

Stichworte

zur Lehrveranstaltung **Programmieren 2**
im zweiten Semester des Studiengangs **Medien-Informatik Bachelor**
im Wintersemester 2009/10 an der **Beuth-Hochschule für Technik Berlin**,
von Ulrich Grude.

Wie bekommt man eine Note für das Fach MB2-PR2?

Im Laufe des Semesters werden (am Anfang bestimmter Übungsblöcke) 7 Tests geschrieben. Bei jedem Test können Sie maximal 20 Punkte erreichen. Am Ende des Semesters werden die Punkte Ihrer 5 besten Tests addiert und aus der Summe (Maximalwert: $5 * 20$ gleich 100) wird nach der folgenden Tabelle eine *Übungsnote* berechnet:

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

Wenn Sie an weniger als 5 der 7 Tests teilnehmen, bekommen Sie (unabhängig von der erreichten Punktzahl) die Übungsnote 5,0. Dabei spielt es keine Rolle, aus welchem Grund Sie nicht an mindestens 5 der Tests teilgenommen haben.

Wenn Ihre Übungsnote besser als 5,0 ist, dürfen Sie an der Klausur am Ende des WS09/10 (und evtl. an der Nachklausur kurz vor dem SS10) teilnehmen. In der Klausur können Sie maximal 100 Punkte erreichen. Aus der Anzahl Ihrer Klausurpunkte wird eine *Klausurnote* berechnet (ebenfalls nach der obigen Tabelle).

In die Berechnung Ihrer *Note für das Fach MB2-PR2* geht die Übungsnote mit 25 % und die Klausurnote mit 75 % ein. Das Ergebnis der Prozentrechnung wird zur nächstliegenden Note gerundet. Es folgend drei Beispiele dafür, wie eine *Note* aus den beiden *Teilnoten* berechnet wird:

Übungsnote	Klausurnote	Rohergebnis	(gerundete) Note
2,3	2,7	$(2,3 + 3 * 2,7) / 4 = 2,60$	2,7
3,7	1,3	$(3,7 + 3 * 1,3) / 4 = 1,90$	2,0
2,0	4,0	$(2,0 + 3 * 4,0) / 4 = 3,50$	3,3

Zu den Tests und den Aufgaben

Die Fragen in den 7 Tests beziehen sich

1. auf die *Aufgaben*, die jeweils vor dem Test gelöst werden sollten,
2. auf den Stoff, der vor dem Test in der *Vorlesung behandelt* wurde und
3. auf den Stoff, der vor dem Test in den *Übungen* behandelt wurde.

Auf der Netzseite <http://public.beuth-hochschule.de/~grude/> finden Sie eine Liste, der Sie entnehmen können, *wann* (in welchen Vorlesungsblöcken) die 7 Tests geschrieben werden und welche Aufgaben wann (d.h. vor welchem Test) gelöst werden sollen.

Änderung (30.09.2009): Die *Tests* werden nicht am Anfang bestimmter *Vorlesungsblöcke* geschrieben, sondern am Anfang bestimmter *Übungsblöcke*. Diese Änderung wurde notwendig, weil der Vorlesungsraum (B554) zu klein ist, um darin mit den zahlreichen TeilnehmerInnen Tests zu schreiben.

Übung 1 (Gruppe 1b: Mo 28.09.2009, 2. Block, Gruppe 1a: Fr 02.10.2009, 1. Block)

1. Falls mehr StudentInnen an dieser Übungsgruppe (1a bzw. 1b) teilnehmen möchten als Plätze vorhanden sind, die vorhandenen Plätze verlosen.

Anmerkung: Die Kapazität der Übungsgruppen (ca. 22 TeilnehmerInnen) wird vor allem durch die *Kapazität des Betreuers* begrenzt, *nicht* durch die Anzahl der Rechner im SWE-Labor. Diese *menschliche Betreuungskapazität* wird *nicht* größer, wenn einige StudentInnen auf eigenen Laptops arbeiten, statt die Labor-Rechner zu benutzen (eigene Laptops benötigen eher ein bisschen mehr menschliche Betreuung als die fertig eingerichteten Laborrechner).

2. Einloggen (Noch fehlende Benutzer-Konten einrichten).

3. Auf Papier **TipsZuEclipse** hinweisen ([http://public.beuth-hochschule.de/~grude/#MB2 PR2](http://public.beuth-hochschule.de/~grude/#MB2_PR2))

4. Eclipse starten und ein `Hallo`-Programm schreiben und laufen lassen (so wie im Papier `TipsZuEclipse` im Abschnitt 2. beschrieben).

5. Das Template `sysout` benutzen (**TipsZuEclipse**, Abschnitt 5.1. **Schablonen anwenden, Beispiel-02 und -03**).

5. Von der Netzseite <http://public.beuth-hochschule.de/~grude/JavaIstEineSprache/> das Archiv `DateienFuerPR2.zip` herunterladen und in ein Verzeichnis `Z:\DateienFuerPR2` entpacken).

6. In Eclipse ein weiteres Projekt erstellen (z.B. eines namens `Proj02`). In dieses Projekt die Quelldatei `Z:\DateienFuerPR2\DialogGeneratorFenster.java` importieren (siehe dazu `TipsZuEclipse.pdf`, Abschnitt 9. **Eine Java-Quelldatei in ein Eclipse-Projekt importieren**). Das Programm `DialogGeneratorFenster` ausführen lassen.

7. Die Datei `AufgabenPR2.pdf` auf obiger Netzseite ansehen und die **Aufgabe 1 (Quiz)** bearbeiten.

Achtung: Eine Lösung dieser Aufgabe sollte nicht nur *Antworten*, sondern vor jeder Antwort auch die zugehörige *Frage* enthalten (damit Ihre Antworten leichter überprüft werden können. Ob z.B. die Antwort 42 richtig ist, hängt ja auch sehr von der zugehörigen Frage ab).

Tip zu den Aufgaben: Sehen Sie sich die Aufgaben (in der Datei `AufgabenPR2.pdf`) zu Hause schon vor der Woche an, in der sie offiziell bearbeitet werden. Nehmen Sie sich fest vor, dem offiziellen Zeitplan für die Bearbeitung der Aufgaben immer ein bisschen *voraus* zu sein und nie hinterher zu hinken.

1. SU Mo 28.09.09

1. Begrüßung

Paßen wir alle in den B554?

2. Die Regeln nach denen man für diese LV eine Note bekommt stehen ma Anfang der Datei `Stichworte.pdf` (auf meiner Netzseite). Kurzfassung:

Am Anfang bestimmter Vorlesungen (hier im Raum B554) 10-Minuten Kurzttests, 7 Stück, die 5 besten zählen. Pro Test max. 20 Punkte. Daraus ergibt sich die Übungsnote.

Wer mind. 50 Punkte erreicht (Übungsnote 4,0) darf an der Klausur und/oder Nachklausur teilnehmen. Die Endnote für das Fach ergibt sich aus der Übungsnote (25 %) und der Klausurnote (75 %).

Jetzt geht es los mit dem Stoff!

Entwurfsmuster

Ein wichtiges Ziel beim Programmieren: *Wiederverwendung* (man sollte 1. möglichst wiederverwendbare Programmteile schreiben und 2. möglichst viele Teile wiederverwenden).

Problem: Wenn man die Programmiersprache wechselt (z.B. von C++ zu Java) kann man "die alten wiederverwendbaren Teile" nicht mehr wiederverwenden.

Man hat aber herausgefunden: Wichtige objektorientierte Sprachen (SmallTalk, C++, Java, C# etc.) haben so viel gemeinsam, dass man nicht nur konkrete Quelldateien wiederverwenden kann, sondern auch "die abstrakte Struktur" eines Programmteils. Für solche weitgehend sprachunabhängige Programmstrukturen wurde der Begriff Entwurfsmuster (engl. *design pattern*) eingeführt.

Wichtigstes Buch: "Design Patterns. Elements of Reusable Object-Oriented Software" von Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides.

Die Autoren werden auch als *Gang of Four* (GoF) und das Buch als *das GoF-Buch* bezeichnet.

Hier die (englischen) Namen einiger Entwurfsmuster (sollte man zumindest mal gehört haben):

Factory Method, Singleton, Adapter, Decorator, Iterator, Visitor.

Im vorigen Semester haben Sie als Aufgabe 2 eine Art Taschenrechner für `int`-Werte geschrieben. In diesem Semester sollen Sie als Aufgabe 2 ebenfalls eine Art Taschenrechner schreiben, aber

- mit einer Grabo (grafischen Benutzeroberfläche)
- mit einer änderungsfreundlichen Struktur entsprechend dem MVC-Muster

Das MVC-Muster sieht drei Komponenten vor: **model**, **view** und **control**, die in unserem konkreten Fall die folgenden Aufgaben erledigen sollen:

model: Legt den Typ der Zahlen fest, mit denen gerechnet wird (z.B. `int` oder `float` oder ...) und führt die Rechnungen (Addition, Subtraktion, ...) durch.

view: Realisiert die Grabo. Ermöglicht die Eingabe von zwei Zahlen und die Anzeige der Rechenergebnisse (Summe, Differenz, ...).

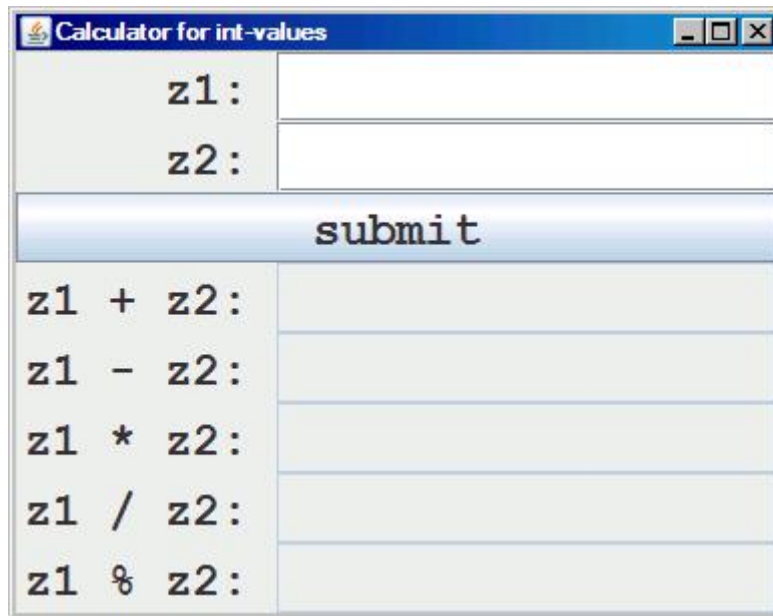
control: Verbindet die anderen beiden Komponenten und steuert sie.

Im einfachsten Fall wird jede *Komponente* durch genau *eine Klasse* realisiert. Die **model**-Klasse und die **view**-Klasse sollen aber leicht austauschbar sein, ohne dass die anderen Komponenten deshalb geändert werden müssten.

Unterschiedliche **model**-Klassen: für `int`-Werte, für `float`-Werte, für ...

Unterschiedliche **view**-Klassen: schlichte Grabo mit Text-Feldern für die Ein-/Ausgabe der Zahlen, oder mit 3D-Effekten, Goldrändern und Sprachausgabe oder ...

Abbildung einer möglichen Grabo:



7 Text-Felder (z.B. `TextField`-Objekte), 2 für die Eingabe ($z1$ und $z2$) und 5 für die Ausgabe ($z1+z2$ bis $z1\%z2$). Ein Knopf (z.B. ein `Button`-Objekt) um eine Berechnung zu veranlassen.

Grundlegendes zu `TextField`:

Bei einem `TextField`-Objekt `jtf` kann man Listener der folgenden Typen anmelden:

Listener-Typ	Wird informiert wenn
<code>ActionListener</code>	der Benutzer auf <code>Return</code> oder <code>Enter</code> drückt*
<code>KeyListener</code>	der Benutzer auf irgendeine Taste der Tastatur drückt*
<code>FocusListener</code>	<code>jtf</code> den Fokus gewinnt oder verliert

* während `jtf` den Fokus hat!

Der Befehl `jtf.requestFocusInWindow()`; bewirkt, dass das Text-Feld `jtf` den Fokus bekommt (wenn das umgebende Fenster den Fokus hat oder bekommt). Mit diesem Befehl in einem `FocusListener` kann man bewirken, dass `jtf` den Fokus nur unter bestimmten Bedingungen verlieren kann, z.B. wenn sein Inhalt eine Prüfung bestanden hat.

Fragen mit mehreren Antworten:

Wann / wie oft sollen die Eingaben des Benutzers (in einem Text-Feld `jtf`) geprüft werden?

- Wenn er auf den submit-Knopf klickt?
- Wenn er (nach der Eingabe von ein paar Zeichen) auf `Return` drückt?
- Wenn `jtf` den Fokus verliert?
- Nach jedem Tastendruck des Benutzers?

Wann / wie oft sollen die nötigen Berechnungen durchgeführt und die Ergebnisse angezeigt werden?

- Wenn der Benutzer auf den submit-Knopf klickt?
- Wenn der Benutzer auf `Return` drückt (und $z1$ und $z2$ Eingaben enthalten)?
- Nach jedem Tastendruck des Benutzers?

Wie kann man bewirken, dass nur die `model`-Komponente "festlegt und weiß" ob mit `int`-Werten oder mit `float`-Werten oder ... gerechnet wird? In welcher Form sollen die `view`- und die `control`-Komponente mit diesen Werten umgehen?

- in Form von `String`-Objekten
- in Form von `Number`-Objekten

Number ist die direkte Oberklasse der Klassen `BigDecimal`, `BigInteger`, `Byte`, `Double`, `Float`, `Integer`, `Long` und `Short`. Das sind praktisch alle Java-Klassen, die interessant sind, wenn man rechnen will.

Die `model`-Klasse und die `view`-Klasse müssen *kooperieren*, dürfen aber nicht *konkret voneinander abhängig sein* (z.B. darf der `Name` der `model`-Klasse in der `view`-Klasse nicht vorkommen, sonst müsste man die `view`-Klasse ja ändern, wenn man eine `model`-Klasse gegen eine andere austauscht).

Wie kann man so eine Kooperation ohne konkrete Abhängigkeit hinkriegen? (Mit Schnittstellen!)

Die Aufgabe 2: Ein Rechner mit Grabo nach dem MVC-Muster strukturiert verteilen und besprechen.

Zur Entspannung: Was ist eine Zahl? Entwicklung des Begriffs einer Zahl

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

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

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

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

Die Namen aller Klassen und Schnittstellen, die zu einer Lösung von Aufgabe 2 gehören, sollten mit "Rechner" beginnen und zum Paket `rechner` gehören.

Die Schnittstelle `RechnerModel_I` ("I" wie interface):

Sollte von der (oder: von jeder) `model`-Klasse implementiert werden.

Methode `getTypeName`:

Nützlich, wenn das Rechner-Fenster einen Titel haben soll (z.B. `Calculator for int-values`) der sich automatisch ändert, wenn man die `model`-Klasse austauscht (z.B. gegen eine, die mit `float`-Werten rechnet).

Methode `isValidEntry`:

In welcher Komponente (M, V oder C?) wird diese Methode gebraucht? Wann? Was macht sie?

Methode `calculate`:

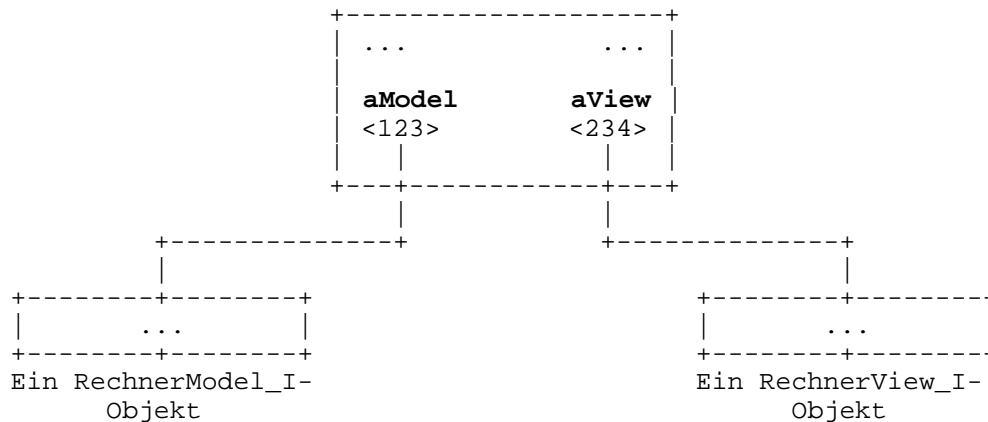
In welcher Komponente (M, V oder C?) wird diese Methode gebraucht? Wann? Was macht sie?

Was braucht man, um ein Objekt der Klasse `RechnerControl` zu erzeugen?

(Ein `RechnerModel_I`-Objekt und ein `RechnerView_I`-Objekt).

Das bedeutet: Das `control`-Objekt "kennt" das `model`-Objekt und das `view`-Objekt (und kann mit den Namen `aModel` bzw. `aView` darauf zugreifen). Aber die anderen beiden Objekte "wissen nicht", mit welchen konkreten Objekten sie zusammenarbeiten.

Ein `RechnerControl`-Objekt



Im RechnerControl-Objekt werden zwei Listener-Klassen (Baupläne für Listener-Objekte) vereinbart. **Wie heißen die?** (MeinActionListener und MeinKeyListener).

In den Methoden dieser Klassen werden Methoden aus dem view und aus dem model aufgerufen. **Wie heißen die?** (aView.getNumber1, aView.getNumber2, aView.setResults, aView.isValidEntry und aModel.calculate). **Diese Aufrufe sind nur möglich, weil das RechnerControl-Objekt mit den Namen aView und aModel auf die anderen beiden Objekte zugreifen kann.**

Jedesmal, wenn der Konstruktor der RechnerControl-Klasse ein neues RechnerControl-Objekt initialisiert, erzeugt er 3 Listener-Objekte namens mal, mk11 und mk12. **Was macht der Konstruktor mit diesen 3 Objekten?** (Er ruft damit die Methode aView.register auf).

