Stichworte

zur Lehrveranstaltung **Programmieren 2** (MB2-PR2, Zug 1)
im zweiten Semester des Studiengangs **Medien-Informatik Bachelor**
im **Wintersemester 2016/17** an der **Beuth Hochschule für Technik** Berlin,
von Ulrich Grude.

In dieser Datei finden Sie
alle Termine dieser Lehrveranstaltung (S. 2),
die Regeln, nach denen man Noten bekommt (ab S. 3) und
nach jedem SU (seminaristischen Unterricht) ein paar Stichworte dazu (ab S. 5).

Anmerkung zum Belegen dieser Lehrveranstaltung

Diese Lehrveranstaltung sollten Sie nur belegen, wenn Sie bereit und in der Lage sind, ein ganzes Semester lang jeden Dienstag um 8 Uhr in der Beuth Hochschule einen kleinen Test zu schreiben und sich fest vorgenommen haben, an allen seminaristischen Unterrichten (SUs) und allen Übungen (Üs) teilzunehmen.

Wenn Sie diese Voraussetzungen erfüllen besteht die Möglichkeit, dass diese Lehrveranstaltung ihnen nützt und (zumindest gelegentlich) Spaß macht. Andernfalls besteht die Gefahr, dass diese Lehrveranstaltung Sie unangenehm belastet und frustriert.

Inhaltsverzeichnis

Termine dieser Lehrveranstaltung

Termine der *seminaristischen Unterrichte* (SU-01, SU-02, ...), der *Übungen* (Ü1a-01, Ü1a-02, ...Ü1b-01, Ü1b-02, ...), wann wird welcher *Test* geschrieben (T01, T02, ...) und wann sollte welche *Aufgabe* abgegeben bzw. vorgeführt werden (A01, A02, ...)?

KW	SW	Dienstag	Block 1: SU	Block 2: Ü1a	Block 3: Ü1b	Aufgaben
40	01	04.10.2016	ausgefallen	ausgefallen	ausgefallen	
41	02	11.10.2016	T01, SU-01	Ü1a-01	Ü1b-01	
42	03	18.10.2016	T02, SU-02	Ü1a-02	Ü1b-02	A01
43	04	25.10.2016	T03, SU-03	Ü1a-03	Ü1b-03	A02, 1. Hälfte
44	05	01.11.2016	T04, SU-04	Ü1a-04	Ü1b-04	A02, 2. Hälfte
45	06	08.11.2016	T05, SU-05	Ü1a-05	Ü1b-05	A03
46	07	15.11.2016	T06, SU-06	Ü1a-06	Ü1b-06	A04
47	08	22.11.2016	T07, SU-07	Ü1a-07	Ü1b-07	A05
48	09	29.11.2016	T08, SU-08	Ü1a-08	Ü1b-08	
49	10	06.12.2016	T09, SU-09	Ü1a-19	Ü1b-09	A06
50	11	13.12.2016	T10, SU-10	Ü1a-10	Ü1b-10	A07
51	12	20.12.2016	T11, SU-11	Ü1a-11	Ü1b-11	A08
51 01	13	23.12.16 - 03.01.17	Ferien			
02	14	10.01.2017	T12, SU-12	Ü1a-12	Ü1b-12	A09
03	15	17.01.2017	T13, SU-13	Ü1a-13	Ü1b-13	A10
04	16	24.01.2017	SU-14	Ü1a-14	Ü1b-14	
05	17	31.01.2017	**Klausur**			
06	18	07.02.2017		**Rückgabe**		

Aküs

Aküs:	Abkürzungen	SU:	seminaristischer Unterricht
KW:	Kalenderwoche	Ü1a:	Übungsgruppe 1a
SW:	Semesterwoche	Ü1b:	Übungsgruppe 1b

Belegungszeitraum WS16/17: **15.2016.-19.10.2016**

Haupt-Klausur: **Di 31.01.2017, 08 Uhr, Raum B101**
Rückgabe: **Di 07.02.2017, 10 Uhr, Raum DE17 (Eingang durch DE16)**
Nachklausur: **Do 30.03.2017, 10 Uhr, Raum DE17 (Eingang durch DE16)**

Hinweis: Den **Zug 2** betreut Herr Steppat.

Wie bekommt man Noten für das Fach MB2-PR2, Zug 1?

Für dieses Fach bekommen Sie bis zu **vier** Noten:
2 inoffizielle Noten: **ÜNd** (Übungsnote differenziert) und **KNd** (Klausurnote differenziert) und
2 offizielle Noten: **ÜNu** (Übungsnote undifferenziert) und **MNd** (Modulnote differenziert).

Dabei bedeutet:
offiziell: Die Note erscheint auf Ihrer Studiendokumentation (SD)
inoffiziell: Die Note dient nur zur Berechnung anderer Noten, erscheint aber nicht auf ihrer SD.
differenziert: einer der 11 Werte **1,0**, **1,3**, **1,7**, **2,0**, **2,3**, **2,7, 3,0**, **3,3**, **3,7**, **4,0**, **5,0**
undifferenziert: einer der 2 Werte **m.E.** ("mit Erfolg"), **o.E.** ("ohne Erfolg").

Im Laufe des Semesters werden 13 kleine **Tests** geschrieben, in denen Sie **Pluspunkte** bekommen können. Außerdem sollen Sie im Laufe des Semesters (als Mitglied einer kleinen Arbeitsgruppe) 10 **Aufgaben** lösen. Wenn Ihre Arbeitsgruppe eine Lösung für eine Aufgabe unpünktlich vorführt, bekommen Sie dafür **Minuspunkte**. Aus diesen Plus- und Minuspunkten wird die differenzierte Übungsnote **ÜNd** berechnet. Und aus der differenzierten Übungsnote **ÜNd** ergibt sich auf naheliegende Weise die offizielle undifferenzierte Übungsnote **ÜNu** (die Werte 1,0 bis 4,0 ergeben **m.E.**, der Wert 5,0 ergibt **o.E.**)

Falls Ihre Übungsnote **ÜNu** gleich **o.E.** ist, dürfen Sie an keiner Klausur teilnehmen und Ihre Modulnote **MNd** ist automatisch gleich **5,0** (d.h. sie sind durchgefallen).

Sonst dürfen Sie an der Hauptklausur und/oder der Nachklausur teilnehmen und bekommen dafür eine (inoffizielle, differenzierte) Klausurnote **KNd**.

Falls Sie in keiner der beiden Klausuren eine mindestens ausreichende Note erreichen (d.h. **4,0** oder besser) ist Ihre Modulnote **MNd** gleich **5,0**.

Andernfalls (wenn Sie eine Klausur bestanden haben und Ihre Übungsnote **ÜNu** gleich **m.E.** ist) wird Ihre Modulnote **MNd** nach folgender Formel berechnet: **MNd** = (**ÜNd** + 3 * **KNd**) / 4. Das bedeutet: In die Modulnote **MNd** geht die Übungsnote **ÜNd** mit 25% und die Klausurnote **KNd** mit 75% ein.

Ergänzende Erläuterungen zu den Noten-Regeln:

1. Die 13 kleinen Tests werden jeweils dienstags ab 8 Uhr geschrieben. Bearbeitungszeit ca. 10 Minuten. In jedem Test können Sie bis zu 10 **Pluspunkte** bekommen. Mit 5 oder mehr Punkten haben Sie den Test *bestanden*, mit weniger als 5 Punkten haben Sie *nicht bestanden*.

2. Wenn Sie weniger als 10 der 13 Tests (also weniger als 77% der Tests) bestehen, bekommen Sie **o.E.** als Übungsnote **ÜNu** und eine **5,0** als Modulnote **MNd** (d.h. sie sind durchgefallen).

3. Wenn Sie an einem Test nicht teilnehmen, haben Sie ihn nicht bestanden. *Warum* Sie nicht teilgenommen haben (war krank, meine S-Bahn fuhr nicht, ...) spielt dabei *keine Rolle*. Sie brauchen von den 13 Tests nur 10 zu bestehen, damit Sie auch mal krank sein oder von der S-Bahn im Stich gelassen werden können, ohne gleich durchzufallen.

4. Zu jedem Test dürfen Sie als Unterlage 1 beliebig beschriebenes Blatt (maximale Größe: DIN A4) mitbringen. Zu den Klausuren dürfen Sie als Unterlage 5 beliebig beschriebene Blätter (maximale Größe: DIN A4) mitbringen. "Beliebig beschriftet" bedeutet: Nur auf einer Seite oder auf beiden Seiten, von Hand oder mit einem Drucker beschrieben, schwarz-weiß oder in Farbe, ...

5. In der ersten Übung sollen Sie eine kleine **Arbeitsgruppe** bilden und einen Namen für Ihre Gruppe festlegen (z.B. Los Olvidados, Gummiadler, ABC, Heroes oder so ähnlich). Normalerweise besteht eine Arbeitsgruppe aus **zwei Personen**. **Dreier-Gruppen** können beantragt und genehmigt (oder abgelehnt) werden. **Einer-Gruppen** sind nicht zulässig.

6. Jede Arbeitsgruppe soll im Laufe des Semesters **10 Aufgaben** bearbeiten und ihre Lösungen während eines bestimmten Übungstermins vorführen (siehe oben S. 2). Bei der Vorführung einer Lösung müssen alle Gruppen-Mitglieder anwesend und bereit sein, Fragen zu der Lösung zu beantworten.

7. Führt eine Arbeitsgruppe eine Lösung n Wochen *verspätet* vor,
so bekommt jedes Mitglied dafür 2*n **Minuspunkte**.

8. Wenn eine Arbeitsgruppe bis zum **Di 24.01.2017** nicht für jede der 10 Aufgaben eine akzeptable Lösung vorgeführt hat, bekommen alle Mitglieder dieser Gruppe **o.E.** als Übungsnote **ÜNu** und eine **5,0** als Modulnote **MNd** (d.h. sie sind durchgefallen).

9. Die (inoffizielle, differenzierte) Übungsnote **ÜNd** wird aus den **Pluspunkten** Ihrer 10 besten Tests, abzüglich der **Minuspunkte** Ihrer Arbeitsgruppe ermittelt (mit der **PN-Tabelle**, siehe nächsten Punkt).

10. In der Hauptklausur am Ende des WS16/17 bzw. in der Nachklausur kurz vor Beginn des SS17 sind jeweils 100 Punkte erreichbar. Aus Ihrer Punktzahl wird nach der folgenden **PN-Tabelle** eine differenzierte Note ermittelt:

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

11. Bei der Berechnung der Modulnote **MNd** aus der Übungsnote **ÜNd** und einer Klausurnote **KNd** wird in "Zweifelsfällen" zu Ihren Gunsten gerundet, etwa wie im letzten der folgenden Beispiele:

Übungsnote ÜNd	**Klausurnote KNd**	**Rohergebnis**	**Modulnote MNd**
2,3	2,7	(2,3 + 3 * 2,7) / 4 = 2,60	2,7
3,7	2	(3,7 + 3 * 2,0) / 4 = 2,425	2,3
2,0	4,0	(2,0 + 3 * 4,0) / 4 = 3,50	3,3

Hinweis auf die Zukunft: Ab dem SS17 (leider noch nicht im WS16/17) gilt (für den *Bachelorstudiengang Medieninformatik*) eine neue *Prüfungs- und Studienordnung*. Die erlaubt es, Regeln für die Notenvergabe festzulegen die (meiner Ansicht nach, U.G.) einfacher und besser sind als die hier beschriebenen Regeln.

1. SU, Di 04.10.2016

Heute haben wir den Test-01 geschrieben (als "virtuellen" Test, alle bekommen 10 Punkte)

Begrüßung

Kurzfassung der Noten-Regeln:
Wir schreiben 13 kleine Tests. Pro Test kann man 10 Pluspunkte bekommen.
Sie müssen mindestens 10 davon bestehen (mit 5 oder mehr Punkten).
In der ersten Übung (heute noch) sollen Sie Arbeitsgruppen (2 oder 3 Personen) bilden.
Jede Arbeitsgruppe soll in diesem Semester 10 Aufgaben lösen.
Für verspätete Vorführung einer Lösung gibt es Minuspunkte.
Wenn eine Gruppe nicht alle Aufgaben löst, sind alle Mitglieder durchgefallen.

Bitte schicken Sie mir eine Email mit Ihrem Namen darin und einem Foto von sich (damit ich Ihre Namen lernen kann). Ausgenommen sind alle, die im 1. Semester bei mir waren.

Verlosung der Plätze in der Übungsgruppe 1a

1. Feststellen: Wer von den Anwesenden möchte in die Übungsgruppe 1a?
(Ich lese die Namen von Zetteln ab und fertige evtl. zusätzliche Zettel mit BewerberInnen an)
2. Aus den Zetteln aller anwesenden BewerberInnen werden 22 zufällig ausgewählt
(am besten von einer Bewerberin für die Übungsgruppe 1b) .
3. Wer keinen Platz in der Übungsgruppe 1a hat, gehört automatisch zu 1b.

Hat jemand noch eine dringende Frage zur Organisation dieser LV?

Dann geht es jetzt los mit dem Stoff dieser LV

Rekursion

Wie kann man (in einem Java-Programm) dem Ausführer befehlen,
eine bestimmte Befehlsfolge *mehrmals* auszuführen?

Aufgabe-01: Schreiben Sie eine Methode entsprechend der folgenden Spezifikation:

```
void mehrmals(int anz) {
   // Führt den Befehl pln("Hallo!"); anz Mal aus.
```

Lösung-01s: Eine Lösung mit einer Schleife:

```
void mehrmalsS(int anz) {
   // Führt den Befehl pln("Hallo!"); anz Mal aus.

   for (int i=1; i<=anz; i++) pln("Hallo!");
}
```

Frage: Was bewirken die folgenden Aufrufe der Methode `mehrmalsS`?

```
mehrmalsS(3);   // Gibt 3 Mal "Hallo!" aus
mehrmalsS(1);   // Gibt 1 Mal "Hallo!" aus
mehrmalsS(0);   // nix
mehrmalsS(-1);  // nix
mehrmalsS(-17); // nix
```

Schon lange bevor es Computer, Programmiersprachen und Schleifen-Befehle gab, haben Mathematiker "Wiederholungen" auf eine *ganz andere Weise* ausgedrückt. Eine Lösung für die **Aufgabe-01** kann "auf diese andere Weise" etwa so aussehen:

Lösung-01r: Eine Lösung mit Rekursion

```
void mehrmalsR(int anz) {
   // Führt den Befehl pln("Hallo!"); anz Mal aus.
```

```
   if (anz < 1) return;
   pln("Hallo!");
   mehrmalsR(anz-1);
}
```

Man sieht: Der Rumpf der Methode besteht aus einer *Fallunterscheidung*.
In diesem Beispiel werden 2 Fälle unterschieden:
Fall 1: `anz` ist kleiner als 1
Fall 2: `anz` ist nicht kleiner als 1

Den **Fall 2** bezeichnet man auch als einen *rekursiven Fall*, weil die Methode sich in diesem Fall selbst aufruft ("rekursiv" bedeutet in etwa "auf sich selbst zurückgreifend").

Den **Fall 1** bezeichnen wir als einen *einfachen Fall*, weil die Methode sich in diesem Fall *nicht* selbst aufruft.

