellung von Variablen sehr nützlich: Wenn man bestimmte Programme "mit Papier und Bleistift" ausführen oder bestimmte Eigenschaften von Java "anschaulich verstehen" will. In diesen Fällen werden wir die **Bojendarstellung** von Variablen verwenden.

Diese Darstellung wurde im Zusammenhang mit der Programmiersprache Algol68 (Ende der 1960-er-Jahren) erfunden und von Studenten der Beuth Hochschule ein bisschen verbessert. Diese Darstellung ist für alle Programmiersprachen geeignet, in denen es (veränderbare) Variablen gibt.

In den ersten Bojen-Beispielen werden wir unter anderem die Referenztypen `String` und `StringBuilder` verwenden. Zu beiden Typen gehören Zeichenketten wie z.B. `"Hallo!"`, `"X"` oder `""`. Die Unterschiede zwischen den beiden Typen kann man besonders leicht erklären und verstehen, wenn alle Beteiligten die Bojendarstellung von Variablen kennen.

Buch S. 119, Beispiel-01

Zeile 1: So kann die Vereinbarung (und Initialisierung) einer `StringBuilder`-Variablen aussehen.

Nach dieser Vereinbarung würde ein Befehl wie `pln(otto);` die Zeichenkette `"Hallo!"` (ohne die doppelten Anführungszeichen) zum Bildschirm ausgeben.

Regeln:

1. Eine Variable besteht aus mindestens 2 Teilen (einer Referenz und einem Wert)
2. Eine primitive Variable kann zusätzlich einen Namen haben.
3. Eine Referenz-Variable kann zusätzlich einen Namen und/oder einen Zielwert haben

Teile einer Variablen und Kästchen-Formen in einer Boje

Variablen-Teil	Kästchen-Form dtsch.	Kästchen-Form engl.
Name	Paddelbbot	canoe
Referenz	Sechseck	hexagon
Wert	Rechteck	rectangle
Zielwert	Rechteck	rectangle

Ein *primitiver* Wert steht nur in einem *Rechteck*. Ein Wert, der eine Referenz ist, steht (als Wert) in einem *Rechteck* und gleichzeitig (als Referenz) in einem *Sechseck*.

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 zerstörte 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.

Übungstermin-08:

1. **Aufgabe 3** abgeben (ist auch noch beim **Übungstermin-09** möglich)
2. **Übung 6: Schleifen B (programmieren)** fertig machen

Do 05.05.16: Himmelfahrt

SU 9 Di 10.05.16, Block 1

Heute schreiben wir den Test-04

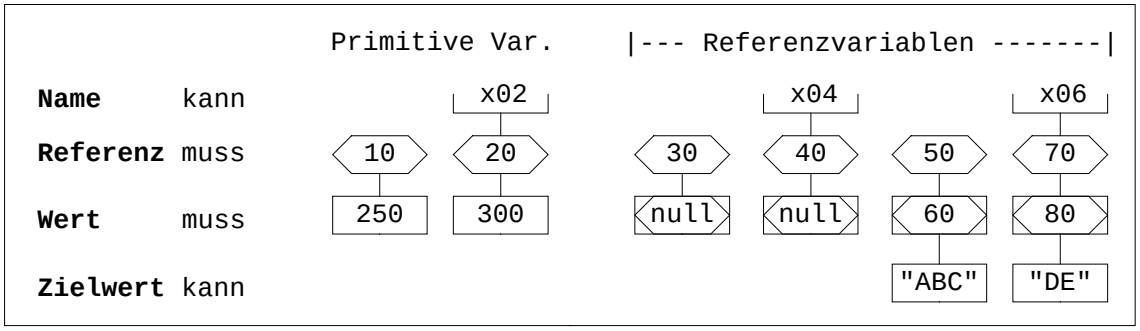
Variablen als Bojen darstellen, Fortsetzung

Eine Variable besteht aus *mehreren Teilen*.
Welche Namen wurden im vorigen SU für diese Teile eingeführt?
(Name, Referenz, Wert, Zielwert)

Wie heißen (in einer Boje) die Kästchen
für Namen? (Paddelboote, engl, canoes)
für Referenzen? (Sechsecke, engl. hexagons)
für Werte? (Rechtecke, engl. rectangles)

Aus wie vielen Teilen besteht eine Variable?

Primitive Variablen: 2-3 Teile, Referenzvariablen: 2-4 Teile



Merke: Viele Variablen (wahrscheinlich sogar die meisten) haben keinen Namen!

Begründung: Viele Programme enthalten Millionen oder noch mehr Variablen. Für jede dieser Variablen einen Namen zu erfinden und mehrmals hinzuschreiben wäre eine unzumutbare Belastung für den Programmierer.

Wenn der Wert einer Referenzvariablen
gleich `null` ist dann darf die Variable keinen Zielwert haben
ungleich `null` ist dann muss die Variable einen Zielwert haben

Der Wert `null` einer Referenzvariablen bedeutet: "Diese Variable zeigt nicht auf einen Zielwert!".

Wichtige "Grundregeln" für Variablen:

1. Eine *Zuweisung* an eine Variable x (`x = ...;`) verändert immer den *Wert* von `x`.

2. Ein *Vergleich* wie `x == y` (oder `x != y`) vergleicht immer die *Werte* von `x` und `y`.

3. In Java ist jeder Zielwert ein Objekt (in anderen Sprachen kann ein Zielwert auch ein `int`-Wert oder ein `double`-Wert etc., d.h. ein primitiver Wert, sein).

Beispiel-01:

```
StringBuilder sb1 = new StringBuilder("Hallo!");
StringBuilder sb2 = new StringBuilder("Hallo!");
```

Jetzt hat der Ausdruck `sb1 == sb2` den Wert `false` (und `sb1 != sb2` hat den Wert `true`), denn `sb1` und `sb2` haben unterschiedliche *Werte* (und die *Zielwerte* werden nicht verglichen).

Jedes Objekt einer Java-Klasse enthält eine equals-Methode

Aber was diese Methode genau macht, legt der Programmierer der betreffenden Klasse fest.

2 Beispiele, die man auswendig kennen sollte:

Beispiel 1: Die `equals`-Methode in `StringBuilder`-Objekten vergleicht die **Werte** von `StringBuilder`-Variablen (nicht die Zielwerte).

Beispiel 2: Die `equals`-Methode in `String`-Objekten vergleicht die **Zielwerte** von `String`-Variablen (nicht die Werte).

```
String s1 = new String("Hallo!");
String s2 = EM.liesString();

if (s1.equals(s2)) ...
if (s2.equals(s1)) ...
if ("Hallo!".equals(s2)) ...
```

Siehe dazu auch im Papier **pr1_JavaBuoys.pdf** die Seite 4.

Viele Variablen auf einmal vereinbaren: Reihungen

Buch S. 150

Zur Entspannung: **Eigenschaften von Qubits (Quanten-Bits)**

1. Mit n "normalen Bits" kann man *eine* Zahl (zwischen 0 und 2^n-1) darstellen. Mit n Qubits kann man gleichzeitig bis zu 2^n Zahlen (zwischen 0 und 2^n-1) darstellen und mit *einer* Operation kann man alle diese Zahlen "gleichzeitig bearbeiten".

Angenommen wir haben einen Speicher S, der aus 20 Qubits besteht. Dann können wir darin etwa 1 Million Zahlen (zwischen 0 und etwa 1 Million) speichern und mit *einer* Operation all diese Zahlen auf einmal verändern. Wenn S aus 30 Qubits besteht, gilt alles entsprechend, aber mit 1 Milliarde Zahlen, und bei 40 Qubits mit 1 Billion Zahlen etc.

2. Wenn man unseren Speicher S "ansieht und ausliest", bekommt man allerdings nur *eine* der Zahlen, die sich im Speicher befinden. Die übrigen Zahlen gehen dabei unvermeidbar und unwiderruflich verloren. Während mit unserem Speicher S gerechnet wird, muss der Speicher deshalb sorgfältig von der Umwelt isoliert werden, denn fast jede Interaktion des Speichers mit der Umwelt zählt als "ansehen und auslesen".

3. Es ist nicht möglich, ein Qubit (mit all seinen Werten) zu *kopieren*. Man kann höchstens *einen* seiner Werte kopieren (und die übrigen Werte gehen dabei verloren).

4. Auf Qubits kann man nur *umkehrbare Verknüpfungen* anwenden.

Zur Zeit (2008) erforschen mehrere Tausend Physiker, Informatiker und Ingenieure in mehr als 100 Forschungsgruppen etwa ein Dutzend Möglichkeiten, Qbits zu realisieren (durch ion traps, quantum dots, linear optics, ...).

Eine interessante und lesbare Einführung in die Quantenmechanik. Die Autorin war Schülerin an einem Berliner Gymnasium, als sie dieses Buch schrieb:

Silvia Arroyo Camejo: "Skurrile Quantenwelt", Springer 2006, Fischer 2007

Anfang Mai 2016 hat die Firma IBM einen Quantencomputer (mit 5 Qubits) ins Internet gestellt (siehe http://www.research.ibm.com/quantum/)

Übungstermin-09

1. Übungsgruppe **1a**, Team Team123: Letzter pünktlicher Abgabetermin für **Aufgabe 3**.
Team Boss: Was ist mit **Aufgabe 2**?

2. **Aufgabe 4**. vorführen. Ein grüner Balken vom Testprogramm ist notwendig, aber nicht hinreichend für eine akzeptable Lösung. Möglicherweise sind kleine Überarbeitungen (Entfernung überflüssiger Teile) notwendig.

3. Alle, die es bisher noch nicht gemacht haben:
In Datei **pr1_Uebungen.odt**, **Übung 2: Gleitpunktzahlen** zumindest teilweise bearbeiten.

4. Im **Übungstermin-01** hat (hoffentlich) jeder einen Ordner **Z:\BspJaSp** angelegt. Darin die Datei **Switch01.java** öffnen die `switch`-Anweisung besprechen (die braucht man für **Aufgabe 5**).

5. **Aufgabe 5** bearbeiten.

SU 10 Do 12.05.16, Block 4

Die equals-Methoden der Klassen String und StringBuilder

Papier **pr1_JavaBuoys.pdf**, S. 4, Example-03.

Der `new`-Befehl garantiert, dass die 4 Variablen `st01`, `st02`, `sb01`, `sb02` mit 4 verschiedenen Werten initialisiert werden (im Beispiel: `<55>, <56>, <57>, <58>`).

Der Ausdruck `st01.equals(st02)` hat den Wert `true` (weil die *Zielwerte* gleich sind).
Der Ausdruck `sb01.equals(sb02)` hat den Wert `false` (weil die *Werte* ungleich sind).

Ausführliche und abgekürzte Bojendarstellung von Reihungen (p. 6, Example-01)

Elaborate and abbreviated buoys.

Initialisierung von Reihungskomponenten

Wenn der Programmierer nicht ausdrücklich etwas anderes befiehlt, werden die Komponenten einer Reihung (bei Erzeugen der Reihung) *standardmäßig* initialisiert, d.h.

Typ der Komponenten	Initialisierungswert
`byte, char, short, int, long`	`0`
`float, double`	`0.0F bzw. 0.0`
`boolean`	`false`
irgendein Referenztyp	`null`

```
// "Gleichwertige Vereinbarungen":
boolean[] bra = new boolean[3];          // So ...
boolean[] brb = {false, false, false};   // ... oder so.
String[]  sra = new String[3];           // So ...
String[]  srb = {null, null, null};      // ... oder so.
```

Objekte die sich (angeblich) gleichzeitig in mehreren Reihungen befinden

p. 7, Exampl-02

In Java gilt: Eine Reihung des Typs `String[]` enthält nicht wirklich `String`-Objekte (die sehr groß sein können), sondern nur (relativ kleine) *Referenzen*, die auf `String`-Objekte zeigen (oder: referieren). Entsprechendes gilt für alle Reihungen, deren Komponenten zu einem *Referenztyp* (und nicht zu einem *primitiven Typ*) gehören.

Die Reihungen `ar` und `ar2` könnten wie folgt vereinbart worden sein:

```
String[] ar  = {"AB", "C", "DEF"};
String[] ar2 = {ar[2], ar[1], ar[0]};
```

Unterschied zwischen null-Komponenten und leeren String-Objekten

p. 7, Example-03

Analogie: "Ich habe keine Kaffee-Tasse" und "Ich habe eine Kaffee-Tasse, aber die ist leer"

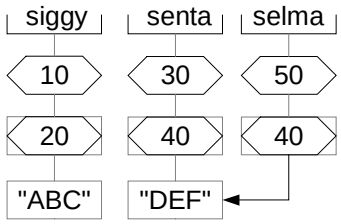
Eine Reihung in drei Schritten oder in einem Schritt erzeugen lassen

p. 8, top

Aufgabe-01: Stellen Sie (jetzt gleich) die folgenden Variablen als Bojen dar:

```
StringBuilder siggy = new StringBuilder("ABC");
StringBuilder senta = new StringBuilder("DEF");
StringBuilder selma = senta;
```

Lösung-01:



Was bezeichnet der Name einer Referenzvariable?

Angenommen, wir haben folgende Referenzvariablen:

```
StringBuilder xxx = new StringBuilder("ABC");
String        yyy = new String("ABC");
String        zzz = ...;              // Mit null oder new ... initialisiert
```

Die Teile der Variablen `xxx`, `yyy` und `zzz` als (ASCI-) Bojen dargestellt:

```
Name          |xxx|     |yyy|     |zzz|
                |         |         |
Referenz      <300>     <400>     <500>
                |         |         |
Wert         [<310>]   [<410>]   [<???>]
                |         |
Zielwert     ["ABC"]   ["DEF"]
```

In jedem der folgenden 8 Befehle kommt der Name einer Referenzvariablen vor.
Welchen *Teil* der Variable bezeichnet dieser Name in den einzelnen Befehlen?
(Auch hier soll `pln` eine Abkürzung für `System.out.println` sein).

```
                            // Der Name xxx bzw. yyy bzw. zzz bezeichnet:
1 if (xxx == ...) ...       // Den Wert [<310>]
2 if (yyy == ...) ...       // Den Wert [<410>]
3 if (xxx.equals(...)) ... // Den Wert [<310>]
4 if (yyy.equals(...)) ... // Den Zielwert ["DEF"]
5 xxx.append("ZZ");         // Den Zielwert ["ABC"]
6 xxx = ... ;               // Die Referenz <300>, die auf das Wertkästchen
                            // zeigt (der "alte Wert" [<310>] ist hier
                            // nicht wichtig, er wird ersetzt).
7 ... = xxx;                // Den Wert [<310>]
8 pln(zzz);                 // Den Wert oder den Zielwert, je nachdem
                            // ob der Wert gleich oder ungleich null ist
                            // (ausgegeben wird null bzw. der Zielwert).
```

Der Name einer Variablen bezeichnet also manchmal die *Referenz*, manchmal den *Wert* und manchmal den *Zielwert* der Variablen. Welcher Teil bezeichnet wird, muss man aus dem Zusammenhang ("aus dem umgebenden Befehl") erkennen.

Zur Entspannung: **Christian Morgenstern** (1871-1914)

Der Werwolf

Ein Werwolf eines Nachts entwich ...

Übungstermin-10:

1. Letzter Termin für die pünktliche Vorführung von **Aufgabe 4** (Dreiseits, Klasse Triple).
2. **Aufgabe-05** (Klasse InWorten) bearbeiten
3. **Aufgabe-06** (Klasse Schleifen) bearbeiten

SU 11 Di 17.05.16, Block 1

Heute schreiben wir den Test-05

Aufgabe-01: Vereinbaren Sie 500 `int`-Variablen und weisen Sie jeder dieser Variablen einen der Werte 10, 20, 30, ... 5000 zu.

Lösung-01:

```
int[] rolf = new int[500];
for (int i=0; i<rolf.length; i++) rolf[i] = 10 * (i+1);
```

Fachbegriffe und Regeln zum Thema Reihungen

1. Ein *Reihungstyp* ist ein Typ, dessen Name mit `[]` endet (z.B. `int[]` oder `String[]` etc.).
2. Reihungstypen sind *Referenztypen* (und keine primitiven Typen).
Auch `int[]` ist ein *Referenztyp*, obwohl `int` ein *primitiver* Typ ist!
3. Reihungstypen kann (und muss) man *nicht* vereinbaren. Wenn man einen Typ `T` vereinbart gibt es sofort und automatisch auch den Reihungstyp `T[]` (Reihung von `T`).
4. Eine Reihung des Typs `int[]` hat Komponenten vom Typ `int`.
Man sagt auch: `int` ist der *Komponententyp* des Reihungstyps `int[]`.
Allgemein: `T[]` ist ein Reihungstyp mit dem Komponententyp `T`.
5. Die *Indizes* einer Reihung sind (in Java) immer vom Typ `int`.
6. Eine Reihung der Länge 10 hat Indizes von 0 bis 9.
Allgemein: Eine Reihung der Länge N hat Indizes von 0 bis N-1.

Mehrstufige Reihungen (engl. nested arrays)

Regel: Zu jedem Typ `T` gibt es einen Reihungstyp `T[]` (gesprochen: Reihung von T-Variablen)

Diese Regel gilt auch für Reihungstypen:

Zum Typ	**gibt es den Reihungstyp**	**gesprochen**
`T[]`	`T[][]`	Reihung von Reihungen von T
`T[][]`	`T[][][]`	Reihung von Reihungen von Reihungen von T
`T[][][]`	`T[][][][]`	Reihung von Reihungen von Reihungen von Reihungen von T
`...`	`...`	`...`

Datei **pr1_JavaBuoys.pdf**, **S. 8** unten: Die Vereinbarung einer 2-stufigen-Reihung **a2b**
und auf **S. 9** oben eine Bojen-Darstellung von **a2b**.