Was sollte die Methode aView.register mit den 3 Listener-Objekten machen, die sie als Parameter übergeben bekommt? (Sie sollte sie beim submit-Knopf bzw. bei den beiden Eingabe-Textfeldern anmelden).

Angenommen, der Methodenaufruf aView.register(mal, mk11, mk12); (am Ende des RechnerControl-Konstruktors) ist fertig ausgeführt, und die register-Methode hat die drei Listener-Objekte richtig angemeldet. Wenn der Benutzer mit der Maus in das erste Eingabe-Textfeld klickt (damit es den Fokus bekommt) und dann auf eine Taste drückt (z.B. auf die Minus-Taste) und die Taste wieder loslässt, was passiert dann? (Die Methode keyReleased im Listener-Objekt mk11 wird aufgerufen).

Was bewirkt diese Methode keyReleased? (Sie prüft, ob der neue Inhalt des ersten Eingabe-Textfeldes, der das gerade eingetippte Minuszeichen enthält, gültig (valid) ist oder nicht).

Was macht die Methode keyReleased, wenn der neue Inhalt nicht gültig ist? (Sie stellt den vorigen Inhalt des Text-Feldes wieder her. Der war bestimmt gültig!).

Was passiert, wenn der Benutzer auf den submit-Knopf klickt (bei dem die register-Methode ja das Listener-Objekt mal angemeldet hat)?

Die actionPerformed-Methode im Listener-Objekt mal wird aufgerufen.

Was bewirkt sie? (Dass die Methoden aView.getNumber1, aView.getNumber2 und aModel.calculate aufgerufen werden).

Wiederholungsfragen, 2. SU, Mo 05.10.09

1. Der Rechner, den Sie zur Lösung von Aufgabe 2 entwickeln, soll aus drei Komponenten bestehen: *model*, *view* und *control*. Welche dieser Komponenten kann auf welche andere(n) Komponenten zugreifen? Füllen Sie die folgende Tabelle mit JA und NEIN aus.

model kann auf view zugreifen	
model kann auf control zugreifen	
view kann auf control zugreifen	
view kann auf model zugreifen	
control kann auf model zugreifen	
control kann auf view zugreifen	

2. Welche der drei Komponenten legt fest, ob mit `int`-Werten oder mit `float`-Werten oder mit ... gerechnet wird?
3. Welche der drei Komponenten ist für das *Einlesen von Zahlen* und das *Anzeigen von Rechenergebnissen* zuständig?
4. Beschreiben Sie ganz kurz und allgemein *die Aufgabe der control-Komponente* (nicht länger und genauer als z.B. "Die control-Komponente kocht Kaffee für die anderen beiden Komponenten").
5. Die control-Komponente ist vorgegeben (in Form der Klasse `RechnerControl`). Wie viele Parameter hat der (einzige) Konstruktor der Klasse `RechnerControl` und von welchem Typ sind sie?

Antworten zu den Wiederholungsfragen, 2. SU, Mo 05.10.09

1. Der Rechner, den Sie zur Lösung von Aufgabe 2 entwickeln, soll aus drei Komponenten bestehen: *model*, *view* und *control*. Welche dieser Komponenten kann auf welche andere(n) Komponenten zugreifen? Füllen Sie die folgende Tabelle mit JA und NEIN aus.

model kann auf view zugreifen	NEIN
model kann auf control zugreifen	NEIN
view kann auf control zugreifen	NEIN
view kann auf model zugreifen	NEIN
control kann auf model zugreifen	JA
control kann auf view zugreifen	JA

2. Welche der drei Komponenten legt fest, ob mit `int`-Werten oder mit `float`-Werten oder mit ... gerechnet wird?

Die model-Komponente.

3. Welche der drei Komponenten ist für das *Einlesen von Zahlen* und das *Anzeigen von Rechenergebnissen* zuständig?

Die view-Komponente.

4. Beschreiben Sie ganz kurz und allgemein *die Aufgabe der control-Komponente* (nicht länger und genauer als z.B. "Die control-Komponente kocht Kaffee für die anderen beiden Komponenten").

Die control-Komponente verbindet die beiden anderen Komponenten (oder: steuert die beiden anderen Komponenten, oder ...)

5. Die control-Komponente ist vorgegeben (in Form der Klasse `RechnerControl`). Wie viele Parameter hat der (einzige) Konstruktor der Klasse `RechnerControl` und von welchem Typ sind sie?

2 Parameter vom Typ `RechnerModel_I` bzw. `RechnerView_I`.

2. SU Mo 05.10.09

Organisation: Die Tests werden am Anfang bestimmter *Übungsblöcke* geschrieben, nicht am Anfang von *Vorlesungen* hier im B554.

Rekursion

Was macht der Programmierer wenn er möchte, dass ein bestimmter Befehl (z.B. `println(" * ");`) *mehrmals* ausgeführt wird? (Er schreibt den Befehl in den *Rumpf einer Schleife*).

Schleifen bewirken (normalerweise), dass Befehle *mehr als einmal* ausgeführt werden. Aber es gibt noch ein anderes Konstrukt, mit dem man das auch bewirken kann: *Rekursion*.

Anmerkung: Die Idee der Rekursion ist viel älter als die Idee von `for`- und `while`-Schleifen.

Beispiel: Die folgende *rekursive Definition* der Fibonacci-Zahlenfolge steht schon in einem Mathematikbuch, welches vermutlich im Jahre 1202 in Pisa erschien (ca. 200 Jahre bevor Johannes Gutenberg geboren wurde, es gab also nur handgefertigte Abschriften von Büchern):

Sei $\text{fib}(0)$ gleich 0 und $\text{fib}(1)$ gleich 1.

Sei $\text{fib}(n+2)$ gleich $\text{fib}(n) + \text{fib}(n+1)$.

Diese Definition ist *rekursiv* ("auf sich selbst zurückgreifend"), weil die Funktion `fib` mit Hilfe der Funktion `fib` definiert wird ($\text{fib}(n+2)$ wird mit Hilfe von $\text{fib}(n)$ und $\text{fib}(n+1)$ definiert).

Berechnen Sie (mit Papier und Bleistift) die Zahlen $\text{fib}(2)$ bis $\text{fib}(10)$.

n	0	1	2	3	4	5	6	7	8	9	11	...
fib(n)	0	1	1	2	3	5	8	13	21	34	55	...

Frage: Wie kann man aus n die Zahl $\text{fib}(n)$ direkt berechnen, ohne vorher alle Zahlen $\text{fib}(0), \text{fib}(1), \text{fib}(2), \dots, \text{fib}(n-1)$ zu berechnen.

Antwort: Na ganz einfach :-) nach der folgenden Formel:

$$\text{fib}(n) = \frac{1}{\sqrt{5}} \left[\left(\frac{1+\sqrt{5}}{2} \right)^n - \left(\frac{1-\sqrt{5}}{2} \right)^n \right]$$

Diese *nicht-rekursive* Formel fand der Mathematiker *Abraham de Moivre* im Jahre 1730 (mehr als 500 Jahre nachdem Fibonacci von Pisa die *rekursive* Definition veröffentlicht hatte).

Merkregel für uns: In manchen Fällen ist eine rekursive Definition *viel einfacher* und leichter verständlich als eine entsprechende *nicht-rekursive* Definition.

Anmerkung: Mit Hilfe von Fibonacci-Zahlen kann man unter anderem *Wachstumsprozesse* besonders verständlich beschreiben, z.B. die Vermehrung von Kaninchen.

Beispiel: Noch eine wichtige rekursive Definition:

Ein **binärer Baum**

ist entweder ein *Leerbaum* oder

er besteht aus einem *Knoten* K , an dem zwei *Bäume* B_1 und B_2 hängen.

Was der Begriff *Baum* bedeutet, wird hier mit Hilfe des Begriffs *Baum* (genauer: Bäume) definiert.

Beim Programmieren unterscheidet man *rekursive* und *nicht-rekursive* Methoden (oder: Unterprogramme). Welche Methoden bezeichnet man als rekursiv? (Solche, die "auf sich selbst zurückgreifen", d.h. die sich selbst aufrufen).

Was versteht man unter der *Fakultät einer natürlichen Zahl n*? (Das Produkt aller natürlichen Zahlen von 1 bis n . Per vernünftiger Definition ist die Fakultät von 0 und die von 1 gleich 1).

Beispiel: Eine rekursive Methode, die die Fakultät einer natürlichen Zahl berechnet

```

1 int fak(int n) {
2     if (n < 2) {
3         return 1;
4     } else {
5         return n * fak(n-1);
6     }
7 }
```

Das Arbeitsblatt (mit `plus1`, `minus1`, `add`, unvollständiger `mul`-Funktion etc.).

Die `add`-Funktion besprechen.

Die `mul`-Funktion vervollständigen lassen.

Die `pot`-Funktion sollte zu Hause gelöst werden.

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

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

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

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

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

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

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

Grundregel für rekursive Methoden:

Der Rumpf einer rekursiven Methode muss immer aus einer *Fallunterscheidung* bestehen (z.B. aus einer `if`-Anweisung oder einer `switch`-Anweisung), die Folgendes unterscheidet:

- nicht-rekursive Fälle (in denen die Methode sich *nicht* rekursiv aufruft) und
- rekursive Fälle (in denen die Methode sich rekursiv aufruft)

Begründung:

Wenn die Methode sich in *keinem Fall* rekursiv aufruft ist sie nicht rekursiv.

Wenn die Methode sich in *allen Fällen* rekursiv aufruft, ist sie "endlos-rekursiv".

Arbeitsblatt (für die 2. VL am Mo 05.10.09)

Angenommen, unser Java-Ausführer wurde von einer hungrigen Maus angeknabbert und funktioniert seitdem nur noch eingeschränkt:

1. Zum Typ `int` gehören nur noch die nicht-negativen ganzen Zahlen `0, 1, 2, 3, ...`.
2. Von den `int`-Operationen `+, -, *, /, %, ++, --, ==` funktionieren nur noch die letzten drei (`++, --, ==`). Alle anderen Rechenoperationen funktionieren nicht mehr, aber man kann noch **Funktionen vereinbaren und aufrufen**.

Für diesen eingeschränkten Java-Ausführer haben wir als Fingerübung die folgenden beiden Funktionen vereinbart:

```
1 int plus1 (int n) {return ++n;}
2 int minus1(int n) {return --n;}
```

die wie erwartet gut funktionieren.

Damit könne wir eine rekursive **Additionsfunktion** `add` vereinbaren, etwa so:

```
3 int add(int m, int n) {
4     if (n == 0) {
5         return m; // m + 0 = m
6     } else {
7         return plus1(add(m, minus1(n))); // m + n = (m + (n-1)) + 1
8     }
9 }
```

Ganz ähnlich (und mit Hilfe dieser `add`-Funktion) kann man auch eine rekursive **Multiplikationsfunktion** `mul` vereinbaren. Es folgt hier eine unvollständige Version einer solchen Vereinbarung:

```
10 int mul(int m, int n) {
11     if (          ) {
12         return
13     } else if (          ) {
14         return
15     } else {
16         return
17     }
18 }
```

Können Sie die fehlenden Ausdrücke (5 Stück, 2 nach `if` und 3 nach `return`) ergänzen?

Es folgt das Skelett einer Potenzierungsfunktion:

```
19 int pot(int m, int n) {
20     // Liefert die n-te Potenz von m (d.h. den Wert m hoch n).
21     ...
22 }
```

Können Sie diese Funktion vereinbaren? Natürlich dürfen Sie alle bisher vereinbarten Funktionen (`plus1`, `minus1`, `add`, `mul`) benutzen. Und die Funktion `pot` darf sich natürlich auch selbst aufrufen (wie `add` und `mul`).

Lösungen für die Aufgaben auf dem Arbeitsblatt (für die 2. VL am Mo 05.10.09):**Die rekursive Multiplikationsfunktion mul:**

```
23 int mul(int m, int n) {
24     // Liefert das Produkt von m und n:
25     if (n == 0) {
26         return 0; // m * 0 = 0
27     } else if (n == 1) {
28         return m; // m * 1 = m
29     } else {
30         return add(m, mul(m, minus1(n))); // m * n = m + (m * (n-1))
31     }
32 }
```

Die rekursive Potenzierungsfunktion pot:

```
33 int pot(int m, int n) {
34     // Liefert die n-te Potenz von m ("m hoch n"):
35     if (n == 0) {
36         return 1; // m hoch 0 = 1
37     } else {
38         return mul(m, pot(m, minus1(n))); // m hoch n = m * (m hoch (n-1))
39     }
40 }
```

Wiederholungsfragen, 3. SU, Mo 12.10.09**1. Warum ist die folgende Methode keine *vernünftige rekursive Methode*?**

```
1 static int berechneA(int n) {
2     if (n == 0) {
3         return berechneA(n-1);
4     } else {
5         return berechneA(n+1);
6     }
7 }
```

2. Warum ist die folgende Methode keine *vernünftige rekursive Methode*?

```
8 static int berechneB(int n) {
9     if (n == 0) {
10        return n - 1;
11    } else {
12        return n + 1;
13    }
14 }
```

3. Warum ist die folgende Methode keine *vernünftig rekursive Methode*?

```
15 static berechneC(int n) {
16     if (n == 0) {
17         return 17;
18     } else {
19         return berechneC(n);
20     }
21 }
```

4. Eine Firma stellt *Roboter* her, die sich *selbst reproduzieren* können. Genauer: 3 solche Roboter brauchen 1 Woche, um einen weiteren Roboter ihresgleichen zu bauen. Die Firma beginnt die Produktion *mit 3 Robotern* und setzt alle ihre Roboter zur Produktion weiterer Roboter ein. Wie viele Roboter hat die Firma nach 1 Woche? Nach 2 Wochen? Nach 3 Wochen? Nach 4 Woche? etc.

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

```
22 static long anzRobos(long w) {
23     // Verlaesst sich darauf, dass w gleich oder groesser 0 ist.
24     // Wie viele Roboter hat die Firma nach w Wochen?
25     // Diese Funktion liefert die Antwort.
26     ...
27 }
```

Tip: Stellen Sie sich die folgenden Fragen:

F1. Für welche Wochenzahl w kann man die Anzahl der danach vorhandenen Roboter leicht angeben?

Auch *ganz einfache, triviale* Antworten auf diese Frage können nützlich sein!

F2. Angenommen, die Firma hat schon n Wochen produziert. Jetzt wollen wir herausfinden, wie viele Roboter in *der letzten* dieser n Wochen (in der n -ten Woche) produziert wurden. Welche Zahlen brauchen wir, um diese Anzahl zu berechnen? Wie können wir dieses Zahlen berechnen?

Rückseite der Wiederholungsfragen, 3. SU, Mo 12.10.09 (noch ein Arbeitsblatt zur Rekursion)

Betrachten Sie die folgende (vernünftige :-) *rekursive Funktion*:

```

1  static String ref37(int n) {
2      if (n % 5 == 0) {
3          return "" + n;
4      } else {
5          return n + " " + ref37(n+1);
6      }
7  } // ref37

```

Aufgabe-01: Führen Sie die Funktion `ref37` mit verschiedenen aktuellen Parametern und mit Hilfe der folgenden *Wertetabelle* aus:

n	ref37(n)
...	
14	
15	
16	
17	
18	
19	
20	
21	
22	
23	
24	
25	
26	
...	

Aufgabe-2: Betrachten Sie noch einmal die Funktion `ref37`:

Welche Parameter-Werte n erfordern 0 rekursive Aufrufe?

Welche Parameter-Werte n erfordern 1 rekursiven Aufruf?

Welche Parameter-Werte n erfordern 2 rekursive Aufrufe?

Welche Parameter-Werte n erfordern 3 rekursive Aufrufe?

Welche Parameter-Werte n erfordern 4 rekursive Aufrufe?

Eine Halbordnung der ganzen Zahlen für `ref37`:

```

...
-5 < -6 < -7 < -8 < -9
 0 < -1 < -2 < -3 < -4
 5 <  4 <  3 <  2 <  1
10 <  9 <  8 <  7 <  6
15 < 14 < 13 < 12 < 11
...

```

Bei dieser Halbordnung sind die Zahlen 0 und 4 oder 9 und 2 oder -5 und -4 etc. *unvergleichbar* und die Zahlen ... -5, 0, 5, 10, 15 ... sind *minimale Zahlen*.

Antworten zu den Wiederholungsfragen, 3. SU, Mo 12.10.09

1. Warum ist die folgende Methode keine *vernünftige rekursive Methode*?

```

1 static int berechneA(int n) {
2     if (n == 0) {
3         return berechneA(n-1);
4     } else {
5         return berechneA(n+1);
6     }
7 }

```

Weil sie sich in jedem Fall rekursiv aufruft (egal ob ihr Parameter n gleich 0 ist oder nicht).

2. Warum ist die folgende Methode keine *vernünftige rekursive Methode*?

```

8 static int berechneB(int n) {
9     if (n == 0) {
10        return n - 1;
11    } else {
12        return n + 1;
13    }
14 }

```

Weil sie sich in keinem Fall rekursiv aufruft (egal ob ihr Parameter n gleich 0 ist oder nicht).

3. Warum ist die folgende Methode keine *vernünftig rekursive Methode*?

```

15 static berechneC(int n) {
16     if (n == 0) {
17         return 17;
18     } else {
19         return berechneC(n);
20     }
21 }

```

Weil sie sich im rekursiven Fall (wenn n ungleich 0 ist) mit dem gleichen Parameter n aufruft, mit dem sie selbst aufgerufen wurde (und nicht mit einem kleineren Parameter).

4. Eine Firma stellt *Roboter* her, die sich *selbst reproduzieren* können. Genauer: 3 solche Roboter brauchen 1 Woche, um einen weiteren Roboter ihresgleichen zu bauen. Die Firma beginnt die Produktion *mit 3 Robotern* und setzt alle ihre Roboter zur Produktion weiterer Roboter ein. Wie viele Roboter hat die Firma nach 1 Woche? Nach 2 Wochen? Nach 3 Wochen? Nach 4 Woche? etc.