Regel: Der Rumpf einer rekursiven Methode muss immer eine *Fallunterscheidung* sein, die mindestens einen *einfachen Fall* und mindestens einen *rekursiven Fall* unterscheidet.

Frage: Warum muss eine rekursive Methode immer mindestens einen *einfachen Fall* enthalten? Was würde passieren, wenn man in der Methode `mehrmalsR` den einfachen Fall wegließe?

Frage: Warum muss eine rekursive Methode immer mindestens einen *rekursiven Fall* enthalten? Was würde passieren, wenn man in der Methode `mehrmalsR` den rekursiven Fall wegließe?

Tipps zum Schreiben rekursiver Methoden

In diesem Abschnitt sind *Schleifen* grundsätzlich *nicht erlaubt*!

Aufgabe-02: Angenommen wir *haben bereits* die folgende Methode:

```
int f01(int n) {
   // Liefert 1, wenn n gleich 0 oder kleiner ist.
   // Liefert sonst das Produkt 1 * 2 * 3 * ... * n

   if (n<=0) return 1;
   ...    // Diese Befehle werden erst später verraten
}
```

Vereinbaren Sie (unter dieser Annahme) eine Methode namens `f02`, die genau das Gleiche leistet wie `f01`.

Lösung-02:

```
int f02(int n) {
   return f01(n);
}
```

OK, das war zu einfach und hat Sie wahrscheinlich unterfordert.
Deshalb führen wir jetzt noch eine kleine "Zusatzbedingung" ein:

Aufgabe-03: Vereinbaren Sie eine Methode namens `f02`, die genau das Gleiche leistet wie `f01`.
Im Rumpf von `f02` sind Methodenaufrufe wie `f01(3)` oder `f01(-17)` oder `f01(n/2)` oder `f01(3*n+5)` ... etc. *erlaubt*,
nur der Aufruf **`f01(n)`** ist *verboten*.
(Zu `f01(n)` *gleichwertige* Aufrufe wie `f01(n+1-1)` oder `f01(5*n/5)` etc. sind natürlich auch verboten).

Lösung-03:

```
int f02(int n) {
   if (n<=0) return 1;
   return f01(n-1) * n;
}
```

Diese Lösung ist eine "ganz normale" Funktion (das soll heißen: keine rekursive Funktion).

Vergewissern Sie sich, dass diese Methode `f02` wirklich für alle möglichen Argumente genau das Gleiche leistet wie `f01`, indem Sie die folgende Tabelle ausfüllen:

`n`	-1	0	1	2	3	4	5
`f02(n)`							

Da `f02` genau das Gleiche leistet wie `f01`, können wir im Rumpf von `f02` den Aufruf `f01(n-1)` durch den Aufruf `f02(n-1)` ersetzen. Damit erhalten wir:

```
int f02(int n) {
   if (n<=0) return 1;
   return f02(n-1) * n;
}
```

Diese Lösung ist eine rekursive Funktion.

Tipp: Wenn man eine rekursive Methode `m` schreibt sollte man
1. ganz genau wissen, was `m` leisten soll und
2. sich einreden (oder: daran glauben), dass man die Methode `m` schon hat und aufrufen darf.

Unterschied zwischen abstrakten Zahlen und konkreten Darstellungen von Zahlen

Welche der folgenden Eigenschaften betreffen *abstrakte Zahlen* (A) bzw. Darstellungen (D)?

Endet mit der Ziffer 3? // D
Ist negativ. // A
Ist größer als siebzehn? // A
Besteht aus sieben Ziffern? // D
Ist ein Primzahl? // A
Hat die Quersumme acht? // D

Die Werte eines Ganzzahltyps wie z.B. `int` sollte man sich als **abstrakte Zahlen** vorstellen, denn einen `int`-Wert kann man z.B. als 10-er-Zahl (oder: Dezimalzahl) oder als 2-er-Zahl (oder: Binärzahl) oder als 17-er-Zahl (kein oder) oder als römische Zahl etc. darstellen.

Sei `n` eine `int`-Variable (die irgendeinen `int`-Wert enthält). Dann gilt:

Der Ausdruck `n%10` bezeichnet die letzte Ziffer in der Darstellung von `n` als 10-er-Zahl
Der Ausdruck `n%2` bezeichnet die letzte Ziffer in der Darstellung von `n` als 2-er-Zahl
Der Ausdruck `n%17` bezeichnet die letzte Ziffer in der Darstellung von `n` als 17-er-Zahl
... etc.

Der Ausdruck `n/10` bezeichnet den Wert von `n` ohne die letzte 10-er-Ziffer
Der Ausdruck `n/2` bezeichnet den Wert von `n` ohne die letzte 2-er-Ziffer
Der Ausdruck `n/17` bezeichnet den Wert von `n` ohne die letzte 17-er-Ziffer

Beispiele:

`123%10` ist gleich `3`, `123/10` ist gleich `12`.

`5%2` ist gleich `1`, `5/2` ist gleich `2`.

`37%17` ist gleich `3`, `37/17` ist gleich `2`.

2. SU, Di 18.10.2016

Heute schreiben wir den Test-02

(den Test-01 haben wir im 1. SU als "virtuellen Test" geschrieben)

Kleine Anmerkungen zu Methoden

Angenommen, eine Methode hat einen numerischen Parameter, z.B. vom Typ `int`, wir wollen aber nur den *Betrag* des Parameters bearbeiten ("ein evtl. Minuszeichen soll beseitigt werden"). Wie kann man das machen?

```
int m37(int n) {
   if (n<0) n = n * -1;  // Ganz schlecht!
   n = Math.abs(n);      // Besser
   if (n<0) n = -n;      // Noch besser
   ...
```

Diese Zuweisungen an `n` sind aber nicht erlaubt, wenn `n` unveränderbar (in Java: `final`) ist. In einem solchen Fall hilft manchmal eine "pseudo-Rekursion", etwa so:

```
int m38(final int N) {
   if (N<0) return m38(-N);
   ...
```

Was bedeutet der folgende Anfangskommentar?

```
int m39(final int N) {
   // Liefert die kleinste Primzahl, die groesser als N ist ...
```

Wer liefert hier? Und was bedeutet "liefern" (engl: to yield)?

Was bedeutet der folgende Anfangskommentar?

```
int m40(final int N) {
   // Verlaesst sich darauf, dass N positiv ist
```

oder

```
   // Verlaesst sich darauf, dass N nicht negativ ist
```

Wer "verlässt sich"? Und was ist der Unterschied zwischen "positiv" und "nicht negativ"?

Merke: Die Zahl 0 ist weder positiv noch negativ (zumindest "unter Mathematikern")

"Mit Rekursion" ist oft viel einfacher als "ohne Rekursion"

Die Idee der Rekursion wird schon sehr lange benutzt, z.B. in einem Mathe-Buch, welches im Jahr 1202 in Italien erschienen ist.

Papier **pr2_RekursionAufgaben16WS.odt** öffnen, **S. 2**.

Definition der Fibonacci-Zahlen mit Rekursion und ohne Rekursion.

S. 4 unten: **Regeln zur Bearbeitung von Ziffern** (Wiederholung, hatten wir schon im SU 1)

Zur Erinnerung:

Was ist mit der **Stellung** und dem **Stellenwert** einer Ziffer innerhalb einer Zahl gemeint?

An welchen Stellen stehen die einzelnen Ziffern der folgenden Zahl: **7358**

Die 8 steht an der Stelle 0, die 5 an der stelle 1, die 3 an der Stelle 2 und die 7 an der Stelle 3

Angenommen, **7358** ist eine 10-er-Zahl. Welche Stellenwerte haben dann die einzelnen Ziffern?

Die 8 hat den Stellenwert 10^0 (gleich 1), die 5 hat den Stellenwert 10^1 (gleich10),
die 3 hat den Stellenwert 10^2 (gleich 100), die 7 hat den Stellenwert 10^3 (gleich 1000).

Angenommen, **7358** ist eine 16-Zahl, welche Stellwerte haben dann die einzelnen Ziffern?

Bearbeiten Sie jetzt (möglichst in Ihren Arbeitsgruppen zu 2 oder 3 Personen) die **Aufgabe-211**, **-212** etc. (ab S. 5 des Papiers **pr2_RekursionAufgaben16WS.odt**).

Zur Entspannung: **Warum ich "Reihung" besser finde als "Array"**

Welche Sätze sind richtig?
1. "Der Array wurde in Zeile 13 vereinbart. Er enthält 5 Werte."
2. "Das Array wurde in Zeile 13 vereinbart. Es enthält 5 Werte."
3. "Die Array wurde in Zeile 13 vereinbart. Sie enthält 5 Werte."

Laut Online-Duden, Wiktionary und anderen Stellen im Internet sind 1. und 2. richtig, 3. ist falsch. Viele Sätze, die das Wort "Array" enthalten, klingen in den Ohren einiger Hörer "schief" (weil diese anderen Hörer das *eine* Geschlecht nicht richtig finden, weil sie das *andere* gewohnt sind).

Das Wort "Reihung" hat im Deutschen ein eindeutiges Geschlecht (**die Reihung** ist weiblich). Es wird z.B. in der **DIN-Norm 66 268, 1988** verwendet, einer deutschen Version des (englischen) Reference Manuals der Sprache Ada.

3. SU, Di 25.10.2016

Heute schreiben wir den Test-03

Organisation

Im SWE-Labor können Sie auch *drucken*, aber nicht ganze Bücher oder Papiere mit 50 oder 100 Seiten. Erledigen Sie umfangreiche Ausdrucke zu Hause oder in einem Copyshop.

Zufall in Java

Typische Anwendungen von Zufallszahlen:
- Ein Würfelspiel programmieren
- Umfangreiche Testdaten (z.B. 100 Tausend `int`-Werte) erzeugen

In diesen beiden Fällen braucht man aber *verschiedene Arten* von Zufall.

```
Random ralf = new Random(123L); // mit  Keim
Random niko = new Random();     // ohne Keim
for (int i=1; i<10; i++) {
   p(ralf.nextInt() + " ");
   p(niko.nextInt() + " ");
}
```

Wenn man diese Befehlsfolge z.B. 5 Mal ausführen lässt, wird die "Folge der `ralf`-Zahlen" immer die gleiche sein, die "Folge der `niko`-Zahlen" wird aber (sehr, sehr wahrscheinlich) bei jeder Ausführung eine andere sein.

Fachbegriffe: *Reproduzierbarer* Zufall (`ralf`) und *nicht reproduzierbarer* Zufall (`niko`).

Bei `ralf` hat der *Programmierer* einen bestimmten Keim angegeben (`123L`).
Bei `niko` berechnet der *Ausführer* einen Keim, indem er die Methode `System.nanoTime()` aufruft.

Wie stellt man **Bereiche** (oder: **Intervalle**) von Zahlen dar? Beispiele:

[10, 50] // einschließlich 10 und 50, abgeschlossen
(10, 50) // ausschließlich 10 und 50, offen
[10, 50) // einschließlich 10, ausschließlich 50, halboffen, gleich [10, 49] gleich (9, 50)
(10, 50] // ausschließlich 10, einschließlich 50, halboffen, gleich [11, 50] gleich (10, 51)

Diese Notationen sind vor allem bei Bruchzahlen sehr nützlich, z.B. [0.0, 1.0) etc.

Ein `Random`-Objekt (wie z.B. `ralf` oder `niko`) enthält mehrere Methoden, die zufällige Werte liefern. Diese Methoden unterscheiden sich vor allem durch den *Typ*, den *Bereich* und die *Verteilung* ihrer Ergebnisse.

Öffnen Sie das Papier **pr2_Zufall.odt**, S. 1, der Kasten etwa in der Mitte der Seite.

S. 2, Abschnitt **4. Methoden eines Random-Objekts**

Aufgabe-01: Vereinbaren Sie eine `int`-Variable namens `ilka` mit einer zufälligen Zahl aus dem Bereich `[-100, +100]` als Anfangswert. Benutzen Sie dazu das `Random`-Objekt `ralf`.

Lösung-01:

```
int ilka = ralf.nextInt(201) - 100;
```

Die Klasse PoxelPanel (im Anhang der Datei pr2_Aufgaben16WS.odt, ca. S.16)

1. Wo werden die `int`-Variablen `anzPoxX` und `anzPoxY` vereinbart?
(ca. 5 Zeilen unter der Zeile `public class PoxelPanel ...`)
Wo werden diese beiden Variablen initialisiert? (im Konstruktor, S. 16 unten)

2. Von welchem Typ ist das Attribut `fTab`? Und `zTab`?
(Reihung von Reihungen von `Color`, Reihung von Reihungen von `char`)

3. Was bedeutet die Zahl `this.anzPoxX*anzPixX`? Und `this.anzPoxY*anzPixX`?
(die Größe des aktuellen `PoxelPanel`-Objekts in Pixeln, in x-Richtung bzw. in y-Richtung)

4. In der Methode `getColor`: Was macht der Befehl `x = Math.abs(x) % anzPoxX`?
(Falls der Wert von x keine geeignete Koordinate für das aktuelle `PoxelPanel` ist, wird er durch eine geeignete Koordinate ersetzt)
Ebenso für das Attribut y.

5. Die Methode `drawRect`: Wie viele und was für Parameter hat sie? Was bewirkt sie?

6. Die Methode `paint`: Ganz besonders: Wird vom Programmierer vereinbart, aber nicht aufgerufen! Wird nur vom Ausführer aufgerufen und ausgeführt, "wenn er das für richtig hält"!

Erläutern, wann der Ausführer "das für richtig hält".

Der Parameter `g` vom Typ `Graphics` "stammt vom Ausführer bzw. vom Betriebssystem" (nicht vom Programmierer) und repräsentiert ein bestimmtes Rechteck auf dem Bildschirm, in das die Methode `paint` zeichnen darf.

Das war der Bauplan-Aspekt der Klasse `PoxelPanel`, oberhalb der folgenden Zeile:

```
// ======================================================================
```

Unterhalb dieser Zeile kommen noch ein paar Klassen-Elemente (static members), die zum "Ausprobieren und Testen" der Klasse dienen:

Die Methode `macheJF` "macht" ein Fenster (vom Typ `JFrame`)
Die Reihung `cTab` enthält 29 Farben
Die `main`-Methode erzeugt ein `PoxelPanel` namens `pop` und ein Fenster `ram` und tut das Objekt `pop` in das Fenster `ram`.

Zur Entspannung: **Wie man mit Zufallszahlen die Kreiszahl π berechnen kann**

Öffnen Sie das Papier **pr2_PiMitZufall.odt**.

4. SU, Di 01.11.2016

Heute schreiben wir den Test-04

Bestimmte Befehle vereinfachen

```
static boolean wenigerGut11(int n) {
   if (n>0) {
      return true;
   } else {
      return false;
   }
}
```

Wie kann man den Rumpf dieser Methode vereinfachen?

```
static boolean besser11(int n) {
   return n>0;
}
```

Schreiben Sie eine vereinfachte Version der folgenden Methode:

```
static boolean wenigerGut12(int n) {
   if (n>0) {
      return false;
   } else {
      return true;
   }
}
```

Regel: In `boolean`-Funktionen kann (und sollte) man häufig auf folgende Dinge verzichten:

`if`-Anweisungen, die Literale `true` und `false`.