Aufgabe-01: Stellen Sie die Variable **a2a** (vereinbart auf **S. 8** unten) als Boje dar.

Mehrdimensionale Reihungen

Gibt es z.B. in Fortran oder Pascal (aber nicht in Java). In C++ gibt es mehrstufige und mehrdimensionale Reihungen. Deshalb sollte man die Begriffe *mehrstufig* (engl. *nested*) und *mehrdimensional* (engl. *multidimensional*) möglichst selten verwechseln.

Datei **pr1_Mehrdimensional.pdf**, S. 2: Eine **2-dimensionale** Reihung

Kurzfassung des Unterschieds: Mit *mehrstufigen* Reihungen kann man ein bisschen mehr machen (Stichwort: Flatterrand), aber *mehrdimensionale* Reihungen sind (ein bisschen und in speziellen Fällen deutlich) effizienter.

for-each-Schleifen zur Bearbeitung von Reihungen (und Sammlungen)

Angenommen, wir haben eine Reihung `rolf` mit irgendwelchen Werten darin:

```
int[] rolf = ... ;
```

Alle Komponenten der Reihung `rolf` kann man z.B. so ausgeben:

```
for (int n : rolf) pln(n);
```

Diese **for-each**-Schleife bewirkt garantiert exakt das Gleiche wie folgende **for-i**-Schleife:

```
for (int i=0; i<rolf.length; i++) {
   int n = rolf[i];
   pln(n);
}
```

Wichtige Vorteile von for-each-Schleifen

1. Sie sind einfacher zu lesen (und zu schreiben)

2. Sie sind garantiert keine Endlos-Schleifen!

Aufgabe-02: Was bewirkt die folgende for-each-Schleife?

```
for (int n : rolf) n = 17;
```

Lösung-02: Sie bewirkt garantiert exakt das Gleiche wie die folgende Schleife:

```
for (int i=0; i<rolf.length; i++) {
   int n = rolf[i];
   n = 17;
}
```

Diese Schleife bewirkt **nichts**: Die Reihung `rolf` bleibt völlig unverändert
(und die Variable `n` wird zerstört, wenn die for-Schleife fertig ausgeführt ist).

Falls noch Zeit ist: **Aufgabe-03**:

```
static void kop(int[] orig, int[] kopi) {
   // Kopiert alle Komponenten von orig nach kopi
```

Wichtiger Nachteil von for-each-Schleifen

Bestimmte Probleme kann man mit **for-i**-Schleifen lösen, aber *nicht* mit **for-each**-Schleifen (siehe z.B. Aufgabe-02 und Aufgabe-03).

Empfehlung: Benutzen Sie wann immer es möglich ist **for-each-Schleifen** (statt for-i-Schleifen). Nur so entwickelt man mit der Zeit ein Gefühl dafür, welche Probleme man mit ihnen lösen kann und welche nicht.

Zur Entspannung: **Der EPR-Effekt**

Albert Einstein mochte die Quantenmechanik nicht. Er dachte sich mehrere Gedankenexperimente aus, die zeigen sollten, dass sie fehlerhaft ist oder zumindest "keine vollständige Beschreibung der Natur" liefert. Nils Bohr vertrat die Quantenmechanik und fühlte sich zuständig für ihre Verteidigung. Mehrmals gelang es ihm, in einem Gedankenexperiment von Einstein einen subtilen Fehler zu entdecken und die Argumente von Einstein dadurch zu entkräften. Bei *einem* der Gedankenexperimente gelang ihm das aber nicht.

Einstein mochte die Quantenmechanik nicht, kannte sie aber offenbar so genau, dass er auch einige ihrer entfernten Konsequenzen erkennen konnte. Den Effekt, der heute als *EPR-Effekt* bezeichnet wird (nach **E**instein, seinem Physiker-Kollegen **P**odolski und einem Studenten **R**osen) ist eine Konsequenz der Quantenmechanik, die Einstein "so unsinnig und unglaubhaft" vorkam, dass er sie als *Argument gegen die Quantenmechanik* veröffentlichte (und dabei ironisch als "spukhafte Fernwirkung" bezeichnete). Inzwischen hat man diesen "spukhaften Effekt" experimentell nachgewiesen und benutzt ihn zur *abhörsicheren Übertragung von Daten*.

Übungstermin-11:

1. Die **Aufgabe 5** (Klasse `InWorten`) vorführen (in Anwesenheit *aller Team-Mitglieder*).

2. Im Internet (z.B. mit Google) nach **Java 8 api docs** suchen und **Java 8 API - Oracle Help Center** wählen oder direkt zu https://docs.oracle.com/javase/8/docs/api/ gehen.

Strg-F und in das Suchfenster **String** eintippen.
Solange weitersuchen, bis die Klasse **String** gefunden ist.

Machen Sie sich mit der *Struktur der Dokumentation* einer Klasse (z.B. **String**) vertraut:

A. Allgemeine Einleitung

B. Field Summery
C. Constructor Summery
D. Method Summery

E. Methods inherited from ...
Methods inherited from ...
...

F. Field Detail
G. Constructor Detail
H. Method Detail

Versuchen Sie, die Antworten auf folgende Fragen herauszufinden:

- Wie viele Attribute (engl. fields) wurden in den Klasse `String` vereinbart?
- Wie viele Konstruktoren wurden in der Klasse `String` vereinbart?
- Wie viele Methode namens `indexOf` wurden in der Klasse `String` vereinbart?

3. Aus der Datei **pr1_AlleUebProgs.txt**
die Dateien **UebReihungenA.java** und **UebReihungenA_Jut.java** extrahieren
und in der Datei **UebReihungenA.java** die mit `// SOLLTE ERSETZT WERDEN` gekennzeichneten Zeilen durch geeignete Befehle ersetzen und mit dem Testprogramm **UebReihungenA_Jut.java** testen.

4. Die **Aufgabe 6** (Schleifen) bearbeiten.

SU 12 Do 19.05.16, Block 4

Mehrstufige Reihungen (engl. nested arrays) in Java

Grundregel: Zu jedem Typ `T` gibt es einen Typ `T[]` (sprich: Reihung von T-Variablen).

Diese Regel gilt insbesondere auch für *Reihungstypen*:

Zum Typ	gibt es den Reihungstyp	gesprochen
`T[]`	`T[][]`	Reihung von Reihungen von T (-Variablen)
`T[][]`	`T[][][]`	Reihung von Reihungen von Reihungen von T (-Variablen)
`T[][][]`	`T[][][][]`	Reihung von Reihungen von Reihungen von Reihungen von T (-Variablen)
`...`	`...`	`...`

Datei **pr1_JavaBuoys.pdf**, **S. 8** unten, **Example-02**: Die Vereinbarung einer 2-stufigen-Reihung **a2b** und auf **S. 9** oben eine Bojen-Darstellung von **a2b**.

Problem-03: Stellen Sie die Variable **a2a** (aus dem **Example-02**) als Boje dar (jetzt gleich, mit Papier und Bleistift).

Wichtige Empfehlung für das Zeichnen von Bojen:

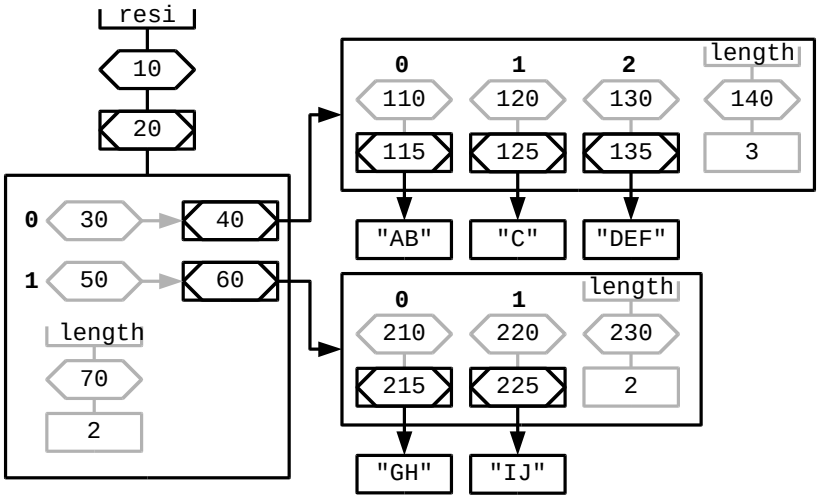
Eine *Reihung* wird immer in einem *Rechteck* dargestellt. Wenn man zuerst das Rechteck zeichnet, und dann seinen Inhalt (die Komponenten der Reihung, ihre Indizes `0, 1, 2,`..., die `length`-Variable), ist das Rechteck immer *zu klein*. (ein bisher noch unerklärtes Naturgesetz :-)).
Deshalb sollte man immer *zuerst den Inhalt* und erst dann das Rechteck drumherum zeichnen!

Example-03: A nested array with empty elements and null-elements (page 9)

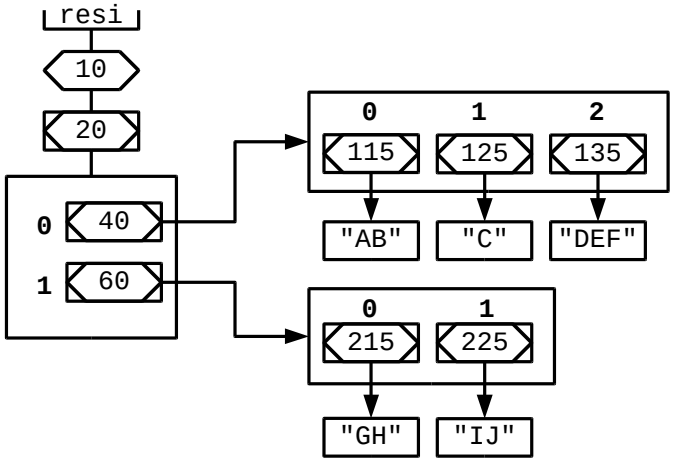
Aufgabe-01: Stellen Sie die folgende 2-stufige Reihung `resi` als Boje dar:

```
String[][] resi = {{"AB", "C", "DEF"}, {"GH", "IJ"});
```

Lösung-01a: (in ausführlicher Darstellung, die grau gezeichneten Teile kann man auch weglassen):



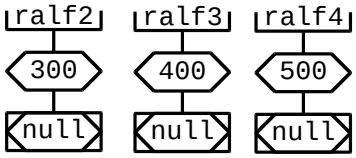
Lösung-01b: (in abgekürzter Darstellung, "ohne die grau gezeichneten Teile"):



Aufgabe-02: Stellen Sie die folgenden Variablen `ralf2` bis bis `ralf4` als Boje dar:

```
String[][]     ralf2 = null;
String[][][]   ralf3 = null;
String[][][][] ralf4 = null;
```

Lösung-02:



Aufgabe-03: Vereinbaren Sie eine Variable namens `ria3` vom Typ `int[][][]` und initialisieren Sie die Variable so, dass sie auf eine 2x3x2-Reihung zeigt und die `int`-Werte 1 bis 12 enthält.

Lösung-03:

```
int[][][] ria = {{{1, 2}, {3, 4}, {5, 6}}, {{7, 8}, {9, 10}, {11, 12}}};
```

oder mit einem etwas anderen Layout:

```
int[][][] ria = {
                   {
                      {1, 2}, {3, 4}, {5, 6}
                   },
                   {
                      {7, 8}, {9, 10}, {11, 12}
                   }
                };
```

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 17.000 Pfund der britischen Regierung, was etwa dem Preis von zwei Schlachtschiffen entsprach. Dann brach die Regierung das Projekt ab.
1833 **Analytical Engine**: Die sollte sogar schon programmierbar werden und Lochkarten lesen.
1847 **Difference Engine no. 2**: Verbesserte Variante der Difference Engine no. 1.
1985 Die **Difference Engine no. 2** wurde im Science Museum in London nach den Plänen von Babbage gebaut (das Rechenwerk war 1991 fertig, das Druckwerk 2002, alles funktioniert).
2012 Es gibt einen Plan, auch die **Analytical Engine** zu bauen (siehe http://plan28.org/).

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 Programmiersprache **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.

Übungstermin-12:

Aufgabe 5 (Klasse `InWorten`) vorführen.

Übung 10: ReihungenA (Dateien **UebReihungenA.java** und **UebReihungenA_Jut.java**).

SU 13 Di 24.05.16., Block 1

Heute schreiben wir den Test-06

Hinweis: Im Tutorium am **Mo 30.05.2016** wird Herr Klatt *Klausuraufgaben* besprechen.

Was ist ("beim Programmieren") eine Klasse?

Def.: Ein **Modul** ist ein Behälter für Variablen, Unterprogramme, Typen, Module etc., der aus mindestens 2 Teilen besteht: Einem öffentlichen und einem privaten Teil. Von außerhalb des Moduls kann man nur auf die Dinge im öffentlichen Teil zugreifen.

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

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

Die nach einem solchen Bauplan gebauten Module bezeichnet man
als **Objekte** oder als **Instanzen der Klasse**.

Eine Klasse ist so etwas wie ein "Kombi-Werkzeug", wie z.B. ein "Kugelschreiber-plus-Flaschenöffner". Wir sagen auch: Eine Klasse hat einen **Modul-Aspekt** und einen **Bauplan-Aspekt**.

Buch S. 203, **Beispiel-01**: Die Klasse `Zaehler01`

In Zeile 1 bis 22 steht *ein einziger Befehl*: Eine Klassen-**Vereinbarung**
(auf Deutsch etwa: "Erzeuge eine Klasse namens `Zaehler01` mit folgenden Eigenschaften ...").

Eine Klassen-Vereinbarung darf direkt nur **Vereinbarungen** enthalten
(keine Ausdrücke und keine Anweisungen).

Aber: Die Vereinbarung der Klasse `Zaehler01` enthält unter anderem (in den Zeilen 16 bis 18) eine Methoden-Vereinbarung und die darf Anweisungen enthalten und Anweisungen dürfen Ausdrücke enthalten. Diese Anweisungen und Ausdrücke stehen aber nicht **direkt** in der Vereinbarung der Klasse!

Wie viele Vereinbarungen enthält die Vereinbarung der Klasse `Zaehler01`? (6 Stück)

Innerhalb einer Klassenvereinbarung kann man (nicht nur Attribute und Methoden sondern) auch **Konstruktoren** vereinbaren (siehe Zeile 8 bis 10). In der Klasse `Zaehler01` wird *ein* Konstruktor vereinbart.

Mit Ausnahme der Konstruktoren bezeichnet man alle Dinge, die innerhalb einer Klassenvereinbarung vereinbart wurden als **Elemente** der Klasse (engl: **members** of the class).

In der Klasse `Zaehler01` werden 5 Elemente und 1 Konstruktor vereinbart.

Wie heißen die 5 Elemente und um was für Elemente handelt es sich?
(`anzahl`: Attribut, `getAnzahl`: Methode, `punkte`: Attribut, `add`: Methode, `getPunkte`: Methode). Insgesamt: 2 Attribute und 3 Methoden.

Konstruktoren und alle mit **static** gekennzeichneten *Element* gehören zum **Modul**-Aspekt der Klasse.

Alle nicht mit **static** gekennzeichneten Elemente gehören zum **Bauplan**-Aspekt der Klasse.

Wenn der Ausführer während der Ausführung eines Programms merkt, dass er eine bestimmte Klasse braucht (z.B. die Klasse `Zaehler01`) dann erzeugt er erst mal nur den Modul `Zaehler01`.

Buch S. 204, **Bild 9.1**

Dieser Modul enthält eine Variable (`anzahl`), eine Methode (`getAnzahl`) und einen Konstruktor (der natürlich `Zaehler01` heißt). Außerdem (so kann man sich vorstellen) enthält dieser Modul einen Bauplan für Objekte (aber nur den Bauplan, keine "fertig gebauten" Objekte).

Buch S. 205, **Beispiel-02**

Den Befehl in Zeile 25 überspringen wir erst mal und betrachten gleich den Befehl in Zeile 26:

```
26   Zaehler01 zelia = new Zaehler01(10);
```

Die Ausführung erfolgt in 4 Schritten

Schritt 1: Der `new`-Befehl erzeugt ein neues `Zaehler01`-Objekt, siehe Bild 9.3 auf S. 206
Schritt 2: Der Konstruktor wird mit dem Argument `10` ausgeführt, siehe Bild 9.4
Schritt 3: Der `new`-Befehl liefert eine Referenz (hier: `<50>`), die auf das neue Objekt zeigt.
Schritt 4: Eine Variable namens `zelia` vom Typ `Zaehler01` mit dem Anfangswert `<50>` wird erzeugt.