Es folgt eine rekursive Funktion, die all diese Fragen beantworten kann:

```

22 static long anzRobos(long w) {
23     // Verlaesst sich darauf, dass w gleich oder groesser 0 ist.
24     // Wie viele Roboter hat die Firma nach w Wochen?
25     // Diese Funktion liefert die Antwort.
26     // "LW" steht fuer "letzter Woche"
27
28     if (w == 0) return 3;
29     long anzahlAmAnfangLW = anzRobos(w - 1);
30     long anzahlProduziertInLW = anzahlAmAnfangLW / 3;
31     return anzahlAmAnfangLW + anzahlProduziertInLW;
32 }

```

3. SU Mo 12.10.09

A. Wiederholung

B. Organisation: Für die Übungsgruppen 1a und 1b: Ab jetzt sollte jeder wissen, zu welcher *Übungsgruppe* er gehört (sonst sehen Sie auf Ihrem Test 1 nach oder fragen Sie mich nach der Vorlesung). Kommen Sie in Zukunft nur zu den Tests *Ihrer Übungsgruppe* (1a bzw. 1b), sonst ist Ihr Test evtl. *ungültig*.

Rekursive Aufrufe nur mit kleineren Parametern erlaubt

Betrachten Sie bitte die Rückseite des Blattes mit den Wiederholungsfragen.

Wenn man die Funktion `ref37` mit dem Parameter 19 aufruft, ist das dann ein *einfacher Fall* oder ein *rekursiver Fall*? (ein rekursiver Fall).

Mit welchem Parameter ruft die Funktion sich in diesem Fall rekursiv auf? (mit dem Parameter 20).

Ist der Parameter 20 für `ref37` ein *einfacher* oder ein *rekursiver* Fall? (Ein einfacher Fall).

Also: Wenn man die Funktion `ref37` mit dem Parameter 19 aufruft, ruft sie sich *genau einmal* rekursiv auf (mit dem Parameter 20) und dann ist sie fertig.

Allgemein muss gelten: Wenn man eine rekursive Funktion mit einem Parameter p_1 aufruft und sie sich daraufhin mit einem Parameter p_2 rekursiv aufruft, dann muss p_2 (in einem bestimmten Sinn) *kleiner* sein, als p_1 .

Ist 20 kleiner als 19? Ja! Wir müssen verstehen, in welchem Sinne 20 hier kleiner ist als 19!

Welche Parameterwerte sind für die Funktion `ref37` einfache Fälle?

(Die Parameterwerte ..., -10, -5, 0, 5, 10, 15, 20, ..., d.h. alle die man mit Rest 0 durch 5 teilen kann).

Welche Parameterwerte erfordern genau 1 rekursiven Aufruf?

(Die Parameterwerte -11, -6, -1, 4, 9, 14, 19, ..., d.h. alle die 1 kleiner sind als bei der vorigen Frage).

Für die Funktion `ref37` ist *20 kleiner als 19*, weil der Parameter 20 *weniger rekursive Aufrufe* erfordert als der Parameter 19.

Lösen Sie (schriftlich, am besten zu zweit) die Aufgabe-02 auf dem Arbeitsblatt.

Zusammenfassung der Lösung der Aufgabe-02: Für `rec37` sind die Parameter ..., -9, -4, 1, 6, 11, ... *am größten*, weil sie für jeden dieser Parameter 5 rekursive Aufrufe braucht.

Die Parameter ..., -8, -3, 2, 7, 12, ... sind *etwas kleiner*, weil sie nur 4 rekursive Aufrufe erfordern.

...

Die Parameter ..., -6, -1, 4, 9, 14, ... sind *noch kleiner*, weil sie nur 1 rekursiven Aufruf erfordern.

Die Parameter ..., -5, 0, 5, 10, 15, ... sind *am kleinsten*, weil sie 0 rekursive Aufrufe erfordern.

Für die rekursive Funktion `ref37` ist der Parameter 20 also kleiner als 19, weil 20 weniger rekursive Aufrufe erfordert (nämlich 0) als der Parameter 19 (der 1 rekursiven Aufruf erfordert).

Wer das gründlicher und mathematisch genauer verstehen will, sollte sich mit folgenden Begriffen vertraut machen: Halbordnung, minimale Elemente, strikte Halbordnung, fundierte Menge. Diese Begriffe werden aber hier im Fach MB2-PR2 nicht näher behandelt (nur in der Datei `Stichworte.pdf` findet man ein paar Definitionen und Beispiele dazu).

Halbgeordnete Mengen, fundierte Mengen, total geordnete Mengen

Beispiele für *total geordnete* Mengen:

Die natürlichen Zahlen mit der Relation kleiner-gleich

Die rationalen Zahlen mit der Relation kleiner-gleich

Die Menge aller Strings mit der Relation ist-kürzer-oder-gleich-lang

Die Menge aller Strings mit der Relation ist-lexikographisch-kleiner-oder-gleich

"total" bedeutet hier:

Nimmt man zwei beliebige Elemente a und b der betreffenden Menge, dann gilt

entweder a kleiner-gleich b

oder b kleiner-gleich a

oder beides

Manche Mengen sind bezüglich bestimmter Relationen "ein bisschen geordnet", aber nicht total:

Beispiele für halbgeordnete Mengen:

Die Menge aller Strings mit der Relation ist-ein-Teilstring-von

Die natürlichen Zahlen mit der Relation ist-ein-Teiler-von

Die Menge der Java-Referenztypen mit der Relation ist-Untertyp-von

Bezüglich der Relation ist-ein-Teilstring-von sind die Strings "BC" und "ABCDE" *vergleichbar* und es gilt: "BC" ist-ein-Teilstring-von "ABCDE".

Dagegen sind die Strings "BC" und "DEF" *unvergleichbar*, weil

weder "BC" ist-ein-Teilstring-von "DEF"

noch "DEF" ist-ein-Teilstring-von "BC"

gilt.

Ganz entsprechend gilt für Java-Referenz-Typen: Object und String sind *vergleichbar*, weil String ein Untertyp von Object ist. Dagegen sind String und Integer *unvergleichbar*, weil keiner dieser beiden Typen ein Untertyp des anderen ist.

Halbgeordnete Mengen: Eine Menge mit einer zweistelligen Relation kg ("kleiner oder gleich") ist *halbgeordnet*, wenn für alle Elemente a, b, c gilt:

kg ist transitiv: Wenn $a kg b$ und $b kg c$ dann auch $a kg c$

kg ist reflexiv: $a kg a$

kg ist antisymmetrisch: ($a kg b$ und $b kg a$) gilt nur wenn a gleich b ist.

Minimale Elemente: Ein Element m (einer halbgeordneten Menge M) heißt minimal, wenn es in M kein Element n gibt, welches noch kleiner als m ist.

Achtung: Diese Definition sagt *nicht*, dass ein minimales Element kleiner als alle anderen sein muss. Ein minimales Element kann mit anderen Elementen unvergleichbar sein. Nur wenn es vergleichbar ist, muss es kleiner sein.

Fundierte Menge: Eine halbgeordnete Menge heißt fundiert, wenn jede nichtleere Teilmenge (mindestens) ein minimales Element enthält. Oder: Wenn es keine unendliche Ketten von Elementen gibt, von denen jedes kleiner als das vorhergehende ist.

Beispiel: Die natürlichen Zahlen (0, 1, 2, ...) mit der Relation ist-kleiner-als ist fundiert (denn zu jeder natürlichen Zahl gibt es nur endlich viele andere natürliche Zahlen, die kleiner sind).

Gegenbeispiel: Die Menge der ganzen Zahlen mit der Relation ist-kleiner-als (kurz: $<$) ist nicht fundiert, weil es z.B. die unendliche Kette $-17 < -20 < -23 < -26 < -29 < -32 < \dots$ gibt.

Zur Entspannung: Die Sütterlin-Schrift und der Mond

In Deutschland gab es oft langwierige Diskussionen und Streit darüber, welche Schrift benutzt und in den Schulen gelehrt werden sollte. 1911 entwarf Ludwig Sütterlin im Auftrag des preußischen Kultusministeriums zwei Schriften. Eine davon wurde 1915 an preußischen Schulen und ab 1930 in ganz Deutschland eingeführt. 1941 wurde diese Sütterlin-Schrift von Hitler verboten. Ab 1954 wurde sie an einigen Schulen wieder gelehrt, wurde aber von der lateinischen Schreibschrift verdrängt.

Im Gedicht "Der Mond" von Christian Morgenstern (1871-1914, München-Meran) spielen die Großbuchstaben A und Z in Sütterlin (\mathcal{A} und \mathcal{Z}) eine wichtige Rolle:

Als Gott den lieben Mond erschuf	/	Gab er ihm folgenden Beruf:
Beim Zu- sowohl wie beim Abnehmen	/	Sich deutschen Lesern zu bequemem
ein \mathcal{A} formierend und ein \mathcal{Z} -	/	daß keiner groß zu denken hätt.
Befolgend dies ward der Trabant	/	ein völlig deutscher Gegenstand.

Sammlungen (engl. collections)

Eine kleine Wiederholung aus MB1-PR1:

Angenommen, wir brauchen in einem Programm 500 `double`-Variablen. Welche Vorteile hat es, wenn wir diese Variablen in Form einer *Reihung* vereinbaren, etwa so:

```
double dr = new double[500];
```

statt die Variablen einzeln zu vereinbaren und mit Namen zu versehen, etwa so:

```
double d0, d1, d2, d3, ..., d499;
```

(Die Vereinbarung der Reihung ist viel einfacher und leichter veränderbar. Jede Bearbeitung der 500 einzelnen Variablen erfordert 500 Befehle. Die Reihung kann man mit Schleifen bearbeiten).

Nachteil einer Reihung im Vergleich zu einzelnen Variablen? (Man kann den Reihungskomponenten keine *individuellen Namen* geben).

In Aufgabe 2 (MVC-Rechner) kommt so ein ähnliches Problem vor: Dort muss man 7 `JTextField`-Objekte erzeugen und bearbeiten. Wenn der *Programmierer* gleich eine Reihung mit 7 `JTextField`-Komponenten vereinbart, kann er sich und dem *Warter* ein paar Code-Wiederholungen ersparen.

Noch ein Nachteil von Reihungen? Die Länge einer Reihung muss bei ihrer Erzeugung festgelegt werden und kann danach nicht mehr geändert werden ("Reihungen sind aus Beton").

Sammlungen haben Ähnlichkeit mit Reihungen, bieten dem Programmierer aber *mehr Komfort* oder sind *für bestimmte Anwendungen optimiert*.

Zur Erinnerung:

Def.: Eine *Sammlung* ist ein Objekt, in dem man andere Objekte sammeln (d.h. hineintun, darin suchen und wieder entfernen) kann.

Def: Eine Sammlung ist ein `Collection`-Objekt (d.h. ein Objekt einer Klasse, die die Schnittstelle `Collection` implementiert).

Zur Zeit (in Java 6) gibt es in der Java-Standardbibliothek 31 Sammlungsklassen. Warum so viele?

Für einige Anwendungen ist das *Einfügen* von Komponenten besonders wichtig (und sollte besonders schnell gehen), das Suchen und Entfernen von Komponenten ist dagegen nicht so wichtig.

Für andere Anwendungen ist das *Suchen* besonders wichtig. Das Einfügen darf ruhig ein bisschen mehr Zeit kosten, wenn dafür das Suchen schneller geht.

Einige Anwendungen wollen, dass eine Sammlung *beliebig vergrößert* werden kann. Andere wollen, dass Sammlungen nur bis zu einer bestimmten Größe wachsen können und sich dann gegen weitere Vergrößerungen sträuben (sie z.B. die Klasse `ArrayBlockingQueue`).

Einige Anwendungen wollen mit *Indizes* auf die Komponenten einer Sammlung zugreifen (ähnlich wie bei Reihungen). Andere wollen nur "oben auf der Sammlung" Komponenten hinzufügen oder entfernen (*Stapel*, stack). Wieder andere wollen Komponenten "vorn" einfügen und "hinten" wieder entfernen (*Schlange*, queue).

Einige für Sammlungen wünschenswerte Eigenschaften konkurrieren miteinander oder schließen sich sogar aus: Wenn man möglichst *wenig Speicherplatz* verbrauchen will, muss man meistens bei der *Geschwindigkeit* (des Einfügens und Suchens etc.) Kompromisse machen. Wenn das *Suchen sehr schnell* gehen soll, muss man beim *Einfügen* meistens *mehr Zeit investieren* etc.

Analogie: Es ist kaum möglich, ein billiges, besonders umweltfreundliches, schnelles und bequemes Auto zu bauen. Autobauer suchen nach "guten Kompromißen" zwischen diesen Eigenschaften.

Für viele Anwendungen ist es sehr wichtig, ihre (umfangreichen) Daten in Sammlungen abzulegen, die *genau die richtige Kombination von Eigenschaften* haben. Deshalb gibt es so viele Sammlungsklassen. Und ab und zu muss ein Programmierer für eine spezielle Anwendung noch eine *neue Sammlungsklasse* mit ganz speziellen Eigenschaften entwickeln (meist als Unterklasse einer Standardklasse).

Alle Java-Standard-Sammlungsklassen sind generisch

Man kann den Typ der zu sammelnden Objekte also einschränken. In einer Sammlung des Typs `ArrayList<String>` kann man nur `String`-Objekte sammeln, in einer Sammlung vom Typ `ArrayList<JButton>` nur `JButton`-Objekte. Solche Einschränkungen sind ein Vorteil, bestimmte Flüchtigkeitsfehler werden vom Ausführer schon bei der Übergabe des Programms erkannt, nicht erst später bei der Ausführung (oder gar nicht).

Rohe Typen wie `ArrayList` (ohne `<String>` oder `<Integer>` dahinter) sollte man nur verwenden, wenn es *unbedingt* sein muss (beim Erweitern von "alten Java-Programmen", die mit einer Java-Version vor Version 5 geschrieben wurden).

In einer Sammlung eines rohen Typs wie z.B. `ArrayList` kann man alle möglichen Objekte sammeln!

Achtung: Der rohe Typ `ArrayList` und der parametrisierte Typ `ArrayList<Object>` sind *nicht* gleich!

Der rohe Typ `ArrayList` ist ein *Obertyp* der parametrisierten Typen `ArrayList<String>`, `ArrayList<Integer>`, ... etc.

Der parametrisierte Typ `ArrayList<Object>` ist *weder Ober- noch Untertyp* der anderen parametrisierten Typen `ArrayList<String>`, `ArrayList<Integer>`, ... etc.

Wiederholungsfragen, 4. SU, Mo 19.10.09

1. Angenommen, eine rekursive Funktion `rf` wurde mit einem (formalen) Parameter vom Typ `int` vereinbart. Bei einer Ausführung des betreffenden Programms wird die Funktion `rf` mit einem (aktuellen) Parameter `n` aufgerufen und ruft sich darauf mit einem Parameter `m` auf. Was muss dann für die Parameter `n` und `m` gelten?

2. Was ist eine *Sammlung*? Geben Sie *zwei* Definitionen an (eine "inhaltliche" und eine "formale").

3. Welchen Vorteil haben *parametrisierte Typen* wie `Set<Integer>`, `Set<JTextField>`, ... etc. im Vergleich zum *rohen Typ* `Set`?

4. Was ist der Unterschied zwischen dem *parametrisierten Typ* `Set<Object>` und dem *rohen Typ* `Set`?

5. Schreiben Sie eine rekursive Methode entsprechend der folgenden Spezifikation:

```
1 static void printNeg(int n) {
2     // Falls n kleiner oder gleich 0 ist werden alle int-Werte
3     // zwischen n und 0 (einschliesslich) durch je ein Blank " " getrennt
4     // ausgegeben (mit der Methode p). Sonst wird nichts ausgegeben.
5     // Beispiel 1: Der Aufruf printNeg(-3) bewirkt folgende Ausgabe:
6     // -3 -2 -1 0
7     // Beispiel 2: Der Aufruf printNeg(0) bewirkt folgende Ausgabe:
8     // 0
9     // Beispiel 3: Der Aufruf printNeg(3) bewirkt keine Ausgabe.
10
11
12
13
14
15
16
17
18
19
20 }
```

Korrektur-Hinweis: Beim 3. SU am Mo 12.10.09 stand auf der Rückseite des Blattes mit den Wiederholungsfragen eine *Halbordnung der natürlichen Zahlen für ref37*. Diese Halbordnung war *völlig falsch*. In der Datei `Stichworte.pdf` steht jetzt an ihrer Stelle eine (hoffentlich korrekte) *Halbordnung der ganzen Zahlen für ref37*. Dank an die Studenten, die mich auf den Fehler aufmerksam gemacht haben.

Rückseite der Wiederholungsfragen, 4. SU, Mo 19.10.09

Die Methoden der Schnittstelle Collection<K>:

```

1  boolean      add      (K      k )           // opt
2  boolean      remove   (Object ob)          // opt
3  boolean      contains (Object ob)
4
5  boolean      addAll   (Collection<? extends K> c) // opt
6  boolean      removeAll (Collection<?> c) // opt
7  boolean      containsAll (Collection<?> c)
8  boolean      retainAll (Collection<?> c) // opt
9  void         clear    ()           // opt
10
11 int          size     ()
12 Iterator<K>  iterator ()
13 boolean      isEmpty ()
14
15 boolean      equals   (Object ob)
16 int          hashCode ()
17 Object[]     toArray ()
18 <T> T[]      toArray  (T[] rei)

```

Die Schnittstelle Set<K> erweitert die Schnittstelle Collection<K> um null Methoden.