Öffnen Sie das Papier **pr2_EinfachSparsamSchoen.odt**, S. 1
Sehen Sie sich die Methoden `besser12a` und `besser12b` an.

Wichtige Regeln für `boolean`-Ausdrücke mit `&&, ||` und `!` (und, oder und nicht)

De Morgansche Gesetze:

`!(A && B)` ist gleich `!A || !B`
`!(A || B)` ist gleich `!A && !B`

Gelten ganz entsprechend auch für `(A && B && C && ...)` bzw. `(A || B || C || ...)`

Schnittstellen (engl. interfaces)

Eine Java-Schnittstelle besteht "in erster Linie und hauptsächlich" aus *abstrakten Methoden*. Eine abstrakte Methode hat keinen Rumpf, nur ihr *Rückgabe-Typ*, ihr *Name* und die *Typen ihrer Parameter* (0 oder mehr) sind festgelegt. Außerdem sollte eine abstrakte Methode einen *Anfangskommentar* haben, der genau beschreibt, was sie machen soll.

In einigen Programmiersprachen kann eine Klasse *mehrere* Klassen beerben (vor allem in C++). Das hat sich als ziemlich gefährlich herausgestellt. Deshalb kann (und muss) in Java jede Klasse nur *eine* Klasse beerben (in C++ gibt es mehrfache Beerbung, in Java gibt es nur einfache Beerbung).

Eine Klasse darf aber beliebig viele (0 oder mehr) *Schnittstellen implementieren*, d.h. Rümpfe für die abstrakten Methoden der Schnittstelle(n) festlegen. Man kann Schnittstellen deshalb als "Trostpflaster für die fehlende mehrfache Beerbung" verstehen.

Außerdem sind Schnittstellen *Verträge zwischen Programmierern*. Schnittstellen sind (in aller Regel) viel leichter zu schreiben als Klassen und müssen nicht getestet werden. Am Anfang eines größeren Projekts legt man deshalb häufig für alle Klassen, die von mehreren Programmierern benutzt werden sollen, Schnittstellen fest. Die **Aufgabe-04** ist ein winziges Beispiel dafür.

Als Typ eines Methoden-Parameters p hat eine Schnittstellen S im Vergleich zu einer Klasse K den folgenden Vorteil: In einem Aufruf der Methode kann man als Argument für p nicht nur Objekte *einer einzigen* Klasse angeben, sondern Objekte aller Klassen die die Schnittstelle S implementieren.

Aktionen, Ereignisse, Listener-Schnittstellen und Adapter-Klassen (Buch S. 546)

Aktionen: Der Benutzer klickt auf einen Knopf, oder bewegt die Maus innerhalb eines Fensters, oder drückt auf eine Taste der Tastatur während ein JTextField den Fokus hat, ... etc.

Ereignis: Daraufhin erzeugt das betreffende Grabo-Objekt (der Knopf bzw. das Fenster bzw. das JTextField-Objekt bzw. .... etc.) ein Ereignis.

Ein Ereignis findet zu einem bestimmten Zeitpunkt an einem bestimmten Ort statt und ist grundsätzlich *nicht wiederholbar*.

Arten von Ereignissen: Jedes Ereignis gehört zu einer bestimmen *Art von Ereignis*, z.B. zur Art `actionPerforemed` oder zur Art `mouseMoved` oder zur Art `keyPressed` oder ... .

Oberarten von Ereignissen: Jede Art von Ereignis gehört zu einer *Oberart von Ereignis* , z.B. zur Oberart Aktionsereignis oder zur Oberart Mausereignis oder zur Oberart Tastatur-Ereignis oder ...

Buch S. 547, die Tabelle ansehen.

Oberarten von Grabo-Ereignissen	**Arten** von Grabo-Ereignissen	**Schnittstelle** für die Oberart	**Adapterklasse** für die Schnittstelle

Fensterereignis	`windowOpened`
	`windowClosing`
	`windowClosed`
	`windowIconified`
	`windowDeiconified`
	`windowActivated`
	`windowDeactivated`
		`WindowListener`	`WindowAdapter`
			`(7 Strohpuppen)`

Mausereignis	`mouseClicked`
	`mousePressed`
	`mouseReleased`
	`mouseEntered`
	`mouseExited`
		`MouseListener`	`MouseAdapter`
			`(5 Strohpuppen)`

Mausbewegungsereignis	`mouseDragged` `mouseMoved`	`MouseMotionListener`	`MouseMotionAdapter` `(2 Strohpuppen)`
Mausradereignis	`mouseWheelMoved`	`MouseWheelListener`	`--`
Aktionsereignis	`actionPerformed`	`ActionListener`	`--`

Zur Entspannung: **Programmiersprachen-Rankings**

Auf verschiedenen Netzseiten werden Rankings von Programmiersprachen ("Welche Sprache ist wie beliebt?") veröffentlicht. Es folgen hier die Adressen einiger solcher Seiten (jeweils mit den 3 beliebtesten Sprachen von 2015 bzw. Anfang 2016):

Tiobe:	http://www.tiobe.com/tiobe-index//	(Java, C, C++)
RedMonk:	http://redmonk.com/sogrady/2016/02/19/language-rankings-1-16/	(JavaScript, Java, PHP)
PopularitY:	http://pypl.github.io/PYPL.html	(Java, Python, PHP)

5. SU, Di 08.11.2016

Heute schreiben wir den Test-05

Eine wichtige "Konsequenz von Schnittstellen"

Jede normale *Klasse* ist auch ein *Typ* (eine generische Klasse definiert sogar viele Typen).
Jede *Schnittstelle* ist auch ein *Typ*.

Buch S. 341 unten, **Beispiel-01**: Eine Schnittstelle namens `Vergroesserbar`

Denken Sie sich zusätzlich eine ganz ähnliche Schnittstelle namens `Verkleinerbar`, die nur eine abstrakte Methode `void halbiere()` enthält.

Buch S. 343 oben, **Beispiel-03**: Eine Klasse namens `GanzZahl02` implementiert zwei Schnittstellen.

Aus der Vereinbarung der Klasse folgt: Jedes `GanzZahl02`-Objekt gehört zu **4 Typen**:

1. Zum Typ `GanzZahl02`
(sonst wäre es kein `GanzZahl02`-Objekt :-))

2. Zum Typ `Vergroesserbar`
(weil die Klasse `GanzZahl02` die Schnittstelle `Vergroesserbar` implementiert)
3. Zum Typ `Verkleinerbar`
(weil die Klasse `GanzZahl02` die Schnittstelle `Verkleinerbar` implementiert)
4. Zum Typ `Object`
(weil die Klasse `GanzZahl02` eine Unterklasse der Klasse `Object` ist)

Die folgenden Variablen-Vereinbarungen sind also erlaubt:

```
GanzZahl02     anna = new GanzZahl02(10);
Vergroesserbar bert = new GanzZahl02(20);
Verkleinerbar  carl = new GanzZahl02(30);
Object         dora = new GanzZahl02(40);
```

Welche Methoden in den vier Modulen (oder: Objekten) `anna` bis `dora` kann man aufrufen?

`anna.verdopple, anna.verdreifache, anna. halbiere, anna.toString`
`bert.verdopple, bert.verdreifache`
`carl.halbiere`
`dora`? Keine der hier interessanten Methoden kann aufgerufen werden

Fachbegriffe

Sei **S** eine **Schnittstelle**.
Dann ist eine **S-Klasse** eine Klasse, die S implementiert.
Und ein **S-Objekt** ist ein Objekt einer S-Klasse.

Zur Aufgabe-04 (Ein Rechner nach dem MVC-Muster strukturiert)

Die Struktur eines Rechners (d.h. einer Lösung für Aufgabe-04)

Ein Rechner besteht aus 3 Objekten:
einem `RechnerControl`-Objekt (kurz: `Control`-Objekt),
einem `RechnerView_I`-Objekt (kurz: `View`-Objekt),
einem `RechnerModel_I`-Objekt (kurz: `Model`-Objekt).

Die Klasse `RechnerControl` ist vollständig vorgegeben.
Sie müssen
1. eine `RechnerView_I`-Klasse vereinbaren.
2. eine `RechnerModel_I`-Klasse vereinbaren.
3. eine Hauptklasse ("eine mit `main`-Methode") vereinbaren und in der `main`-Methode

ein Objekt Ihrer `RechnerView_I`-Klasse, ein Objekt Ihrer `RechnerModel_I`-Klasse und ein `RechnerControl`-Objekt erzeugen lassen.

Öffnen Sie die Datei **pr2_Aufgaben.odt**, **S. 35** (die vorgegebene Klasse `RechnerControl`)

Beschreiben Sie (kurz) alle Dinge, die in der Klasse `RechnerControl` vereinbart sind:

1. Ein Klassen-Attribut `TST` (zum Ein-/Ausschalten von Test-Befehlen).
2. Ein privates Objekt-Attribut `aModel`
3. Ein privates Objekt-Attribut `aView`
4. Ein öffentlicher Konstruktor `RechnerControl`
5. Eine private Objekt-Klasse (non-static class) `MyActionListener`
6. Eine private Objekt-Klasse (non-static class) `MyKeyListener`
7. Eine Klassen-Methode `printf` ("als Abkürzung")

Wenn der Programmierer ein `Control`-Objekt erzeugen lässt (mit `new`), dann muss er dem Konstruktor ein `Model`-Objekt und ein `View`-Objekt übergeben. D.h. das `Control`-Objekt kann auf die anderen beiden Objekte zugreifen, aber die anderen beiden Objekte sind "sich gegenseitig unbekannt" und "kennen auch nicht das `Control`-Objekt".

Die zwei wichtigsten Grabo-Klassen: Component und Container

Buch S. 544, Bild 22.4

Grabo-Klassen: Die Klasse `java.awt.Component` und alle ihre Unterklassen
Behälter-Klasse: Die Klasse `java.awt.Container` und alle ihre Unterklassen

Ein Behälter-Objekt (kurz: **Behälter**, engl. **container**) ist
ein Grabo-Objekt, in das man Grabo-Objekte hineintun kann.

Beispiel-01: In ein **JFrame**-Objekt kann man u.a. **Label**-, **Button**- und **Box**-Objekte hineintun.

Beispiel-02: In ein **Box**-Objekt kann man u.a. **Label**-, **Button**- und **Box**-Objekte hineintun.

Behälter-Objekte kann man *schachteln* (d.h. man kann "Behälter in Behälter hineintun",
z.B. "Box in JFrame" oder "Box in Box" etc.).

Sichtbare und unsichtbare Behälter

`JFrame`-Objekte sind Behälter und sind auf dem Bildschirm *sichtbar*.
`Box`-Objekte sind Behälter, selbst aber *nicht sichtbar*.
Es gibt 2 Arten von Box-Objekten:
Horizontale Box-Objekte (oder X-Box-Objekte), werden von *links* nach nach *rechts* "befüllt"
Vertikale Box-Objekte (oder Y-Box-Objekte), werden von *oben* nach *unten* "befüllt"

Zurück zum MVC-Rechner (Aufgab-04)

Öffnen Sie die Datei **pr2_Aufgabe04-Bauplan.odt**, S. 1

Wie soll ein Rechner (genauer: ein `RechnerControl`-Objekte) auf dem Bildschirm aussehen?
Der *Bauplan* eines Rechners zeigt auch die *unsichtbaren Box-Objekte*.

Im Konstruktor der `Control`-Klasse werden drei Behandler-Objekte (Objekte die eine Behandler-Methode enthalten) namens `mal`, `mkl1` und `mkl2` vereinbart:

`mal` soll `actionPerformed`-Ereignisse des `JButton`-Objekts `submit` behandeln
`mkl1` soll `keyReleased`-Ereignisse des 1. `JTextField`-Objekts behandeln
`mkl2` soll `keyReleased`-Ereignisse des 2. `JTextField`-Objekts behandeln

Im Behandler-Objekt `mal` (wie "my action listener") werden die Methoden `calculate` (im `Model`-Objekt) und `setResults` (im `View`-Objekt) aufgerufen.
In den Objekten `mkl1` und `mkl2` (wie "my key listener 1 bzw. 2") wird die Methode `isValidEntry` (im `Model`-Objekt) aufgerufen.

Nur in der Klasse `Control` sind alle drei dieser Methoden (`calculate`, `setResultView` und `isValidEntry`) sichtbar. Deshalb müssen die drei Behandler-Objekte `mal`, `mkl1` und `mkl2` in der `Control`-Klasse *vereinbart* werden (und können nicht in der `View`-Klasse vereinbart werden, denn dort sind `calculate` und `isValidEntry` nicht sichtbar).

Andererseits müssen die drei Behandler-Objekte im `View`-Objekt bei den entsprechenden Grabo-Objekten angemeldet werden (mit den Methoden `addActionListener` bzw. `addKeyListener`). Das wird von der Methode `register` erledigt, die im `View`-Objekt *vereinbart* werden muss (siehe Schnittstelle `RechnerView-I`) und im Konstruktor der `Control`-Klasse *aufgerufen* wird.

Klasse Control
Konstruktor Control
erzeugt 3 Behandler-Objekte **mal**, **mkl1**, **mkl1**, in denen die Methoden **calculate**, **setResults** und **isValidEntry** aufgerufen werden und
ruft mit den 3 Objekten die Methode **register** auf.

Ein **View-Objekt** enthält u.a.
Methode **setResults**
Attribut **JTextField 1**
Attribut **JTextField 2**
Attribut **JButton submit**
Methode **register**

Ein **Model-Objekt** enthält u.a.
Methode **isValidEntry**
Methode **calculate**

6. SU, Di 15.11.2016

Heute schreiben wir den Test-06

Behandler-Methoden, Behandler-Objekte und Behandler-Klassen

Was ist eine **Aktion**? Beschreiben Sie typische Beispiele!
(Der **Benutzer** klickt auf einen Knopf oder auf einen Menüpunkt, drückt auf eine Taste der Tastatur, bewegt die Maus, ...)

Eine Aktion betrifft immer ein Grabo-Objekt (einen Knopf, ein Fenster, einen Menüpunkt und häufig) *das Objekt welches gerade den Focus* hat).

Was wird durch eine Aktion (des Benutzers) von dem betroffenen Grabo-Objekt **erzeugt**?
(Ein **Ereignis**, eng. an event).

Die wichtigsten **Eigenschaften eines Ereignisses**?
(Es findet an einem bestimmten Ort und zu einem bestimmten Zeitpunkt statt. Es ist *nicht wiederholbar*)

Jedes Ereignis gehört zu einer **Art** von Ereignissen. Nennen Sie Beispiele für Ereignis-Arten
(`windowsOpened`, `mouseClicked`, `actionPerformed`, ...)

Beispiel: Wenn man dreimal auf einen Knopf klickt, dann werden dadurch drei verschiedene Ereignisse erzeugt, die aber alle zur selben Art (`actionPerformed`) gehören.

Jede Ereignis-**Art** gehört zu einer **Oberart**. Nennen Sie Beispiele für solche Oberarten.
(Fenster-Ereignis, Maus-Ereignis, Mausbewegungs-Ereignis, Mausrad-Ereignis, Aktions-Ereignis, ...)