Aufgabe-00: Führen Sie (mit Papier und Bleistift) den Befehl auf S. 205, in **Zeile 25** (Vereinbarung der Variablen `zafer`) aus und erklären Sie Ihrem Nachbarn, warum die `long`-Variable `zafer.punkte` am Ende den Wert `0` enthält und nicht `-5`:

Zu Konstruktoren:

1. Unmittelbar nach jedem `new`-Befehl muss man einen Konstruktor aufrufen.
2. Einen Konstruktor darf man nur unmittelbar nach einem `new`-Befehl aufrufen.
3. Der Programmierer darf in einer Klassenvereinbarung beliebig viele Konstruktoren vereinbaren (0 oder mehr).
4. Die Bezeichnung "Konstruktor" ist unglücklich gewählt: Objekte werden vom `new`-Befehl erzeugt (oder: konstruiert) und standardmäßig initialisiert. Ein Konstruktor dient nur dazu, das gerade neu erzeugte Objekt "anders als standardmäßig" zu initialisieren. Konstruktoren sollten besser "Initialisierer" oder so ähnlich heißen.

Aufgabe-01: Führen Sie (mit Papier und Bleistift) den Befehl auf S. 205, in **Zeile 27** (Vereinbarung der Variablen `zelma`) aus und stellen Sie die Variable als Boje dar.

Aufgabe-02: Welchen Wert hat das Attribut `Zaehler01.anzahl`, nachdem der Ausführer die Zeilen 25 bis 27 (d.h. die Vereinbarungen von `zafer`, `zelia` und `zelma`) ausgeführt hat?

Buch S. 208, Beispiel-03: Die Objekte `zafer`, `zelia` und `zelma` benutzen

Buch S. 208, Aufgabe3: Warum bewirkt der Befehl `zelma.add(20);` (in Zeile 33) keine Erhöhung der Variablen `zelma.punkte`?

(Die `add`-Methode, siehe S. 203, Zeilen 16 bis 18, erhöht die `punkte` nur, wenn der Parameter `n` einen Wert zwischen 1 und 10 hat).

Buch S.209, Aufgabe-04: Ein Programm namens `Zaehler01Tst02` schreiben.

Was leistet oder nützt die Klasse Zaehler01?

Sie erlaubt es, mit wenig Aufwand, viele "Punktekonten" (d.h. `Zaelehr01`-Objekte) einzurichten.
Sie garantiert, dass ein Punktekonto
- nie eine negative Zahl enthalten kann.
- am Anfang einen Wert zwischen 0 und 10 enthält
- bei jeder Erhöhung um höchstens 10 Punkte erhöht werden kann
- nie vermindert werden kann.

Zur Entspannung: **Was ist eine Zahl? Entwicklung des Zahlenbegriffs**

Griechische Mathematiker kannten die natürlichen Zahlen 1, 2, 3, ..., und außerdem sog. *Verhältnisse* oder *Proportionen* (z.B. 3:2 oder 4:7 etc.), aber diese "Verhältnisse" wurden von Euklid und Aristoteles nicht als Zahlen anerkannt. Heute bezeichnen wir diese "Verhältnisse" als **rationale Zahlen** ("vernünftige Zahlen").

Für griechische Mathematiker schockierend war die Erkenntnis, dass es außer ganzen Zahlen und Verhältnissen noch andere Größen gab, z.B. die Diagonale eines Quadrats mit Kantenlänge 1 (d.h. die Wurzel aus 2). Diese "anderen Größen" wurden später als **irrationale Zahlen** ("unvernünftige Zahlen") bezeichnet, um sie von den rationalen Zahlen zu unterscheiden. Heute erscheinen uns Zahlen wie die Wurzel aus 2 keineswegs mehr als *unvernünftig*, aber die Bezeichnung **irrationale Zahlen** wurde beibehalten.

Vermutlich wurden Zahlen mit einem Minuszeichen davor längere Zeit benutzt, ohne als "eigenständige Zahlen" anerkannt zu sein. Das Minuszeichen wurde als "Modifizierer der Bedeutung der Zahl" verstanden. Bei Geldbeträgen bedeutete +12 ein Guthaben, -12 dagegen eine Schuld. Bei Höhenangaben bedeutet +250m "über dem Meeresspiegel", -250m dagegen "unter dem Meeresspiegel" etc.

Im 16. Jahrhundert begann man, mit Wurzeln aus negativen Zahlen zu rechnen, ohne diese Gebilde gleich als Zahlen anzuerkennen. Später bezeichnete man diese Gebilde als **imaginäre Zahlen** ("eingebildete, unwirkliche Zahlen"). Heute kommt uns die Wurzel aus -1 (die Zahl i) nicht mehr als "*unwirklich*" oder "*nur eingebildet*" vor, aber die Bezeichnung **imaginäre Zahlen** wurde beibehalten.

Übungstermin-13:

Aufgabe 6 (Schleifen) bearbeiten. evtl. schon abgeben.

Übung 10: ReihungenA (Dateien **UebReihungenA.java** und **UebReihungenA_Jut.java**).

SU 14 Do 26.05.16, Block 4

Def: Unter einem **Standard-Konstruktor** versteht man einen Konstruktor, der 0 Parameter hat.

Die Hartz-IV-Regel für Konstruktoren:
Wenn der Programmierer eine Klasse K *ohne einen Konstruktor* vereinbart, dann bekommt die Klasse K vom Ausführer einen *Standard-Konstruktor mit leerem Rumpf* "geschenkt".

Damit gilt: Jede Klasse hat mindestens *einen* Konstruktor.

Besondere Regeln für Strings und String-Literale

String ist der einzige Referenztyp, für den es viele Literale gibt (z.B. `"ABC"` oder `""` etc.). Für jeden anderen Referenztyp gibt es nur das *eine* Literal `null`.

Weil es `String`-Literale gibt, kann man eine Stringvariable auf 2 Weisen initialisieren:
Mit `new` oder nur mit einem String-Literal.

Buch S. 230, Beispiel-03

Regeln:
1. Alle mit dem gleichen `String`-Literal (z.B. `"Hallo!"`) initialisierten `String`-Variablen haben garantiert *gleiche Werte* (und zeigen damit auf *dasselbe* Objekt).

2. Alle mit `new` initialisierten `String`-Variablen haben garantiert *unterschiedliche Werte* (und zeigen damit auf *verschiedene* Objekte).

Elemente nach verschiedenen Gesichtspunkten in Gruppen einteilen (S. 216)

Innerhalb einer Klassenvereinbarung kann man *Konstruktoren* ("Initialisatoren") und
Elemente (engl. members) vereinbaren.
Die **Elemente** kann man nach verschiedenen Gesichtspunkten in Gruppen einteilen:

Einteilung nach	**Gruppen**
Art	Attribute (engl. field), Methoden, Klassen, Schnittstellen.
Aspekt	Klassenelemente (static members), Objektelemente (non-static members).
Erreichbarkeit	öffentliche (public), geschützte (protected), paketweit erreichbare, private (private).

Es gibt alle möglichen Kombinationen dieser Eigenschaften, also z.B.
öffentliche Klassenattribute, öffentliche Objektattribute, öffentliche Klassenmethoden, ...
paketweit erreichbare Objektmethoden, ...,
private Klassen-Klassen (private static classes), geschützte Objekt-Schnittstellen, ...

Insgesamt gibt es also 4x2x4 gleich 32 verschiedene Charaktere von Elementen (wir verwenden hier den Begriff *Charakter* (Mehrzahl: *Charaktere*), weil die Worte *Typ, Art, Gruppe* schon für andere Bedeutungen reserviert sind).

Buch S. 218, Beispiel-01

Was versteht man unter "objektorientierter Programmierung"? (S. 219)

Die Klasse String (S. 222)

Bitte durchlesen. Wenn Fragen auftauchen: In der nächsten Ü fragen.

S. 225, Beispiel-06

`String`-Objekte mit der Objektmethode `compareTo` vergleichen

Ein Ausdruck wie `sa.compareTo(sb)` liefert einen negativen `int`-Wert, den Wert `0` bzw. einen positiven `int`-Wert, je nachdem ob der String `sa` kleiner, gleich oder größer `sb` ist.

Die Klasse StringBuilder (S. 233)

Bitte durchlesen.

Ein `StringBuilder`-Objekt enthält eine Reihung von `char`-Werte (eine Reihung vom Typ `char[]`). Die *Länge* einer Reihung kann nicht verändert werden ("Reihungen sind aus Beton"). Deshalb macht der Ausführer dieses Reihung meistens (wenn der Programmierer ihm nicht ausdrücklich etwas anderes befiehlt) etwas länger als nötig und merkt sich in einer `int`-Variablen den *größten belegten Index* (**Beispiel**: Die Reihung hat die Länge 100, ist aber nur bis zum Index 80 "belegt". D. h. die Komponenten mit den Indizes 81 bis 99 "zählen nicht zum Inhalt des `StringBuilder`-Objekts).

Wenn viele `append`-Befehle ausgeführt werden und die Reihung "platzt", wird 1. eine neue, deutlich längere Reihung angelegt, 2. die alte Reihung in die neue kopiert und 3. die alte Reihung "weggeworfen".

Zur Entspannung: **Ein Blatt Papier 50 Mal halbieren und stapeln**

Stellen Sie sich vor: Wir haben ein großes Blatt Papier (z.B. eine Blatt der Zeitung "Die Zeit"). Wir reißen oder schneiden das Papier in zwei Hälften und legen die beiden Hälften übereinander. Dann reißen oder schneiden wir diesen kleinen Stapel ebenso in zwei Hälften und legen sie übereinander, dann reißen oder schneiden wir diesen Stapel ebenso in zwei Hälften etc. etc. Insgesamt wiederholen wir diesen Vorgang 50 Mal. Wie hoch ist der Papierstapel am Ende ungefähr?

Tabelle mit 2-er und 10-er-Potenzen, die sich ungefähr entsprechen:

2^{10}	2^{20}	2^{30}	2^{40}	2^{50}	2^{60}
10^{3}	10^{6}	10^{9}	10^{12}	10^{15}	10^{18}
1 Tausend	1 Million	1 Milliarde	1 Billion	1Billiarde	1 Trillion

2^{50} Schichten Zeitungspapier ist ungefähr gleich
10^{15} Schichten Zeitungspapier (siehe obige Tabelle).

Angenommen, **10 Schichten sind 1 mm dick**. Dann gilt:

1 mm	10	Schichten
1 m	10 000	Schichten
1 km	10 000 000	Schichten (d.h. 10^{7} Schichten)

10^{15} / 10^{7} ist gleich 10^{8} gleich **100 Millionen km**

Übungstermin-14: Aufgabe 6 vorführen

Aufgabe 7 bearbeiten

SU 15 Di 31.05.16, Block 1

Heute schreiben wir den Test-07

Die Klasse ArrayList (S. 237)

`ArrayList` ist eine *Sammlungs-Klasse* (engl. collection class) und ihre Instanzen sind *Sammlungen*. Eine Sammlung (engl. collection) ist ein Objekt, in dem man andere Objekte sammeln (hineintun, wieder herausnehmen, suchen) kann. Es gibt in Java etwa 30 Sammlungs-Klassen.

`ArrayList` ist auch eine *generische* Klasse. Das bedeutet Folgendes:

Eine nicht-generische ("gewöhnliche") Klasse wie `String`, oder `StringBuilder` oder `Integer` etc. definiert nur *einen* Typ (den Typ `String` bzw. `StringBuilder` bzw. `Integer` etc.).

Die Klasse `ArrayList` hat einen Typ-Parameter und definiert (unbegrenzt) *viele Typen*, z.B. die Typen `ArrayList<String>`, `ArrayList<Integer>`, `ArrayList<Arraylist<String>>` etc.

In den spitzen Klammern darf man nur *Referenztypen* angeben (keine primitiven Typen).
Z.B. gibt es *keine* Typen namens `ArrayList<int>` oder `ArrayList<boolean>` etc..
Es *gibt aber* Typen namens `ArrayList<Integer>` und `ArrayList<Boolean>` etc.

In einem Objekt des Typs `ArrayList<String>` kann man nur `String`-Objekte sammeln,
in einem Objekt des Typs `ArrayList<Integer>` kann man nur `Integer`-Objekte sammeln etc.

Objekte von welchem Typ kann man in einer Sammlung vom Typ
`ArrayList<ArrayList<String>>` sammeln? (Objekte vom Typ `ArrayList<String>`).

Eine Sammlung des Typs `ArrayList<String>` hat hat eine gewisse Ähnlichkeit mit einer Reihung des Typs `String[]`, enthält aber viel mehr Methoden (etwa 45) als eine Reihung und "ist aus Gummi", nicht "aus Beton" wie Reihungen.

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

Die Klasse Random (S. 245)

Zwei Weisen, ein Random-Objekt zu erzeugen:

```
Random r03 = new Random(124); // mit Keim 124, reproduzierbarer Zufall
Random r05 = new Random();    // ohne Keim,    nicht-reproduzierbarer Zufall

for (int i=1; i<=100; i++) pln(r03.nextInt()); // immer gleiche Zahlen
for (int i=1; i<=100; i++) pln(r05.nextInt()); // meistens verschiedene Zahlen
```

Nach etwa 250 Billionen Aufrufen von `nextInt()` wiederholen sich die Zahlen.

Die Klassen BigInteger und BigDecimal

`BigInteger`-Objekte: Wenn man mit Ganzzahlen rechnen will, die außerhalb des Bereichs von `long` liegen (bis etwa 40 Milliarden Dezimalziffern). Beispiel: Eine Ganzzahl mit 50 Ziffern:

```
BigInteger b1=new BigInteger("11111222233333444445555566666777778888899999000000");
```

`BigDecimal`: Wenn die Größe oder Genauigkeit von `double`-Zahlen nicht ausreicht. Beispiele finden Sie in Ihrem Ordner **Z:\BspJaSp**.

Die Klasse Formatter und die printf-Methode

Wenn man Daten "anspruchsvoll formatiert" ausgeben will. Beispiel:

```
                                                  // Ausgabe:
System.out.printf("%,6d Euro %02d Cent%n", 12345, 6); // 12.345 Euro 07 Cent
System.out.printf("%,6d Euro %02d Cent%n",   123,45); //    123 Euro 45 Cent
```

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 (wörtlich: für die Vögel), bringt nichts, unnütz, z. B. im Wortspiel *nesting (e.g. of functions) is for the birds*.

Übungstermin-15:

Aufgabe 7 (StringBuilder Florian) bearbeiten.

Datei **pr1_AlleUebProgs.txt**,
Übung Reihungen B (Datei **UebReihugenB.java** und **UebReihungenB_Jut.java**).

Übungstermin-16

Aufgabe 7 (StringBuilder Florian) vorführen.

Übung Reihungen B (siehe vorigen Übungstermin)

SU 16 Do 02.06.16, Block 4

Klassen erweitern (beerben) und Typgrafen (Buch S. 276)

Angenommen, wir haben eine alte und bewährte Klasse `K01.` Jetzt brauchen wir eine Klasse, die sich nur ein bisschen von `K01` unterscheidet ("wir brauchen eine Erweiterung von `K01`").
Wie man eine solche Erweiterung *nicht* erstellen sollte:

`K01` verändern

Eine Kopie `K02` von `K01` erstellen und verändern.

Wie man die neue Klasse erstellen sollte:

Wir vereinbaren eine Klasse `K02` die die Klasse `K01` erweitert (man sagt auch: beerbt), etwa so:

```
class K02 extends K01 {
    ...
```

Jetzt ist die Klasse `K02` eine Erweiterung der Klasse `K01`.

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

Üben Sie, beim Lesen einer Klassenvereinbarung den Modulaspekt und den Bauplanaspekt ("reflexartig") auseinander zu halten.

Modul-Aspekt (der Modul `Person01`): S. 278, Bild 12.1

Bauplan-Aspekt (ein Objekt der Klasse `Person01`): S. 279, Bild 12.2

Wie könnte man die Methode `anzahlPersonen01` z.B. aufrufen?

```
pln(Person01.anzahlPersonen());
int anz = Person01.anzahlPersonen());
```

Wie könnte man die Methode `druckeName` z.B. aufrufen?

```
personA.druckeName();
```

Angenommen, einige Personen haben nicht nur einen Vor- und einen Nach-Namen, sondern auch noch eine *Personal-Nummer*. Wie können wir eine Klasse erstellen, deren Objekte solche Personen beschreiben? **S. 280, Bsipiel-02**.

Zeile 32: Die Klasse `Person02` ist eine *Erweiterung* der `Person01`.
oder: Die Klasse `Person02` *beerbt* die Klasse `Person01`.
oder: Die Klasse `Person01` *vererbt* alle ihre Elemente an die Klasse `Person02`.

Achtung: Dadurch, dass sie ihre Elemente *vererbt*, ändert sich die Klasse `Person01` *nicht* (sie "verliert dadurch keine Elemente"). Wenn z.B. eine Großmutter einem Enkel ihre Silberbesteck *vererbt* ist das anders. Daraus folgt: Eine Großmutter ist keine Java-Klasse :-).

Modul-Aspekt (der Modul `Person02`): S. 281, Bild 12.3

Bauplan-Aspekt (ein Objekt der Klasse `Person02`): S. 281, Bild 12.4

Die beiden Aspekte (der Klassen `Person01` und `Person02`) wurden bisher absichtlich ein bisschen vereinfacht dargestellt. Genauer sind die Darstellungen auf **S. 282, Bild 12.5**: und **S. 283, Bilds 12.6**.
Die Klasse `Person01` beerbt die Klasse `Object`, und deshalb
enthält der Modul `Person01` den Modul `Object` und
jedes `Person01`-Objekt enthält ein `Object`-Objekt.