Die Schnittstelle Queue<K> erweitert die Schnittstelle Collection<K> um folgende Methoden:

```

1  K            element  ()
2  boolean      offer   (K k)
3  K            peek    ()
4  K            poll    ()
5  K            remove  ()

```

Die Schnittstelle List<K> erweitert die Schnittstelle Collection<K> um folgende Methoden:

```

1  void         add      (int index, K k);           //opt
2  K            set      (int index, K k);           //opt
3  K            remove   (int index);               //opt
4  K            get      (int index);
5  int          indexOf  (Object ob);
6  int          lastIndexOf (Object ob);
7
8  boolean      addAll   (int index,
9                        Collection<? extends K> c) //opt
10 List<K>      subList  (int vonIndex, int bisIndex);
11
12 ListIterator<K> listIterator();
13 ListIterator<K> listIterator(int index);

```

Die Schnittstelle SortedSet<K> erweitert die Schnittstelle Set<K> um folgende Methoden:

```

1  K            first    ();
2  K            last     ();
3
4  SortedSet<K> headSet  (K bisKompo);
5  SortedSet<K> tailSet  (K vonKompo);
6  SortedSet<K> subSet   (K vonKompo, K bisKompo);
7
8  Comparator<? super K> comparator ();

```

Antworten zu den Wiederholungsfragen, 4. SU, Mo 19.10.09

1. Angenommen, eine rekursive Funktion `rf` wurde mit einem (formalen) Parameter vom Typ `int` vereinbart. Bei einer Ausführung des betreffenden Programms wird die Funktion `rf` mit einem (aktuellen) Parameter `n` aufgerufen und ruft sich darauf mit einem Parameter `m` auf. Was muss dann für die Parameter `n` und `m` gelten?

Der Parameter `m` muss bezüglich einer Halbordnung kleiner sein als `n`.

2. Was ist eine Sammlung? Geben Sie zwei Definitionen an (eine "inhaltliche" und eine "formale").

Def. 1: Eine Sammlung ist ein Objekt, in dem man Objekte sammeln kann (d.h. in das man Objekte einfügen, aus dem man Objekte entfernen und in dem Objekte suchen kann).

Def. 2: Eine Sammlung ist ein Objekt einer Sammlungsklasse (d.h. einer Collection-Klasse).

3. Welchen Vorteil haben parametrisierte Typen wie `Set<Integer>`, `Set<JTextField>`, ... etc. im Vergleich zum rohen Typ `Set`?

Sie ermöglichen es dem Ausführer, bestimmte Flüchtigkeitsfehler schon bei der Übergabe des Programms ("zur Compilezeit") zu entdecken. Beispiel für einen solchen Flüchtigkeitsfehler: Der Programmierer versucht, ein String-Objekt in eine Sammlung des Typs `Set<Integer>` einzufügen. Verwendet man Sammlungen eines rohen Typs wie `Set` kann der Ausführer solche Fehler nicht entdecken.

4. Was ist der Unterschied zwischen dem parametrisierten Typ `Set<Object>` und dem rohen Typ `Set`?

Der rohe Typ `Set` ist ein Obertyp aller parametrisierten Typen wie `Set<Integer>`, `Set<JTextField>`, `Set<Object>`, Der parametrisierte Typ `Set<Object>` ist weder Ober- noch Untertyp anderer parametrisierten Typen wie `Set<Integer>` oder `Set<JTextField>` etc.

5. Schreiben Sie eine rekursive Methode entsprechend der folgenden Spezifikation:

```
1 static void printNeg(int n) {
2     // Falls n kleiner oder gleich 0 ist werden alle int-Werte
3     // zwischen n und 0 (einschliesslich) durch je ein Blank " " getrennt
4     // ausgegeben (mit der Methode p). Sonst wird nichts ausgegeben.
5     // Beispiel 1: Der Aufruf printNeg(-3) bewirkt folgende Ausgabe:
6     // -3 -2 -1 0
7     // Beispiel 2: Der Aufruf printNeg(0) bewirkt folgende Ausgabe:
8     // 0
9     // Beispiel 3: Der Aufruf printNeg(3) bewirkt keine Ausgabe.
10
11     if (n > 0) return;
12     if (n == 0) {
13         p(0);
14     } else {
15         p(n);
16         printNeg(n+1);
17     }
18
19 }
```

4. SU Mo 19.10.09

A. Wiederholung

B. Organisation : Tutorien (Frau Nusch):

Do 14.15-15.45 Uhr, B240, für Zug 2

Do 16.00-17.30 Uhr, B240, für Zug 1

Sammlungsklassen in Java sind keine Einzelgänger. Sie bilden ein System.

Die folgenden drei Regeln verbinden Java-Sammlungsklassen zu einem System:

Regel 1: Jede Sammlungsklasse $SK\langle K \rangle$ muss ("als Minimum" oder als "gemeinsame Grundlage") eine bestimmte *Schnittstelle* (`java.util.Collection\langle K \rangle`) implementieren.

Regel 2: Jede Sammlungsklasse $SK\langle K \rangle$ sollte einen *Konstruktor* mit einem Parameter c des Typs `Collection\langle ? extends K \rangle` haben, und die Komponenten der Sammlung c in die neue Sammlung (die gerade initialisiert wird) einfügen.

Damit ist folgendes möglich ($SK1$ und $SK2$ sollen zwei *beliebige Sammlungsklassen* sein):

```
1 SK1<String> sob1 = new SK1<String>(...);
2 ... die Sammlung sob1 wird irgendwie bearbeitet ...
3 SK2<String> sob2 = new SK2<String>(sob1);
```

Der Befehl in Zeile 3 erzeugt aus der $SK1$ -Sammlung $sob1$ eine $SK2$ -Sammlung $sob2$, die die selben Komponenten enthält wie $sob1$ (ansonsten aber ganz andere Eigenschaften haben kann als $sob1$).

Der *Programmierer* kann also ganz leicht aus einer Sammlung eines Typs $SK1\langle String \rangle$ eine Sammlung eines anderen Typs $SK2\langle String \rangle$ erzeugen. Für den *Ausführer* kann der Befehl in Zeile 3 allerdings ziemlich aufwendig sein (wenn $sob1$ sehr groß ist).

Regel 3: Jede Sammlung S muss 4 Methoden (namens `addAll`, `removeAll`, `containsAll` und `retainAll`) enthalten, die eine Sammlung P als Parameter und bearbeiten S mit Hilfe von P .

Sammlungen und Reihungen

Vorbereitung: `List` ist eine Erweiterung der Schnittstelle `Collection`. `List`-Sammlungen bieten "ein bisschen mehr Komfort" als "nur-`Collection`-Sammlungen". Später mehr dazu.

1. Reihungen sind (genau genommen) keine Sammlungen, aber ...

Der Typ `ArrayList\langle String \rangle` *ist ein* Untertyp des Typs `Collection\langle String \rangle`, aber der Typ `String[]` *ist kein* Untertyp des Typs `Collection\langle String \rangle`.

Aber es ist leicht möglich, Reihungen als (`List`-) Sammlungen *zu betrachten*, etwa so:

```
4 String[] sr = {"aa", "bb", "cc"};
5 List<String> lr = Arrays.asList(sr);
```

sr ist eine Reihung und lr ist eine `List`-Sammlung von `String`-Objekten. Aber lr ist keine Kopie von sr , sondern eine *Sammlungs-Sicht* (engl. a collection view) auf die Reihung sr . Das bedeutet:

Wenn man die Reihung sr ändert, ändert man damit auch die Sammlung lr , und wenn man die Sammlung lr ändert, ändert man damit auch die Reihung sr .

```
6 sr [0] = "xx"; // Die Reihung sr wird veraendert
7 lr.set(2, "yy"); // Die Sammlung lr wird veraendert
8
9 pln("sr: " + Arrays.toString(sr)); // sr wird ausgegeben
10 pln("lr: " + lr); // lr wird ausgegeben
```

Auf dem Bildschirm erscheint:

```
sr: [xx, bb, yy]
lr: [xx, bb, yy]
```

Siehe auch S. 461, Beispiel-01 im Buch. Vorsicht, Druckfehler:

statt	sLst in Zeile 17	rS[0] in Zeile 20	lS.set(2, in Zeile 22
sollte es heißen	sLst	sRei[0]	sLst.set(2,

2. Sammlungen können relativ leicht in Reihungen umgewandelt werden, aber ...

In der Schnittstelle `Collection<K>` würde man folgende Methode erwarten und sich wünschen:

```
K[] toArray()
```

Die ist aber leider nicht möglich (die Gründe dafür haben mit generischen Einheiten in Java zu tun und sind nicht ganz leicht zu erklären). Deshalb hat man als Ersatz die folgenden beiden Methoden vereinbart:

```
Object[] toArray()
<T> T[] toArray(T[] r)
```

Diese Methoden gibt es also in jedem Sammlungs-Objekt. Beide haben Vor- und Nachteile:

Methode	Vorteil	Nachteil
<code>Object[] toArray()</code>	keine Parameter	Rückgabotyp <code>Object[]</code>
<code><T> T[] toArray(T[] r)</code>	Rückgabotyp <code>T[]</code>	Angabe eines Parameters vom Typ <code>T[]</code> nötig

Erläuterung: Die zweite Methode ist eine generische Methode mit einem eigenen Typ-Parameter `T`. Für den darf man irgendeinen (Referenz-) Typ einsetzen, unabhängig vom Typ für den Parameter `K`.

Mit der zweiten Methode kann man also z.B. versuchen, aus einer Sammlung des Typs `ArrayList<Object>` (für den Typparameter `K` wurde der Typ `Object` eingesetzt) eine Reihung vom Typ `String[]` zu erzeugen (für den Typparameter `T` haben wir `String` eingesetzt). Das gelingt natürlich nur, wenn die Sammlung ausschließlich `String`-Objekte enthält. Andernfalls wird eine `ClassCastException` geworfen.

3. Reihungen können auch primitive Komponenten haben, Sammlungen nicht!

Die Schnittstelle `Collection` (S. 441 bzw. auf der Rückseite der Wiederholungsfragen)

Als Beispiel betrachten wir die folgende Sammlung:

```
1 Collection<String> sam = new ArrayList<String>();
```

In der Sammlung `sam` gibt es dann garantiert auch eine `add`-Methode. Welchen Rückgabotyp hat die? (`boolean`) Wie viele Parameter hat sie und zu welchen Typen gehören die? (1 vom Typ `String`).

Was bewirkt ein Methodenaufruf wie `sam.add(otto)` wohl? Offizielle Formulierung:

Die `add`-Methode muss sicherstellen, dass `otto` sich nachher in `sam` befindet.

Falls die Methode `add` das nicht erreichen kann, muss sie eine Ausnahme werfen!

Wann liefert sie `true` und wann `false`?

`true` wenn sie die Sammlung (durch das Einfügen von `otto`) verändert hat, `false` sonst (wenn `otto` schon in der Sammlung war und Doppelgänger verboten sind).

Ganz Entsprechendes gilt auch für die anderen "verändernden Methoden" mit Rückgabotyp `boolean`. (`remove`, `addAll`, `removeAll`, `retainAll`, `clear`)

Anmerkung: Die Schnittstelle `Collection` muss von *allen* Sammlungsklassen implementiert werden. Die Beschreibung der `add`-Methode in der Schnittstelle muss also für alle möglichen Sammlungen "passen", z.B. für Sammlungen

bei denen *Doppelgänger* erlaubt bzw. verboten sind

bei denen das Einfügen von `null` erlaubt bzw. verboten ist

...

Was macht wohl die Methode `remove`? Wann liefert sie `true`, wann `false`? Was ist merkwürdig an dieser Methode? Genauer: An ihrem Parameter? (Er ist nicht vom Typ `K`, sondern vom Typ `Object`).

Es ist also erlaubt zu versuchen, aus der Sammlung `sam` z.B. ein `JButton`-Objekt zu entfernen (obwohl `JButton`-Objekte in `sam` gar nicht vorkommen dürfen).

Grundsatz: Die Schnittstelle versucht, die Methoden so *wenig* einzuschränken, wie möglich. Warum sollte man nicht versuchen dürfen, ein `JButton`-Objekt aus `sam` zu entfernen?

Denkbar: Eine Reihung, in der sich viele `String`-Objekte und ein paar andere Objekte befinden. Es ist günstig, wenn man die Methode `sam.remove(...)` auf *alle* Komponenten dieser Reihung anwenden darf.

Von welchem Typ ist der Parameter der Methode `sam.removeAll`? (von irgendeinem `Collection`-Typ, das Fragezeichen `?` darf durch irgendeinen Komponententyp ersetzt werden (z.B. durch `String` oder `JButton` oder ...).

Von welchem Typ ist der Parameter der Methode `sam.addAll`?

Vom Typ `Collection<? extends String>`

Der Komponententyp der Sammlung, die als Parameter angegeben wird, muss also eine Erweiterung des Typs `String` sein.

Zur Erinnerung: Der Typ `String` gilt auch als eine Erweiterung des Typs `String`!

Nach einem Methodenaufruf wie `sam.retainAll(emil)` enthält `sam` den *Durchschnitt* der Sammlungen `sam` und `emil`. Wie bei `removeAll` darf `emil` eine Sammlung mit beliebigem Komponententyp sein.

Beispiel für eine Anwendung der Methode `iterator`:

```
2 Iterator<String> it = sam.iterator();
3
4 while (true) {
5     if (!it.hasNext()) break;
6     String s = it.next();
7     ... // s irgendwie bearbeiten
8 }
```

Zur Entspannung: Das Ziegen-Problem

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

Statt die Tür gleich zu öffnen, versucht der Showmaster die Spannung zu heben, mit der Frage: "Sind Sie sicher, dass die Tür 1 richtig ist?". Und dann bietet er an: "Sehen Sie, ich zeige ihnen mal was!" und öffnet eine der beiden nicht-gewählten Türen, z. B. Tür 2. Dahinter steht eine Ziege.

Jetzt kommt die entscheidende Frage des Showmasters an 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?

Wiederholungsfragen, 5. SU, 26.10.09

1. `ArrayList<K>` und `TreeSet<K>` sind zwei Sammlungsklassen in der Java-Standardbibliothek. Angenommen, Sie haben (in einem Java-Programm) eine Sammlung namens `samA` vom Typ `ArrayList<String>`. Wie können Sie daraus eine Sammlung namens `samT` erzeugen, die zum Typ `TreeSet<String>` gehört und dieselben `String`-Komponenten enthält wie `samA`?
2. Jede Sammlungsklasse `S<K>` sollte einen Konstruktor haben, der eine *Sammlung* als Parameter erwartet. Geben Sie den genauen Typ dieses Parameters an.
3. Jede Sammlungsklasse `S<K>` muss unter anderem zwei (Objekt-) Methoden namens `addAll` und `removeAll` mit je einem Parameter vereinbaren (oder erben).
Zu welchem Typ gehört der Parameter von `addAll`?
Zu welchem Typ gehört der Parameter von `removeAll`?
4. In der Schnittstelle `Collection<K>` sind 6 Methoden als *optional* gekennzeichnet. Was haben diese Methoden gemeinsam? Was machen sie alle?
5. Zu den optionalen Methoden in der Schnittstelle `Collection<K>` gehört auch die (parameterlose) Methode `clear`, die normalerweise die aktuelle Sammlung *leer macht* (d.h. alle Komponenten entfernt). Angenommen, Sie vereinbaren eine neue Sammlungsklasse `S<K>`. Wie dürfen Sie die Methode `clear` vereinbaren, weil sie *optional* ist? Ergänzen Sie das folgende Methoden-Skelett:

```
public void clear() {  
  
  
}
```

Tip: Falls Sie jetzt nach dem Namen einer bestimmten Ausnahmeklasse suchen, ist es wahrscheinlich die Klasse `UnsupportedOperationException`.

6. Sei `samX` eine Sammlung des Typs `ArrayList<String>`, die die drei Komponenten "A", "B" und "C" enthält.

Sei `samY` eine Sammlung des Typs `TreeSet<String>`, die die vier Komponenten "A", "C", "D" und "E" enthält.

Was gibt dann die folgende Befehlsfolge aus:

```
1 samX.retainAll(samY);  
2 pln(samX);
```

Rückseite der Wiederholungsfragen, 5. SU, Mo 26.10.09

Eine Klasse Mango so vereinbaren, dass Mango-Objekte sortiert werden können

```

1 class Mango implements ... {
2     int    saftmenge;
3     char   guete;
4     double preis;
5     ...
6 }
```

In dieser Klasse könnten wir einige der folgenden Methoden vereinbaren:

```

1 public boolean isLessThan      (Mango that) { ... }
2 public boolean isLessOrEquals (Mango that) { ... }
3 public boolean isEqual       (Mango that) { ... }
4 public boolean isNotEquals    (Mango that) { ... }
5 public boolean isGreaterOrEquals (Mango that) { ... }
6 public boolean isGreaterThan  (Mango that) { ... }
```

Wie viele dieser Methoden müssten wir mindestens implementieren?

Angenommen, wir haben zwei Mango-Objekte *ma* und *mb*. Wie viele dieser Methoden müssten wir *auf-rufen*, um *alle drei* möglichen Vergleichsergebnisse (*ma* ist *kleiner* *mb*, *ma* ist *gleich* *mb*, *ma* ist *größer* als *mb*) zu unterscheiden?

Bei großen Objekten (z.B. bei String-Objekten, die 10 Mega Zeichen enthalten) kann *ein einziger* Vergleich ziemlich *teuer* sein. Man hat deshalb nach Methoden gesucht, bei denen man mit *einem einzigen* (teuren) Aufruf *alle drei* Vergleichsergebnisse unterscheiden kann. Solche Methoden werden von den Schnittstellen `Comparable<T>` und `Comparator<T>` beschrieben:

```

7 public interface Comparable<T> {
8     public int compareTo(T that);
9     // Liefert eine negative Zahl bzw. 0 bzw. eine positive Zahl,
10    // wenn this kleiner bzw. grösser bzw. gleich that ist.
11 }
12
13 public interface Comparator<T> {
14     public int compare(T tob1, T tob2);
15     // Liefert eine negative Zahl bzw. 0 bzw. eine positive Zahl,
16     // wenn tob1 kleiner bzw. grösser bzw. gleich tob2 ist.
17 }
```

Hinweis: Im Buch auf S. 227, im Beispiel-03, werden zwei String-Objekte miteinander verglichen.

Wie kann man Mango-Objekte *vergleichbar* machen?

Möglichkeit 1: Man implementiert in der Klasse Mango die Schnittstelle `Comparable<Mango>`.