Zu welcher Oberart gehört die Ereignisart `windowOpened`? (Fenster-Ereignis)
Zu welcher Oberart gehört die Ereignisart `mouseClicked`? (Maus-Ereignis)
Zu welcher Oberart gehört die Ereignisart `actionPerformed`? (Aktions-Ereignis)

"Ereignisse behandeln", was ist damit gemeint?

Um Ereignisse einer bestimmten Art (z.B. der Art `ActionPerformed`) zu **behandeln**, muss man eine (öffentliche Objekt-) **Prozedur** mit einem bestimmten Namen und einem bestimmten Parameter schreiben. Durch bestimmte Maßnahmen (siehe unten) kann man dann bewirken, dass diese Prozedur immer dann ausgeführt wird, wenn ein Ereignis der betreffenden Art eintritt.

Regel-01:
Die Methode zur Behandlung von `windowOpened`-Ereignissen muss `windowOpened` heißen.
Die Methode zur Behandlung von `mouseClicked`-Ereignissen muss `mosueClicked` heißen.
Die Methode zur Behandlung von `Karl-Heinz`-Ereignissen muss `Karl-Heinz` heißen.
(OK, eine Ereignis-Art `Karl-Heinz` gibt es nicht. Aber wenn es sie gäbe, dann müsste ... :-)

Regel-02:
Wenn man Ereignisse einer bestimmten **Art** behandeln will, muss man auch die Ereignisse aller anderen Arten behandeln, die zur selben **Oberart** gehören.

Beispiel-01: Wenn man `windowOpened`-Ereignisse behandeln will, muss man auch Ereignisse der Arten `windowClosing`, `windowClosed`, ... `windowDeactivated` behandeln
(sieh oben die Tabelle auf **S. 13**).

Beispiel-02: Wenn man `mouseMoved`-Ereignisse behandeln will, muss man auch Ereignisse der Art `mouseDragged` behandeln.

Beispiel-03: Wenn man `actionPerformed`-Ereignisse behandeln will, muss man keine weiteren Arten von Ereignissen behandeln.

Der Fluch der Objektorientierung

In Java kann man eine Methode nur innerhalb einer Klasse vereinbaren, und eine Objekt-Methode gibt es erst, wenn ein Objekt der betreffenden Klasse erzeugt wurde. Das gilt leider auch für Behandler-Methoden, und macht den Umgang mit ihnen ein bisschen umständlich.

Anmerkung: In einigen Programmiersprachen (z.B. in C/C++) sind *Unterprogramme* (Methoden, Funktionen, Prozeduren) "first class citizens", die man z.B. einer Variablen zuweisen, als Argument an ein Unterprogramm übergeben oder in Reihungen speichern kann. In Java sind *Objekte* "first class citizens", Methoden haben dagegen nur "eingeschränkte Bürgerrechte".

Regel-03: Wenn man Ereignisse **behandeln** will, die von einem bestimmten Grabo-Objekt (z.B. von einem Fenster oder von einem Knopf oder ...) erzeugt werden, muss man entsprechende Methoden "in ein Behandler-Objekt einpacken" und dann das Behandler-Objekt bei dem Grabo-Objekt **anmelden**.

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    });
```

Ereignisse behandeln (Fortsetzung)

Öffnen Sie das Papier **pr2_Behandler.odt**. Auf der S.1 sehen Sie eine kleine Grabo, die nur aus einem *Fenster* (vom Typ `JFrame`) und einem *Knopf* (vom Typ `JButton`) besteht.

Wenn man auf diesen Knopf klickt erzeugt er ein `actionPerformed`-Ereignis. Diese Ereignisse sollen behandelt werden. Die Programme `Grabo30` und `Grabo31` (auf den Seiten 2 und 3 des Papiers) demonstrieren 2 verschiedene Weisen wie man ein geeignetes Behandler-Objekt vereinbaren und beim Knopf-Objekt anmelden kann (einmal "mit Namen" und einmal "anonym").

Buch S. 536, **Beispiel-04**, **Grabo02**, mit Namen `Terminator` und `arnold`

Unterschied zwischen den Klassen
`BehandlerKlasse` in **Grabo30** und
`Terminator` in **Grabo02**?

(**Grabo30**: `class BehandlerKlasse implements ActionListner`,
Grabo02: `class Terminator extends WindowAdapter`
weil die Ereignis-Art `windowClosing` zur Oberart Fenster-Ereignis gehört, zu der mehrere Ereignis-Arten gehören (genauer: 7 Ereignis-Arten), siehe Tabelle im **Buch** auf **S. 547**).

Buch S. 538, **Beispiel-05**, **Grabo03**, anonymes Objekt einer anonymen Klasse..

Zur Entspannung: **Prozedurale und funktionale Programmier-Sprachen**

Zur Erinnerung: Es gibt 3 Arten von Befehlen: *Vereinbarungen*, *Ausdrücke*, *Anweisungen.*
Es gibt 2 Arten von Methoden (oder: Unterprogrammen): *Funktionen* und *Prozeduren.*
Ein Aufruf einer *Funktion* ist ein *Ausdruck,* ein Aufruf einer *Prozedur* ist eine *Anweisung.*

Es gibt zwei Arten von Programmiersprachen: *Prozedurale* und *funktionale* Sprachen.
Prozedurale Sprachen enthalten Befehle aller 3 Arten, Prozeduren und Funktionen, veränderbare Variablen und die Zuweisungs-Anweisung. Die meisten weit verbreiteten Sprachen sind prozedural, z.B. Fortran, Cobol, Pascal, C, C++, Ada, Java, JavaScript, PHP etc.

In *funktionalen Sprachen* gibt es nur Vereinbarungen und Ausdrücke, aber *keine Anweisungen* (und deshalb nur Funktionen, keine Prozeduren und keine veränderbaren Variablen und keine Zuweisungs-Anweisung). Außer gewöhnlichen Funktionen (die z.B. Ganzzahlen auf Ganzzahlen abbilden) enthalten funktionale Sprachen auch *höhere Funktionen* (die Funktionen auf Funktionen abbilden). Beispiele für funktionale Sprachen sind Lisp, Erlang, F-Sharp, Miranda, Opal, Scheme, Haskell. Die relativ weit verbreiteten Sprachen SQL, Scala, Python und JavaScript enthalten "erhebliche funktionale Anteile".

7. SU, Di 22.11.2016

Heute schreiben wir den Test-07

Oberarten von Ereignissen und "ihr Anhang"

Was ist jeder *Oberart von Ereignissen* zugeordnet?
Siehe die Tabelle oben auf **S. 13** oder im Buch auf **S. 547**.
(Eine *Schnittstelle*. Genauer: Eine *Listener-Schnittstelle*)

Wovon hängt die Anzahl der *abstrakten Methoden* in dieser Listener-Schnittstelle ab?
(Von der Anzahl der *Ereignis-Arten*, die zu der betreffenden Oberart gehören).

Was ist vielen (aber nicht allen) Oberarten von Ereignissen noch zugeordnet?
(Eine *Adapter-Klasse*)

Welchen Oberarten ist *keine* Adapter-Klasse zugeordnet?
(Den Oberarten, die nur *eine einzige Ereignis-Art* enthalten)

Beispiele für Oberarten ohne Adapter-Klasse?
(Mausrad-Ereignis, Aktions-Ereignis)

Angenommen, Sie wollen sich eine Liste aller **Listener-Schnittstellen** übers Bett hängen (z.B. um zu üben, von Java-Fachbegriffen zu träumen). Wie können Sie alle Listener-Schnittstellen finden?

Öffnen Sie die Online-Dokumentation der Java-Standardbibliothek (hoffentlich genügt dafür ein Klick auf einen Link auf dem Desktop Ihres Computers. Sonst sollten Sie nach "**Java 8 docs api**" googeln).

Suchen Sie in der Liste aller Klassen und Schnittstellen die Schnittstelle **EventListener**
(zur Erinnerung: Die Namen von Schnittstellen sind *kursiv* geschrieben, z.B. *EventListener*).

Unter **All Known Subinterfaces** finden Sie eine Liste von etwa 70 Listener-Schnittstellen.
Daraus können wir schließen: In Java 8 gibt es etwa 70 Oberarten von Ereignissen.

Wie viele und welche **Arten** (von Ereignissen) gehören zu der **Oberart** der eine bestimmte **Schnittstelle** zugeordnet ist?

Schnittstelle	**Anzahl Methoden**	**Arten / Methoden**	**Typ des Parameters**
CaretListener	1	caretUpdate	CaretEvent
FocusListener	2	focusGained, focusLost	FocusEvent
KeyListener	3	keyPressed, keyReleased, keyTyped	KeyEvent
ComponentListener	4	componentHidden, componentMoved, ComponentResizede, ComponentShow	ComponentEvent
DragSourceListener	5	dragDropEnd, dragDropEnter, ...	DragSourceDropEvent
DropTargetListener	5	dragEnter, dragExit, dragOver, ...	DropTargetDragEvent
IIOReadProgress-Listener	9	imageComplete, imageProgress, imageStarted, readAborted, ...	ImageReader
IIOWriteProgress-Listener	7	imageComplete, imageProgress, imageStarted, thumbnailComplete, ...	ImageWriter
MouseInputListener	7	mouseClicked, mouseEntered, mouseExited, mousePressed, mouseReleased, mouseDragged, mouseMoved	MouseEvent

Angenommen, Sie haben eine Grabo-Klasse und möchten gerne wissen: Welche **Arten** von Ereignissen ein Objekt dieser Klasse möglicherweise erzeugt. Wie können Sie diese Arten herausfinden?

Beispiel: Grabo-Klasse **JButton**: Suchen Sie (in der Online-Doku der Java-Standardbibliothek) in der Doku der Klasse `JButton` alle Methoden, deren Namen die Form **`add`**`XXX`**`Listener`** (mit irgendeinem XXX) haben. Jede solche Methode entspricht einer **Oberart** von Ereignissen.

Geerbte Listener-Methoden:
Von **AbstractButton**: addActionListener, addChangeListener, addItemListener,
Von **JComponent**: addAncestorListener, addVetoableChangeListener,
Von **Container**: addContainerListener, addPropertyChangeListener,
Von **Component**: addComponentListener, addFocusListener, addHierarchyBoundsListener, add HierarchyListener, addInputListener, addKeyListener, addMouseListener, addMouseMotionListener, addMouseWheelListener

16 Oberarten von Ereignissen

Beispiel: Grabo-Klasse **JFrame**:
Geerbte Listener-Methoden
Von **Frame**: --
Von **Window**: addPropertyChangeListener, addWindowFocusListener, addWindowListener, addWindowStateListener
Von **Container**: addContainerListener
Von **Component**: genau wie oben bei JButton

15 Oberarten von Ereignissen

Frage:
Die Klasse **JButton** erbt von der Klasse **Container** eine Methode **addPropertyChangeListener** Warum ist das bei der Klasse **JFrame** anders (sie erbt diese Methode *nicht* von **Container**)?

(Weil **JFrame** von **Window** eine Methode **addPropertyChangeListener** erbt, und diese Methode überschreibt die gleichnamige Methode der Klasse **Container**).

Anmerkung zur Klasse JFrame

Ein `JFrame`-Objekt ist *kein* Behälter (container), in den man andere Grabo-Objekte (z.B. der Typen `JButton`, `JLabel`, `Box` etc.) hineintun könnte, obwohl einige Methoden (`add`, `remove`, `setLayout`) versuchen, einem das vorzugaukeln. In Wahrheit *enthält* ein `JFrame`-Objekt `jf` einen Behälter `jf.getContentPane()`. Einzelheiten findet man in der Online-Dokumentation.

Behälter und Layout-Manager

Jedes Behälter-Objekt (engl. container object) `berta` hat (normalerweise) ein `LayoutManager`-Objekt, das man sich mit `berta.getLayout()` holen und das man mit `berta.setLayout(...)` (er-) setzen kann. Nach Ausführung von `berta.setLayout(null)` hat der Behälter `berta` *kein* `LayoutManager`-Objekt mehr.

Mit einem Layout-Manager kann man sich viel "fitzelige Programmierarbeit" ersparen.
Wenn der Benutzer z.B. die Größe eines `JFrame`-Behälters verändert, dann verändert der Layout-Manager die Größen der darin enthaltenen Objekte entsprechend.

Wenn man ohne Layout-Manager programmiert, muss man für jedes Objekt, welches man in einen Behälter einfügt, genaue Pixel-Koordinaten (seiner linken oberen Ecke) angeben. Wenn man das z.B. für 20 Objekte gemacht hat und dann noch ein weiteres Objekt dazwischen schieben will, muss man viele dieser Koordinaten neu berechnen und eingeben. Das ist mühselig und fehleranfällig.

Wenn man Layout-Manager verwendet, gibt man beim Einfügen (eines Grabo-Objekts in den betreffenden Behälter) keine fitzeligen Pixel-Koordinaten an, sondern viel einfachere Informationen. Abhängig vom verwendeten Layout-Manager kann ein Einfüge-Befehl etwa so aussehen:

```
                              // LayoutManager von berta:
berta.add(knopf16);           // BoxLayout
berta.add(knopf17);           // GridLayout
berta.add(knopf18, "Center"); // BorderLayout
berta.add(knopf19, "North");  // BorderLayout
```

`LayoutManager` ist eine Schnittstelle, die in der Java-Standardbibliothek von etwa **30 LayoutManager-Klassen** implementiert wird. Hier ein paar wichtige Beispiele solcher Klassen:

`BorderLayout`, `BoxLayout` (X- und Y-Varianten), `GridLayout` (ein einfaches "Gitter"), `GridBagLayout` (ein ziemlich kompliziertes "Gitter").

Sammlungen

Öffnen Sie das Papier **pr2_Sammlungen.odt** (5 Seiten).
Das Papier wurde teilweise, aber noch nicht ganz, besprochen.

8. SU, Di 29.11.2016

Heute schreiben wir den Test-08

Sammlungen (Fortsetzung)

Frage: Vermutlich haben Sie im Fach *Algorithmen und Datenstrukturen* (bei Frau Ripphausen-Lipa) schon etwas über Sammlungen (engl. collections) gelernt. Was genau?

Hinweis: In Java gehören alle Sammlungsklassen zum sogenannten **Collections Framework** .
Informationen zu diesem Framework findet man unter anderem an der folgenden Stelle:
https://docs.oracle.com/javase/8/docs/technotes/guides/collections/

Klassen und Typen, Gemeinsamkeiten und Unterschiede

Die Klasse `String` definiert genau *einen* Typ.
Zum Typ `int` und zum Typ `String[]` gibt es *keine* Klasse.
Die Klasse `ArrayList<K>` definiert unbegrenzt *viele* **parametrisierte Typen** wie z.B.
`ArrayList<String>, ArrayList<Integer>,`
`ArrayList<ArrayList<String>>, ...`

`ArrayList<K>` und `HashMap<S, W>` sind **generische Klassen**,
`Iterable<K>` und `Map<S, W>` sind **generische Schnittstellen**.

Dabei sind `K`, `S` und `W` **Typ-Parameter**. An ihrer Stelle muss man einen *Referenztyp* angeben (*primitive Typen* sind an diesen Stellen *nicht* erlaubt).

Es ist üblich, als Namen von Typ-Parametern *einzelne Großbuchstaben* (wie `K`, `S`, `W`) zu verwenden (der Java-Ausführer würde auch Namen wie `a` oder `Anton` oder `anton` akzeptieren).

Variablen eines parametrisierten Typs vereinbaren:

```
ArrayList<String> als1 = new ArrayList<String>(); // früher
ArrayList<String> als2 = new ArrayList<>();       // seit Java 8 erlaubt
```

Die meisten *generischen Klassen* in der Java-Standard-Bibliothek sind *Sammlungsklassen* (d.h. sie implementieren die generische Schnittstelle `Collection<K>` ("K" wie Komponenten-Typ, auf Englisch heißt die Schnittstelle `Collection<E>` mit "E" wie element type).

Methoden, die es in jeder Sammlung-Klasse gibt

Öffnen Sie (wie schon im vorigen SU) das Papier **pr2_Sammlungen.odt** auf **S. 4**. oben
(**6. Die generische Schnittstelle<K>**)

Angenommen, wir haben eine Sammlungen `has` vereinbart wie folgt:

```
HashSet<Number>  han = new HashSet<>();
```

Zur Erinnerung:
Die Klassen `Byte, Short, Integer, Long, Float, Double, BigInteger, BigDecimal` und noch ein paar sind Erweiterungen der (abstrakten) Klasse `Number`.

Welche Methoden der Schnittstelle `Collection<K>` enthält das Objekt `han`?