Allgemeine Erbregeln (S. 283) 1 bis 6

Buch S. 286, Ein größeres Beispiel für Beerbung (Anfang)

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	Plattformunabhängigkeit	
1995	Ada	Klassen	

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

SU 17 Di 07.06.16, Block 1

Heute schreiben wir den Test-08

Klassen beerben (oder: erweitern), Fortsetzung

Buch S. 288, die Klasse `E01Rechteck`

Wie viele Attribute *erbt* die Klasse `E01Rechteck` von ihrer direkten Oberklasse? (`2: x, y`)

Wie viele Attribute werden in jedes Objekt der Klasse `E01Rechteck` *eingebaut*?
(`4: x, y, seiteX, seiteY`).

Wie viele Methoden werden in jedes `E01Rechteck`-Objekt eingebaut?
(9: `urAbstand, urSpiegeln, text, toString, getSeiteX, getSeiteY, getUmfang, getFläche, to String`).

Die in `E01Rechteck` vereinbarte `toString`-Methode **überschreibt** (engl. **overrides**) die (von `E01Punkt`) geerbte `toString`-Methode. Ein Programmierer, der die Klasse `E01Rechteck` benutzt, kann die überschriebene Methode *nicht aufrufen*.

Konstruktor E01Rechteck: Dieser Konstruktor (eigentlich: Initialisator) soll die 4 Attribute eines neu erzeugten `E01Rechteck`-Objekts initialisieren. Deshalb hat er 4 Parameter.

Kleines Problem: Im Konstruktor ist der Name `seiteX` *zweideutig*. Soll er das **Attribut** `seiteX` (vereinbart in Zeile 34) bezeichnen oder den **Parameter** `seiteX` (vereinbart in Zeile 38)?

Regel: Der Name `seiteX` bezeichnet den **Parameter**.
Der Ausdruck `this.seiteX` bezeichnet das **Attribut**.

Man könnte dieses kleine Problem leicht *vermeiden*, indem man die Parameter "ein bisschen anders benennt". Es ist aber üblich, das Problem *nicht* zu vermeiden sondern so zu lösen, wie in dieser Klasse ("mit und ohne `this`").

Der erste Befehl (in Zeile 39) `super(x, y);` ruft einen Konstruktor der direkten Oberklasse (`E01Punkt`) auf. Das ist möglich, weil dieser Konstruktor **öffentlich** ist, siehe S. 287, Zeile 5) (und obwohl die zu initialisierenden Attribute `x` und `y` **privat** sind).

Hätte der Programmierer den Befehl `super(x, y);` weggelassen, so hätte der Ausführer an seiner Stelle den Befehl `super();` eingefügt (d.h. er hätte einen Standardkonstruktor der direkten Oberklasse aufgerufen). Was wäre dann passiert? Fehlermeldung bei der Übergabe des Programms:

```
D:\BspJaSp\E01Rechteck.java:24:
error: constructor E01Punkt in class E01Punkt cannot be applied to given types;
  public E01Rechteck(double x, double y, double seiteX, double seiteY) {
                                                                        ^
  required: double,double
  found: no arguments
  reason: actual and formal argument lists differ in length
```

Aufgabe-01: Schreiben Sie eine Klasse `E01Quadrat` als Erweiterung (oder: als Unterklasse) der Klasse `E01Rechteck`. Jedes Objekt dieser Klasse soll ein Quadrat darstellen.

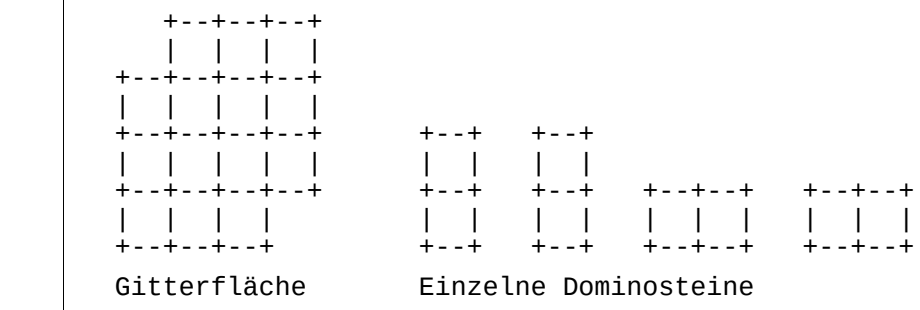
Lösung-01: Siehe S. 290.

Was ist besser?
`E01Quadrat extends E01Rechteck` oder
`E01Rechteck extends E01Quadrat`?

Programmiertechnische Argumente sprechen für die erste Lösung.
Soziale Argumente sprechen für die zweite Lösung.

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?

Übungstermin-17: **Aufgabe-08**: Reihungen und Sammlungen (Veronika01) bearbeiten.

In der Datei **pr1_UebProgs.txt** die Dateien **UebMethodenA.java** und **UebMethodenA_Jut.java** bearbeiten.

Übungstermin-18: **Aufgabe-08**: Reihungen und Sammlungen (Veronika01) vorführen.

SU 18 Do 09.06.16, Block 4

Wiederholung: Drei Arten von Befehlen

Was befiehlt der Programmierer dem Ausführer mit einer *Vereinbarung*? (Etwas zu erzeugen)
Und mit einem *Ausdruck*? (Einen Wert zu berechnen)
Und mit einer *Anweisung*? (Die Inhalte von Wertebehältern zu verändern)

Was bewirkt ein Ausdruck normalerweise *nicht*? (Dass Inhalte von Wertebehältern verändert werden).
Wie nennt man Ausdrücke, die gegen diese Grundregel verstoßen? (Ausdrücke mit Seiteneffekt).

Einfache Beispiele für Ausdrücke mit Seiteneffekt? (`i++, ++i, i--, --i`).

Wie heißen die 4 Teile einer `for`-Schleife? (Initialisierungsteil, Bedingung, Fortschaltungsteil, Rumpf)
Was soll der *Fortschaltungsteil* in vielen `for`-Schleifen bewirken? (Er soll den Wert einer Variablen, die häufig `i` heißt, verändern und sie z.B. um 1 erhöhen oder vermindern).

Was für Befehle eignen sich als *Forstschaltungsteil*?
(Entweder ein *Ausdruck mit Seiteneffekt*, z.B. `i++` oder `i--`, oder eine *Anweisung*, z.B. `i=2*i`).

Ein normaler Ausdruck (einer ohne Seiteneffekt, z.B. `i+3`) eignet sich **nicht** als Fortschaltungsteil!!

Klassen beerben (oder: erweitern), Fortsetzung

Wichtige Regel: Jedes Objekt des Typs `E01Quadrat` ist (nicht nur ein `E01Quadrat`-Objekt, sondern auch) ein `E01Rechteck`-Objekt, ein `E01Punkt`-Objekt und ein `Object`-Objekt.

Diese "Mehrtypigkeit" folgt daraus, dass die betreffenden Klassen
sich *erweitern* (oder: *beerben*) wie folgt:

```
E01Quadrat --> E01Rechteck --> E01Punkt --> Object.
```

Buch S. 286, Bild 12.8: Zu welchen Typen (Plural!)
gehört jedes `E01Elipse`-Objekt? (`E01Elipse, E01Punkt, Objekt`).
Und jedes `E01Kreis`-Objekt? (`E01Kreis, E01Elipse, E01Punkt, Object`).

Wozu Untertypen gut sind (S. 294)

Angenommen, wir Vereinbaren ein Variable des Typs `E01Punkt[]`
(d.h. vom Typ Reihung von `E01Punkt`-Variablen), die auf eine Reihung zeigt:

```
E01Punkt[] tab = new E01Punkt[6];
```

Objekte welcher Typen können wir in der Reihung `tab` aufbewahren?

Die Objekte in der Reihung `tab` müssen "mindestens vom Typ `E01Punkt`" sein, sie dürfen aber auch von irgendeinem *Untertyp* von `E01Punkt` sein.

Von diesen Objekten `tab[i]` dürfen wir aber nur die `E01Punkt`-Eigenschaften ausnutzen:
Erlaubt sind z.B. folgende Methodenaufrufe:
`tab[i].urAbstand(), tab[i].urSpiegeln, ...`
Nicht erlaubt sind z.B. folgende Methodenaufrufe:
`tab[i].getUmfang(), tab[i].getFlaeche(), ...`

Ein (hoffentlich) abschreckendes Beispiel
(S. 296), Beispiel-02: Wie man es nicht machen soll
(S. 298), Beispiel-03: Wie man es machen soll

Kritik an der E01Punkt-Hierarchie

Seite 299, Aufgabe-05: Warum kann man diese Aufgabe nur so hässlich wie im **Beispiel-02** (S. 296) lösen, aber nicht so schön wie im **Beispiel-03** (S. 298)?

Weil die Wurzel unserer Hierarchie (die Klasse `E01Punkt`) *keine* Methode `getFläche()` enthält.

Das ist ein Hinweis darauf, dass die Hierarchie (vor allem die Wurzelklasse) nicht besonders gut entworfen wurde und verbessert werden sollte.

Abstrakte Klassen

Motivation für abstrakte Klassen (S. 335)

Seite 336, Beispiel-01: Die abstrakte Klasse `E02GeoFigur`
Seite 337, Die Klasse `E02Rechteck`

Abstrakte Klassen kann man auch *abstrakt erweitern*.

S. 338, Die Klasse `E02GeoFigurMitFarbe`

Auch mit abstrakten Klassen gilt weiterhin die Regel: Eine Klasse darf (und muss) genau **eine** (konkrete oder abstrakte) Klasse erweitern. In Java ist es (aus guten Gründen) nicht erlaubt, dass eine Klasse **mehrere** Klassen beerbt (keine Mehrfachbeerbung!).

Ein Trostpflaster für die fehlende Mehrfachbeerbung: Schnittstellen (engl. interfaces)

Schon seit Java 1 gibt es ein Konstrukt namens `interface` (Schnittstelle). Mit Java 8 ist dieses Konstrukt erheblich erweitert worden. Wir befassen uns erst mal nur mit den "Kern-Eigenschaften" dieses Konstrukts, die es schon immer gab und die besonders wichtig sind.

Eine Schnittstelle ist im Wesentlichen *eine Menge von abstrakten Methoden*.
Eine Schnittstelle ist (ähnlich wie wie eine Klasse) auch ein Typ.

Buch S. 341, Beispiel-01: Die Schnittstelle `Vergroesserbar` enthält 2 abstrakte Methoden.

Eine Schnittstelle kann von einer Klasse *implementiert* werden.

S. 342, Beispiel-02: Die Klasse `GanzZahl01` implementiert die Schnittstelle `Vergroesserbar`.
Das bedeutet: Die Klasse `GanzZahl01` muss alle (abstrakten) Methoden der Schnittstelle mit konkreten Methoden ("Methoden die einen Rumpf haben") überschreiben.

Die Objekte der Klasse `GanzZahl01` gehören zu 3 Typen:
`GanzZahl01`, `Object` und `Vergroesserbar`.

Wichtigste Schnittstellen-Regel:
Eine Klasse darf *beliebig viele* (0 oder mehr) Schnittstellen implementieren.

Hinweis: Die Schnittstelle `Verkleinerbar` (im Buch nicht wiedergegeben, die Datei `Verkleinerbar.java` steht im Ordner `BspJaSp`) enthält nur 1 abstrakte Methode und die heißt `halbiere`.

S. 343, Beispiel-03: Die Klasse `GanzZahl02` implementiert zwei Schnittstellen.

Zu welchen Typen gehören die Objekte der Klasse `GanzZahl02`?
(`GanzZahl02, Object, Vergroesserbar, Verkleinerbar`).

Man beachte: In den Klassen `GanzZahl01` und `GanzZahl02` werden gleiche Objekt-Methoden vereinbart (`verdopple, verdreifache, halbiere, toString`). Warum gehören `GanzZahl02`-Objekte u.a. zum Typ `Verkleinerbar`, `GanzZahl01`-Objekte aber nicht?
(Weil die Klasse `GanzZahl02`-Objekte mit `...implements ... Verkleinerbar...` beginnt, `GanzZahl01` aber nicht).

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: **"The Annotated Alic"** von Martin Gardner, 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 Antwort **42** und erklären dann die dazugehörige Frage.

SU 19 Di 14.06.16, Block 1

Heute schreiben wir den Test-09

Schnittstellen, Wiederholung und Fortsetzung

Was enthält eine *Schnittstelle* (engl. an interfac) hauptsächlich? (Abstrakte Methoden).
Was kann eine *Klasse* mit einer *Schnittstelle* machen (oder "zu tun haben")? (Die Klasse kann die Schnittstelle *implementieren*)
Was bedeutet das konkret? Was muss der Programmierer machen, damit eine konkrete Klasse K eine Schnittstelle S *implementiert*? (Er muss in K alle in S enthaltenen *abstrakten Methoden* durch konkrete Methode *überschreiben*).
Wie viele *Klassen* darf/muss eine Klasse *beerben* (oder: erweitern)? (Genau **eine**).
Wie viele *Schnittstellen* darf eine Klasse *implementieren*? (Beliebig viele: 0, 1, 2, 3, ...).

Erb-Regel für Schnittstellen:
Eine *Schnittstelle* darf beliebig viele (0, 1, 2, 3, ...) andere *Schnittstellen* beerben (oder. erweitern).

Buch S. 343 unten, Beispiel-04: Die Schnittstelle `Veraenderbar` beerbt (oder: erweitert) die beiden Schnittstellen `Vergroesserbar` und `Verkleinerbar`.

Zur Erinnerung: Die Schnittstelle `Vergroesserbar` enthält 2 abstrakte Methoden (S. 341), `Verkleinerbar` enthält 1 abstrakte Methode (siehe Datei `Verkleinerbar.java`).

Wie viele abstrakte Methoden muss man in einer konkreten Klasse überschreiben, um die Schnittstelle `Veraenderbar` zu implementieren?
(2+1+1 gleich 4 Stück: `verdopple, verdreifache, halbiere, aendere`).

Schnittstellen als Parametertypen (S. 350)

Die Methode `int anzahlZiff(String s) { ... }`
kann man nur auf `String`-Objekte anwenden.
Die Methode `int anzahlZiff(StringBuilder s) { ... }`
kann man nur auf `StringBuilder`-Objekte anwenden.

Die Schnittstelle `CharSequence` wird (unter anderem) von den Klassen `StringBuilder` und `String` implementiert und enthält (unter anderem) die Methoden `length` und `charAt`.

Die Methode `int anzahlZiff(CharSequence s) { ...}`
kann man (unter anderem) auf `String` und auf `StringBuilder`-Objekte anwenden.

Anstelle eines Parameters eines *Klassen-Typs* (z.B. `String` oder `StringBuilder`) ist ein Parameter von einem *Schnittstellen-Typ* (z.B. `CharSequence`) günstiger, weil man die Methode dann auf Objekte unterschiedlicher Klassen anwenden kann.

Der Typ und der Zieltyp einer Variablen

S. 304, Beispiel-01

Zeile 1 bis 4: Einfache Beispiele mit *Zieltyp gleich Typ*
Zeile 5: Zieltyp `void`
Zeile 6 bis 11: *Zieltyp ungleich Typ*

Pakete (bei Java, nicht bei der Post)

Ein Ziel der Java-Entwickler: Es soll möglich sein, Java-Klassen von verschiedenen Firmen zu kaufen (oder umsonst zu übernehmen) und daraus ein Java-Programm zusammenzustellen.

Problem: Klassen-Namen sind dann möglicherweise nicht eindeutig. Z.B. können 2 Firmen je eine Klasse namens `Test` (oder `String` oder `Integer` etc.) entwickelt haben, die nichts miteinander zu tun haben, und wir haben ein Problem, wenn wir beide Klassen in einem Programm verwenden wollen.

Lösung des Problems:
1. Jede Klasse gehört zu einem Paket.
2. Ein Paket ist ein Behälter für Klassen, Schnittstellen und Pakete (ähnlich wie ein Ordner, der "normale Dateien" und Ordner enthalten kann).
3. Wenn eine Institution (Firma, Hochschule, ...) Java-Klassen weltweit weitergeben will, dann sollte die Institution aus ihrer (eindeutigen) *Internet-Adresse* (nach einfachen Regeln) einen *Paketnamen* P erzeugen und alle ihre Klassen sollten zum Paket P (oder zu Unterpaketen von P) gehören.

Beispiel-01: Alle an der Beuth Hochschule entwickelten Java-Klassen (die weltweit vertrieben werden sollen), sollten zu einem Paket namens **de.beuth_hochschule** gehören (weil die Beuth Hochschule die Internet-Adresse `www.beuth_hochschule.de` hat).

Beispiel-02: Die Beuth Hochschule könnte die Regel aufstellen, dass alle von **Karl Schulz** am Fachbereich **VI** entwickelten Java-Klassen (die weltweit vertrieben werden sollen) zu einem Paket namens **de.beuth_hochschule.fb6.Schulz.Karl** gehören sollen.

Zuordnung einer Klasse (oder Schnittstelle) zu einem Paket

Mit einem `package`-Befehl, z.B.

```
package de.beuth_hochschule.fb6.Schulz.Karl;
```

Dieser Befehl ist nur als erster Befehl in einer `.java`-Datei erlaubt. *Davor* dürfen nur Kommentare stehen.

Top-Pakete und Pakete, die in anderen Paketen enthalten sind.
In der Java-Standard-Bibliothek gibt es **3 Top-Pakete**: (`java`, `javax`, `org`) und
etwas mehr als 200 Pakete.