Ein Vergleich der Mango-Objekte *ma* und *mb* sieht dann z.B. so aus:

```
18 int erg = ma.compareTo(mb);
```

Möglichkeit 2: Man implementiert in einer beliebigen Klasse XXX die Schnittstelle `Comparator<Mango>` und erzeugt ein Objekt *xxx* der Klasse XXX.

Ein Vergleich der Mango-Objekte *ma* und *mb* sieht dann z.B. so aus:

```
19 int erg = xxx.compare(ma, mb);
```

Von der Möglichkeit 1 kann man nur *einmal* Gebrauch machen. Die dadurch definierte totale Ordnung (z.B. für Mango-Objekte) wird auch als *die natürliche Ordnung* (für Mango-Objekte) bezeichnet.

Von der Möglichkeit 2 kann man *beliebig oft* Gebrauch machen. Die dadurch definierten totalen Ordnungen (z.B. für Mango-Objekte) bezeichnen wir als *sonstige Ordnungen* (für Mango-Objekte).

Antworten zu den Wiederholungsfragen, 5. SU, 26.10.09

1. `ArrayList<K>` und `TreeSet<K>` sind zwei Sammlungsklassen in der Java-Standardbibliothek. Angenommen, Sie haben (in einem Java-Programm) eine Sammlung namens `samA` vom Typ `ArrayList<String>`. Wie können Sie daraus eine Sammlung namens `samT` erzeugen, die zum Typ `TreeSet<String>` gehört und dieselben `String`-Komponenten enthält wie `samA`?

`TreeSet<String> samT = new TreeSet<String>(samA);`

2. Jede Sammlungsklasse `S<K>` sollte einen Konstruktor haben, der eine *Sammlung* als Parameter erwartet. Geben Sie den genauen Typ dieses Parameters an.

`Collection<? extends K>`

3. Jede Sammlungsklasse `S<K>` muss unter anderem zwei (Objekt-) Methoden namens `addAll` und `removeAll` mit je einem Parameter vereinbaren (oder erben).

Zu welchem Typ gehört der Parameter von `addAll`? **Zum Typ `Collection<? extends K>`**

Zu welchem Typ gehört der Parameter von `removeAll`? **Zum Typ `Collection<?>`**

4. In der Schnittstelle `Collection<K>` sind 6 Methoden als *optional* gekennzeichnet. Was haben diese Methoden gemeinsam? Was machen sie alle?

Die optionalen Methoden verändern (normalerweise) die betreffende Sammlung.

5. Zu den optionalen Methoden in der Schnittstelle `Collection<K>` gehört auch die (parameterlose) Methode `clear`, die normalerweise die aktuelle Sammlung *leer macht* (d.h. alle Komponenten entfernt). Angenommen, Sie vereinbaren eine neue Sammlungsklasse `S<K>`. Wie dürfen Sie die Methode `clear` vereinbaren, weil sie optional ist? Ergänzen Sie das folgende Methoden-Skelett:

```
public void clear() {
    throw new UnsupportedOperationException("Clear wird hier " +
        "nicht unterstuetzt!");
}
```

6. Sei `samX` eine Sammlung des Typs `ArrayList<String>`, die die drei Komponenten "A", "B" und "C" enthält.

Sei `samY` eine Sammlung des Typs `TreeSet<String>`, die die vier Komponenten "A", "C", "D" und "E" enthält.

Was gibt dann die folgende Befehlsfolge aus:

```
1 samX.retainAll(samY);
2 pln(samX);
```

Ausgabe: [A, C] (das ist der *Durchschnitt* der Sammlungen `samX` und `samY` bevor Zeile 1 ausgeführt wurde)

5. SU Mo 26.10.09

A. Wiederholung

B. Organisation: Klausurtermin: Mo 08.02.2010, ab 12 Uhr, im Beuth-Saal

Anfangskommentare für Methoden

1. Jede Methode sollte mit einem Anfangskommentar versehen werden (ganz wenige Ausnahmen)
2. Dieser AK sollte beschreiben, *was* die Methode macht, nicht *wie* sie es macht.
3. Der AK sollte (in aller Regel) Regel nicht beschreiben, ob die Methode andere Methoden aufruft oder die betreffende Arbeit selbst erledigt (weil das zum *wie* gehört, nicht zum *was*).
4. Der AK sollte unmittelbar vor dem Rumpf der Methode (vor der öffnenden geschweiften Klammer) stehen. Bevor die LeserIn den AK liest, sollte sie gelesen haben und wissen:
 - Wie heißt die Methode?
 - Welchen Rückagebetyp hat sie?
 - Anzahl, Namen und Typen der Parameter?

Der AK sollte die Antworten auf diese Fragen als bekannt voraussetzen und nicht wiederholen!

5. Der AK kann in vielen Fällen mit einem Verb (Tätigkeitswort) anfangen, z.B. "Gibt ... aus", "Fügt ... ein", "Löscht ..." etc. Funktionen können in vielen Fällen mit dem Verb "Liefert" anfangen.

6. Jeder Parametername sollte (in aller Regel) im AK *mindestens einmal vorkommen!*

7. Wichtige Vokabeln: Eingeben, übergeben, ausgeben, zurückgeben:

Der Benutzer gibt Daten ein (z.B. über eine Tastatur), ein Programm liest Daten ein (z.B. von einer Tastatur oder aus einer Datei etc.) und gibt Daten aus (z.B. zu einem Bildschirm oder in eine Datei). Der Programmierer übergibt Parameter an ein Unterprogramm, eine Funktion gibt ein Ergebnis zurück (oder: liefert ein Ergebnis).

Wozu optionale Methoden?

In der Schnittstelle `Collection` sind 6 Methoden als *optional* gekennzeichnet. Diese Methoden darf man so implementieren, dass sie *nicht richtig funktionieren*, sondern nur eine Ausnahme des Typs `UnsupportedOperationException` werfen.

Was haben diese 6 Methoden gemeinsam? (Sie *verändern* die Sammlung).

Problem: Manchmal will man eine kostbare Sammlung `sam` einer Methode `met` als Parameter übergeben, will aber auf keinen Fall, dass `sam` von `met` verändert wird.

Lösung: In der Klasse `Collections` (mit `s`, nicht verwechseln mit der Schnittstelle `Collection` ohne `s`!) gibt es eine Klassen-Methode

```
static public <T> Collection<T> unmodifiableCollection(Collection<? extends T> c)
```

die liefert einem eine "unveränderbare Sicht" auf die Sammlung, die man ihr als Parameter übergibt. Diese unveränderbare Sicht ist ein Objekt, welches die Sammlung `c` enthält, alle Aufrufe von verändernden Methoden mit einer `OperationNotSupportedException` beantwortet und alle anderen Aufrufe an `c` weitergibt.

Anmerkung: Im Buch habe ich das Konzept der optionalen Methoden kritisiert (S. 446). Inzwischen ist mir klarer geworden, warum dieses Konzept eingeführt wurde und warum es sehr schwer, wahrscheinlich sogar unmöglich gewesen wäre, eine bessere Lösung zu finden.

Wann liefert die Methode `add` das Ergebnis `true`? (Wenn die einzufügende Komponente schon in der Sammlung war und Doppelgänger verboten sind).

Wann liefert die Methode `remove` das Ergebnis `true`? (Wenn tatsächlich eine Komponente entfernt wurde).

Empfehlung: Ehe man die Schnittstelle `Collection<K>` selbst implementiert, sollte man sehr genau nachlesen, was die einzelnen Methoden machen sollen bzw. nicht machen sollen.

Die Schnittstelle `Collection` beschreibt nur "das Allernötigste", was jede Sammlung braucht. Es fehlen aber noch ein paar Dinge, die man sehr häufig benötigt. Vor allem möchte man häufig:

- eine bestimmte Komponente `k1` durch eine andere Komponente `k2` **ersetzen**.
- eine Komponente an einer bestimmten **Stelle der Sammlung** einfügen (z.B. ganz vorn oder ganz hinten oder vor/nach einer bestimmten anderen Komponente etc.)
- auf eine bestimmte **Stelle der Sammlung** zugreifen und die dort stehende Komponente ansehen, ersetzen oder entfernen.

Von der Schnittstelle `Collection<K>` gibt es zur Zeit 10 Erweiterungen, die solche zusätzlichen Möglichkeiten bieten, darunter die Schnittstellen `Set<K>`, `List<K>` und `Deque<K>` (wie "double ended queue", sprich: deck).

Um **wie viele Methoden** erweitert die Schnittstelle `Set<K>` die Schnittstelle `Collection<K>`? Sie dürfen auf der Rückseite des Blattes mit den Wiederholungsfragen nachsehen. (Um 0 Methoden).

Um **was** erweitert die Schnittstelle `Set<K>` dann die Schnittstelle `Collection<K>`? (Um die folgende **weiche Vertragsbedingung**: `Set`-Sammlungen sollen keine Doppelgänger enthalten).

Die Schnittstelle `List<K>` ist eine "erhebliche Erweiterung" der Schnittstelle `Collection<K>`. Sie führt die Vorstellung ein, dass jede Komponente der Sammlung "an einer bestimmten Stelle steht, die man durch einen **Index** bezeichnen kann" (ganz ähnlich wie bei Reihungen).

Anmerkung: Der Name `List` der Schnittstelle `List<K>` ist nicht sehr glücklich gewählt (`Indexed` wäre besser gewesen).

Kurzer Blick auf die Schnittstelle `List<K>` (auf der Rückseite der Wiederholungsfragen von letzter Woche oder im Buch auf S. 450).

Was macht wohl die Methode `indexOf?` `lastIndexOf?` `subList?` (`subList` liefert **eine Sicht** auf eine Teil-Liste, **keine Kopie!**).

Zur Entspannung: Das Ziegen-Problem (Lösung)

Angenommen, das Ziegen-Spiel wird mehrmals von zwei Kandidaten `K1` und `K2` **gleichzeitig** gespielt und zwar so: `K1` und `K2` wählen zuerst immer dieselbe Tür und verhalten sich später so:

`K1` wählt immer "Ich bleibe bei meiner ursprünglichen Wahl" und

`K2` wählt immer "Ich ändere meine Wahl"

Dann gilt doch:

1. Bei jedem Spiel gewinnt entweder `K1` oder `K2` einen Preis (statt einer Ziege).
2. `K1` gewinnt offenbar bei etwa einem Drittel der Spiele einen Preis (weil er eine von drei Möglichkeiten rät, die gleich wahrscheinlich sind).
3. Also muss `K2` bei etwa zwei Dritteln aller Spiele einen Preis gewinnen.
4. Also gewinnt man mehr, wenn man seine Wahl ändert.

Wiederholungsfragen, 6. SU, Mo 02.11.09

1. Schreiben Sie eine **öffentliche Klassenprozedur** namens `gibAus` mit zwei Parametern: Einem Namen (vom Typ `String`) und einer beliebigen Sammlung. Die Methode soll den Namen und die Sammlung hintereinander ausgeben (z.B. mit der Methode `println`), etwa so:

```
sam27: [A, B, C, D]
otto: [12, 17, 33]
```

Das Problem ist eigentlich nur: Welchen *Typ* legt man für den zweiten Parameter fest? Dieser Typ sollte es möglich machen, dass man der Methode **eine beliebige Sammlung** (als zweiten Parameter) übergeben kann, z.B. eine Sammlung des Typs `ArrayList<String>` oder eine des Typs `Stack<Long>` oder eine des Typs `TreeSet<Integer>` oder

2. Schauen Sie sich die Schnittstelle `List` an (auf der Rückseite der Wiederholungsfragen vom 4. SU, Mo 19.10.09 oder im Buch auf S. 450), vor allem die Methoden `get` und `set`. Schreiben Sie dann eine öffentliche Klassenprozedur namens `drehRum` mit **einem** Parameter. Als Parameter soll man dieser Methode eine beliebige `List`-Sammlung übergeben können (z.B. eine des Typs `Stack<Long>` oder eine Sammlung des Typs `ArrayList<String>` oder ...). Die Methode soll **die Reihenfolge der Komponenten** dieser Sammlung **rumdrehen** (aus `[A, B, C]` soll `[C, B, A]` werden).

Rückseite der Wiederholungsfragen, 6. SU, Mo 02.10.09

Typische Zusammenhänge zwischen einer Problemgröße n (z.B. der Länge von Reihungen, die sortiert werden sollen) und der Schrittzahl s , die ein (Sortier-) Algorithmus zum Lösen von Problemen der Größe n benötigt:

Wenn man n so vergrößert	dann vergrößert sich s so	Zeitkomplexität	in Worten
mal 2 mal m	mal 2 mal m	$O(n)$	Oh von n
mal 2 mal m	mal 2^2 mal m^2	$O(n^2)$	Oh von n quadrat
mal 2 mal m	mal 2^3 mal m^3	$O(n^3)$	Oh von n hoch 3
mal 2 mal m	mal 2^k mal m^k	$O(n^k)$	Oh von n hoch k
mal 2^1 mal 2^m	plus 1 plus m	$O(\log n)$	Oh von $\log n$
plus 1 plus m	mal 2^1 mal 2^m	$O(2^n)$	Oh von zwei hoch n
mal 2 mal m	mal 1 mal 1	$O(1)$	Oh von eins

Angenommen, wir wollen die (Zeit-) Komplexität eines Algorithmus ermitteln. Welche der folgenden Befehlsfolgen können wir dabei als einen Schritt bewerten (JA) und welche nicht (NEIN)?

Nr	Befehlsfolge	Schritt?
1	Zwei int-Werte vergleichen?	
2	500 int-Werte vergleichen?	
3	Zwei BigInteger-Objekte vergleichen?	
4	Zwei BigInteger-Objekte vergleichen, von denen jedes weniger als 500 Ziffern enthält?	
5	Zwei BigInteger-Objekte addieren?	
6	800 BigInteger-Objekte addieren, von denen jedes weniger als 500 Ziffern enthält?	
7	Zwei String-Objekte vergleichen?	
8	Zwei String-Objekte vergleichen, von denen jedes kürzer als 500 Zeichen ist?	
9	In einer beliebigen Reihung von int-Variablen den größten Wert finden?	
10	In einer Reihung der Länge 500 den größten Wert finden?	

Antworten zu den Wiederholungsfragen, 6. SU, Mo 02.10.09

1. Schreiben Sie eine *öffentliche Klassenprozedur* namens `gibAus` mit zwei Parametern: Einem Namen (vom Typ `String`) und einer beliebigen Sammlung. Die Methode soll den Namen und die Sammlung hintereinander ausgeben (z.B. mit der Methode `println`), etwa so:

```
sam27: [A, B, C, D]
otto: [12, 17, 33]
```

Das Problem ist eigentlich nur: Welchen *Typ* legt man für den zweiten Parameter fest? Dieser Typ sollte es möglich machen, dass man der Methode *eine beliebige Sammlung* (als zweiten Parameter) übergeben kann, z.B. eine Sammlung des Typs `ArrayList<String>` oder eine des Typs `Stack<Long>` oder eine des Typs `TreeSet<Integer>` oder

```
1 public static <T> void gibAus(String name, Collection<T> lt) {
2     println(name + ": " + lt);
3 }
```

2. Schauen Sie sich die Schnittstelle `List` an (auf der Rückseite der Wiederholungsfragen vom 4. SU, Mo 19.10.09 oder im Buch auf S. 450), vor allem die Methoden `get` und `set`. Schreiben Sie dann eine öffentliche Klassenprozedur namens `drehRum` mit *einem* Parameter. Als Parameter soll man dieser Methode eine beliebige `List`-Sammlung übergeben können (z.B. eine des Typs `Stack<Long>` oder eine Sammlung des Typs `ArrayList<String>` oder ...). Die Methode soll *die Reihenfolge der Komponenten* dieser Sammlung *rumdrehen* (aus `[A, B, C]` soll `[C, B, A]` werden).

```
4 public static <T> void drehRum(List<T> list) {
5     for (int vorn=0, hinten=list.size()-1; vorn<hinten; vorn++, hinten--) {
6         T tmp = list.get(vorn);
7         list.set(vorn, list.get(hinten));
8         list.set(hinten, tmp);
9     }
10 }
```

Angenommen, wir wollen die (Zeit-) Komplexität eines Algorithmus ermitteln. Welche der folgenden Befehlsfolgen können wir dabei als *einen* Schritt bewerten (JA) und welche nicht (NEIN)?

Nr	Befehlsfolge	Schritt?
1	Zwei int-Werte vergleichen?	JA
2	500 int-Werte vergleichen?	JA
3	Zwei BigInteger-Objekte vergleichen?	NEIN
4	Zwei BigInteger-Objekte vergleichen, von denen jedes weniger als 500 Ziffern enthält?	JA
5	Zwei BigInteger-Objekte addieren?	NEIN
6	800 BigInteger-Objekte addieren, von denen jedes weniger als 500 Ziffern enthält?	JA
7	Zwei String-Objekte vergleichen?	NEIN
8	Zwei String-Objekte vergleichen, von denen jedes kürzer als 500 Zeichen ist?	JA
9	In einer beliebigen Reihung von int-Variablen den größten Wert finden?	NEIN
10	In einer Reihung der Länge 500 den größten Wert finden?	JA

6. SU Mo 02.11.09

A. Wiederholung

B. Organisation

Von der Schnittstelle `Collection<K>` gibt es **10 Erweiterungen**, darunter die Schnittstellen `Set<K>`, `List<K>` und `Deque<K>`.

Die Schnittstelle `Set<K>`: Erweitert die Schnittstelle `Collection` um 0 (in Worten: null) Methoden und die "weiche Bedingung": `Set`-Sammlungen sollen **keine Doppelgänger** enthalten.

Die Schnittstelle `List<K>`: Erweitert die Schnittstelle `Collection` um 10 (in Worten: zehn) Methoden. Diese Methoden führen das Konzept (oder die Vorstellung) ein, dass jede Komponente in einer `List`-Sammlung "an einer bestimmten Stelle" steht, die durch einen Index (0, 1, 2, ...) bezeichnet wird (ganz ähnlich wie bei Reihungen).