```
boolean          add        (Number ob)
boolean          remove     (Object ob)
boolean          contains   (Object ob)
boolean          addAll     (Collection<? extends Number> c)
boolean          removeAlle (Collection<?> c)
...
Iterator<Number> iterator   ()
Object[]         toArray    ()
<T> T[]          toArray    (T[] rei)
...
```

Welche der folgenden Methoden-Aufrufe sind erlaubt?

```
... han.add(123)          ...   // erlaubt
... han.add(3.5)          ...   // erlaubt
... han.add("Hallo")      ...   // nicht erlaubt
... han.remove(123)       ...   // erlaubt
... han.remove("Hallo")   ...   // erlaubt
... han.contains(123)     ...   // erlaubt
... han.contains("Hallo") ...   // erlaubt
```

Wann liefert der Funktionsaufruf `han.contains(otto)` das Ergebnis `true`?
(Wenn `otto` in `han` vorkommt)

Wann liefert der Funktionsaufruf `han.remove(emil)` das Ergebnis `true`?
(Wenn der Aufruf die Sammlung `han` *verändert* hat, d.h. wenn `emil` gelöscht werden konnte).

Was darf/muss man als Parameter der Methode `addAll` angeben?
(Eine Sammlung von Objekten einer *Erweiterung* von `Number`, d.h. eine Sammlung des Typs `Collection<Byte>` oder `Collection<Integer>` oder `Collection<Number>`).

Zur Erinnerung: `Number` ist auch eine Erweiterung von `Number`!

Die optionalen Methoden in der Schnittstelle Collection<K>

Die volle Wahrheit über Schnittstellen (in Java 8)

Bisher haben wir so getan, als ob eine Schnittstelle nur *abstrakte Objekt-Methoden* enthält. Tatsächlich konnten Schnittstellen schon seit Java 1 auch noch *Klassen-Konstanten* (genauer: unveränderbare Klassen-Attribute) enthalten. Und ab Java 8 können sie auch noch *konkrete Klassen-Methoden* und *konkrete Objekt-Methoden* enthalten.

Anmerkung: Klassen-Methoden sind immer *konkret* (es gibt keine *abstrakten* Klassen-Methoden)

Eine **Beispiel-Schnittstelle**, die Elemente aller möglichen Arten enthält:

```
 1 interface I16 {
 2    // Eine Schnittstelle kann folgende Arten von Elementen enthalten:
 3
 4    // Abstrakte Objekt-Methoden (seit Java 1):
 5    abstract public int add1(int n);
 6
 7    // Konkrete unveraenderbare Klassen-Variablen (seit Java 1):
 8    // (gehören zu den implementierenden Klassen, z.B. K16.PI, K16.NAM)
 9    static final double PI = 3.141_592_653_589_793_238;
10    static final String NAM = "Kreiszahl";
11
12    // (Konkrete) Klassen-Methoden (seit Java 8).
13    // Die gehoeren aber nicht zu den implementierenden Klassen sondern
14    // zur Schnittstelle I16 (z.B. I16.mul2, nicht K16.mul2)
15    static public int mul2(int n) {return 2* n;}
16
17    // Konkrete Objektmethoden:
18    default public int div3(int n) {return n/3;}
19    default public int mul4(int n) {return mul2(n)*2;}
20 }
21
22 // Eine Klasse, die die Schnittstelle I16 implementiert:
23 class K16 implements I16 { ... }
```

9. SU, Di 06.12.2016

Heute schreiben wir den Test-09

Harte und weiche Bedingungen, die eine Sammlung erfüllen muss

Harte Bedingung: Jedes Sammlung-Objekt muss zu einem `Collection`-Typ (z.B. `Collection<String>` oder `Collection<Integer>` oder ...) gehören.
Diese Bedingung wird vom Ausführer (bei der Übergabe eines Programms) überprüft.

Weiche Bedingungen: S. 2, **Collection §1** und **Collection §2**
Diese Bedingungen werden *nicht* vom Ausführer überprüft.

Die Abschnitte 10. bis 12. im Papier **pr2_Sammlungen.odt** besprechen:

10. Erweiterungen der Schnittstelle Collection<K>
11. Wie definiert man Totalordnungen für die Objekte einer bestimmten Klasse?
12. Beispiel: Drei Totalordnungen für Objekte der Klasse Mensch

Zur Entspannung: **Wie lernt man? Ein simples Modell.**

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

1. Sie verbinden je zwei Einzelteile zu einem 2-er-Teil. Das sind 512 Arbeitsschritte. Sie sehen kaum einen Fortschritt, weil ein 2-er-Teil nicht viel länger aussieht als ein Einzelteil.

2. Sie verbinden je zwei 2-er-Teile zu einem 4-er-Teil. Das sind etwa 256 Arbeitsschritte. Immer noch ist kaum ein Fortschritt zu sehen, denn die 4-er-Teile sind noch sehr kurz im Vergleich zum ein Meter langen Endergebnis.

3. Sie verbinden je zwei 4-er-Teile zu einem 8-er-Teil.128 Arbeitsschritte.

...

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

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

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

Zur **Aufgabe-07: Die toString-Methode der Klasse HashSet testen**

Eine Lösung ist teilweise vorgegeben (als Klasse `HashSetJut05`, Sie können den Namen aber auch ändern, z.B. zu `HashSetJut01`).

In der vorgegebenen Klasse brauchen Sie nur noch die Methode `doTheTesting` zu programmieren.

Die vorgegebene Klasse enthält 7 Testfälle. Davon sind 2 "aktiviert", die restlichen 5 sind auskommentiert (siehe die Methode `suite()` am Anfang der Klasse). Den letzten Testfall (`testToString07`) dürfen Sie immer auskommentiert lassen ("um bestimmte Probleme zu vermeiden").

Finden Sie heraus: Was soll die Methode `doTheTesting` (mit ihrem Parameter `sr`) genau machen?

Wurden im Fach **Algorithmen** *Hash-Tabellen* schon behandelt?

Problem: Im Ergebnis der Methode `toString` eines `HashSet`-Objekts stehen die einzelnen Komponenten nicht in *einer bestimmten* Reihenfolge, sondern in *irgendeiner* Reihenfolge.

Plan für die Methode `doTheTesting`:

Grobe (ungenaue) Zusammenfasung: Wir erzeugen aus der Reihung `sr` eine Sammlung des Typs `HashSet<String>` namens `hs`. Dann entfernen wir aus dem String `hs.toString()` die einzelnen Komponenten (von `sr` bzw. `hs`) und sollten dadurch einen *leeren String* bekommen.

1. Erzeugen Sie ein `HashSet<String>`-Objekt namens `hs`.

2. Fügen Sie alle Strings aus `sr` in die Sammlung `hs` ein und berechnen Sie dabei gleich, wie lang der String `hs.toString()` (ausschließlich der eckigen Klammern, aber einschließlich der der Trennstrings `", "` zwischen den einzelnen Komponenten) sein müsste. Berücksichtigen Sie dabei auch den im Punkt 6. (siehe unten) beschriebenen "kleinen Gleichmacher-Trick".

3. Speichern Sie das (zu testende) Ergebnis von `hs.toString()` in einer `String`-Variablen namens `st`.

4. Erledigen Sie den Fall, dass die Reihung `sr` die Länge 0 hatte (d.h. dass `hs` eine leere Sammlung ist).

5. Testen Sie, ob `st` wirklich mit `"["` beginnt und mit `"]"` endet. Dafür gibt es in der Klasse `String` zwei sehr gut geeignete Methoden namens `startsWith` und `endsWith`. Testen Sie auch, ob st die richtige Länge hat (die Sie im Punkte 2. berechnet haben).

6. Vereinbaren Sie einen `StringBuilder` namens `sb`, der im Wesentlichen die Zeichen des Strings `st` enthält, aber minus die eckigen Klammern und plus einem zusätzlichen Trennstring `", "` am Ende.
Begründung: Jetzt steht nach *jeder* Komponenten ein Trennstring `", "` (und die letzte Komponente ist keine "lästiger Sonderfall" ohne Trennstring dahinter).

Gefahr: Wenn `sr` z.B. die Strings `"AA"`, `"X"` und `"A"` enthält, dann könnte der Inhalt von `sb` jetzt gleich `"AA, X, A, "` sein.
Wenn wir jetzt das erste `"A, "` entfernen hat `sb` den Inhalt `"A, X, A"`.
Wenn wir jetzt versuchen `"AA, "` zu entfernen wird das nicht mehr funktionieren.

Wie können wir diese Gefahr abwehren? (Indem wir die zu entfernenden Strings absteigend nach ihrer Länge sortieren und dann die längeren Strings vor den kürzeren entfernen).

7. Vereinbaren Sie eine Klasse `StringNachLaenge`, die die Schnittstelle `Comparator<String>` implementiert und eine geeignete `compare`-Methode enthält.

8. Sortieren Sie die Reihung `sr` mit folgendem Befehl:

```
Arrays.sort(sr, new StringNachLaenge());
```

9. Entfernen Sie jede Komponente der (jetzt sortierten) Reihung `sr` aus dem StringBuilder `sb`. Das ist gar nicht soo einfach, aber man kann es mit den Methoden `sb.indexOf` und `sb.delete` hinkriegen. Wenn sich eine Komponente nicht entfernen lässt (weil sie in `sb` nicht oder *nicht mehr* vorkommt) sollte der Befehl `fail();` ausgeführt werden. Wenn alle Komponenten entfernt werden konnte, sollte `sb` die Länge 0 haben.

Bibliotheken und Frameworks

Bibliothek: Mit Klassen daraus ergänzt man eine selbstgeschriebene Hauptklasse.

Framework: Es enthält eine Hauptklasse, die man mit selbst geschriebenen Klassen ergänzt.

10. SU, Di 13.12.2016

Heute schreiben wir den Test-10

Markierungsschnittstellen und Anmerkungen

Markierungsschnittstellen (engl. marker interfaces), **Buch S. 352**.

Anmerkungen (engl. annotations), **Buch S.353**

Die Besprechung des Papiers **pr2_JUnit3undJUnit4.odt** musste verschoben werden, weil das Internet gestört und das Papier deshalb nicht zugänglich war (nur ich hatte einen Ausdruck).

Ein-/Ausgabe mit Strömen (streams) (im Buch Kapitel 19, ab S. 470)

Beim Einlesen von Daten (und ganz entsprechend auch beim Ausgeben von Daten) sind verschiedene Arbeitsschritte (Umwandlungen, Zwischenspeicherungen etc.) und vor allem *Kombinationen solcher Arbeitsschritte* nötig. Im Buch auf S. 470 werden 3 Beispiele solcher Kombinationen beschrieben.

In Java gibt es etwa 2 Dutzend Klassen, deren Objekte jeweils einen Eingabe-Arbeitsschritt durchführen, und etwa 1 1/2 Dutzend Klassen, deren Objekte jeweils einen Ausgabe-Arbeitsschritt durchführen. Diese knapp 4 Dutzend Klassen bezeichnet man als **Strom-Klassen** (engl. stream classes) und ihre Objekte als **Ströme** (engl. streams). Alles Strom-Klassen gehören zum Paket **java.io**.

Anmerkung: Das *Einlesen* von Daten ist grundsätzlich etwas schwieriger als das *Ausgeben*, weil man beim Ausgeben die Daten im Programm "genau kennt", beim Einlesen dagegen "mit verschiedenen Daten" rechnen muss, insbesondere mit unerwarteten oder falschen Daten.

Einzelne Eingabe-Strom-Objekte kann man zusammensetzen zu einem längeren Eingabe-Strom.. Einzelne Ausgabe-Strom-Objekte kann man zusammensetzen zu einem längeren Ausgabe-Strom.

Strom-Objekte sind so etwas wie **Bausteine**, die man auf viele Weisen kombinieren kann.

Def. Strom, (Buch S. 471)
Beispiel-02 (Buch S. 472)
Darstellung von Strömen die aus mehreren Strom-Objekten bestehen, z.B.

```
Datei <-- felix <-- oskar <-- bruno <-- Programm
```

Beispiel-03 (Buch S. 473).