Wenn man keinen `package`-Befehl angibt, gehören die betreffenden Klassen zum **namenlosen Paket**.

Innerhalb eines Pakets kann man eine Klasse als **öffentlich** (`public`) oder als **paketweit erreichbar** (ohne Modifizierer) vereinbaren (die Modifizierer `protected` und `private` sind nicht erlaubt):

```
public classe Karola { ... } // öffentlich
class Karl { ... }           // paketweit erreichbar
```

Klassen im namenlosen Paket sind grundsätzlich nur *paketweit erreichbar* (von Klassen in anderen Paketen aus kann man grundsätzlich *nicht* auf Klassen im namenlosen Paket zugreifen).

Im namenlosen Paket kann man auf (öffentliche) Klassen in Paketen-mit-Namen zugreifen.

Alle Java-Klassen bilden einen **Baum** (mit `Object` als Wurzel).
Alle Pakete bilden einen **Wald** (mehrere Bäume, mit den Top-Paketen als Wurzeln).

Der **Klassenbaum** und der **Paketwald** sind völlig *unabhängig voneinander*! (S. 432).

Regel: Das namenlose Paket ist sehr nützlich, wenn man Java lernt und einfache Beispiel-Programme schreiben will. Bei industriellen Anwendungen müssen alle Klassen und Schnittstellen zu Paketen mit (eindeutigen) Namen gehören.

Java-Pakete und der TextPad

Wenn man den TextPad richtig eingestellt hat, nimmt er einem alle mit Paketen zusammenhängenden Probleme ab und löst sie unauffällig.

Wenn man im TextPad an der richtigen Stelle (**Konfiguration**, **Einstellungen**, **+ Extras**, **Java kompilieren**, **Parameter**) die Option `-d \Z:Klassen` angegeben hat und eine Datei namens `Otto.java` compiliert, die die Vereinbarung einer Klasse namens `Otto` enthält und mit dem folgenden `package`-Befehl beginnt:

```
package pak01.pak02;
```

dann wird die vom Compiler erzeugt Datei `otto.class` im Ordner `Z:\Klassen\pak01\pak02` abgelegt. Falls die Ordner `pak01` und `pak02` noch nicht existieren, werden sie automatisch (vom Java-Compiler) erzeugt.

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 `beuth-hochschule.de`	K-vorn
Eine arabische Zahl im Deutschen 12345	G-vorn (Leserichtung: von links nach rechts)
Eine arabische Zahl im Arabischen 12345	K-vorn (Leserichtung: von rechts nach links)

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)

Übungstermin-19: **Aufgabe-09**: Reihungen und Sammlungen (Veronika02) bearbeiten.

In der Datei **pr1_UebProgs.txt** die Dateien **UebMethodenB.java** und **UebMethodenB_Jut.java** bearbeiten.

Übungstermin-20: **Aufgabe-09**: Reihungen und Sammlungen (Veronika02) vorführen.

SU 20 Do 16.06.16. Block 4

"Es läuft aber!": Eine kurze Predigt

Beim Schreiben eines Programms sollte man nicht nur an den *Ausführer* denken, sondern auch an den *Warter* und den *Wiederverwender.* Das Argument "Es läuft aber!" trifft auch auf viele sehr schlechte Programme zu und klingt so, als ob hauptsächlich der Ausführer (meistens eine *Maschine*) wichtig sei. Aus ökonomischer Sicht sind der Warter und der Wiederverwender (meistens *Menschen*) in aller Regel wichtiger und teurer (eine IT-Fachkraft kostet pro Monat in aller Regel deutlich mehr als ein guter PC).

Deshalb sollte man (in Programmen) Formulierungen vermeiden, die dem Ausführer zwar das Richtige befehlen, für andere Menschen aber unnötig schwer zu lesen sind. Beispiel (sei `c` eine `char`-Variable):

Statt `if (96 < c && c < 123)` sollte man fast immer
besser `if ('a' <= c && c <= 'z')` schreiben.

Die `char`-Literale wie `'a', 'z', '0'` etc. wurden extra erfunden, um die zweite (menschenfreundliche) Formulierung zu ermöglichen. Ein Programmierer sollte das wissen und ausnutzen.

Vergleich: Manche Programme ähneln einem stark verbeulten Auto, das vom Händler mit dem Argument angepriesen wird: "Es läuft aber!".

Bereichsprüfungen

Häufig muss man prüfen, ob eine Zahl `n` in einem bestimmten Bereich (z.B. zwischen 100 und 200) liegt oder nicht. Das kann man man mit zwei Vergleichen und einem und (`&&`) bzw. einem oder (`||`) erledigen. Besonders empfehlenswert sind die folgenden beiden Formulierungen (für eine `int`-Variable `n`):

```
if (100 <= n && n <= 200) ... // Wenn n       im Intervall [100, 200] liegt ...
if (n < 100  || 200 < n)  ... // Wenn n nicht im Intervall [100, 200] liegt ...
```

Ein Beispiel mit einer `char`-Variablen c:

```
if ('a' <= c && c <= 'z') ... // Wenn c       im Intervall ['a', 'z'] liegt ...
if (c < 'a'  || 'z' < c)  ... // Wenn c nicht im Intervall ['a', 'z'] liegt ...
```

Schlunzige "Doppelvergleiche" sind in Java (und anderen Programmiersprachen) verboten

In Mathe-Texten findet man manchmal einen Ausdruck wie z.B. 0 < n < 10
Der Teilausdruck `0 < n` ist offenbar vom Typ `boolean`. Soll geprüft werden, ob sein Wert kleiner als 10 ist? In Java muss man schreiben: `0 < n && n < 10`.

Ausnahmen sind keine Ausnahmen sondern eher die Regel

"Befehle für die normalen Fälle" (BNs) und "Befehle für Ausnahmefälle" (BAs)

Buch S. 362, Ausnahmefall 1, 2, 3, 4, 5.

Altmodisches Zebra-Muster für die Behandlung von Ausnahmefällen:

```
methodeA(status, ..., ...);  BN 1
if (status==-1) {
   ...                       BA 11
} else if (status==-2) {
   ...                       BA 12
} else if (status==-3) {
   ...                       etc.
}
methodeB(status, ..., ...);  BN 2
if (status==-1) {
   ...                       BA 21
} else if (status==-2) {
   ...                       BA 22
} else if (status==-3) {
   ...                       etc.
}
...
```

Nachteil: So ein Zebra-Muster von BNs und BAs ist schwer lesbar.

Neuere Technik: Mit Hilfe spezieller Konstrukte trennt man die BNs von den BAs. Dadurch werden beide Arten von Befehlen leichter lesbar.

Man behandelt Ausnahmen nicht mehr an der Stelle, wo sie entdeckt werden, sondern anderswo (z.B. viele oder alle Ausnahmen an einer bestimmten Stelle des Programms).

Wenn man in BNs eine Ausnahme entdeckt, packt man Informationen in ein spezielles *Ausnahme-Objekt* und *wirft* dieses Objekt. Das Objekt fliegt dann (nach genauen Regeln) an andere Stellen des Programms und kann dort *gefangen* und *behandelt* werden.

Def. *Ausnahmeklasse* und *Ausnahmeobjekt* (**S. 363**)

Ausnahmen werfen

Viele Java-Befehle werfen in bestimmten Ausnahmefällen ein bestimmtes Ausnahmeobjekt.

Beispiel-01 (Ganzzahldivision durch 0)
Beispiel-02 (Integer.parseInt(s)) (**S. 364**)

Mit dem throw-Befehl kann der Programmierer Ausnahmen werfen

Beispiel-03 (throw new ArithmeticException ...)

Wenn eine Ausnahme *geworfen*, aber nirgends *gefangen* wird, wird das ganze Programm mit einer Fehlermeldung abgebrochen

Beispiel-04: (Exception in thread "main" ...) (**S. 364**)

In der zweiten Hälfte des Blocks soll die Übung
Übung 9: Methoden B (Dateie **UebMethodenB.java** in Datei **pr1_AlleUebProgs.txt**)
bearbeitet werden

Zur Entspannung: **Christian Morgenstern (1871-1914): Anto-logie**

"Im Anfang lebte, wie bekannt, / als größter Säuger der Gig-ant. ..."

SU 21 Di 21.06.16

Heute schreiben wir den Test-10

Ausnahmen fangen (S. 365)

Beispiel-01: Eine einfache, aber gefährliche Funktion

Je nachdem, mit welchen Argumenten man diese Funktion aufruft, wirft sie eine `NumberFormatException` (in Zeile 2 oder 3) oder eine `ArithmeticException` (in Zeile 4).

Beispiel-02: Die gefährliche Funktion aufrufen (und Ausnahmen auffangen und behandeln) (**S. 366**)

Wie oft wird der Rumpf der `for`-Schleife (d.h. der Befehl in Zeile 14 bis 24) ausgeführt? (5 mal)

D.h. der `try-catch`-Befehl (Zeile 14 bis 24) wird 5 mal ausgeführt.

Mit welchen Argumenten wird die Funktion `dividiere` (in Zeile 16) aufgerufen?
(mit `("9", "4"), ("a", "6"), ("3", "0"), ("1", "b"), ("8", "2")`)

Der Aufruf `dividiere("9", "4")` wirft keine Ausnahme. Der `try`-Block wird vollständig ausgeführt und damit ist auch der `try-catch`-Befehl vollständig ausgeführt.

Der Aufruf `dividiere("a", "6")` wirft eine Ausnahme des Typs `NumberFormatException`.
Der `try`-Block wird abgebrochen, der *erste* `catch`-Block wird ausgeführt.
Damit ist der `try-catch`-Block fertig ausgeführt.

Der Aufruf `dividiere("3", "0")` wirft eine Ausnahme des Typs `ArithmeticException`.
Der `try`-Block wird abgebrochen, der *zweite* `catch`-Block wird ausgeführt.
Damit ist der `try-catch`-Block fertig ausgeführt.

Ein `try-catch`-Befehl besteht normalerweise aus
einem **try-Bloc**k gefolgt von einem oder mehreren **catch-Blöcken**.

Der `try-catch`-Befehl in Zeile 14 bis 23 enthält 2 `catch`-Blöcke.

Ein `catch`-Block fängt und behandelt alle Ausnahme-Objekte, die
zu einem bestimmten Typ (oder zu einem Untertyp dieses Typs) gehören.

Beispiel-03 (**S. 368**)

Im **Beispiel-02** wurden alle Ausnahmen sofort gefangen und behandelt.
Was passiert, wenn eine Ausnahme geworfen wird und **nicht** sofort gefangen wird?

Dann fliegt die Ausnahme nach bestimmten Regeln an bestimmte Stellen des Programms. Um diese Regeln zu verstehen muss man sich "die abstrakte Struktur einer Programmausführung" klar machen. Diese Struktur ist bei vielen Programmiersprachen gleich oder sehr ähnlich.

S. 369, Bild 15.1: Methoden, die weitere Methoden aufrufen

S. 370, Beispiel-01: Ein kleines Java-Programm `Stapel01`, welches aus 3 Methoden besteht.

Die Rolle des Stapels bei der Ausführung eines Programms: Der Stapel enthält Informationen zu jeder Methode, deren Ausführung begonnen aber noch nicht abgeschlossen wurde.

Anmerkung: 1957 haben Friedrich Bauer und Klaus Samelson das Kellerprinzip zum Patent angemeldet. Später verbreitete sich die Bezeichnung "Stapel" (engl. stack) anstelle von "Keller".

Allgemeine Wirkung einer Ausnahme

Angenommen, der Stapel des Ausführer befindet sich im Zustand **z5** (d.h. der Ausführer führt gerade die Methode `metB` aus, Zeile 20) und dabei tritt eine Ausnahme `a` auf. Was macht der Ausführer jetzt?

Der Ausführer fragt sich:

Trat die Ausnahme `a` innerhalb eines `try`-Blocks auf, dem ein (zum Fangen von `a`) geeigneter `catch`-Block folgt?

Diese Frage unbedingt **auswendig lernen**!

S. 372, Bild 15.2 Was der Ausführer machen muss, wenn eine Ausnahme auftritt

Den **Ja-Zweig** dieses Diagramms haben wir oben im Beispiel-02 auf S. 366 schon besprochen.
Wir können uns hier also auf den **Nein-Zweig** des Diagramms konzentrieren.

Wenn der Stapel sich im Zustand **z5** befindet: Von wo aus (Zeilen-Nr.?) wurde die Methode `metB` dann aufgerufen (von **Zeile 13** aus).

Der Ausführer würde also zur **Zeile 14** (unmittelbar hinter die Stelle, an der die Methode `metB` aufgerufen wurde) zurückkehren und dort die Ausnahme erneut werfen.

Das Programm `Ausnahmen03.java` (im Ordner **Z:\BspJaSp**) ansehen:

```
 1 // Datei Ausnahmen03.java
 2 /* ------------------------------------------------------------------------
 3 Was kann man mit einem Ausnahmeobjekt alles machen, nachdem man es
 4 gefangen hat? Dieses Programm demonstriert die Methoden
 5 getMessage, printStackTrace und getStackTrace, die in jedem
 6 Ausnahme-Objekt (Throwable-Objekt) enthalten sind.
 7
 8 Throwable ist per Definition (die direkte oder eine indirekte) Oberklasse
 9 aller Ausnahmeklassen, insbesondere also auch der Ausnahmeklasse
10 ArithmeticException.
11
12 Siehe auch Ausnahmen04.
13 ------------------------------------------------------------------------ */
14 class Ausnahmen03 {
15    // ---------------------------------------------------------------------
16    static public void main(String[] sonja) {
17       // Ruft die gefaehrliche Methode metA auf. Faengt und behandelt
18       // die dadurch geworfene Ausnahme.
19
20       pln("Ausnahmen03: Jetzt geht es los!\n");
21
22       try {
23          metA(12, 3);
24          metA(17, 0);
25          metA(14, 2);
26       } catch (ArithmeticException ex) {
27          bearbeiteAusnahmeobjekt(ex);
28       } // try-catch-Befehl
29    } // main
30    // ---------------------------------------------------------------------
31    static void metA(int a, int b) {
32       // Ruft die gefaehrliche Methode metB auf:
33       metB(a, b);
34    } // metA
35    // ---------------------------------------------------------------------
36    static void metB(int a, int b) {
37       // Ruft die gefaehrliche Methode metC auf:
38       metC(a, b);
39    } // metB
40    // ---------------------------------------------------------------------
41    static void metC(int a, int b) {
42       // Gibt auf jeden Fall eine kleine Meldung ("A ... ist") aus.
43       // Wirft ein Ausnahme, falls b gleich 0 ist.
```

```
 44       p("A " + a + " / " + b + " ist ");
 45       int quot = a / b;        // ArithmeticException falls b gleich 0 ist!
 46       pln("gleich " + quot);
 47    } // metB
 48    // ------------------------------------------------------------------
 49    static void bearbeiteAusnahmeobjekt(Throwable ausOb) {
 50       // Ruft einig Objektmethoden des Ausnahmeobjekts ausOb auf:
 51       pln();
 52       pln("-----------------------------------------------------------------");
 53       pln("B ausOb:\n");
 54       pln("  " + ausOb);
 55       pln("-----------------------------------------------------------------");
 56       pln("C ausOb.getMessage():\n" );
 57       pln("  " + ausOb.getMessage());
 58       pln("-----------------------------------------------------------------");
 59       pln("D ausOb.printStackTrace():\n");
 60       p  ("  "); ausOb.printStackTrace();
 61       pln("-----------------------------------------------------------------");
 62       pln("E ster = ausOb.getStackTrace():\n");
 63       pln("  Die Reihung ster (stack trace element Reihung) sieht so aus:");
 64       StackTraceElement[] ster = ausOb.getStackTrace();
 65       for (StackTraceElement ste : ster) {
 66          pln ("\t   " + ste);
 67       }
 68       pln("-----------------------------------------------------------------");
 69
 70    } // bearbeiteAusnahmeobjekt
 71    // ------------------------------------------------------------------
 72    // Mehrere Methoden mit kurzen Namen:
 73    static void pln(Object ob) {System.out.println(ob);}
 74    static void p  (Object ob) {System.out.print  (ob);}
 75    static void pln()          {System.out.println();  }
 76    // ------------------------------------------------------------------
 77 } // class Ausnahmen03
 78 /* ------------------------------------------------------------------
 79 Ausgabe des Programms Ausnahmen03:
 80
 81 Ausnahmen03: Jetzt geht es los!
 82
 83 A 12 / 3 ist gleich 4
 84 A 17 / 0 ist
 85 -----------------------------------------------------------------
 86 B ausOb:
 87
 88   java.lang.ArithmeticException: / by zero
 89 -----------------------------------------------------------------
 90 C ausOb.getMessage():
 91
 92   / by zero
 93 -----------------------------------------------------------------
 94 D ausOb.printStackTrace():
 95
 96   java.lang.ArithmeticException: / by zero
 97    at Ausnahmen03.metC(Ausnahmen03.java:45)
 98    at Ausnahmen03.metB(Ausnahmen03.java:38)
 99    at Ausnahmen03.metA(Ausnahmen03.java:33)
100    at Ausnahmen03.main(Ausnahmen03.java:24)
101 -----------------------------------------------------------------
102 E ster = ausOb.getStackTrace():
103
104   Die Reihung ster (stack trace element Reihung) sieht so aus:
105       Ausnahmen03.metC(Ausnahmen03.java:45)
106       Ausnahmen03.metB(Ausnahmen03.java:38)
107       Ausnahmen03.metA(Ausnahmen03.java:33)
108       Ausnahmen03.main(Ausnahmen03.java:24)
109 -----------------------------------------------------------------
110 ------------------------------------------------------------------- */
```

Wenn noch Zeit ist:

Der `try-catch-finally`-Befehl (**S. 373**)

Zur Entspannung: **Ist der folgende Satz wahr oder nicht?**

Dieser Satz enthält trei Fähler!. Stimmt das oder nicht? Offenbar enthält der Satz zwei syntaktische Fehler (*trei* statt *drei* und *Fähler* statt *Fehler*) und einen semantischen Fehler (drei statt zwei). Damit ist er also wahr. Aber wenn er wahr ist, enthält er keinen semantischen Fehler und ist somit falsch. Aber wenn er falsch ist, enthält er einen semantischen Fehler und ist somit wahr. Aber wenn er ... .