Algorithmen bewerten und vergleichen, Komplexität eines Algorithmus

Angenommen, wir wollen bestimmte (große) Reihungen sortieren. Dafür gibt es mehrere Algorithmen ("abstrakte Programme, die man unabhängig von der konkreten Sprache oder Notation, in der sie geschrieben sind, verstehen sollte"). Wie können wir diese Algorithmen vergleichen, um den besten (oder zumindest einen besonders guten) herauszufinden?

(Wir können die Algorithmen z.B. in Java programmieren und z.B. im SWE-Labor auf verschiedene große Reihungen anwenden und die zum Sortieren benötigte Zeit messen. Vorteile dieser Vorgehensweise? Nachteile?

Problem: *Abstrakte Algorithmen* haben keinen konkreten Zeit- und Speicherbedarf (weil dieser Bedarf immer ganz wesentlich auch von dem konkreten Ausführer abhängt, den man verwendet, und nicht nur vom Algorithmus).

Wie kann man Algorithmen trotzdem bewerten und miteinander vergleichen?

Lösung: Man ermittelt keinen konkreten Zeitbedarf sondern (nur) den abstrakten **Zusammenhang** zwischen der **Problemgröße** n und der **Anzahl der Rechenschritte**, die der Algorithmus zur Lösung von Problemen der Größe n benötigt.

Dazu sehen wir uns die 1. Tabelle auf der Rückseite der Wiederholungsfragen an.

Def.: Ein (Ausführungs-) **Schritt** ist eine **beliebig lange Folge von Befehlen**, von der man annehmen kann, dass ein Ausführer sie immer in (ungefähr) der **gleichen Zeit** ausführen kann. Insbesondere darf die Ausführungszeit eines Schritts **nicht** von der Problemgröße n abhängen.

Beispiele und Gegenbeispiele für Schritte

Füllen Sie die 2. Tabelle auf der Rückseite der Wiederholungsfragen (mit JA bzw. NEIN) aus.

Ein zusätzliches Arbeitsblatt (**Zeitkomplexitäten**) austeilen und bearbeiten lassen.

Zur Entspannung: Der EPR-Effekt

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

Einstein mochte die Quantenmechanik nicht, kannte sie aber offenbar so genau, dass er auch einige ihrer entfernten Konsequenzen erkennen konnte. Den Effekt, der heute als **EPR-Effekt** bezeichnet wird (nach Einstein, seinem Physiker-Kollegen Podolski und einem Studenten Rosen) ist eine Konsequenz der Quantenmechanik, die Einstein "so unsinnig und unglaublich" vorkam, dass er sie als **Argument gegen die Quantenmechanik** veröffentlichte. Inzwischen hat man diesen merkwürdigen Effekt experimentell nachgewiesen und benutzt ihn zur **abhörsicheren Übertragung von Daten**.

Wiederholungsfragen, 7. SU, Mo 09.11.09

1. Man kann abstrakte Algorithmen (z.B. zum Sortieren von Reihungen) unter anderem dadurch vergleichen, dass man sie konkret implementiert (z.B. in Java, mit dem Java-Ausführer von Sun, unter Windows-Vista auf einem PC mit einem Pentium 4), die Implementierungen ausführen läßt (mehrmals, mit verschiedenen Testdaten) und die Zeit misst.

Welche Nachteile hat diese Vorgehensweise?

2. Angenommen, sie wollen die Zeitkomplexität eines Algorithmus (z.B. eines Algorithmus zum Sortieren von Reihungen) ermitteln. Was dürfen Sie in diesem Zusammenhang als einen Schritt betrachten?

3. Angenommen, Sie wissen von einem Algorithmus namens *Akayat* (zum Kasumtieren von Yakk-Tensoren unter Erhaltung der Li-Alpha-Eigenschaft), dass er eine Zeitkomplexität von $O(n^2)$ hat und für die Lösung eines Problems der Größe 100 etwa 1000 Schritte braucht. Wie viele Schritte wird dieser Algorithmus dann für die Lösung eines Problems der Größe 200 brauchen? Und für ein Problem der Größe 500? Füllen Sie die folgende Tabelle aus:

Für den Algorithmus Akayat mit einer Zeitkomplexität von $O(n^2)$ gilt:

Problemgröße n	Anzahl Schritte (die zur Lösung eines Problems der Größe n benötigt werden):
100	1000
200	
300	
500	

Rückseite der Wiederholungsfragen, 7. SU, Mo 09.11.09

Profile verschiedener Sammlungsstrukturen

Wieviele Schritte kosten die Operationen *Einfügen*, *Suchen in einem positiven Fall*, *Suchen in einem negativen Fall* und *Entfernen* bei Sammlungen verschiedener Strukturen in einem schlimmsten Fall und im Durchschnitt?

Die Variable n bezeichnet immer die Anzahl der Objekte, die sich bereits in der Sammlung befinden.

Tabelle mit Schrittzahlen für 6 verschiedene Sammlungsstrukturen (6 Profile):

Struktur der Sammlung	Operation	Anz. Schritte in einem schlimmsten Fall	Anz. Schritte im Durchschnitt
Reihung (unsortiert)	Einfügen		
	Suchen pos.		
	Suchen neg.		
	Entfernen		
Reihung (sortiert)	Einfügen		
	Suchen pos.		
	Suchen neg.		
	Entfernen		
verkettete Liste (unsortiert)	Einfügen		
	Suchen pos.		
	Suchen neg.		
	Entfernen		
verkettete Liste (sortiert)	Einfügen		
	Suchen pos.		
	Suchen neg.		
	Entfernen		
binäre Bäume (müssen immer sortiert sein)	Einfügen		
	Suchen pos.		
	Suchen neg.		
	Entfernen		
Hash-Tabellen (müssen immer unsortiert sein)	Einfügen		
	Suchen pos.		
	Suchen neg.		
	Entfernen		

Antworten zu den Wiederholungsfragen, 7. SU, Mo 09.11.09

1. Man kann abstrakte Algorithmen (z.B. zum Sortieren von Reihungen) unter anderem dadurch vergleichen, dass man sie konkret implementiert (z.B. in Java, mit dem Java-Ausführer von Sun, unter Windows-Vista auf einem PC mit einem Pentium 4), die Implementierungen ausführen läßt (mehrmals, mit verschiedenen Testdaten) und die Zeit misst.

Welche Nachteile hat diese Vorgehensweise?

Nachteil 1: Die gemessenen Zeiten werden nicht nur vom Algorithmus abhängen, sondern auch von der verwendeten Programmiersprache (z.B. Java), dem verwendeten Ausführer (z.B. dem Java-Ausführer von Sun), dem Betriebssystem (z.B. Windows-Vista), der Hardware (Pentium) und möglicherweise von weiteren Faktoren.

Nachteil 2: Die gemessenen Zeiten werden vermutlich ziemlich schnell ungültig (oder: unrealistischen), weil die Entwicklung von Programmiersprachen, Ausführern, Betriebssystem und Hardware ziemlich schnell weiter geht.

2. Angenommen, sie wollen die Zeitkomplexität eines Algorithmus (z.B. eines Algorithmus zum Sortieren von Reihungen) ermitteln. Was dürfen Sie in diesem Zusammenhang als einen Schritt betrachten?

Eine beliebige Befehlsfolge, von der es plausibel ist anzunehmen, dass sie von einem Ausführer in einer bestimmten, festen Zeit ausgeführt werden kann. Dieses Zeit darf insbesondere nicht von der Größe des gerade bearbeiteten Problems abhängen.

3. Angenommen, Sie wissen von einem Algorithmus namens *Akayat* (zum Kasumieren von Yakk-Tensoren unter Erhaltung der Li-Alpha-Eigenschaft), dass er eine Zeitkomplexität von $O(n^2)$ hat und für die Lösung eines Problems der Größe 100 etwa 1000 Schritte braucht. Wie viele Schritte wird dieser Algorithmus dann für die Lösung eines Problems der Größe 200 brauchen? Und für ein Problem der Größe 500? Füllen Sie die folgende Tabelle aus:

Für den Algorithmus Akayat mit einer Zeitkomplexität von $O(n^2)$ gilt:

Problemgröße n	Anzahl Schritte (die zur Lösung eines Problems der Größe n benötigt werden):
100	1000
200	4 000
300	9 000
500	25 000

7. SU Mo 09.11.09

A. Wiederholung

B. Organisation: Zu den kommenden Vorlesungsterminen sollten Sie immer alle *Blätter mit Wiederholungsfragen* mitbringen, auf deren Rückseiten etwas steht. Diese Rückseiten brauchen wir ab und zu.

Sortierte Sammlungen

Ob zwei beliebige Objekte `ob1` und `ob2` **gleich** oder **ungleich** sind, kann man mit den `equals`-Methoden der Objekte (und dem `not`-Operator `!`) feststellen, z.B. so:

```
1  if ( ob1.equals(ob2)) ...
2  if (! ob2.equals(ob1)) ...
```

Ein solcher Vergleich *auf gleich oder ungleich* funktioniert *für alle Objekte* (weil jedes Objekt eine `equals`-Methode mit einem `Object`-Paramter enthält).

Ein Vergleich *auf kleiner oder größer* funktioniert dagegen nur für Objekte *bestimmter Klassen*.

Ein problematischer Fall: Angenommen, ein `Punkt`-Objekt besteht im Wesentlichen aus zwei `double`-Werten namens `x` und `y`. Warum kann man eine Sammlung solcher `Punkt`-Objekte (normalerweise) *nicht* sortieren? (Weil es für Punkte in einer Ebene keine natürliche oder naheliegende *totale Ordnung* gibt).

Ähnliches gilt auch für Sammlungen von `JButton`-Objekten, Sammlungen von `Auto`-Objekten etc.

Damit man eine Sammlung `sam` sortieren kann, muss es eine *totale Ordnung* für die Komponenten von `sam` geben.

Wie könnte man eine totale Ordnung z.B. für `String`-Objekte in Java programmieren?

Z.B. indem man in der Klasse `String` Methoden wie die folgenden vereinbart:

```
1 public boolean isLessThan      (String that) { ... }
2 public boolean isLessOrEqual  (String that) { ... }
3 public boolean isEqual        (String that) { ... }
4 public boolean isNotEqual     (String that) { ... }
5 public boolean isGreaterOrEqual (String that) { ... }
6 public boolean isGreaterThan  (String that) { ... }
```

Angenommen, wir wollen zwei `String`-Objekte `s1` und `s2` vergleichen und dabei *alle möglichen Vergleichsergebnisse* unterscheiden (`s1` ist *kleiner* als `s2`, `s1` ist *gleich* `s2`, `s1` ist *größer* als `s2`). Wie viele dieser Methoden müsste man dazu aufrufen? (Um alle drei Ergebnisse zu unterscheiden, müssen wir *zwei* dieser Methoden aufrufen, z.B. `isLessThan` und `isGreaterThan`).

Die Rückseite des Blattes mit den Wiederholungsfragen für den 5. SU (Mo 26.10.09) besprechen.

(teure Vergleiche, die Methoden `compareTo` bzw. `compare` in den Schnittstellen `Comparable` bzw. `Comparator`, 2 Möglichkeiten, `Mango`-Objekte vergleichbar zu machen).

Hinweis: Im Buch auf S. 227, im Beispiel-03, werden zwei `String`-Objekte miteinander verglichen.

Mango-Objekte sortiert sammeln

```
7 TreeSet<Mango> ts1 = new TreeSet<Mango>();
8 // Geht nur, wenn Mango eine Comparable-Klasse ist
9
10 class Karl implements Comparator<Mango> {
11     public int compare(Mango m1, Mango m2) {
12         return m1.saftmenge - m2.saftmenge;
13     }
14 }
15
16 TreeSet<Mango> ts2 = new TreeSet<Mango>(new Karl());
```

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

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

Im Abendrot

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

...

Dieses Gedicht wurde 1948 von Richard Strauss vertont (als das letzte seiner "Vier letzten Lieder"). Der Filmkomponist Maurice Jarre verwendete u.a. dieses Lied von Strauss für seinen Soundtrack des Films "I Dreamed of Africa" ("Ich träumte von Afrika"), der im Jahr 2000 herauskam.

Wiederholungsfragen, 8. SU, Mo 16.11.09

1. Welche Java-Objekte enthalten eine öffentliche Methode mit dem Profil

`boolean equals Object`?

2. Die Schnittstelle `Comparable<T>` enthält nur *eine* Methode. Geben Sie ihr Profil an.

3. Die Schnittstelle `Comparator<T>` enthält nur *eine* Methode. Geben Sie ihr Profil an.

4. Betrachten Sie den folgenden Befehl:

```
1    TreeSet<Auto> tsa = new TreeSet<Auto>();
```

Vereinbaren Sie eine Klasse namens `Auto` so, dass der Befehl in Zeile 1 erlaubt ist.

Zur Erinnerung: Der Befehl in Zeile 1 ist nur erlaubt, wenn es für `Auto`-Objekte eine *natürliche Ordnung* gibt. Das wird meistens dadurch erreicht, dass die Klasse `Auto` die Schnittstelle `Comparable<Auto>` implementiert.

5. Betrachten Sie den folgenden Befehl:

```
2    TreeSet<String> tss = new TreeSet<String>(absteigend);
```

Dabei soll `absteigend` ein Objekt des Typs `Comparator<String>` sein welches bewirkt, dass die Komponenten der Sammlung `tss` in *absteigender Reihenfolge* sortiert werden (also genau andersherum als wenn man sie nach ihrer *natürlichen Reihenfolge* sortiert).

Vereinbaren Sie eine geeignete Klasse namens `StringsAbsteigend` und `absteigend` als Objekt dieser Klasse.

Rückseite der Wiederholungsfragen, 8. SU, Mo 16.11.09

Die Objekte einer Klasse `Mango` haben eine natürliche Ordnung, wenn jedes `Mango`-Objekt `m1` eine Methode `compare` enthält, mit der man es mit einem beliebigen `Mango`-Objekt `m2` vergleichen kann (etwa so: `m1.compare(m2)`). Das ist der Fall, wenn die Klasse `Mango` "mindestens" die Schnittstelle `Comparable<Mango>` implementiert. Aber was ist, wenn die Klasse `Mango` "etwas *anderes* als `Comparable<Mango>`" oder "etwas *mehr* als `Comparable<Mango>`" implementiert?

1. Angenommen, eine Klasse `Zitrone` implementiert die Schnittstelle `Comparable<String>` und wir haben folgende Objekte erzeugen lassen:

```
1 Zitrone z1 = new Zitrone(...);
2 Zitrone z2 = new Zitrone(...);
3 String s1 = new String (...);
```

Welche der folgenden Vergleiche sind dann erlaubt und welche nicht?

```
4 if (z1.compareTo(z2)) ...
5 if (z1.compareTo(s1)) ...
6 if (s1.compareTo(z1)) ...
```

2. Warum sind `Zitrone`-Objekte *nicht* natürlich geordnet (obwohl die Klasse `Zitrone` die Schnittstelle `Comparable<String>` implementiert)?

3. Angenommen, eine Klasse `Orange` implementiert die Schnittstelle `Comparable<Object>`. Wird dadurch eine natürliche Ordnung für `Orange`-Objekte definiert?

4. Generische Methoden mit unbeschränkten und beschränkten Typ-Parametern

```
1 // Generische Methode mit einem unbeschaenkten Typ-Parameter:
2 <T> void pA(T t) {println("pA: " + t);}
3
4 // Generische Methoden mit ein bis zwei beschaenkten Typ-Parametern:
5 <T extends Number>
6 void pB(T t) {println("pB: " + t);}
7
8 <T extends Number & Serializable & Remote>
9 void pC(T t) {println("pC: " + t);};
10 // Nach extends darf der 1. Name eine Klasse oder eine Schnittstelle
11 // bezeichnen. Alle weiteren Namen muessen Schnittstellen bezeichnen.
12
13 <K extends Number, S extends ArrayList<K>>
14 void pD(S s) {println("pD: " + s);};
15
16 <S extends ArrayList<?>>
17 void pE(S s) {println("pE: " + s);};
18
19 <T extends Comparable<T>>
20 void pF(T t) {println("pF: " + t);};
21
22 <T extends Comparable<?>>
23 void pG(T t) {println("pG: " + t);};
24
25 <T extends Comparable<? extends T>>
26 void pH(T t) {println("pH: " + t);};
27
28 <T extends Comparable<? super T>>
29 void pI(T t) {println("pI: " + t);}
30
31 // Ein unbeschaenakter Typ-Parameter <T>, aber beschaenkte Paramertypen:
32 <T> void pJ(List<T> lst, Comparator<? super T> ctor) {println("pI: " + lst);};
```

Beschreiben Sie für jede der Methoden `pA`, `pB`, ..., `pI`, mit Parametern von welchen Typen man sie aufrufen darf.

Antworten zu den Wiederholungsfragen, 8. SU, Mo 16.11.09

1. Welche Java-Objekte enthalten eine öffentliche Methode mit dem Profil
`boolean equals Object`?

Alle.

2. Die Schnittstelle `Comparable<T>` enthält nur *eine* Methode. Geben Sie ihr Profil an.

`int compareTo T`

3. Die Schnittstelle `Comparator<T>` enthält nur *eine* Methode. Geben Sie ihr Profil an.

`int compare T T`

4. Betrachten Sie den folgenden Befehl:

```
1    TreeSet<Auto> tsa = new TreeSet<Auto>();
```

Vereinbaren Sie eine Klasse namens `Auto` so, dass der Befehl in Zeile 1 erlaubt ist.

Zur Erinnerung: Der Befehl in Zeile 1 ist nur erlaubt, wenn es für `Auto`-Objekte eine *natürliche Ordnung* gibt. Das wird meistens dadurch erreicht, dass die Klasse `Auto` die Schnittstelle `Comparable<Auto>` implementiert.

```
1    class Auto implements Comparable<Auto> {
2        int preis; // Wird von den Konstruktoren initialisiert
3        ...
4        public int compareTo(Auto that) {
5            return this.preis - that.preis;
6        }
7        ...
8    }
```

5. Betrachten Sie den folgenden Befehl:

```
9    TreeSet<String> tss = new TreeSet<String>(absteigend);
```

Dabei soll `absteigend` ein Objekt des Typs `Comparator<String>` sein welches bewirkt, dass die Komponenten der Sammlung `tss` in *absteigender Reihenfolge* sortiert werden (also genau andersherum als wenn man sie nach ihrer *natürlichen Reihenfolge* sortiert).