```
Datei --> fiona -->  ilse --> britta --> Programm
```

Das Ende eines *Ausgabe-Stroms* (z.B. die Datei, in die man Daten schreibt) bezeichnet man allgemein als **Datensenke**. Den Anfang eines *Eingabe-Stroms* (z.B. die Datei aus der man Daten liest) bezeichnet man allgemein als **Datenquelle**.

Ich empfehle, in Strom-Darstellungen das *Programm* immer **rechts** (und somit die *Datenquelle* bzw. *Datensenke* immer **links**) zu zeichnen. Dann fließen Ausgabe-Daten immer *von rechts nach links* und Eingabedaten von *links nach rechts*. Diese Regelung soll die Wahrscheinlichkeit von Flüchtigkeitsfehlern (beim Lesen von Strom-Darstellungen) verringern.

Zeichenorientierte und Byte-orientierte Ströme (Buch S. 474)

Zeichenorientierte Ströme dienen zum Ein- und Ausgeben von **Text-Daten**. Damit sind alle Daten gemeint, die von Menschen (mit einem einfachen Editor wie z.B. TextPad oder NotePad++ oder ...) gelesen werden können.

Byte-orientierte Ströme dienen zum Ein- und Ausgeben von **Binär-Daten**. Damit sind alle Daten gemeint, die von Menschen nur mit einem ganz speziellen Programm angehört oder angesehen oder anderweitig benutzt werden können, z.B Audio-Daten oder Video-Daten oder `.odt`-Dateien oder `.doc`-Dateien etc. etc.

Es ist im Prinzip möglich, auch Binär-Daten mit zeichenorientierten Strömen zu lesen und z.B. auf dem Bildschirm anzuzeigen, das Ergebnis wird aber in aller Regel ein "kaum lesbarer Zeichensalat" sein.

Wenn man umgekehrt versucht Text-Daten z.B. von einem MP3-Player abspielen zu lassen, wird der Player in aller Regel eine Fehlermeldung ausgeben.

Besipiel: `.class`-Dateien enthalten Binär-Daten. Wie erkennt der Java-Ausführer, ob eine Datei wirklich eine "richtige `.class`-Datei" ist (und nicht nur eine "umbenannte `.txt`-Datei")?
Öffnen Sie mit dem **TextPad** irgendeine `.class`-Datei, stellen Sie aber (im *Datei öffnen - Fenster* des TextPads, ganz unten) als Dateiformat nicht **Auto** sondern **Binär** ein). Daraufhin sollte die Datei byteweise in *hexadezimaler Darstellung* und in *Zeichen-Darstellung* angezeigt werden, jeweils 16 Byte pro Zeile. In der Zeichen-Darstellung werden *nicht-darstellbare* Zeichen durch je einen Punkt `.` dargestellt. Was steht in den ersten 4 Bytes der Datei (in hex-Darstellung)? (`CA FE BA BE`).
Diese **magic number** prüft der Java-Ausführer immer, bevor er eine `.class`-Datei "ernst nimmt". Viele Programme, die Binärdaten lesen, benützen ähnliche **magic numbers**.

Objekte in Ströme schreiben / aus Strömen lesen (Buch S. 484)

In ein Strom-Objekt des Typs `ObjectOutputStream` kann man beliebige Objekte schreiben. Dabei werden die Objekte **serialisiert**, d.h. in eine bestimmte "Serie von Bits und Bytes" umgewandelt. Das geht allerdings nur, wenn die Objekte serialisierbar sind, d.h. wenn ihre Klasse die Schnittstelle `Serializable` implementiert.

Großartig-01: Angenommen, ein Objekt `anna` enthält ein Attribut namens `bert`, welches auf ein anderes Objekt zeigt. Wenn man dann den Befehl gibt, `anna` in einen Strom zu schreiben, wird außer `anna` automatisch auch das Objekt `bert` geschrieben. Und diese Regel gilt rekursiv: Wenn `bert` ein Attribut namens `carl` enthält, welches auf ein Objekt zeigt, dann wird auch `carl` geschrieben, und wenn `carl` ... etc.

Mit einem einzigen Schreibbefehl kann man also sehr viele "zusammenhängende" Objekte (z.B. in eine Datei) schreiben.

Großartig-02: Wenn man später (möglicherweise mit einem anderen Programm) das Objekt `anna` wieder einliest, wird nicht nur `anna` eingelesen, sondern auch `bert` und `carl` und ... .

Beim Lesen wird **deserialisiert**, d.h. aus jeder "Serie von Bits und Bytes" ein Objekt erzeugt.

Weitere Ströme

Es gibt in Java auch Ströme, mit denen man Daten in Archive (`.zip`- bzw. `gunzip`-Dateien) schreiben und von dort wieder einlesen kann.

Anmerkung: OpenOffice-Dateien (z.B. `.odt`- oder `.odg`-Dateien etc.) haben genau das Format von `.zip`-Archiven. Man kann sie also mit den entsprechenden Strom-Objekten lesen und schreiben. Das Gleiche gilt für `.jar`-Dateien ("jar" soll an "Java Archiv" erinnern).

Mit sogenannten `pipes` kann man Daten von einem Teil eines Programms zu eine anderen Teil desselben Programms schicken. Eine solche `pipe` besteht aus zwei (miteinander verbundenen) Strom-Objekten, einem zum Hineinschreiben und einem zum Herauslesen.

Die Klasse RandomAccesFile

Im Paket **java.io** gibt es außer den Strom-Klassen noch eine "nicht-Strom-Klasse" namens `RandomAccessFile`. Mit einem Objekt dieser Klasse kann man eine beliebige Datei als eine Reihe von Bytes bearbeiten. Mit der Methode `void seek(long pos)` kann man z.B. zum Byte Nr. `3580` "gehen" und dieses Byte mit der Methode `int read()` lesen oder mit der Methode `void write(int n)` einen neuen Wert hineinschreiben.

Das Paket java.nio (new io)

Das Java-System von Strom-Klassen ist sehr flexibel (man kann die Strom-Bausteinen zu sehr vielen verschiedenen Ein- und Ausgabe-Routinen zusammenbauen). Weil Java auf allen Plattformen laufen soll, können die Strom-Klassen aber keine "Möglichkeiten und Tricks" ausnützen, die nicht auf allen Plattformen funktionieren. Wenn die Ein-/Ausgabe von sehr großen Datenmengen sehr schnell gehen soll, ist der Baukasten der Strom-Klassen nicht gut geeignet, weil er besonders *flexibel* aber nicht besonders *schnell* ist.

Seit Java 4 gibt es deshalb zusätzlich zum Paket `java.io` ein Paket `java.nio`. Die Klassen in diesem (damals neuen) Paket ermöglichen eine "Betriebssystem-nahe" Ein-/Ausgabe und das Anwenden von "Möglichkeiten und Tricks" die es nur in einigen Betriebssystemen, aber nicht in allen gibt.

Zwei Bücher:

Java NIO von Ron Hitchens, Oreilly-Verlag 2002 (ca. 280 Seiten)

Pro Java NIO.2 von Anghel Leonard, Apress-Verlag 2011 (ca. 296 Seiten)

Abschließende Bemerkung zu Strömen

Mit Java 8 wurde unter anderem ein sehr interessantes neues Konstrukt eingeführt:
Ströme (engl. streams, im Paket `java.util.streams`). Leider haben diese Ströme nichts mit den "alten Strömen" (im Paket `java.io`) zu tun, die es schon seit Java 1 gibt. Vorschlag: Ab jetzt sollte man die alten Ströme als **I/O-Ströme** (engl. I/O streams) bezeichnen, und die neuen einfach als **Ströme**.

Zur Entspannung: **Das Ziegen-Problem**

Bei einer Gewinnshow im Fernsehen werden einem Kandidaten vom Showmaster drei (verschlossene) Türen gezeigt. Hinter einer der Türen steht ein wertvolles Auto (Hauptgewinn), hinter den anderen beiden zwei Ziegen (d. h. Nieten). Der Kandidat wählt z. B. die Tür 1.

Daraufhin öffnet der Showmaster z.B. die Tür 2, hinter der eine Ziege steht, und fragt dann den Kandidaten: "Wollen Sie bei Ihrer Entscheidung für Tür 1 bleiben oder wollen Sie ihre Entscheidung ändern (und Tür 3 wählen)?"

Frage: Was sollte der Kandidat tun? Bei seiner Entscheidung für 1 bleiben oder lieber die 3 wählen? Macht es einen Unterschied, ob er bei seiner ersten Wahl bleibt oder wechselt, oder ist das egal?

Lösung: Angenommen, die Ziegen-Show wird häufig wiederholt (z. B. 100 Mal) und zwar jeweils mit 2 gleichzeitigen Kandidaten, die zuerst immer dieselbe Tür wählen. Wenn der Showmaster dann fragt, bleibt K1 bei dieser Tür, K2 wählt dagegen die andere (noch verschlossene Tür). Bei jeder Show gewinnt also entweder K1 oder K2. Offenbar gewinnt K1 in etwa 1/3 aller Fälle. In allen anderen Fällen gewinnt K2, also in 2/3 aller Fälle. Also ist es besser, seine Wahl zu ändern.

11. SU, Di 20.12.2016

Heute schreiben wir den Test-11

Die Methode toString in der Klasse HashSet

Finden Sie (mit einem Laptop oder ähnlichen Gerät) heraus, was die Methode `toString` in der Klasse `HashSet` *genau* machen soll.

In der **Aufgabe-07 Die toString-Methode der Klasse HashSet testen**
mussten Sie *ohne Iteratoren* und somit auch *ohne for-each-Schleifen* auskommen, weil die (so sollten Sie annehmen) noch nicht getestet waren. In einer "ähnlichen Situation in der Praxis" hätte man wahrscheinlich zuerst die Klasse der `HashSet-Iterator`-Objekte getestet und erst danach die `toString`-Methode der Klasse `HashSet`. Weil in einem Ergebnis der `toString`-Methode die einzelnen Komponenten in der Reihenfolge stehen müssen, in der ein `Iterator`-Objekt sie liefert, wäre das Testen viel einfacher (als wenn man nichts über die Reihenfolge weiß).

Die Klasse RandomAccessFile

Das Paket java.nio

E/A-Ströme (I/O streams, seit Java 1, java.io) und Ströme (streams, seit Java 8,)

Das Papier pr2_JUnit3und4.odt

Dieses Papier enthält im Wesentlichen

ein JUnit 3 Testprogramm namens **StringBuilderTestJU3** und
ein JUnit 4 Testprogramm namens **StringBuilderTestJU4**

die beide ziemlich genau das Gleiche machen.

Vergleichen Sie die Zeilen 19 und 118 (`extends TestCase` / `extends Object`)
Wie erkennt die JUnit *die* Methode, die *vor jedem Testfall ausgeführt* werden soll?
(Name `setUp` / Anmerkung (engl. annotation) `@Before`)
Wie erkennt die JUnit die einzelnen *Testfälle*?
(Name muss mit "`test`" beginnen / Anmerkung `@Test`)

Kovariant, kontravariant und invariant (Buch Abschnitt 7.9, S. 173)

Beispiel für **kovariante** Größen: Geschwindigkeit und Luftwiderstand
Je *schneller* sich ein Mensch (oder ein Auto, ein Flugzeug etc.) durch die Luft bewegt, desto *höher* wird der Luftwiderstand. Wenn die Geschwindigkeit *kleiner* wird, wird auch der Luftwiderstand *kleiner*.

Beispiel für **kontravariante** Größen: Höhe (im Luftraum) und Luftdruck
Je *höher* ein Bergsteiger (oder ein Flugzeug, ein Ballon etc.) steigt, desto *kleiner* wird der Luftdruck. Wenn die Höhe *kleiner* wird, wird der Luftdruck *größer*.

Die meisten Größen-Paare sind weder kovariant noch kontravariant, sondern haben "ein komplizierteres Verhältnis zueinander" (sie sind **invariant**).

Beispiel: Tageszeit (Stunde 0 bis 23) und Helligkeit. Wenn die Tageszeit größer wird, wird die Helligkeit auch größer, aber z.B. nach 17 Uhr wird die Helligkeit (mit wachsender Tageszeit) wieder kleiner.

Die (Daten-) Typen in Java sind zwar nicht total geordnet, sondern nur partiell geordnet, aber von einem Klassen-Typ T sagt man manchmal, er sei **größer** als seine Untertypen (d.h. die Typen, an die T seine Elemente vererbt hat). Der Typ `Object` ist größer als alle anderen Referenztypen. Der Typ `Number` ist größer als die Typen `Integer`, `Float`, `BigInteger` etc. Die Typen `String` und `Integer` sind unvergleichbar (`String` ist nicht größer als `Integer` und `Integer` ist nicht größer als `String`).

Zur Erinnerung:
A: Der *Reihungstyp* `T[]` (Reihung von T) hat den *Komponententyp* T.
B: Der Typ `Number` ist größer als der Typ `Double`.

Wichtige Frage (bei der Entwicklung von Java):

Wie soll sich die Größe eines Reihungstyps mit der Größe seines Komponententyps verändern?

1. Soll der Typ `Number[]` **größer** sein als `Double[]`?
(d.h. **kovariantes** Verhältnis: Je **größer** der Komponententyp, desto **größer** der Reihungstyp)

2. Soll der Typ `Number[]` **kleiner** sein als `Double[]`?
(d.h. **kontravariantes** Verhältnis: Je **größer** der Komponententyp, desto **kleiner** der Reihungstyp)

3. Sollen die Typen `Number[]` und `Double[]` **unvergleichbar** sein?
(d.h. **invariantes** Verhältnis: obwohl die Komponententypen `Number` und `Double` **vergleichbar** sind)

Problem-01 (mit Kovarianz):

```
Double[] rda = {new Double(2.5), new Double(3.4)};
Number[] rna = rda;
rna[0] = new Long(123L); // Ein Long-Objekt in einer Reihung von Double?
```

Problem-02 (mit Kontravarianz):

```
Number[] rnb = {new Long(123), new Double(2.4)};
Double[] rdb = rnb;      // incompatible types: Number[] cannot be converted
Double   d   = rdb[0];   // Ein Long-Objekt in einer Double-Variablen?
```

Um **Problem-01** und **-02** zu vermeiden, sollte man eigentlich die Festlegung **3. (Invarianz**) wählen. Die Java-Entwickler haben aber die Festlegung **1. (Kovarianz)** gewählt und das damit verbundene Problem in Kauf genommen. Warum?

1. Was ist sehr gut an Typen? Der Ausführer kann bestimmte Fehler des Programmierers erkennen und melden. Dadurch wird das Beseitigen dieser Fehler viel billiger.

2. Was ist nicht so gut an Typen? Wenn man Werte *unterschiedlicher* Typen *gleich* (oder "ganz ähnlich") *bearbeiten* will, muss man evtl. *für jeden Typ* eine eigene Methode schreiben.

Problem-03: Wie kann man **eine** Methode schreiben, die man auf Objekte **unterschiedlicher Typen** anwenden kann (statt für jeden Typ eine eigene Methode zu schreiben)?

3. Wenn man nur **ein** Objekt ("von irgendeinem Typ") bearbeiten will, kann man in Java ("schon immer", d.h. seit Java 1) **eine** Methode mit einem Parameter vom Typ `Object` vereinbaren (und damit Objekte unterschiedlicher Typen bearbeiten).

4. Aber was ist, wenn man **viele** Objekte ("von irgendeinem Typ") bearbeiten will?
Dann kann man wegen der Entscheidung für die **Kovarianz bei Reihungstypen** eine Methode mit einem Parameter vom Typ `Object[]` ("Reihung von `Object`") vereinbaren. Diese Methode kann man dann auch mit einem Argument vom Typ `String[]` oder `Integer[]` oder ... etc. aufrufen. **Ohne Kovarianz** (mit Invarianz oder Kontravarianz) wäre das nicht möglich.

Anmerkung-01: In Java gibt es **eine** oberste Klasse (`Object`), aber **viele** unterste Klassen (z.B. alle Klassen die als `final` definiert wurden wie etwa `String`). Deshalb wäre eine Entscheidung für die **Kontravarianz bei Reihungstypen** viel weniger nützlich als die für die **Kovarianz** (aber genauso problembehaftet, siehe **Problem-02**).

Anmerkung-02: Java-Programme können das **Problem-01** (oder ähnliche Probleme) enthalten und doch vom Ausführer *akzeptiert werden*. Erst bei der Ausführung des Programms ("zur Laufzeit") wird das Problem erkannt und eine Ausnahme des Typs **ArrayStoreException** geworfen.

Mit **Java 5** hat man das **Problem-03** noch einmal "viel gründlicher" gelöst, nämlich mit **generischen Einheiten** (Klassen und Methoden). Generische Einheiten kann man als eine Möglichkeit verstehen, bestimmte **Härten und Nachteile eines strengen Typensystems** zu zu mildern.

Seit es in Java generische Einheiten gibt, käme man auch gut *ohne die Kovarianz* (bei Reihungstypen) aus (und wäre gern das damit verbundene **Problem-01** los.). Leider kann man diese Kovarianz aber nicht mehr abschaffen, weil sehr viele Java-Programme (deren Entwicklung sehr, sehr viel Geld gekostet hat) sich darauf verlassen.

Was man aber beim Einführen von generischen Einheiten richtig machen konnte und gemacht hat:

Regel: Ein parametrisierter Typ wie z.B. `HashSet<String>` ist *kein Untertyp* (und *kein Obertyp*) des Typs `HashSet<Object>` (obwohl `String` ein Untertyp von `Object` ist). Mit anderen Worten: Ein generischer Typ (z.B. `HashSet<K>`) und sein Parametertyp `K` sind nicht ko- oder kontravariant, sondern **invariant**.

Anmerkung: Für generische Typen mit mehreren Typ-Parametern gilt Entsprechendes.

Wird nicht offiziell behandelt, hab ich nur stehen lassen (falls jemand Interesse hat).

Mit den Begriffen kovarianz und kontravarianz kann man auch eine sehr tiefgehende Eigenschaft von Funktionen beschreiben:

Beispiel-02: Eine Funktion mit einem `Number`-Ergebnis und einem `Number`-Parameter