Übungstermin-21: Aufgabe-10 (**Klasse Punkt3D**) bearbeiten

Übungstermin-22: Aufgabe-10 vorführen.

SU 22 Do 23.06.16

Geprüfte und ungeprüfte Ausnahmen (checked and unchecked exceptions)

Was fragt sich der Ausführer, wenn er eine Ausnahme wirft (oder: wenn eine Ausnahme auftritt)?
("Trat die Ausnahme in einem `try`-Block auf, dem ...").

Was macht der Ausführer, wenn die Antwort auf diese Frage NEIN ist?
(Er bricht die Ausführung der Methode, die er gerade ausführt, ab und kehrt hinter die Stelle zurück, an der sie aufgerufen wurde)

Jeder, der eine Methode `m` aufruft, möchte und sollte wissen, was für Ausnahme-Objekte möglicherweise aus dieser Methode zu seinem Aufruf geflogen kommen können. Das sind die Ausnahmen, die in `m` auftreten können, dort aber nicht gefangen und behandelt werden.

ADOC-Regel, Version 1: Zur Dokumentation jeder Methode `m` gehört eine Beschreibung aller Ausnahmen, die aus ihr zu ihren Aufrufen geflogen kommen können.

Die Entwickler von Java wussten, dass viele Programmierer nicht gern Dokumentationen schreiben und beschlossen, die Programmierer zur Dokumentation von Ausnahmen ("Welche Ausnahmen werden von dieser Methode eventuell geworfen?") zu zwingen. Der Java-Ausführer sollte schon bei der Übergabe eines Programms prüfen, ob am Anfang jeder Methode eine vollständige Liste aller möglichen Ausnahmen steht.

Nun gibt es aber einige Ausnahmen, die aus sehr vielen oder fast allen Methoden geflogen kommen können, z.B. Ausnahmen des Typs `OutOfMemoryError` oder `NullPointerException`. Es wäre sehr unpraktisch, diese Ausnahmen vor *jeder* Methode in einer Liste erwähnen zu müssen. Deshalb führten die Entwickler von Java folgende Regel ein: Es gibt 2 Arten von Ausnahmen:
Geprüfte und *ungeprüfte* (checked and unchecked exceptions). Die geprüften *muss* man am Anfang jeder Methode dokumentieren. Die ungeprüften *darf* man (muss man aber nicht) dokumentieren.

Buch S. 378, Beispiel-01: Eine Methode mit `throws`-Klausel

ADOC-Regel, Version 2: Zur Dokumentation jeder Methode `m` gehört eine Beschreibung aller **geprüften** Ausnahmen, die aus ihr zu ihren Aufrufen geflogen kommen können.

Achtung: Mit der Anweisung `throw` (ohne `s`!) wirft man ein Ausnahme-Objekt. Eine `throws`-Klausel (`throws` mit `s`!) ist kein Befehl an den Ausführer, sondern ein Teil der Dokumentation. Das Besondere an dieser Dokumentation: Sie wird vom Ausführer überprüft und "wenn sie nicht stimmt", lehnt er das Programm ab.

Eine wichtige Fehlermeldung des Compilers

```
D:\Quellen\Ausnahmen08.java:59: unreported exception
java.io.IOException; must be caught or declared to be thrown
      met30();
      ^
```

In Zeile 59 steht ein Methodenaufruf `met30()`. Die Methode `met30` wirft manchmal eine Ausnahme des Typs `IOException`. Dies ist eine *geprüfte* Ausnahme. Den Fehler in Zeile 59 kann man auf zwei Weisen beseitigen:

1. Man erwähnt den Ausnahmetyp `IOException` in der `throws`-Klausel der aktuellen Methode.
2. Man platziert den Methodenaufruf `met30()` in einen `try`-Block und *fängt* Ausnahmen des Typs `IOException` in einem nachfolgenden `catch`-Block.

Lesen und beachten Sie nicht nur den *ersten* Teil der Fehlermeldung ("must be caught", muss gefangen werden), sondern auch den *zweiten* Teil ("or declared to be thrown", muss in einer `throws`-Klausel erwähnt werden).

Übersicht über ein paar wichtige Ausnahmeklassen (S. 382)

Geprüft sind: Die Typen `Throwable`, `Exception` und fast alle Untertypen von `Exception` (Ausnahme: `RuntimeException`)

Ungeprüft sind: Der Typ `Error` und alle seine Untertypen,
der Typ `RuntimeException` und alle seine Untertypen.

Die Methoden format und printf

Mit der Methode `format` kann man aus irgendwelchen Daten einen formatierten String erzeugen.
Die Methode `printf` bewirkt das Gleiche, gibt den String aber auch noch aus (zur Standardausgabe).

Diese beiden Methoden werden in den Klassen `Console`, `PrintStream`, `PrintWriter` vereinbart, `format` zusätzlich auch in den Klassen `String` (!) und `java.util.Formatter`.

Besonders häufig benutzt wird die Methode `System.out.printf`
(`out` ist ein öffentliches Klassenattribut vom Typ `PrintStream` in der Klasse `System`).

"Daten formatieren" kann z.B. bedeuten:

Werte verschiedener Typen in Strings umwandeln
Mehrere Strings konkatenieren ("zu *einem* String zusammenpappen")
Strings auf eine bestimmte "Mindestbreite" verlängern
Zahlen in einem bestimmten Zahlensystem (2-er-, 8-er- 10-er, 16-er-System) darstellen
Bruchzahlen mit einer bestimmten Anzahl von Nachpunktstellen darstellen
Zahlen durch das "Gruppieren" der Ziffern leichter lesbar machen etc. etc.

Die Methode `printf` kann man wahlweise mit *einem* oder mit *mehreren* Parametern aufrufen. Der erste Parameter ist immer vom Typ `String` und wird als **Formatstring** bezeichnet.

Der Formatstring kann **Umwandlungsbefehle** enthalten (die mit einem Prozentzeichen % beginnen). Normalerweise muss man für jeden Umwandlungsbefehl im Formatstring einen weiteren zusätzlichen Parameter angeben.

S. 257, Beispiel-01: Ein Formatstring mit `%%` und `%n` darin

S. 258, Beispiel-02: Ganzzahlen formatieren

S. 259, Beispiel-03: Ein Formatstring mit mehreren (2) Umwandlungsbefehlen
... etc.

Übersicht über die Umwandlungsbefehle (S. 266)

Struktur eines Formatstrings:
`%[Index][Schalter][Breite][.Genauigkeit]UBuchstabe`
Structure of a format string:
`%[index][flags][width][.precision]cletter`
("cletter" like "conversion letter")

Alle Angaben in eckigen Klammern sind optional
(nur das Prozentzeichen `%` und der `UBuchstabe` sind *nicht* optional).

Der UBuchstabe legt fest: "Welche Typen von Daten sind erlaubt?" und "Was wird gemacht?"

Große UBuchstaben (z.B. **B, H, S** etc.) haben fast die gleiche Wirkung wie kleine (z.B. **b, h. s**).

Es gibt 7 Schalter: `-#+ 0,(`
Minus, Nummernzeichen, Plus, Blank, Null, Komma, Runde Klammer auf

Man darf mehrere angeben, einige Kombinationen sind nicht erlaubt (z.B. Plus und Blank),
Reihenfolge der Schalter ist beliebig.

Auf was für Daten darf man einen Umwandlungsbefehl `%x` anwenden?
(Auf `byte, Byte, short, Short, int, Integer, long, Long, BigInteger`)

Auf was für Daten darf man einen Umwandlungsbefehl `%f` anwenden?
(Auf `float, Float, double, Double, BigDecimal`)

Was bewirkt der Schalter + ? (Vorzeichen `+` bzw. `-`)
Was bewirkt der Schalter - ? (linksbündig)

Zur Entspannung: **MOOCs: Kostenlose online-Kurse von Spitzenunis**

MOOC: massive open online courses.
Seit Anfang 2012 (oder etwas früher) haben verschiedene Unis, vor allem amerikanische Spitzenunis wie Stanford, Harvard, MIT, Yale, .., damit angefangen, zahlreiche (tausende!) Vorlesungen aus vielen Fachgebieten im Internet zu veröffentlichen. *Die Teilnahme an einem solchen Kurs ist kostenlos*. Typischerweise gibt es zu jeder solchen Vorlesung umfangreiche Unterlagen: Videos mit den eigentlichen Vorlesungen, dazu .pdf-Dateien mit Folien, Aufgaben, multiple choice tests, die automatisch korrigiert und bewertet werden, etc. Viele dieser Kurse wurden bereits von vielen hundert TeilnehmerInnen gleichzeitig absolviert, die populärsten mit 200 Tausend TeilnehmerInnen oder noch mehr. Die meisten Kurse sind auf Englisch, einige auf Deutsch. Hier ein paar Adressen, unter denen man solche Kurse und weitere Informationen findet:

https://www.coursera.org/ (Stanford, Caltech, University of London, ...)
https://www.edx.org/ (Harvard, MIT, ...)
http://oyc.yale.edu/ (Yale)
https://www.canvas.net/
http://www.udacity.com/
https://www.mooc-list.com/language/german

SU 23 Di 28.06.16, Block 1

Heute schreiben wir den Test-11

Ausnahmen (Fortsetzung)

Der Programmierer kann eigene Ausnahme-Klassen vereinbaren, indem er eine schon vorhandene Ausnahmeklasse erweitert. Dabei gilt: Eine Unterklasse einer *geprüften* Ausnahmeklasse ist auch *geprüft*, eine Unterklasse einer *ungeprüften* Ausnahmeklasse ist auch *ungeprüft*.

Buch S. 379: Beispiel-01: Vier eigene Ausnahmeklassen

Die Methoden format und printf (Fortsetzung)

Fehler im Formatstring eines `format-` oder `printf`-Befehls werden in aller Regel (nicht schon bei der *Übergabe* des Programms an den Ausführer sondern) erst bei der *Ausführung* des Programms entdeckt und gemeldet.

Wenn man ein Objekt `ob` einer eigenen Klasse (z.B. ein `Punkt3D`-Objekt) zum Bildschirm ausgeben will, dann kann man den String `ob.toString()` ausgeben (denn jedes Java-Objekt enthält eine Methode `toString`).

Es gibt aber noch eine andere (zusätzliche) Möglichkeit: Man implementiert in der betreffenden Klasse (z.B. `Punkt3D`) die Schnittstelle `Formattable` (voller Name: `java.util.Formattable`). Diese Schnittstelle enthält nur *eine* abstrakte Methode:

```
void formatTo(Formatter f, int flags, int width, int precision)
```

Scheiben Sie die Methoden, die in der Datei **pr1_AufgabenPrintf.txt** spezifiziert werden.

```
static public void printf01(int n, double d, String s) {
   // Gibt ihre Parameter n, d und s (jeden auf einer neuen Zeile)
   // mit printf aus. Die Einzelheiten der Formatierung koennen Sie
   // frei waehlen.

   printf("n: %+,d%n",    n);
   printf("d: %+,6.2f%n", d);
   printf("s: %-10s%n",   s);
}

static public void gibMoeglichstSchmalAus(int[] ir) {
   // Die Komponenten von ir werden lesbar formatiert ausgegeben, eine Zahl
   // pro Zeile, die Einerstellen genau untereinander. Alle Zahlen werden
   // "so schmal wie moeglich" formatiert, d.h. nur so breit wie die
   // breiteste Zahl in ir es erfordert, z.B. so:
   //
   // Beispiel 1: Schmale Zahlen (Breite: 4)
   //   17
   // -123
   //  520
   //    6
   //
   // Beispiel 2: Breite Zahlen (Breite 14)
   //  1.234.567.890
   //             34
   // -1.234.567.890
   //    987.654.321
   //            278

   // Die Breite der breitesten Zahl in ir ermitteln:
   int maxBreite = 1;
   for (int i : ir) {
      String s = format("%,d", i);
      if (maxBreite < s.length()) maxBreite=s.length();
   }
```

```
      // Einen Formatstring fs mit maxBreite zusammenbauen:
      String fs = "%," + maxBreite + "d%n";

      // Die Komponenten von ir mit dem Formatstring fs ausgeben:
      for (int i : ir) printf(fs, i);
   } // gibMoeglichstSchmalAus

   static void pmi(String[] fsr, int[] zr) {
      // Die Komponenten von fsr sollten Formatstrings (fuer format-
      // und printf-Befehle) sein, mit denen man einen int-Wert
      // formatieren kann.
      // Gibt (fsr.length mal zr.length) viele Zeilen aus.
      // Jede Zeile enthaelt:
      // 1. Einen printf-Befehl und danach
      // 2. die Ausgabe dieses printf-Befehls
      // z.B. so:
      // printf(%,14d, -1234567890)    : -1.234.567.890

      for (String fs : fsr) {
         for (int z : zr) {
            // Wie sieht ein printf-Befehl aus, der z mit fs formatiert?
            String s = String.format("printf(%s, %d)", fs, z);
            // Gib diesen printf-Befehl (als String) aus:
            printf("%-30s: ", s);
            // Fuehre diesen printf-Befehl (und einen Zeilenwechsel) aus:
            printf(fs+"%n", z);
         }
         printf("%n");
      }
   }

   static void pmd(String[] fsr, double[] zr) {
      // Die Komponenten von fsr sollten Formatstrings (fuer format-
      // und printf-Befehle) sein, mit denen man einen double-Wert
      // formatieren kann.
      // Gibt (fsr.length mal zr.length) viele Zeilen aus.
      // Jede Zeile enthaelt:
      // 1. Einen printf-Befehl und danach
      // 2. die Ausgabe dieses printf-Befehls
      // z.B. so:
      // printf(%,14d, -1234567890)    : -1.234.567.890

      for (String fs : fsr) {
         for (double z : zr) {
            // Wie sieht ein printf-Befehl aus, der z mit fs formatiert?
            String s = String.format("printf(%s, %f)", fs, z);
            // Gib diesen printf-Befehl (als String) aus:
            printf("%-30s: ", s);
            // Fuehre diesen printf-Befehl (und einen Zeilenwechsel) aus:
            printf(fs+"%n", z);
         }
         printf("%n");
      }
   }
```

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

SU 24 Do 30.06.16, Block 4

Fragen zu den Methoden "mit printf"?

`printf01, gibMoeglichstSchmalAus, pmi und pmd?`

Die Applets printfApplet und eTeachMePrintf

Mit diesen Applets kann man die Methode printf "erkunden und kennen lernen". Das `printfApplet` ist in einem bestimmten Sinn *vollständig*, aber seine Benutzung ist gewöhnungsbedürftig.
Das Applet `eTeachMePrintf` hat eine bessere Oberfläche, ist aber nicht ganz vollständig.

Wie erlaubt man einem Windows-Rechner,
Java-Applets von einer vertrauenswürdigen Seite auszuführen?

Start-Knopf / **Alle Programme** / **Java** / **Configure Java** (oder: **Java konfigurieren**)

Dadurch sollte das `Java Control Panel` geöffnet werden.

Sicherheit / **Siteliste bearbeiten**

Unter **Verzeichnis** das betreffende Verzeichnis
(z.B. `http://public.beuth-hochschule.de/~grude/`)
eintragen (am besten kopieren) und auf **Hinzufügen** klicken.

Die Klasse java.util.Stack

Wie viele Konstruktoren hat die Klasse `Stack`? (Genau einen)

Wie viele Methoden werden in dieser Klasse vereinbart? (5)

Was für Ausnahmen können auftreten, wenn man die Methode `pop` aufruft?
(`EmptyStackException`)

Ist `EmptyStackException` geprüft oder ungeprüft?
(`ungeprüft`, Oberklasse `RunTimeException`)

Was für Ausnahmen können auftreten, wenn man die Methode `push` aufruft?
(keine geprüften Ausnahmen, nur die üblichen ungeprüften. z.B. `OutOfMemoryError`).

Wie heißt die direkte Oberklasse der Klasse Stack? (`Vector`)

Was ist "besonders" an dieser Klasse? (Stack ist eine generische Klasse mit *einem* Typ-Parameter)

Wie alle generischen Klassen definiert die Klasse Stack nicht nur einen Typ, sondern unbegrenzt viele Typen, z.B. `Stack<String>, Stack<Integer>, Stack<Character)` etc.

Verwenden Sie beim Lösen der **Aufgabe 12 Klammern prüfen**
ein Objekt des Typs `Stack<Character>` als Stapel.