Vereinbaren Sie eine geeignete Klasse namens `StringsAbsteigend` und `absteigend` als Objekt dieser Klasse.

```
10   class StringsAbsteigend implements Comparator<String> {
11       public int compare(String s1, String s2) {
12           return -s1.compareTo(s2);
13       }
14   }
15
16   StringsAbsteigend absteigend = new StringsAbsteigend();
```

Lösungen zu den Aufgaben auf der Rückseite der Wiederholungsfragen, 8. SU, Mo 16.11.09

1. Angenommen, eine Klasse `Zitrone` implementiert die Schnittstelle `Comparable<String>` und wir haben folgende Objekte erzeugen lassen:

```
1 Zitrone z1 = new Zitrone(...);
2 Zitrone z2 = new Zitrone(...);
3 String s1 = new String (...);
```

Welche der folgenden Vergleiche sind dann erlaubt und welche nicht?

```
4 if (z1.compareTo(z2)) ...           nicht erlaubt
5 if (z1.compareTo(s1)) ...           erlaubt
6 if (s1.compareTo(z1)) ...           nicht erlaubt
```

2. Warum sind `Zitrone`-Objekte *nicht* natürlich geordnet (obwohl die Klasse `Zitrone` die Schnittstelle `Comparable<String>` implementiert)?

Weil ein `Zitrone`-Objekt keine Methode `compare` enthält, mit der man es mit einem anderen `Zitrone`-Objekt vergleichen kann (man kann es nur mit `String`-Objekten vergleichen).

3. Angenommen, eine Klasse `Orange` implementiert die Schnittstelle `Comparable<Object>`. Wird dadurch eine natürliche Ordnung für `Orange`-Objekte definiert?

Ja, denn jedes `Orange`-Objekt enthält eine Methode `compare`, mit der man es mit beliebigen Objekten vergleichen kann, unter anderem auch mit anderen `Orange`-Objekten.

4. Generische Methoden mit unbeschränkten und beschränkten Typ-Parametern

```
1 // Generische Methode mit einem unbeschaenkten Typ-Parameter:
2 <T> void pA(T t) {pLn("pA: " + t);}
3
4 // Generische Methoden mit ein bis zwei beschaenkten Typ-Parametern:
5 <T extends Number>
6 void pB(T t) {pLn("pB: " + t);}
7
8 <T extends Number & Serializable & Remote>
9 void pC(T t) {pLn("pC: " + t);};
10 // Nach extends darf der 1. Name eine Klasse oder eine Schnittstelle
11 // bezeichnen. Alle weiteren Namen muessen Schnittstellen bezeichnen.
12
13 <K extends Number, S extends ArrayList<K>>
14 void pD(S s) {pLn("pD: " + s);};
15
16 <S extends ArrayList<?>>
17 void pE(S s) {pLn("pE: " + s);};
18
19 <T extends Comparable<T>>
20 void pF(T t) {pLn("pF: " + t);};
21
22 <T extends Comparable<?>>
23 void pG(T t) {pLn("pG: " + t);};
24
25 <T extends Comparable<? extends T>>
26 void pH(T t) {pLn("pH: " + t);};
27
28 <T extends Comparable<? super T>>
29 void pI(T t) {pLn("pI: " + t);}
30
31 // Ein unbeschaenakter Typ-Parameter <T>, aber beschaenkte Paramertypen:
1 <T> void pJ(List<T> lst, Comparator<? super T> ctor) {pLn("pI: " + lst);};
```

Beschreiben Sie für jede der Methoden `pA`, `pB`, ..., `pI`, mit Parametern von welchen *Typen* man sie aufrufen darf.

Ein aktueller Parameter für **pA** darf zu einem beliebigen Typ gehören (primitive Parameter werden vom Java-Ausführer automatisch in ein entsprechendes Hüllobjekt umgewandelt).

Ein aktueller Parameter für **pB** muss zu einer Erweiterung (d.h. zu einem Untertyp) des Typs `Number` gehören. Beispiele für solche Erweiterungen sind die Typen `Long`, `Double` und `Number`.

Zur Erinnerung: Jeder Typ gilt als *Untertyp* (Erweiterung) und als *Obertyp* von sich selbst.

Ein aktueller Parameter für **pC** muss gleichzeitig zu *drei Typen* gehören:

1. Zu einer Erweiterung des Typs `Number` (z.B. `Long`, `Double`, ..., `Number`)
2. Zum Schnittstellen-Typ `Serializable`
3. Zum Schnittstellen-Typ `Remote`

Da `Serializable` und `Remote` sogenannte Markierungsschnittstellen (und damit leer) sind, sind sie besonders einfach zu implementieren. Beispiel für eine Klasse, auf der man die Methode `pC` anwenden darf:

```
1 class DreiTyp extends Number implements Serializable, Remote {}
```

Ein aktueller Parameter für **pD** kann z.B. zu einem der Typen `ArrayList<Long>`, `ArrayList<Double>`, `ArrayList<Number>` gehören. Allgemein: Ein aktueller Parameter für **pD** muss eine `ArrayList`-Sammlung sein, deren Komponenten zu einer Erweiterung (d.h. zu einem Untertyp) von `Number` gehören.

Ein aktueller Parameter für **pE** muss eine `ArrayList`-Sammlung sein. Die Komponenten der Sammlung dürfen zu einem beliebigen Typ (`<?>`) gehören.

Ein aktueller Parameter für **pF** muss ein Objekt einer Klasse `C` sein, die genau die Schnittstelle `Comparable<C>` implementiert ("nicht mehr und nicht weniger").

Achtung: Falls die Klasse `C` "mehr implementiert", z.B. die Schnittstelle `Comparable<Object>` (und selbst nicht gleich `Object` ist), dann darf man **pF** *nicht* auf `C`-Objekte anwenden!

Ein aktueller Parameter für **pG** muss ein Objekt einer Klasse `C` sein, die irgendeinen `Comparable`-Typ implementiert (z.B. den Typ `Comparable<String>` oder `Comparable<C>` oder ...).

Ein aktueller Parameter für **pH** muss ein Objekt einer Klasse `C` sein, deren Objekte man mit Objekten irgendeiner Unterklasse von `C` vergleichen kann. Das ist ein Fall, der in der Praxis vermutlich nie vorkommt.

Ein aktueller Parameter für **pI** muss zu einer Klasse `C` gehören, die für ihre Objekte eine natürliche Ordnung definiert (indem sie z.B. die Schnittstelle `Comparable<C>` oder die Schnittstelle `Comparable<Object>` oder ... implementiert).

Der Typ-Parameter `<T>` von **pJ** ist unbeschränkt (d.h. man darf für `T` jeden beliebigen Referenztyp einsetzen). Aber der erste ("normale") Parameter `lst` muss zum Typ `ArrayList<T>` gehören. Und der zweite Parameter `ctor` muss ein `Comparator`-Objekt sein, mit dessen `compare`-Methode man Objekte von einem Obertyp (engl. *super type*) von `T` vergleichen kann.

Beispiel: Wenn man sich `T` durch `Long` ersetzt denkt, dann muss man für `lst` ein Objekt des Typs `ArrayList<Long>` und für `ctor` ein Objekt des Typs `Comparator<Long>` oder `Comparator<Number>` oder `Comparator<Object>` angeben (weil `Long`, `Number` und `Object` die einzigen Obertypen von `Long` sind).

Siehe dazu auch das Programm `GenParams01.java`.

8. SU Mo 16.11.09

A. Wiederholung B. Organisation

Sammlungen sortieren und sortierte Sammlungen in Java

In der Klasse `Collections` (die man möglichst selten mit der Schnittstelle `Collection` verwechseln sollte) gibt es zahlreiche (Klassen-) Methoden zum Bearbeiten von Sammlungen, u.a. die folgenden beiden, die wir im Folgenden als `sort-1` (mit einem Parameter) und `sort-2` (mit zwei Parametern) bezeichnen werden:

```
1 static public <T extends Comparable<? super T>> void sort(List<T> list)
2 static public <T> void sort(List<T> list, Comparator<? super T> c)
```

Um zu lernen, wie man diese beiden Zeilen lesen und verstehen sollte, bearbeiten wir jetzt erstmal gemeinsam die Rückseite des Blattes mit den Wiederholungsfragen für den heutigen SU.

Verschiedene Strukturen für Sammlungen

In Java zählen Reihungen nicht zu den Sammlungen. Aber wenn man Sammlungen unabhängig von einer bestimmten Programmiersprache untersucht und diskutiert, unterscheidet man häufig folgende Grundformen von Sammlungen:

Reihungen	(unsortiert und sortiert)
Verkettete Listen	(unsortiert und sortiert)
Binäre Bäume	(müssen immer sortiert sein)
Hash-Tabellen	(dürfen nicht sortiert sein)

Die wichtigsten Operationen, die man auf Sammlungen anwenden möchte:

Einfügen	
Suchen	(in einem positiven Fall, wenn man das Gesuchte findet)
Suchen	(in einem negativen Fall, wenn man das Gesuchte nicht findet)
Entfernen	

Wir haben im letzten SU schon damit angefangen, diese Operationen für die verschiedenen Grundformen von Sammlungen zu besprechen und dabei die Tabelle auf der Rückseite der Wiederholungsfragen zum 7. SU, Mo 09.11.09 auszufüllen. Das wollen wir (ich zumindest :-) jetzt fortsetzen.

Zur Entspannung: Eigenschaften von Qbits (Quanten-Bits)

- Mit n "normalen Bits" kann man *eine* Zahl (zwischen 0 und 2^n-1) darstellen.
Mit n Qubits kann man gleichzeitig bis zu 2^n Zahlen (zwischen 0 und 2^n-1) darstellen und mit *einer* Operation kann man alle diese Zahlen "gleichzeitig bearbeiten".
- Wenn man ein n -Qubit-Speicher "ansieht und ausliest", bekommt man nur *einen* seiner Werte. Alle anderen Werte gehen dabei unvermeidbar und unwiderruflich verloren.
- Man kann ein Qubit (mit all seinen Werten) *nicht* kopieren.
- Auf Qubits kann man nur *umkehrbare Verknüpfungen* anwenden.

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

Eine interessante Einführung in die Quantenmechanik von einer berliner Schülerin:

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

Wiederholungsfragen, 9.SU, Mo 23.11.09

Betrachten Sie die folgende (noch unvollständige) Klassenvereinbarung:

```
1 class Papaya implements Comparable<Papaya> {
2     private int saft; // Anzahl Kubikzentimeter Saft
3     private int preis; // Anzahl Cents
4
5     public int getSaft () {return saft;}
6     public int getPreis() {return preis;}
7
8     public Papaya(int saft, int preis) {
9         this.saft = saft;
10        this.preis = preis;
11    }
12
13    public String toString() {
14        return "Papaya: " + saft + ", " + preis;
15    }
16
17    ...
18 }
```

1. Weil in Zeile 1 ... `implements Comparable<Papaya>` ... steht, muss die Klasse `Papaya` eine öffentliche Objekt-Methode namens `compareTo` enthalten. Schreiben Sie diese Methode so, dass sie `Papaya`-Objekte *aufsteigend nach ihrem Saft* vergleicht.

2. Vereinbaren Sie ein Objekt namens `papSam` des Typs `ArrayList<Papaya>`, fügen Sie drei `Papaya`-Objekte mit unterschiedlichen Saftmengen in diese Sammlung ein und sortieren Sie sie anschließend entsprechend ihrer *natürlichen Ordnung* (mit einer der `sort`-Methoden aus der Klasse `Collections`).

3. Vereinbaren Sie eine Klasse namens `PapNachPreis`, die die Schnittstelle `Comparator<Papaya>` implementiert. Die Methode `compare` in dieser Klasse soll `Papaya`-Objekte *absteigend nach ihrem Preis* vergleichen.

4. Sortieren Sie die Sammlung `papSam` (siehe oben 2.) *absteigend nach Preisen* (mit Hilfe eines `PapNachPreis`-Objekts).

Antworten zu den Wiederholungsfragen, 9.SU, Mo 23.11.09

1. Weil in Zeile 1 ... implements Comparable<Papaya> ... steht, muss die Klasse Papaya eine öffentliche Objekt-Methode namens compareTo enthalten. Schreiben Sie diese Methode so, dass sie Papaya-Objekt *aufsteigend nach ihrem Saft* vergleicht.

2. Vereinbaren Sie ein Objekt namens papSam des Typs ArrayList<Papaya>, fügen Sie drei Papaya-Objekte mit unterschiedlichen Saftmengen in diese Sammlung ein und sortieren Sie sie anschließend (mit einer der sort-Methoden aus der Klasse Collections).

3. Vereinbaren Sie eine Klasse namens PapNachPreis, die die Schnittstelle Comparator<Papaya> implementiert. Die Methode compare in dieser Klasse soll Papaya-Objekte *absteigend nach ihrem Preis* vergleichen.

4. Sortieren Sie die Sammlung papSam (siehe oben 2.) nach den Preisen ihrer Komponenten (mit Hilfe eines PapNachPreis-Objekts).

```

1 class Papaya implements Comparable<Papaya> {
2     private int saft; // Anzahl Kubikzentimeter Saft
3     private int preis; // Anzahl Cents
4
5     public int getSaft () {return saft;}
6     public int getPreis() {return preis;}
7
8     public Papaya(int saft, int preis) {
9         this.saft = saft;
10        this.preis = preis;
11    }
12
13    public String toString() {
14        return "Papaya: " + saft + ", " + preis;
15    }
16
17    public int compareTo(Papaya that) { // Loesung zu 1.
18        return this.saft - that.saft;
19    }
20
21    static public void main(String[] _) {
22
23        ArrayList<Papaya> papSam = // Loesung zu 2.
24            new ArrayList<Papaya>();
25        papSam.add(new Papaya(50, 220));
26        papSam.add(new Papaya(30, 180));
27        papSam.add(new Papaya(40, 200));
28
29        Collections.sort(papSam); // Loesung zu 4.
30
31        Collections.sort(papSam, new PapNachPreis());
32    } // main
33 } // class Papaya
34
35 class PapNachPreis implements Comparator<Papaya> { // Loesung zu 3.
36     public int compare(Papaya p1, Papaya p2) {
37         return p2.getPreis() - p1.getPreis();
38     }
39 }

```

9. SU Mo 23.11.09

A. Wiederholung

B. Organisation

Strukturen von Sammlungen (Fortsetzung)

In Java gelten Reihungen *nicht* als Sammlungen. Aber wenn man unabhängig von einer bestimmten Programmiersprache über Sammlungen spricht, zählt man Reihungen meistens dazu.

Wie hatte wir die *wichtigsten Operationen* genannt, die man auf Sammlungen anwenden will? (Einfügen, Suchen, Entfernen).

Welche beiden Varianten der Operation *Suchen* hatten wir unterschieden? (Suchen in einem *positiven* Fall und Suchen in einem *negativen* Fall, wenn man das Gesuchte *findet* bzw. *nicht findet*).

Die Sammlungsform *unsortierte Reihung* haben wir schon vor 2 Wochen besprochen und die Schrittzahlen für die einzelnen Operationen in eine Tabelle eingetragen (siehe S. 36

Rückseite der Wiederholungsfragen, 7. SU, Mo 09.11.09)

Einfügen: $1 \mid 1$, Suchen pos.: $n \mid n/2$, Suchen neg.: $n \mid n$

Bei Entfernen tragen wir ein: Suchen +1 | Suchen + 1

Anmerkung: Die fertig ausgefüllte Tabelle ist irgendwo im Internet zugänglich. Wer sie sich schon runtergeladen hat, sollte seine Kopie der Tabelle jetzt weglegen oder rumdrehen und sie möglichst nicht benutzen, sonst nützen die folgenden Fragen weniger, als sie sollten und könnten.

Jetzt wollen wir die Sammlungsform

Sortierte Reihung untersuchen.

Beim Einfügen müssen wir jede der n Komponenten, die schon in der Sammlung sind, mind. einmal anfassen. Wieso? (Entweder beim Suchen nach "der richtigen Stelle" oder beim Verschieben)

Einfügen: $n \mid n$

Binäres Suchen besprechen.

Suchen neg: $\log_2 n \mid \log_2 n$

Suchen pos: $\log_2 n \mid$ etwas weniger als $\log_2 n$

Entfernen: Suchen + $n \mid$ Suchen + $n/2$

Unsortierte und sortierte Reihungen vergleichen:

Wann ist die eine und wann die andere Struktur schneller?

Nachteile von Reihungen allgemein (egal ob sortiert oder unsortiert)? (unveränderbare Länge, "Beton").

Verkettete Listen

Sind "aus Gummi statt aus Beton".

Die Übung03: Verkettete Listen austeilen und bearbeiten.

Erreichbarkeit/Aspektzugehörigkeit/Art der einzelnen Elemente der Klasse `MeineListe`?

(private Klassen-Klasse `Knoten`, pak.er. Objektattribut `end`, pak.er. Objektattribut `anf`, öffentliche Objektmethoden `fuegeEin`, `sucheVor`, `istDrin`, `entferne`, `toString`).

Üb03-1: Aus wie vielen `Knoten`-Objekten besteht eine leere Liste? (2)

Welchen Wert hat die Variable `anf.next` (in der Bojendarstellung) am Anfang? (30, siehe Zeile 19)

Achtung: Diesen Wert dürfen nur BesitzerInnen eines *Bleistifts* eintragen, da er gleich noch mehrmals verändert werden muss. Studierende, die im 2. Semester immer noch keinen Bleistift bei sich tragen, machen damit ja deutlich, dass sie nicht wirklich InformatikerInnen werden wollen.

Üb03-2: Drei fuegeEin-Aufrufe ausführen.

Den ersten Aufruf führen wir zusammen (an der Tabel) aus, die anderen jeder allein oder mit seinen Nachbarn.