```
Number f37(Number n) { ... }
```

Diese Funktion kann man also mit einem `Number`-Argument aufrufen und ihr Ergebnis in einer `Number`-Variablen speichern, z.B. so:

```
Number na = ...;
Number nb = f37(na);
```

Als Argument darf man aber auch ein Objekt eines Untertyps von `Number` angeben, und das Ergebnis der Funktion darf man auch in einer Variablen eines Obertyps von `Number` speichern, z.B. so:

```
Double da = new Double(2.5); // Double ist ein Untertyp von Number
Object ob = f37(da);         // Object ist ein  Obertyp von Number
```

Man kann das auch so beschreiben:

Funktionen sind
kontravariant bezüglich ihres Ergebnistyps und
kovariant bezüglich ihrer Parameter-Typen.

Abbildungen (engl. maps)

Auf die Komponenten einer *Reihung* kann man mit *Indizes* zugreifen.
Indizes sind `int`-Werte (die in Java immer mit 0 beginnen).

Eine *Abbildung* ist eine Art "verallgemeinerte Reihung": Anstelle von Indizes vom Typ `int` gibt es *Schlüssel-Objekte*, die zu einem beliebigen Referenztyp (z.B. `String` oder `Integer` oder `JButton` oder ...) gehören können.

Def. (inhaltlich): Eine **Abbildung** ist eine Menge von *Einträgen*. Jeder *Eintrag* besteht aus 2 Objekten: einem *Schlüssel* (-Objekt) und einem *Wert* (-Objekt).

Achtung: Der *Wert eines Eintrags* ist etwas ganz anderes als der *Wert einer Variablen*.
Der Wert eines Eintrags ist (fast) *immer ein Objekt*, der Wert einer Variablen ist *nie ein Objekt*.

Def. (formal): Eine **Abbildung** ist ein Objekt einer `Map`-Klasse
(d.h. einer Klasse, die die Schnittstelle `Map` implementiert.

Abbildungs-Regel-01: Zwei verschiedene Einträge müssen *verschiedene* Schlüssel haben.
Abbildungs-Regel-02: Zwei verschiedene Einträge dürfen (*verschiedene* oder) *gleiche* Werte haben.

Beispiel-01: Eine **Abbildung** (d.h. ein `Map`-Objekt) vereinbaren und bearbeiten:

```
 1 import        java.util.HashMap;
 2 import static java.util.HashMap.Entry;
 3 import        java.util.Collection;
 4 import        java.util.Set;
 5 ...
 6       // Eine Abbildung (a Map object) abel vereinbaren;
 7       HashMap<String, Integer> abel = new HashMap<>();
 8
 9       // Ein paar Eintraege (Entry-Objekte) hineintun:
10       abel.put("Anna", 12); // Der Eintrag Anna=12 wird eingefuegt
11       abel.put("Bert", 15);
12       abel.put("Carl", 13);
13       abel.put("Bert", 24); // Bert=24 ersetzt Bert=15
14
15       // Aus der Abbildung abel drei Sammlungen erzeugen:
16       Set<Entry<String, Integer>> entries = abel.entrySet();
17       Set      <String>           keys    = abel.keySet();
18       Collection       <Integer>  values  = abel.values();
19
20       // Die drei Sammlungen ausgeben:
21       printf("abel.entrySet(): %s%n", entries);
22       printf("abel.keySet():   %s%n", keys);
23       printf("abel.values():   %s%n", values);
```

Ausgabe auf dem Bildschirm:

```
abel.entrySet(): [Carl=13, Bert=24, Anna=12]
abel.keySet():   [Carl, Bert, Anna]
abel.values():   [13, 24, 12]
```

12. SU, Di 10.01.2017

Heute schreiben wir den Test-12.

Zur Erinnerung: Objektorientierte Programmierung

Buch Abschnitt 9.5 (ab **S. 219**)

In einem objektorientierten Programm zum Verwalten der Beuth Hochschule würde man wahrscheinlich Klassen namens `StudentIn`, `DozentIn`, `LehrVeranstaltung`, `Raum`, ... etc. so vereinbaren, dass Objekte dieser Klassen die in der Beuth Hochschule wichtigen Personen und Dinge repräsentieren können.

Was für Klassen würde man in einem objektorientierten Programm zum Verwalten von Java-Programmen vereinbaren? Aus was für Dingen bestehen Java-Programme?

Genau diese Klassen gibt es schon in der Java-Standard-Bibliothek:

`Class, Constructor, Field, Method, Parameter, Modifier, ...`

Reflexion

Gemeinsam bezeichnet man diese Klassen als die **Reflexions-Schnittstelle von Java**
("Schnittstelle" ist hier in einem allgemeinen Sinne gemeint, nicht im engeren, technischen Sinn des englischen Wortes interface).

Erläuterung der Bezeichnung Reflexion: Ein Objekt der Klasse `Class` *reflektiert* eine Klasse, d.h. das Objekt enthält alle wichtigen Informationen über die Klasse. Häufig sagt man auch einfach: Ein `Class`-Object *ist* eine Klasse. Entsprechend: Ein `Method`-Objekt reflektiert eine Methode, oder: Ein `Method`-Objekt ist eine Methode. ... etc.

Ein Programmierproblem: Ein Programm enthält viele (z.B. 100 oder 1000 etc.) Methoden namens `m1`, `m2`, `m3`, ... Der Name einer Methode wird (als String) eingelesen und die entsprechende Methode soll aufgerufen werden. Eine mögliche Lösung:

```
if (eingabe.equals("m1")) m1(); else
if (eingabe.equals("m2")) m2(); else
...
```

Mit Reflexion kann man dieses Problem deutlich eleganter lösen.

Mit Hilfe von Reflexion kann man "wundersame Dinge" programmieren:

Beispiel-01: Einen Klassen-Browser

Dem muss man den vollen Namen einer Klasse eingeben (z.B. `"java.lang.Math"`). Der Browser zeigt einem dann alle Konstruktoren und Elemente an, die von dieser Klasse geerbt oder in dieser Klasse vereinbart wurden. Und wenn man die Klasse verändert und erneut anschaut sieht man immer "die neuste Version". Ein solches Programm z.B. in C++ zu schreiben ist praktisch unmöglich.

Beispiel-02: Ein Funktions-Ausführer

Eingabe: Der volle Name einer Klasse, der Name einer Funktion (die in der Klasse vereinbart ist) und geeignete Argumente für diese Funktion. Daraufhin wird die Funktion (mit den angegebenen Argumenten) ausgeführt und das Ergebnis wird angezeigt.

Beispiel-03: Eine Funktion, die ein beliebiges Objekt liefert

Als Parameter erwartet diese Funktion den vollen Namen einer Klasse (als `String`). Als Ergebnis liefert die Funktion (meistens, aber nicht immer) ein (zufällig gewähltes) Objekt der Klasse.

Wie funktioniert die Reflexions-Schnittstelle von Java?

Für jeden Typ `T` gibt es ein `Class`-Objekt, welches den Typ `T` reflektiert (d.h. alle möglichen Informationen über der Typ T enthält).

Eine reflektive Methode beginnt in aller Regel damit, dass man sich Zugriff auf das `Class`-Objekt verschafft, welches einen bestimmten Typ `T` reflektiert. Dazu ein paar Beispiele.

Beispiel-01: Das `Class`-Objekt eines "normalen" Typs besorgen

```
static void m01() throws ClassNotFoundException {
   // 3 Moeglichkeiten, auf das Class-Objekt eines Typs zuzugreifen
   //(je nachdem, was man von den Typ weiss oder hat):

   // 1. Wenn wir (beim Schreiben eines reflektiven Programms)
   // den Namen des Typs (z.B. Integer) kennen:
   Class<Integer> kob1 = Integer.class;

   // 2. Wenn wir ein Objekt des Typs haben (eventuell ohne
   // seinen genauen Typ zu kennen):
   Object   ob   = new Integer(123);
   Class<?> kob2 = ob.getClass();

   // 3. Wenn wir nur eine String-Variable haben, in der
   // der volle Name des Typs steht (dieser Name ist eventuell
   // beim Schreiben des Programms noch nicht bekannt und wird
   // erst zur Laufzeit des Programms eingelesen):
   String    kName = "java.lang.Integer";
   Class<?>  kob3  = Class.forName(kName); // Evtl. ClassNotFoundException
   ...
}
```

Beispiel-02: Das `Class`-Objekt eines primitiven Typs besorgen:

```
static void m02() {
   // Wie bekommt man Zugriff auf das Class-Objekt eines
   // primitiven Typs, z.B. des Typs int oder des Typs boolean?

   Class<?> kob11 = int.class;
   Class<?> kob12 = Integer.TYPE;
   Class<?> kob21 = boolean.class;
   Class<?> kob22 = Boolean.TYPE;
   ...
}
```

Beispiel-03: Was man (unter anderem) mit einem `Class`-Objekt machen kann

```
static void m03() throws Exception {

   Class<?> kob = Integer.class;

   Field   fo1 = kob.getDeclaredField("MIN_VALUE");
   Field[] fr1 = kob.getDeclaredFields();
   Field   fo2 = kob.getField("SIZE");
   Field[] fr2 = kob.getFields();

   Method   mo1 = kob.getDeclaredMethod("sum", int.class, int.class);
   Method[] mr1 = kob.getDeclaredMethods();
   Method   mo2 = kob.getMethod("wait", long.class);
   Method[] mr2 = kob.getMethods();

   Constructor<?>   co1 = kob.getDeclaredConstructor(String.class);
   Constructor<?>[] cr1 = kob.getDeclaredConstructors();
   Constructor<?>   co2 = kob.getConstructor(int.class);
   Constructor<?>[] cr2 = kob.getConstructors();
   ...
}
```

Zur Entspannung: **Die Polymerase Kettenreaktion (PKR, engl. PCR)**

Wurde 1983 von *Karry Mullis* in Kalifornien (angeblich während einer langweiligen Autofahrt) erfunden. 1993 erhielt er den Nobelpreis dafür (obwohl er "nur eine Technik erfunden", aber keine "grundlegenden Prinzipien entdeckt" hatte).

Ein *Polymer* ist eine chemische Substanz, bei der die Länge der einzelnen Moleküle nicht genau festliegt, weil eine bestimmte Gruppe von Atomen ("ein Kettenglied") sich fast beliebig oft wiederholen kann.

DNS-Moleküle (Desoxyribonukleinsäure) sind wichtige Beispiele für solche Kettenmoleküle. Ein DNS-Molekül besteht aus zwei Teilsträngen (etwa wie Eisenbahnschienen), die aber nicht gleich, sondern "Spiegelbilder voneinander" sind, etwa so:

```
ATGC ...
|||| ...
TACG ...
```

Eine *Polymerase* ist ein Enzym, welches z.B. DNS-Moleküle an einer bestimmten Stelle durchtrennen kann, oder ähnliche chemische Reaktionen stark begünstigt. Z.B. gibt es eine Polymerase, mit der man die beiden Teilstränge ("Eisenbahnschienen") von DNS-Molekülen voneinander trennen kann. Beide Teilstränge haben dann eine starke Tendenz dazu, einen Ersatz für "den abgetrennten Partner-Strang" an sich "anzulagern" und wieder "eine vollständige Schiene" zu werden.

Die PKR ist eine Technik, mit der man DNS-Moleküle sehr schnell vermehren kann. Im Extremfall beginnt man mit einem einzigen DNS-Molekül und hat nach wenigen Stunden Milliarden oder Trillionen Kopien davon.

Die PKR läuft in Schritten ab, von denen jeder aus 3 Teilschritten besteht. Durch Erwärmen und Abkühlen der beteiligten Stoffe auf bestimmte Temperaturen kann man die einzelnen Teilschritte einleiten bzw. beenden. Nach jedem Schritt (gleich 3 Teilschritten) hat sich die Anzahl der DNS-Moleküle verdoppelt. Ein solcher Schritt dauert z.B. 5 Minuten. In 50 Minuten (ca. 1 Stunde) kann man also 10 Verdopplungen bewirken, die Anzahl der DNS-Moleküle also um den Faktor 2^{10} (ungefähr 1000) vermehren. Innerhalb von 6 Stunden kann man die Ausgangsmenge theoretisch um einen Faktor von etwa 2^{60} ($\approx 10^{18}$ gleich eine Trillion) vermehren, praktisch ist die Ausbeute etwas geringer.

13. SU, Di 17.01.2017

Heute schreiben wir den Test-13 (d.h. den letzten Test in diesem Semester)

Termine und Orte der **Hauptklausur** und ihrer **Rückgabe** stehen jetzt in der Datei **Stichworte.pdf**.

Die Online-Dokumentation der Java Standardbibliothek

Schauen Sie möglichst oft in dieser Dokumentation nach.
Machen Sie sich vertraut mit der Struktur dieser Dokumentation.
Lernen Sie die englischen Vokabeln, die in dieser Dokumentation benutzt werden.
Diese Dokumentation ist die beste Quelle von Informationen über Java.
Im Internet gibt es sehr viel deutlich schlechtere Informations-Quellen.

Warum ist die Klasse Class generisch (Class<T>)?

Von welchem Typ ist das `Class`-Objekt `String.class`? (Vom Typ `Class<String>`)
Von welchem Typ ist das `Class`-Objekt `Integer.class`? (Vom Typ `Class<Integer>`)

Angenommen, Sie haben eine Klasse namens `Carola` vereinbart.
Von welchem Typ ist dann das `Class`-Objekt `Carola.class`? (Vom Typ `Class<Carola>`)

Warum ist das so? Warum ist die Klasse `Class` generisch?

Öffnen Sie die Online-Doku der Klasse `Class<T>` und schlagen Sie die Seite **Method Summery** auf.
Was haben die beiden Methoden `cast` und `newInstance` gemeinsam?
(Beide haben den Ergebnistyp `T`)
Generische Klassen (und Methoden) gibt es erst seit **Java 5**.
Welchen Ergebnistyp hatte die Methode `newInstance` vorher (z.B. in **Java 4**)?
(Sie hatte den Ergebnistyp `Object`)
Und was war mit der Methode `cast`?
(Die gab es nicht, weil bei ihr der Ergebnistyp `Object` sinnlos wäre)

Also: Die Klasse `Class<T>` ist generisch, weil nur so die Methoden `cast` und `newInstance` den "richtigen Ergebnistyp" `T` haben können.

Zur Klausur (am Di 31.01.2017, im Raum B101)

6 Aufgaben. Pro Aufgabe 20 bzw. 15 Punkte, insgesamt 100 Punkte.

Bringen Sie Papier mit, um darauf die Lösungen zu schreiben. Empfehlung:Verwenden Sie weißes, kariertes Papier (keine Umweltpapier, Probleme beim Radieren).

Sie dürfen mit Bleistift schreiben (und radieren).

Kennzeichnen Sie jedes Blatt, welches Sie abgeben, mit Ihrem **Nachnamen** (am besten in der rechten oberen Ecke). Das können Sie schon vorher zu Hause erledigen.

Schreiben Sie die Lösung von jeder Aufgabe auf die Vorderseite eines neuen Blattes (auch wenn dadurch Teile Ihrer Blätter leer bleiben). Lassen Sie die Rückseiten Ihrer Blätter leer.

Was sollten Sie wissen und können?

Sie sollten mit den **10 Aufgaben** vertraut sein, die im Laufe des Semesters bearbeitet wurden.

Sie sollten mit den **Aufgaben in den 13 Tests** vertraut sein, die geschrieben wurden.

Sie sollten mit allen **Fachbegriffen**, die im Laufe des Semesters behandelt wurden, vertraut sein.

Sie sollten sich mit **Zahlensystemen** auskennen

Sie sollten **rekursive Methoden** schreiben können.

Sie sollten **reflektive Methoden** schreiben können ("Methoden, in der Reflexion vorkommt").

Sie sollten die Methoden `format` und `printf` kennen.

Sie sollten sich mit **Zufall** und **Zufallswerten** auskennen.

Sie sollten mit der vollen Wahrheit über **Schnittstellen** (interfaces) vertraut sein.

Sie sollten die Unterschiede zwischen **for-i-Schleifen** und **for-each-Schleifen** kennen.

Zum nächsten SU (dem letzten vor der Klausur) können Sie **alle Fragen** mitbringen (am besten schriftlich), auf die Sie beim Wiederholen des Stoffs gestoßen sind. Ihre Fragen haben dann höchste Priorität.

Zur Entspannung: **Mit DNS-Molekülen kombinatorische Probleme lösen**

Die Gene aller Lebewesen bestehen aus DNS (Desoxyribonukleinsäure) Molekülen. Solche Moleküle können große Mengen von Informationen auf sehr kleinem Raum speichern. *Leonard M. Adleman* hat 1994 zuerst gezeigt, dass man mit solchen Molekülen auch bestimmte algorithmische Probleme lösen kann, z.B. das *Problem des Handlungsreisenden* (ein Handlungsreisender will n Städte besuchen und sucht nach einem kürzesten Weg dafür). In ein Reagenzglas passen mehrere Trillionenen DNS-Moleküle, die (unter geeigneten Bedingungen) alle versuchen, sich miteinander zu verbinden. Mit DNS Ligase kann man bestimmte Verbindungen erheblich erleichtern und damit beschleunigen.

Ein DNS-Moleküle besteht aus 2 Ketten von vier Nukleotiden: A, G, C, T. Ketten verbinden sich, wenn sich überall *komplementäre* Nukleotide gegenüberstehen, etwa so:

```
Kette 1: ACGT ...
         ||||
Kette 2: TGCA ...
```

Grundtechnik zur Lösung des Problems des Handlungsreisenden: Jede Stadt wird durch eine einfache (nicht doppelte!) Kette von 20 Nukleotiden dargestellt, z.B. durch

```
Stadt A: TTGACGAATG ATGCTAGAAA (Komplement: AACTGCTTAC TACGATCTTT)
Stadt B: AATCCATGCG AAATTAGCCC (Komplement: TTAGGTACGC TTTAATCGGG)
Stadt C: TATGACCTAG CTAGCATAGC (Komplement: ATACTGGATC GATCGTATCG)
```

Eine Straße von Stadt x nach Stadt y wird ebenfalls durch 20 Nukleotide dargestellt: Die letzten 10 von x und die ersten 10 von y. Hier zwei Straßen (von A nach B und von B nach C):

```
A-nach-B: ATGCTAGAAA AATCCATGCG
B-nach-C: AAATTAGCCC TATGACCTAG
```

Ein `A-nach-B`- Molekül kann sich mit dem Komplement von `Stadt B` verbinden wie folgt:

```
A-nach-B          : ATGCTAGAAA AATCCATGCG
                              ||||||||||
Komplement von B  :           TTAGGTACGC TTTAATCGGG
```

Dieses Molekül kann sich mit einer Straße `B-nach-C` verbinden:

```
A-nach-B, B-nach-C: ATGCTAGAAA AATCCATGCG AAATTAGCCC TATGACCTAG
                              |||||||||| ||||||||||
Komplement von B  :           TTAGGTACGC TTTAATCGGG
```

Man erzeugt (mit der PKR) von jedem *Städte*-Molekül `A`, `B`, `C`, ... und von jedem *Straßen*-Molekül `A-nach-B`, `B-nach-C`, ... ein paar Billiarden oder Trillionen, füllt alle Moleküle in ein Reagenzglas und "schüttelt ein bisschen". Dann bilden sich u.a. Moleküle, die einem kürzesten Weg "an allen Städten vorbei" entsprechen. Die kann man mit Standard-Techniken der Molekular-Biologie (Gel-Elektrophorese) "herausfiltern" und analysieren.

BitSet-Objekte

Beispiel-01: Angenommen,
Sie wollen etwa *500 Tausend* Zahlen (z.B. Telefon-Nrn) aus dem Bereich *0 bis 1 Million* speichern. Wie würden Sie das machen, und wie viel Speicher (in Bytes, Kilo-Bytes oder Mega-Bytes) würde Ihre Lösung ungefähr kosten?

Lösung-01: Sie speichern die Zahlen in einer Reihung von `int`-Variablen,
jede Zahl in einer der `int`-Variablen (á 4 Byte).
Dann brauchen Sie insgesamt ungefähr 500.000 x 4 = 2.000.000 Bytes = 2 MB.

Lösung-02: Sie vereinbaren ein `BitSet`-Objekt mit 1 Million Bits und setzen darin jedes Bit, welches einer der (500 Tausend) Zahlen entspricht, auf 1 (und alle anderen Bits auf 0).
Das `BitSet`-Objekt belegt nur etwa 1.000.000 / 8 = 125.000 Bytes = 125 KB (statt 2 MB).

14. SU, Di 24.01.2017

Heute schreiben wir (ähnlich wie im 1. SU) **keinen Test**.

Empfehlung: Ich würde allen TeilnehmerInnen dieser LV (die PR2 nicht noch mal wiederholen wollen oder müssen) raten, an der **Hauptklausur** am Di 31.01.2017 teilzunehmen und nicht von vornherein die **Nachklausur** (am Do 30.03.2017) einzuplanen. Erfahrungsgemäß ist es sehr, sehr schwierig, am **Ende** der Semesterferien z.B. im Fach PR2 deutlich mehr zu wissen, als kurz vor dem **Anfang** der Semesterferien.

Wer hat vor, die Hauptklausur am Di 31.01.2017 mitzuschreiben?

Hat jemand Fragen zum Stoff dieser Lehrveranstaltung?

Was ist besonders an BitSet-Objekten?

Ein `BitSet`-Objekt ähnelt einer Reihung vom Typ `boolean[]` (die aber typischerweise besonders platzsparend implementiert ist). Bei einer Reihung stehen normalerweise die zu speichernden Daten in den Komponenten der Reihung, und die Indizes sind nur Hilfsgrößen, mit denen man auf diese Daten zugreifen kann.

Bei einem `BitSet`-Objekt ist das genau umgekehrt: Die Indizes (der einzelnen Bits) sind die zu speichernden Daten, und die Bits enthalten nur Hilfsgrößen `true` oder `false`.

Diese "Umkehrung" (Vertauschung von Daten und Hilfsgrößen) kommen auch bei anderen Datenstrukturen vor, z.B. bei Häufigkeitstabellen.

Beispiel-01: Angenommen wir haben einen String `s`, der (neben anderen Zeichen) auch (0 oder mehr) **Dezimalziffern** enthält. Dann kann es interessant sein herauszufinden und darzustellen, wie oft die einzelnen Ziffern in s vorkommen, etwa so:

```
String s: "3x0000y99"
Häufigkeitstabelle:{4, 0, 0, 1, 0, 0, 0, 0, 0, 2}
```

Diese Häufigkeitstabelle kann z.B. als Reihung des Typs `int[]` der Länge `10` implementiert werden.

Aufgabe-01: Schreiben Sie eine Methode entsprechend der folgenden Spezifikation:

```
static public int[] histoZiffern(String s) {
   // Wie oft kommt in s die Ziffer '0' vor? Und die Ziffer '1'?
   // Und ... Und die Ziffer '9'? Diese Funktion liefert die Antworten
   // auf diese 10 Fragen in Form einer Reihung der Laenge 10.
   // Beispiel:
   // String s21 = "A1221B9990";
   // histoZiffern(s21) ist gleich {1, 2, 2, 0, 0, 0, 0, 0, 0, 3},
   // den s21 enthaelt 1 mal '0', 2 mal '1', 2 mal '2' und 3 mal '9'.
```

Aufgabe-01: Schreiben Sie eine Methode entsprechend der folgenden Spezifikation:

```
static public int[] histoZahlen(int[] ir) {
   // Wie oft kommt in ir die Zahl 0 vor? Und die Zahl 1?
   // Und ... Und die Zahl 15? Diese Funktion liefert die Antworten
   // auf diese Fragen in Form einer Reihung der Laenge 16.
   // Beispiel:
   // int[] ir11 = {2, 15, 37, 2, 15, 2, -3};
   // histoZahlen(ir11) ist gleich {0, 0, 3, 0, ... 0, 2}
   // denn ir11 enthaelt 3 mal die 2 und 2 mal die 15.
   // Die anderen Zahlen (37 und -3) werden nicht gezaehlt.
```

Bei einer Häufigkeitstabelle (implementiert als Reihung) werden die Daten (d.h. die Zeichen im String `s` oder die `int`-Werte in `ir`) als Indizes (d.h. "als Hilfsgrößen") verwendet.

Lösung-01:

```
 1    static public int[] histoZiffern(String s) {
 2       // Wie oft kommt in s die Ziffer '0' vor? Und die Ziffer '1'?
 3       // Und ... Und die Ziffer '9'? Diese Funktion liefert die Antworten
 4       // auf diese 10 Fragen in Form einer Reihung der Laenge 10.
 5       // Beispiel:
 6       // String s21 = "A1221B9990";
 7       // histoZiffern(s21) ist gleich {1, 2, 2, 0, 0, 0, 0, 0, 0, 3},
 8       // den s21 enthaelt 1 mal '0', 2 mal '1', 2 mal '2' und 3 mal '9'.
 9
10       int[] erg = new int[10];
11
12 //    for (int i=0; i<s.length(); i++) {
13 //       char c = s.charAt(i);
14       for (char c : s.toCharArray()) {
15          if ('0' <= c && c <= '9') erg[c - '0']++;
16       }
17
18       return erg;
19    }
```

Lösung-02:

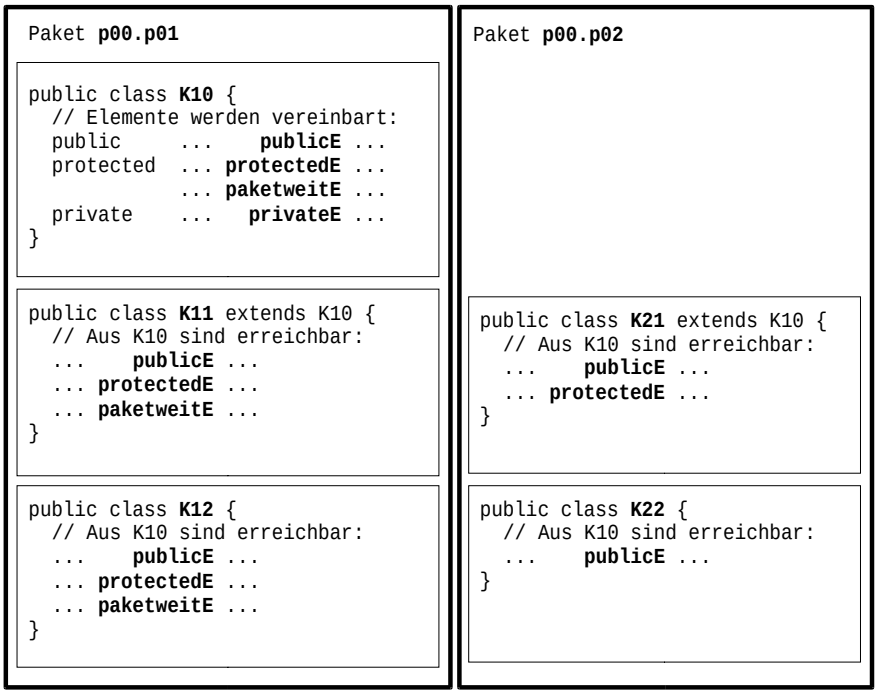
```
 1    static public int[] histoZahlen(int[] ir) {
 2       // Wie oft kommt in ir die Zahl 0 vor? Und die Zahl 1?
 3       // Und ... Und die Zahl 15? Diese Funktion liefert die Antworten
 4       // auf diese Fragen in Form einer Reihung der Laenge 16.
 5       // Beispiel:
 6       // int[] ir11 = {2, 15, 37, 2, 15, 2, -3};
 7       // histoZahlen(ir11) ist gleich {0, 0, 3, 0, ... 0, 2}
 8       // denn ir11 enthaelt 3 mal die 2 und 2 mal die 15.
 9       // Die anderen Zahlen (37 und -3) werden nicht gezaehlt.
10
11       int[] erg = new int[16];
12
13       for (int n : ir) {
14          if (0<=n && n <= 15) erg[n]++;
15       }
16
17       return erg;
18    }
```

Die Erreichbarkeiten protected und "paketweit erreichbar"? (Buch S. 434)

Eine Klasse enthält Konstruktoren (engl. constructuors) und Elemente (engl: members).
Jeder Konstruktor und jedes Element ist auf eine von 4 Weisen erreichbar:

- öffentlich (Schlüsselwort `public`)
- geschützt (Schlüsselwort `protected`)
- paketweit erreichbar (ohne Schlüsselwort)
- privat (Schlüsselwort `private`)

Die folgende Darstellung (mit 2 Paketen und 5 Klassen) soll deutlich machen, was diese Erreichbarkeiten bedeuten:



Wir gehen aus von der Klasse `K10` im Paket `p00.p01`.
Die anderen vier Klassen haben folgende "Beziehung zu `K10`":

`K11`: Ist im selben Paket wie `K10` und erbt von `K10`
`K12`: Ist im selben Paket wie `K10.`
`K21`: Ist in einem anderen Paket als `K10` und erbt von `K10`
`K22`: Ist in einem anderen Paket als `K10.`

Von den 4 in `K10` vereinbarten Elementen sind in den anderen Klassen alle eingezeichnet, die dort erreichbar sind (d.h. "auf die man dort zugreifen kann").