Objekte auf "größer oder kleiner" vergleichen

Objekte darf man in Java grundsätzlich nicht mit den Operationen `<, <=, >, >=` vergleichen.
Dafür gibt es folgenden guten Grund (erläutert am Beispiel von `String`-Objekten):

1. Die `boolean`-Operationen <, <= etc. liefern nur eines von **zwei** Ergebnissen (`true` oder `false`).
2. Der Vergleich zweier Strings `s1` und `s2` kann aber **drei** verschiedene Ergebnisse haben:
`s1` ist kleiner, gleich oder größer als `s2`.
3. Um diese **drei** Ergebnisse zu unterscheiden würde man 2 Vergleiche mit den `boolean`-Operationen benötigen, z.B. so:

```
if (s1<s2) {
   ...
} else if (s1>s2) {
   ...
} else {
   ...
}
```

Da Strings ziemlich groß sein können, können diese zwei `String`-Vergleiche ziemlich viel Zeit kosten.

Deshalb vergleicht man `String`-Objekte (statt mit `boolean`-Operationen) mit `int`-Funktionen, die einen negativen Wert, der Wert 0 bzw. einen positiven Wert liefern, je nachdem ob `s1` kleiner, gleich bzw. größer als `s2` ist. Ein Vergleich kann dann z.B. so ablaufen:

```
int erg = s1.compareTo(s2); // Ein teuerer Vergleich
if (erg<0) {                // Ein billiger Vergleich
   ...
} else if (erg>0) {         // Noch ein billiger Vergleich
   ...
} else {
   ...
}
```

Wie macht man Objekte einer selbst geschriebenen Klasse größer/kleiner vergleichbar?

```
class MeineKlasse implements Comparable<MeineKlasse> {
   ...
   public int compareTo(MeineKlasse that) {
      // Liefert einen negativen Wert, den Wert 0 bzw. einen positiven Wert
      // je nachdem ob this kleiner, gleich bzw. groesser that ist.
      ...
   }
   ...
}
```

Siehe auch **Buch S. 435, Beispiel-02**

Wie rüstet man Objekte mit weiteren Vergleichsfunktionen aus?

Für jede weitere Vergleichsfunktion vereinbart man eine weitere Klasse wie die folgende:

```
class WK7 implements Comparator<MeineKlasse> {
   public int compare(MeineKlasse ob1, MeineKlasse ob2) {
      // Liefert einen negativen Wert, den Wert 0 bzw. einen positiven Wert
      // je nachdem ob ob1 kleiner, gleich bzw. groesser ob2 ist.
      ...
   }
}
```

Achtung:
Die Methode `compareTo` (Schnittstelle `Comparable`) hat nur *einen* Parameter.
Die Methode `compare` (Schnittstelle `Comparator`) hat zwei Parameter.

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" nahm man bis etwa 2014 "18 Monate" an. Offenbar galt damals:

1. Prozessoren werden schneller.
2. Plattenspeicher werden größer und schneller.
3. Aber: Prozessoren werden schneller schneller als Plattenspeicher.

Daraus folgt:

"Ergebnisse neu berechnen, wenn man sie braucht"	wird im Vergleich zu
"Ergebnisse einmal berechnen und in Dateien sammeln"	immer günstiger.

Ein aktueller Trend (2009): Die Prozessoren werden nicht mehr schneller, sondern zahlreicher. Damit wird das Konzept der Nebenläufigkeit noch wichtiger als es schon war.

Noch ein aktueller Trend (2012): Anstelle von Festplatten verwendet man zunehmend *Solid State Disks* (SSDs, Flash-Speicher). Die sind fast so schnell wie der Hauptspeicher.

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).

SU 25 Di 05.07.16

Heute schreiben wir den Test-12

Was man über den Unicode wissen sollte

Ein **Zeichen-Code** legt Code-Zahlen für bestimmte Zeichen fest,
aber nicht die Zeichen selbst ("ihre Form").

Man sagt deshalb auch: Ein Zeichen-Code bildet *Namen-für-Zeichen* auf *Code-Zahlen* ab
(z.B. den Namen LATIN CAPITAL LETTER A auf die Zahl 65).

Ein **Font** bildet *Code-Zahlen* auf konkrete Zeichen ab (z.B. bildet der Font **Gabriola** die Codezahl 65 auf das Zeichen A ab.

Der Unicode umfasst etwa 1 Million Code-Zahlen (genau: $1.114.112_{10}$, 11.0000_{16}).
Davon werden etwa 65 Tausend durch **einen** `char`-Wert dargestellt.
Die übrigen (etwa 1,04 Millionen) Code-Zahlen werden durch **zwei** `char`-Werte dargestellt.

Für die (aktuelle) **Version 8 des Unicode** gilt:
Von den 1-char-Code-Zahlen sind etwa 61 Tausend **belegt**
(mit besonders *häufig verwendeten* Zeichen) und etwa 2 Tausend sind (noch) **nicht belegt**.
Von den 2-char-Code-Zahlen sind etwa 180 Tausend **belegt**
(mit *weniger häufig verwendeten* Zeichen) und etwa 860 Tausend noch **nicht belegt**.

Wird eine Code-Zahl durch 2 `char`-Werte dargestellt, so bezeichnet man den
ersten als H-Teil (H wie high) und den zweiten als L-Teil (L wie low).
Der H-Teil ist immer eine der (1.024) Zahlen zwischen 0xD800 und 0xDBFF.
Der L-Teil ist immer eine der (1.024) Zahlen zwischen 0xDC00 und 0xDFFF.

Man kann also jedem `char`-Wert ansehen, ob er "allein ein Zeichen codiert" oder ob er "die erste oder zweite Hälfte eines Paares ist".

Beispiel-01: Der lateinische **Buchstabe A** wird durch den **einen** `char`-Wert `'\u0041'` dargestellt, und der latainische **Buchstabe B** durch `'\u0042'` etc.

Beispiel-02: Das Zeichen **Violin-Schluessel** (welches in Musik-Noten vorkommt) wird durch den H-Teil `'\uD834'` und den L-Teil `'\uDD1E'` dargestellt.
Ein **Bass-Schlüssel** wird durch den H-Teil `'\uD834'` und und den L-Teil `'\uDD22'` dargestellt.

Zeichen und char-Werte

Beispiel: Ein String der Länge 10 enthält mindestens 5 und höchstens 10 Zeichen
(z.B. 5 Violin-Schlüssel oder 10 Buchstaben A oder 4 Violin-Schlüssel und 2 Buchstaben A etc.).

Sei `s` ein `String`. Dann bezeichnet
der Ausdruck `s.length()` die *Länge* von `s` (d.h. die *Anzahl der char-Werte* in `s`) und
der Ausdruck `s.codePoints()` die *Anzahl der Zeichen* in `s`.

Zur Zeit legt der Unicode unter anderem Code-Zahlen für folgende Zeichen fest:

- Schriftzeichen aller heute (auf der Erde) benutzten Schriftsprachen
  (**lateinische** Buchstaben, **arabische**, **griechische**, **koptische**, **kyrillische**, **koreanische**, ...)
- insbesondere für mehr als 30 Tausend **chinesische** Schriftzeichen
- Schriftzeichen für viele "ausgestorbene" Schriftsprachen (z.B. **Keilschrift**, **Hieroglyphen**, ...)
- Schriftzeichen für "Fantasie-Sprachen" (z.B. **Tengwar**, **Klingonisch**)
- andere Zeichen (mathematische Zeichen, Noten, Emojis, Spielkarten, ...)

Innerhalb eines Java-Programms werden Zeichen grundsätzlich im Unicode dargestellt (jedes Zeichen durch einen oder zwei `char`-Werte). Beim Einlesen und Ausgeben von Daten wird zwischen Unicode und verschiedenen anderen Codes (UTF-8 oder 8-Bit-ASCII oder ...) umgewandelt.

Wenn Sie möchten, dass ein Zeichen oder eine Gruppe von Zeichen (z.B. ein neues Alphabet) in den Unicode aufgenommen wird, können Sie beim **Unicode Consortium** einen entsprechenden Antrag stellen (siehe vorher: http://www.unicode.org/pending/proposals.html)

Cast-Befehle (Typ-Umwandlungsbefehle)

Syntax: Ein Cast-Befehl besteht aus einem Typ-Namen in runden Klammern,
z.B `(int)` oder `(String)` oder `(E01Rechteck)` etc., gefolgt von einem Ausdruck.

Den Cast-Befehl `(int)` darf man nur auf einen Wert anwenden, dessen Typ "eng verwandt ist" mit dem Typ `int`. Alle (primitiven) numerischen Typen (`byte, short, char, int, long, float, double`) gelten als "nah miteinander verwandt".

Beispiel-01:

```
 1 float  f = EM.liesFloat();
 2 int    n = (int)    f;  // Cast notwendig
 3 char   c = (char)   f;  // Cast notwendig
 4 long   m = (long)   n;  // Cast nicht notwendig
 5 String s = (String) n;  // Verboten, String und int sind nicht "nah verwandt"!
```

Den Cast-Befehl `(E01Rechteck)` darf man nur auf einen Wert anwenden, dessen Typ "eng verwandt ist" mit dem Typ `E01Rechteck`. Als "nah verwandt mit dem Typ `E01Rechteck`" gelten alle seine Unter- und Obertypen (d.h. der Untertyp `E01Quadrat` und die Obertypen `E01Punkt` und `Object`). Auf Werte anderer Typen darf man den Cast-Befehl `(E01Rechteck)` nicht anwenden.

Beispiel-02:

```
 6 E01Punkt[]  tab = ...;
 7 int         i   = ...;
 8 E01Rechteck r   = (E01Rechteck) tab[i]; // Cast notwendig
 9 E01Quadrat  q   = (E01Quadrat)  r;      // Cast nicht notwendig
10 String      s   = (String) q;          // Verboten, String und E01Quadrat sind
                                          // nicht "nah verwandt"!
```

Wichtiger Unterschied:

Primitive Cast-Befehle erzeugen aus einem (primitiven) Wert `a` einen primitiven Wert `b`. Dabei können sich `a` und `b` erheblich unterscheiden (z.B. ist `(char) 4_259_905.0;` gleich `'A'`)

Cast-Befehle mit Referenztypen ändern nur, *wie ein Wert interpretiert wird*, an dem betroffenen Objekt wird aber kein einziges Bit verändert.

Zeile 8 im Beispiel-02: Die Variable `tab[i]` hat den Typ `E01Punkt`. Wenn sie auf ein `E01Rechteck`-Objekt zeigt (was durchaus möglich ist), dann geht der Cast-Befehl gut, sonst wird eine Ausnahme (des Typs `ClassCastException`) geworfen.

Allgemein gilt: Eine Referenz-Variable `v` hat (nicht nur einen, sondern) **zwei** Typen:

den Typ, mit dem sie vereinbart wurde (das ist der **Typ von v**) und
den Typ des Objekts, auf das sie zeigt (das ist der **Ziel-Typ von v**)

Buch Seite 306, unten, **Beispiel-01** (fortgesetzt auf Seite 307).

Wenn noch Zeit ist:
Aufgaben im Archiv **pr1_UebMitJut.zip** bearbeiten (mit Papier, Bleistift und Radiergummi)

Zur Entspannung.: **Gottfried Wilhelm Leibniz** (1646-1716, Leipzig-Hannover), Philosoph, Politiker, Forscher

Erfand (zeitgleich mit und unabhängig von Isaac Newton, 1643-1727) die Differentialrechnung. Konzipierte einen Logikkalkül und betrieb Wahrscheinlichkeitsrechnung (mit Anwendung z. B. auf Würfelspiele).

Führte 1673 der Royal Society in London eine Rechenmaschine vor. Heute ist nicht ganz klar, was diese Maschine praktisch konnte und was nicht. 1894 wurde die Maschine von Leibniz an der TU Dresden restauriert. 19?? fand ein Prof. Lehmann an der TU Dresden heraus, dass die Restaurierungsarbeiten auf einem Denkfehler beruhten und das Funktionieren der Maschine verhinderten. Prof. Lehmann baute dann nach den Plänen von Leibniz eine funktionierende Maschine.

Leibniz beschrieb als erster das binäre Zahlensystem.

SU 26 Do 07.07.16, Block 4

Typ und Ziel-Typ einer Referenz-Variablen

Allgemein gilt: Eine Referenz-Variable v hat (nicht nur einen, sondern) **zwei** Typen:

den Typ, mit dem sie vereinbart wurde (das ist der **Typ von v**) und
den Typ des Objekts, auf das sie zeigt (das ist der **Ziel-Typ von v**)

Buch Seite 306, unten, **Beispiel-01** (fortgesetzt auf Seite 307).

Steuerfäden (threads of control)

Alle bisher behandelten Programme waren *sequentielle* Programme. Die Befehle eines solchen Programms werden in einer ganz bestimmten, vorhersehbaren Reihenfolge ausgeführt.

Es gibt Probleme, die man mit einem solchen sequentiellen Programm nicht wirklich lösen kann, z.B.: Ein Programm soll von 2 Tastaturen (an denen 2 Benutzer sitzen) Zahlen einlesen und zu einer Summe addieren.

Buch S. 495, oben, **Beispiel-01**: Von zwei Tastaturen Zahlen einlesen

Warum ist das skizzierte Programm keine gute Lösung? Was passiert, wenn einer der beiden Benutzer auf die Toilette geht? (dann muss der andere warten). Was passiert, wenn der eine Benutzer viel schneller tippen kann als der andere? (dann muss der schnellere sich der Geschwindigkeit des langsameren anpassen).

Um solche Probleme gut zu lösen hat man Ausführer (Hardware und Software) erfunden, die bestimmte Programme oder Programmteile **nebenläufig** (d.h. zeitlich unabhängig, engl. **concurrently**) ausführen können.

Beispiel-02: Eine Lösung mit nebenläufigen Programmteilen

Ganz kurz: Der **Unterschied zwischen Prozessen** (processes) **und Fäden** (threads).

Prozesse sind nebenläufige Einheiten, die *keinen gemeinsamen Variablen* haben (teuer, sicher).

Fäden sind nebenläufige Einheiten, die *gemeinsamen Variablen* haben (billig, unsicher).

Wie sich zwei Fäden stören können ("mit Hilfe von gemeinsamen Variablen")

Buch S. 499, Beispiel-01

S. 500, Beispiel-01: Eine Faden-Klasse in Java

In Zeile 13 wird der Faden eine zufällige gewählte Anzahl von Millisekunden "schlafen gelegt". An dieser Stelle könnte der Faden auch "irgendeine nützliche Arbeit verrichten" (aber dann wäre das Beispiel viel schwerer zu durchschauen).

S. 501, Beispiel-02: Zwei Fäden erzeugen und starten

S. 501, Beispiel-03: Und so sieht die Ausgabe der beiden Fäden aus

Grabos (grafische Benutzeroberflächen) in Java

Kurze Geschichte von Grabos in Java

AWT (Abstract Window Toolkit), von Sun. Seit Java 1 (**1995**).
Unter hohem Zeitdruck in wenigen Monaten entwickelt. "Schwere Komponenten".

Swing (keine Abkürzung), von Sun, seit Java 2 (**1998**)
Baut auf dem **AWT** auf, "Leichte Komponenten" (ganz in Java programmiert).

SWT (Standard Widget Toolkit), von IBM für Eclipse entwickelt, (**2001**)
"Schwere Komponenten", heute Open Source (Eclipse Foundation)

JavaFX 1.0, von Sun (**2008**)
JavaFX 2.0, von Sun (2011) (es gibt keine Versionen 3.0 bis 7.0)
JavaFX 8.0, von Sun, seit Java 8 (**2014**)

AWT und **Swing** hängen eng zusammen.

SWT ist völlig unabhängig von **AWT/Swing**.

JavaFX ist unabhängig von **AWT/Swing**, aber man kann in eine **AWT/Swing**-Grabo **JavaFX**-Komponenten einbauen (andersrum, **AWT/Swing-Komponenten** in eine **JavaFX**-Grabo einbauen, geht **nicht**).

Alle verbreiteten Betriebssysteme (MacOS, Linux, Windows, ...) stellen **native** ("eingeborene") Grabo-Klassen und -Objekte zur Verfügung, die sich ähneln, aber auch unterscheiden.
Z.B. verhält sich ein *MacOS-Knopf* nicht in allen Situationen ganz genau so wie ein *Linux-Knopf* oder ein *Windows-Knopf* etc.

Die Grabo-Klassen und Objekte einer Programmiersprache müssen eng mit dem jeweiligen Betriebssystem (BS) zusammenarbeiten, weil das BS den Bildschirm "besitzt und verwaltet".

Zur Entspannung: **Nicht-transitive Würfel**

Viele "Vergleichs-Relationen" wie *istSchneller, istHöher, istGrößer* etc. sind *transitiv*, d.h.
wenn A schneller ist als B und B schneller ist als C dann gilt auch: A ist schneller als C.

Betrachten wir noch eine andere *istBesser*-Relation: Zwei Personen würfeln mehrmals mit zwei Würfeln W1 und W2 gegeneinander. Wer die höhere Augenzahl würfelt, hat den Wurf gewonnen und bekommt einen Plus-Punkt (der Verlierer bekommt einen Minus-Punkt). Die Würfel W1 und W2 können unterschiedliche Zahlen auf ihren sechs Seiten haben und somit besser oder schlechter sein. Z.B. ist ein Würfel, der auf allen sechs Seiten eine 6 hat offensichtlich besser als einer, der auf allen sechs Seiten eine 1 hat. Aber was ist mit den folgenden 4 Würfeln?

W1: `0 0 4 4 4 4`
W2: `3 3 3 3 3 3`
W3: `2 2 2 2 6 6`
W4: `1 1 1 5 5 5`

In einem Kampf W1 gegen W2 wird W1 im Durchschnitt
2/3 aller Würfe gewinnen (mit 4 zu 3) und 1/3 aller Würfe verlieren (mit 0 zu 3).
W1 ist also besser als W2.

Aus ganz ähnlichen Gründen gilt auch:

W2 ist besser als W3.
W3 ist besser als W4.
Und erstaunlicherweise auch: W4 ist besser als W1.

Quelle: Martin Gardner, "The Colossal Book of Mathematics",
W. W. Norton & Company, ca. 700 Seiten, 35,- US$

SU 27 Di 12.07.16, Block 1

Heute schreiben wir den Test-13 (d.h. den letzten Test)

Wiederholungsfragen

Vereinbaren Sie eine Variable mit dem Typ `Object` und dem Zieltyp `StringBuilder`.
```
Object ob = new StringBuilder();
```

Warum kann man keine Variable mit dem Typ `Quader` und dem Zieltyp `Punkt3D` vereinbaren?
(Weil `Punkt3D` kein Untertyp von `Quader` ist)

Was kann schief gehen, wenn zwei Fäden (engl. threads) Zugriff auf eine Variable haben, z.B. auf eine `int`-Variable mit dem Wert 17?

Wie heißen die drei "Bibliotheken" zum Schreiben von Grabos, die die Firma Sun für Java entwickelt hat? (AWT, Swing, JavaFX)

Wie hängen diese drei zusammen? (Swing baut auf AWT auf, JavaFX ist unabhängig von AWT und Swing, in eine AWT-Swing-Grabo kann man man JavaFX-Elemente einbauen.

Warum ist das Ausführen und Verwalten von Grabo-Programmen für ein Betriebssystem viel schwieriger, als das Ausführen und Verwalten von Programmen ohne Grabo?
(Weil die Fenster verschiedener Programme sich gegenseitig teilweise oder ganz verdecken können. Wenn eine solche "Verdeckung" verschwindet, müssen die verdeckten Fenster (oder die verdeckten Teile der Fenster) neu gezeichnet werden. Ein einzelnes Programm weiß aber nicht, welche Teile seiner Fenster gerade verdeckt sind bzw. gerade wieder sichtbar wurden, nur das Betriebssystem weiß das).

Grabos (Fortsetzung)

Eine **Grabo** (graphische Benutzeroberfläche, engl. GUI, Graphical User Interface)
besteht aus Grabo-Objekten. (Fenstern, Menüs, Knöpfen, Textfeldern, ... etc.).

Buch
S. 541, Zwei Definitionen für die Begriffe **Grabo-Objekt** und **Grabo-Klasse**

S. 542, Definitionen der Begriffe (Grabo-) **Behälterklasse** und **Behälter** (-Objekt).

S. 544, **Bilds 22.4**: Ein Typgraf mit wichtigen AWT- und Swing-Klassen

Ein Behälter (container) kann Grabo-Komponenten (components) enthalten.

Die Klasse `Container` ist eine Erweiterung (oder: eine Unterklasse) von `Component`.
Was bedeutet das? Was folgt daraus?
(Jeder Behälter ist auch eine Komponente, und somit kann ein Behälter kann auch Behälter enthalten, d.h. man kann Behälter *schachteln*).

Die Swing-Klassen `JPanel`, `JLabel`, `JToggelButton`, `JButton` etc. sind
Erweiterungen (oder: Unterklassen) von `Container`? Was folgt daraus?
(z.B.: Jeder `JButton` etc. ist auch ein Behälter).

Ein Grabo-Programm wird von mindestens 2 Fäden ausgeführt:

Der **Haupt-Faden** (main thread) führt (wie bei sequentiellen Programmen) die `main`-Methode aus.

Der **Ereignis-Faden** (event thread) zeichnet die Grabo auf den Bildschirm, wartet dann auf *Aktionen* des Benutzers und behandelt die dadurch erzeugten Ereignisse (events).

Der Befehl
```
pln(Thread.currentThread());
```
gibt den Namen und weitere Informationen über *den* Faden aus, der ihn (den Befehl) ausführt.

Beispiel für ein Grabo-Objekt, sein Abbild und ein Ereignis
In einem Java-Programm kann man einen Knopf (z.B. vom Typ `JButton`) vereinbaren.
Bei diesem Knopf kann man *Behandlungsmethoden* für Ereignisse der Art `actionPerformed` anmelden.
Wenn der Benutzer eine bestimmte *Aktion* ausführt (indem er auf den Knopf klickt oder ein entsprechendes Tastenkürzel eingibt) tritt ein Ereignis der Art `actionPerformed` ein und alle (bei dem Knopf) angemeldeten Behandlungsmethoden namens `actionPerformed` werden aufgerufen.

Kurz: Eine *Aktion* des Benutzers kann bewirken, dass ein *Ereignis*-Objekt erzeugt wird und dass *Behandler-Methoden* aufgerufen werden.

Anonyme Objekte und anonyme Klassen

"anonym" bedeutet: namenlos, hat keinen Namen.

Beispiele für *anonyme Objekte* benannter Klassen:

```
 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    });
```

Eine Behandler-Methode bei einer Grabo-Komponente anmelden

Die Programme `Grabo30` und `Grabo31` ausgedruckt austeilen bzw. im Internet ansehen.

Zur Entspannung: **Was ist schrecklich an Grabos (engl. GUIs)?**

1. Es gibt keine einfache, klare Sprache, in der man die *Bedienung* einer Grabo beschreiben könnte.

Anleitungen wie "Öffnen Sie Menü A, wählen sie Punkt B, klicken Sie auf Knopf C, geben Sie "ABC" in das Fensterchen neben "Eingabe" ein, ..." sind schwer auszuführen und erklären einem nichts über die Struktur der Grabo, mit der man gerade arbeitet.

Screenshots sind häufig sehr umfangreich, selbst für einfache Funktionen muss man mehrere Seiten davon anlegen/lesen. Ein einzelner Screenshot enthält häufig sehr viele Details, von denen nur ganz wenige relevant sind. Man lernt vor allem die Farbe bestimmter Knöpfe und die Positionen gewisser Fenster, die nichts mit dem zu lösenden Problem zu tun haben und in der nächsten Version der Grabo anders sein können.

2. Es gibt keine Möglichkeit, komplizierte Bedienschritte zu automatisieren (z.B. durch ein Skript). Heutige Makro-Recorder leisten zu wenig.

3. Grabos lassen dem Programmierer sehr viel Wahlmöglichkeiten und es gibt keine Regeln, wie er wählen sollte. Deshalb kann ein Benutzer kaum vorhersehen, welches Problem durch welche Art von Grabo-Objekt gelöst wurde (durch einen Knopf? einen Menüpunkt? ein Eingabefeld? eine ComboBox? ...)

4. Viele Grabos enthalten außerdem elementare Fehler, z.B.:

- Bestimmte Fenster sind zu klein und können nicht vergrößert werden (auch wenn man sich gerade einen sehr großen Bildschirm gekauft hat)
- Texteingaben müssen in einem schwer handhabbaren Format erfolgen und werden nicht sofort geprüft (z.B. die Werte von Umgebungsvariablen bei Windows und Linux)
- Manchmal muss man einen bestimmten Arbeitsgang für viele Objekte wiederholen, statt dass man alle Objekte auswählt und dann den Arbeitsgang einmal durchführt.

SU 28 Do 14.07.16, Block 4

Klausur-Vorbereitungen

Am **Do 21.07.2016**, ab **16 Uhr** im **Raum B101** schreiben wir die (Haupt-) Klausur.
Als Unterlagen dürfen Sie mitbringen: **5 Blätter**, max. Format DIN A4, beliebig beschriftet.
Sie sollten alles wissen und können, was im Laufe des Semesters in den SUs und Üs behandelt wurde, insbesondere sollten Sie folgendes können:
- Methoden schreiben (mit Papier und Stift, *ohne* Tastatur und Bildschirm).
- Variablen als Bojen darstellen
- Programme ausführen (und so z.B. ermitteln, was zum Bildschirm ausgegeben wird)
- Fragen zum behandelten Stoff beantworten.

Anmerkung 1: Beantworten Sie Fragen möglichst *kurz*, aber auch möglichst *genau* und verwenden Sie dabei möglichst die *Fachbegriffe*, die behandelt wurden. Es ist nicht empfehlenswert, "möglichst viel und alles was einem einfällt" zu schreiben, denn alle Fehler zählen negativ, auch wenn sie gar nichts mit der Frage zu tun haben.

Beispiel: Betrachten Sie folgenden Befehl:

```
int otto = 17;
```

Frage: Mit welchem Wert wird die Variable `otto` initialisiert?

Schlechte Antwort: Der Befehl ist eine Zuweisung, `otto` ist eine Referenzvariable und wird mit dem Wert 17 initialisiert.

Gute Antwort: 17

Anmerkung 2: Geben Sie (für eine Aufgabe bzw. Teilaufgabe) immer nur *eine* Lösung an.
Falls Sie mehrere Lösungen angeben, zählen alle Fehler in allen Lösungen gegen Sie.

Anmerkung 3: Bringen Sie eigene Blätter mit, auf die Sie Lösungen schreiben können.
Empfehlung: Kariertes, radiergummifestes Papier (kein Umweltpapier!).
Sie dürfen mit Bleistift schreiben (und radieren)!

Anmerkung 4: Die Klausur besteht aus 6 Aufgaben. Einige davon bestehen aus mehreren Teilaufgaben. Schreiben Sie die Lösung jeder Aufgabe auf die *Vorderseite eines neuen Blattes*. Lassen Sie die *Rückseite* Ihrer Lösungsblätter grundsätzlich *leer*. Falls die Lösung einer Aufgabe nicht auf eine Seite passt, dann schreiben Sie sie auf die Vorderseiten von 2 Blättern. Normalerweise sollten Sie am Ende der Klausur 6 Blätter (alle mit leerer Rückseite) abgeben.
Kennzeichnen Sie jedes Lösungsblatt (in der rechten oberen Ecke) mit Ihrem *Nachnamen*. Weitere Angaben (Schuhgröße, Steuer-Nummer etc. sind erlaubt, aber *nicht nötig*.

Ereignisse, Ereignis-Arten, Ereignis-Oberarten

S. 546, Def. **Ereignis** (*grundsätzlich nicht wiederholbar!*).
"Man kann nicht zweimal in denselben Fluss steigen" (Heraklit, Platon, Simplikios).

Man unterscheidet zahlreiche **Arten von Ereignissen** (S. 547),
z.B. die Arten `windowOpened` oder `MouseClicked` etc.

"Eine Art von Ereignissen behandeln" heißt:
Eine Methode schreiben, die immer dann ausgeführt wird, wenn ein Ereignis dieser Art eintritt.

Jede **Art** (von Ereignissen) gehört zu einer bestimmten **Oberart**.
Beispiel: Die Art `windowOpened` gehört zur Oberart Fensterereignis.

Zu jeder **Oberart** gibt es eine **Schnittstelle**.
Beispiel: Zur Oberart Fensterereignis gibt es die Schnittstelle `WindowListener`.

Diese Schnittstelle enthält für jede Art (die zu der betreffenden Oberart gehört), eine (abstrakte Objekt-) Methode.

Zu vielen **Oberarten** gibt es eine **Adapter-Klasse**, die die betreffende Schnittstelle ("mit Strohpuppen") implementiert.
Beispiel: Zur Oberart *Fensterereignis* gibt es die Adapter-Klasse `WindowAdapter`.

In AWT/Swing gilt:

Wenn man *eine* Art von Ereignissen behandeln will, muss man *alle* Arten behandeln, die zur gleichen Oberart gehören (oder: man muss alle Methoden überschreiben, die zur selben Schnittstelle gehören).

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 und 9 ersetzen, dass der Satz wahr wird? Ein Programm schreiben, welches Lösungen findet?

SU 29 Di 19.07.16, Block 1

Noch ein paar Hinweise zur Klausur

Was Sie können sollten:

Strings ("auf größer/kleiner") vergleichen
Sammlungen (eines `ArrayList`-Typs) bearbeiten.
Bojen zeichnen (wahlweise in vereinfachter oder ausführlicher Darstellung, empfohlen: vereinfachte)
`try-catch`-Befehle und `try-catch-finally`-Befehle ausführen.
`printf`-Befehle ausführen ("relativ einfache").
Den Charakter eines Elements erkennen können.
Elemente mit einem bestimmten Charakter vereinbaren können.

Sie sollten sich auch an den Stoff der ersten SUs erinnern
Rollenspiel,
die wichtigsten Grundkonzepte der Programmierung,
die drei Arten von Befehlen (wie man sie ins Deutsche übersetzen kann, was man damit befiehlt).

Noch eine Regel: Missbrauchen Sie das bedruckte Klausurblatt nicht als "Schmierzettel".
Schreiben Sie "Nebenrechnungen und Notizen" auf ein separates Blatt, das Sie nicht abgeben.

Rückgabe der Klausur: Di 26.07.2016, 10 Uhr im SWE-Labor (Raum DE16)

Möglicherweise werde ich schon vorher eine Email mit den Klausur-Noten (und den letzten vier Ziffern Ihrer Matrikel-Nrn, aber ohne Namen) verschicken. Wer in dieser Email nicht vorkommen möchte sollte mir eine entsprechende Email schreiben oder auf das bedruckte Klausurblatt (vorn, ganz oben) **"Keine Benachrichtigung per Email!"** schreiben.

Hat jemand Fragen zum Stoff, der
in dieser LV behandelt wurde (und somit möglicherweise in der Klausur erwähnt wird)?

Ein paar Aufgaben aus der Datei **pr1_Klausur01.pdf**:

Aufgabe 1 (20 Punkte): Schreiben Sie eine Methode entsprechend der folgenden Spezifikation:

```
22    static public boolean istGroesser(ArrayList<String> als, String[] rs) {
23       // Liefert true genau dann wenn
24       // als mehr Komponenten enthaelt als rs   oder wenn
25       // als gleich viele Komponenten enthaelt wie rs und jede Komponente
26       // von als groesser ist als die entsprechende Komponenten von rs.
27       // Beispiele:
28       //
29       // String[]          rs1  = {"AA", "BBB"};
30       // String[]          rs2  = {"BB", "CCC"};
31       // String[]          rs3  = {"AA", "AAA", "A"};
32       // ArrayList<String> als1 = new ArrayList<String>(Arrays.asList(rs1));
33       // ArrayList<String> als2 = new ArrayList<String>(Arrays.asList(rs2));
34       // ArrayList<String> als3 = new ArrayList<String>(Arrays.asList(rs3));
35       //
36       // istGroesser(als3, rs1) ist gleich true
37       // istGroesser(als1, rs2) ist gleich false
38       // istGroesser(als2, rs1) ist gleich true
39       // istGroesser(als3, rs3) ist gleich false
40       ...
41    } // istGroesser
```

Aufgabe 5 (15 Punkte): Geben Sie von jeder der folgenden 4 Schleifen an, was sie zum Bildschirm ausgibt.

```
1       // Schleife 5.1:
2       int n = -3;
3       for (int i=0; i<=10; i+=n) {
4          n +=2;
5          p(i + " ");
6       }
7       pln();
8
9       // Schleife 5.2:
10      int[] rei = {2, 0, 3, 1};
11      for (int i : rei) rei[i] = 3-i;
12      pln(Arrays.toString(rei));
13
14      // Schleife 5.3:
15      StringBuilder[] sbr = {
16         new StringBuilder("A+"),
17         new StringBuilder("B+"),
18         new StringBuilder("C+"),
19      };
20
21      for (int i=1; i<sbr.length; i++) {
22         sbr[i].append(sbr[i-1]);
23      }
24
25      for (StringBuilder sb : sbr) p(sb);
26      pln();
27
28      // Schleife 5.4:
29      for (char c1='A'; c1<='H'; c1+=2) {
30         for (char c2=c1; c2<='H'; c2++) {
31            p(c2);
32         }
33         pln();
34      }
```

Lösungen zu Aufgabe 1 und Aufgabe 5

```
1    static public boolean istGroesser(ArrayList<String> als, String[] rs) {
2       // Liefert true genau dann wenn
3       // als mehr Komponenten enthaelt als rs   oder wenn
4       // als gleich viele Komponenten enthaelt wie rs und jede Komponente
5       // von als groesser ist als die entsprechende Komponenten von rs.
6       // Beispiele:
7
8       if (als.size() > rs.length) return true;
9       if (als.size() < rs.length) return false;
10
11      for (int i=0; i<rs.length; i++) {
12         if (als.get(i).compareTo(rs[i]) <= 0) return false;
13      }
14      return true;
15
16   } // istGroesser

17 // Schleife 5.1: 0 -1 0 3 8
18 // Schleife 5.2: [3, 2, 1, 1]
19 // Schleife 5.3: A+B+A+C+B+A+
20 // Schleife 5.4:
21 ABCDEFGH
22 CDEFGH
23 EFGH
24 GH
```

Die **Nachklausur** findet statt am **Mi 28.09.2016**, ab **10 Uhr** im Raum **DE17** (SWE-Labor).
Als Unterlagen sind erlaubt. 5 Blätter, max. Format DIN A 4, beliebig beschriftet.