Zur Entspannung: Richard Strauss (1864-1949)

Deutscher Musiker, *nicht* verwandt mit den beiden Musikern namens *Johann Strauss*, begann mit 6 Jahren zu komponieren, komponierte zahlreiche *Tondichtungen* (Don Juan, Till Eulenspiegels lustige Streiche, Also sprach Zarathustra, Ein Heldenleben, Sinfonia domestica, Eine Alpensinfonie, ...) und Opern (Salome, Elektra, Der Rosenkavalier, ...). Teile der Tondichtung *Also sprach Zarathustra* wurden im Film 2001 und von Elvis Presley verwendet (und werden heute in einer Bierreklame zitiert). War weltberühmt, galt bei vielen als "der größte lebende deutsche Komponist". Setzte sich intensiv für die Rechte von Musikern und Komponisten ein (Genossenschaft deutscher Tonsetzer, GEMA).

Sein Verhältnis zu den Nazis sieht heute alles andere als vorbildlich aus. Als 1933 die Nazis einen Auftritt des jüdischen Dirigenten *Bruno Walter* verboten, sprang Strauss als Ersatz ein. Als *Toscanini* seine Teilnahme an den Bayreuther Wagner-Festspielen (aus Protest gegen die Nazis) absagte, sprang er ebenfalls ein. Er war von 1933 bis 1935 Präsident der Reichsmusikkammer, was für die Nazis ein wichtiger Propaganda-Erfolg war.

Komponierte auch viele Lieder ("Lieder" ist eine eigene Musikgattung, die auch im Englischen und in anderen Sprachen als "Lieder" bezeichnet wird), unter anderem "Vier letzte Lieder" (für Sopran und großes Orchester), zu denen auch das Lied "Im Abendrot" (Text von Joseph von Eichendorff) gehört.

Wiederholungsfragen, 11. SU, Mo 30.11.09

1. Geben Sie die folgenden Logarithmen zur Basis 2 (gerundet zur nächsten Ganzzahl) an:

$\log_2(1 \text{ Tausend})$:

$\log_2(1 \text{ Million})$:

$\log_2(1 \text{ Milliarde})$:

$\log_2(1 \text{ Billion})$:

2. Betrachten Sie die folgende (sortierte) Reihung:

Kompos | 19 | 21 | 25 | 32 | 36 | 40 | 47 | 53 | 58 | 63 | 65 | 69 | 74 | 77 | 85 |
Indizes 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14

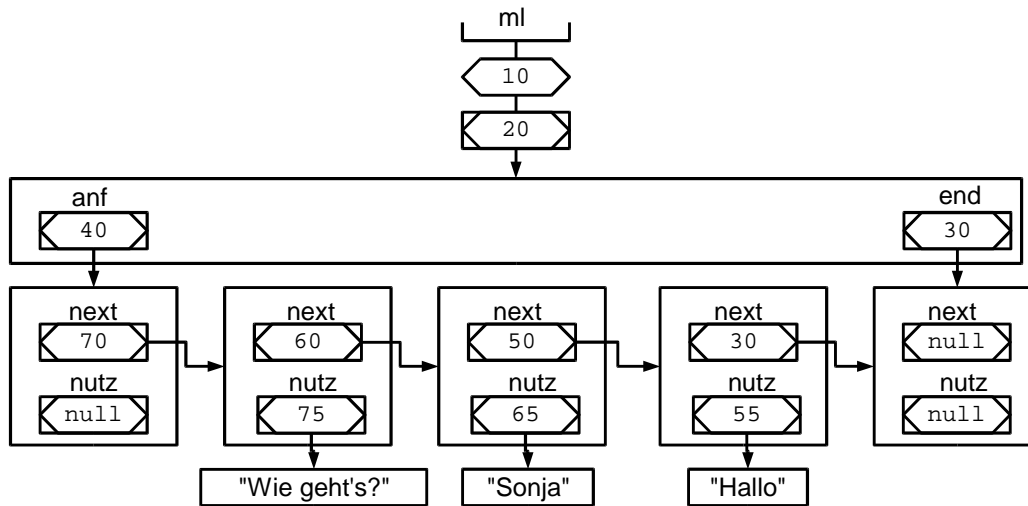
Sie sollen in dieser Reihung verschiedene Zahlen suchen, und zwar nach dem Verfahren *Binäres Suchen*. Dabei sollen Sie angeben, mit welchen *Zahlen in der Reihung* Sie die *gesuchte Zahl* vergleichen. Füllen Sie die folgende Tabelle (entsprechend dem Beispiel in der ersten Zeile) aus:

Wenn man die folgende Zahl sucht	vergleicht man sie der Reihe nach mit den folgenden Zahlen:
36	53 32 40 36
74	
69	
47	
24	
26	

3. Wo müsste man eine Zahl z , die man nicht gefunden hat (wie z.B. 24 oder 26), in die Reihung einfügen?

Rückseite der Wiederholungsfragen, 11. SU, Mo 30.11.09

Eine mögliche Lösung zur Übung **Üb03-2** (das `MeineListe`-Objekt `ml` nach dem Einfügen von 3 `String`-Komponenten als Boje dargestellt). Diese Lösung sollte als Grundlage für die Übungen **Üb3-3**, und **Üb3-4** genommen werden, damit wir alle dieselben Referenzwerte haben und unsere Lösungen vergleichen können:



Aufgabe 1: Füllen Sie die noch freien Felder der folgende Tabelle mit Bojenteilen (in ASCII-Bojen-Schreibweise) aus:

Variable	Wert der Variablen	Zielwert der Variablen
<code>ml</code>	<code>[<20>]</code>	<code>[anf: [<40>], end: [<30>]]</code>
<code>ml.anf</code>	<code>[<40>]</code>	<code>[next: [<70>], nutz: [<null>]]</code>
<code>ml.end</code>		
<code>ml.anf.next</code>		
<code>ml.anf.next.next</code>		
<code>ml.anf.next.nutz</code>		

Aufgabe 2: Welche Referenz hat das `Knoten`-Objekt, an dem

- 2.1. die Nutzdaten "Wie geht's?" hängen?
- 2.2. die Nutzdaten "Sonja" hängen?
- 2.3. die Nutzdaten "Hallo" hängen?

Antworten zu den Wiederholungsfragen, 11. SU, Mo 30.11.09

1. Geben Sie die folgenden Logarithmen zur Basis 2 (gerundet zur nächsten Ganzzahl) an:

$\log_2(1 \text{ Tausend})$: **10**

$\log_2(1 \text{ Million})$: **20**

$\log_2(1 \text{ Milliarde})$: **30**

$\log_2(1 \text{ Billion})$: **40**

2. Betrachten Sie die folgende (sortierte) Reihung:

Kompos | 19 | 21 | 25 | 32 | 36 | 40 | 47 | 53 | 58 | 63 | 65 | 69 | 74 | 77 | 85 |
Indizes 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14

Sie sollen in dieser Reihung verschiedene Zahlen suchen, und zwar nach dem Verfahren *Binäres Suchen*. Dabei sollen Sie angeben, mit welchen *Zahlen in der Reihung* Sie die *gesuchte Zahl* vergleichen. Füllen Sie die folgende Tabelle (entsprechend dem Beispiel in der ersten Zeile) aus:

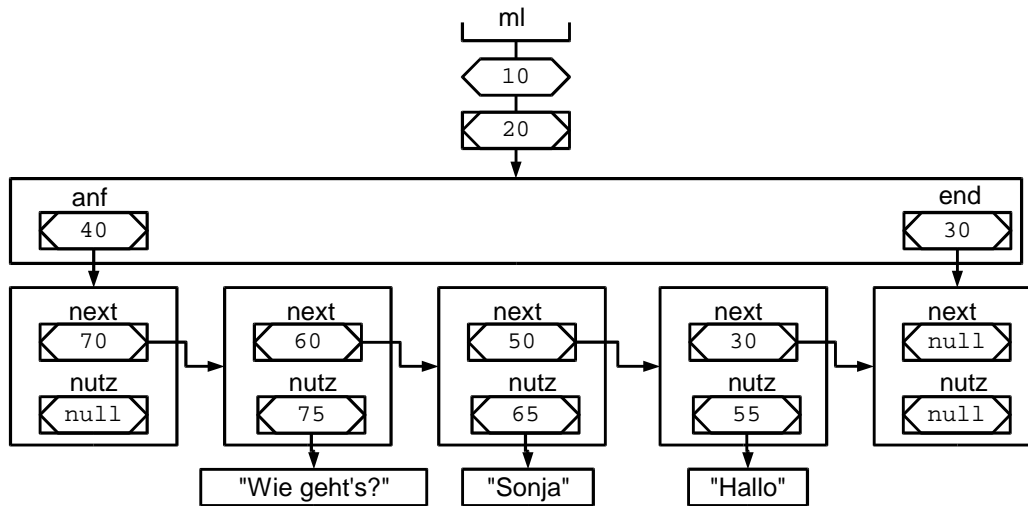
Wenn man die folgende Zahl sucht	vergleicht man sie der Reihe nach mit den folgenden Zahlen:
36	53 32 40 36
74	53 69 77 74
69	53 69
47	53 32 40 47
24	53 32 21 25
26	53 32 21 25

3. Wo müsste man eine Zahl z , die man nicht gefunden hat (wie z.B. 24 oder 26), in die Reihung einfügen?

Genau *an* der Stelle, mit der man z zuletzt verglichen hat (falls z kleiner ist als die dort stehende Zahl) oder *hinter* der Stelle, mit der man z zuletzt verglichen hat (falls z größer ist als die letzte Vergleichspartnerin).

Lösung zur Aufgabe auf der Rückseite der Wiederholungsfragen, 11. SU, Mo 30.11.09

Eine mögliche Lösung zur Übung **Üb03-2** (das `MeineListe`-Objekt `ml` nach dem Einfügen von 3 `String`-Komponenten als Boje dargestellt). Diese Lösung sollte als Grundlage für die Übungen **Üb3-3**, und **Üb3-4** genommen werden, damit wir alle dieselben Referenzwerte haben und unsere Lösungen vergleichen können:



Füllen Sie die noch freien Felder der folgende Tabelle mit Bojenteilen (in ASCII-Bojen-Schreibweise) aus:

Variable	Wert der Variablen	Zielwert der Variablen
<code>ml</code>	[<20>]	[anf: [<40>], end: [<30>]]
<code>ml.anf</code>	[<40>]	[next: [<70>], nutz: [<null>]]
<code>ml.end</code>	[<30>]	[next: [<null>], nutz: [<null>]]
<code>ml.anf.next</code>	[<70>]	[next: [<60>], nutz: [<75>]]
<code>ml.anf.next.next</code>	[<60>]	[next: [<50>], nutz: [<65>]]
<code>ml.anf.next.nutz</code>	[<75>]	["Wie geht's?"]

Aufgabe 2: Welche Referenz hat das Knoten-Objekt, an dem

- 2.1. die Nutzdaten "Wie geht's?" hängen? <70>
- 2.2. die Nutzdaten "Sonja" hängen? <60>
- 2.3. die Nutzdaten "Hallo" hängen? <50>

10. SU Mo 30.11.09

A. Wiederholung

B. Organisation: Nur *eine* Studentin hat Interesse an einer Wiederholung des vorigen SU (9. SU, Mo 23.11.09) gezeigt. Die Wiederholung fand deshalb nicht statt.

Was bedeutet oder bezeichnet der Name einer Referenzvariablen?

Angenommen, wir haben folgende Referenzvariable:

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

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

```
2 |sb|--<100>--[<110>]--[ "ABC" ]
```

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

```
3 if (sb == ...) ...
4 sb.append("ZZ");
5 sb = ...
6 println(sb);
```

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

(Zeile 3: Der *Wert* wird verglichen, Zeile 4: Der *Zielwert* wird verändert, Zeile 5: Der *Wert* wird durch einen neuen Wert ersetzt, Zeile 6: Meist wird eine `String`-Repräsentation des *Zielwertes* ausgegeben, aber falls `sb` nicht auf einen Zielwert zeigt, wird eine Repräsentation des *Wertes* `null` ausgegeben).

In Zeile 1: Den *Wert* [`<110>`], denn der wird durch den Operator `==` verglichen.

In Zeile 2: Den *Zielwert* [`"ABC"`]. Dieser Zielwert ist ein Objekt des Typs `StringBuilder`, d.h. ein *Modul*, in dem sich u.a. eine Methode namens `append` (mit einem `String`-Parameter) befindet. Diese Methode `append` im Zielwert-Modul von `sb` wird aufgerufen.

In Zeile 3: Den *Wert* [`<110>`], denn dieser Wert wird durch einen anderen Wert ersetzt.

In Zeile 4: Ein Teil des *Zielwertes* d.h. Moduls [`"ABC"`], falls ein Zielwert vorhanden ist, sonst der *Wert* der Variablen `sb` (der dann gleich `null` ist).

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

Verkettete Listen (Fortsetzung)

Betrachten Sie die Rückseite des Blattes mit den Wiederholungsfragen.

Sie sehen eine Bojendarstellung von *vielen Variablen*, die miteinander zusammenhängen.

Nur *eine* dieser Variablen hat einen *einfachen Namen*. Welche? (`ml`)

Die anderen Variablen kann man mit *zusammengesetzten Namen* bezeichnen, z.B.

```
ml.anf, ml.end, ml.anf.next, ml.anf.nutz, ml.anf.next.next,
ml.anf.next.nutz, ml.anf.next.next.next, ml.anf.next.next.nutz, ...
```

Angenommen, wir wollen einen bestimmten Knoten aus der Liste *entfernen*, und zwar *den* Knoten, an dem die Nutzdaten `"Hallo"` hängen. Dazu brauchen wir nur den Wert *einer* einzigen Variablen zu verändern?

Welchen *alten Wert* müssen wir durch welchen *neuen Wert* ersetzen?

(Den alten Wert [`<50>`] durch den neuen Wert [`<30>`]).

Allgemein: Wenn wir einen Knoten `K` entfernen wollen, in welchem Knoten müssen wir dann etwas verändern? (Im Vorgänger-Knoten von `K`).

Deshalb gibt es in der Klasse `MeineListe<K>` keine Methode namens `suche`, sondern eine Methode `sucheVor`. Sie liefert nicht den Knoten, an dem bestimmte Nutzdaten hängen, sondern den *Vorgänger* eines solchen Knotens.

Welchen Wert liefert `sucheVor("Wie gehts?")`? (40, nicht 70!). Und `sucheVor("Sonja")`? (70, nicht 60!)

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

Wurde 1983 von Karry Mullis in CA (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 solche Kettenmoleküle. Ein DNS-Molekül besteht aus zwei Teilsträngen (etwa wie Eisenbahnschienen), die aber nicht gleich, sondern "Spiegelbilder voneinander" sind.

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.

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. Durch Erwärmen und Abkühlen der beteiligten Stoffe auf bestimmte Temperaturen kann man die einzelnen Schritte einleiten bzw. beenden. Nach jeweils 3 solchen Schritten hat sich die Anzahl der DNS-Moleküle verdoppelt. Diese drei Schritte dauern 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} (eine Trillion) vermehren (praktisch ist die Ausbeute etwas geringer).

Aufgabe: Programmieren Sie die Methode `istDrin`, die in der Klasse `MeineListe` noch fehlt.

Wiederholungsfragen, 11 SU, Mo 07.12.09

1. Eine Variable eines Referenztyps kann aus bis zu 4 Teilen bestehen. Der Name einer solchen Variablen (falls vorhanden) bezeichnet je nach Zusammenhang den einen oder anderen der vier Teile. Geben Sie von jedem der Befehle in den Zeilen 4 bis 7 an, welchen Teil der Variablen der Name "als" dort bezeichnet:

```
1  ArrayList<String> als = new ArrayList<String>();
2  als.add("Hallo");
3  ...                               // Von "als" bezeichneter Variablen-Teil
4  if (als.equals(...)) ... ;       //
5  als = ... ;                       //
6  als.add(...);                    //
7  boolean b = (als == ...) ... ;   //
8  System.out.print(als);           //
```

2. Schreiben Sie (für die Klasse `MeineListe`) die Methode `public boolean istDrin(K nutz)` (falls Sie beim letzten SU bzw. zu Hause damit noch nicht ganz fertig geworden sein sollten).

3. Schreiben Sie (für die Klasse `MeineListe`) die Methode `public boolean entferne(K nutz)`.

Tip zu den Methoden: Sie dürfen natürlich alle anderen Methoden der Klasse `MeineListe` benutzen!

Antworten zu den Wiederholungsfragen, 11 SU, Mo 07.12.09

1. Eine Variable eines Referenztyps kann aus bis zu 4 Teilen bestehen. Der Name einer solchen Variablen (falls vorhanden) bezeichnet je nach Zusammenhang den einen oder anderen der vier Teile. Geben Sie von jedem der Befehle in den Zeilen 4 bis 7 an, welchen Teil der Variablen der Name "als" dort bezeichnet:

```
1  ArrayList<String> als = new ArrayList<String>();
2  als.add("Hallo");
3  ... // Von "als" bezeichneter Variablen-Teil
4  if (als.equals(...)) ... ; // der Zielwert
5  als = ... ; // der Wert
6  als.add(...); // der Zielwert
7  boolean b = (als == ...) ... ; // der Wert
8  System.out.print(als); // der Wert (falls er gleich null ist), sonst
9  // der Zielwert.
```

2. Schreiben Sie (für die Klasse MeineListe) die Methode

public boolean istDrin(K nutz) (falls Sie beim letzten SU bzw. zu Hause damit noch nicht ganz fertig geworden sein sollten).

```
10 public boolean istDrin(K nutz) {
11     // Liefert true wenn nutz in dieser Sammlung vorkommt, sonst false.
12     Knoten vor = sucheVor(nutz);
13     return vor.next != end;
14 }
```

3. Schreiben Sie (für die Klasse MeineListe) die Methode

public boolean entferne(K nutz).

```
15 public boolean entferne(K nutz) {
16     // Wenn nutz an mind. einem Knoten dieser Liste haengt, wird einer
17     // dieser Knoten geloescht und true geliefert.
18     // Sonst wird false geliefert.
19     Knoten vor = sucheVor(nutz);
20     if (vor.next == end) return false; // nutz nicht in dieser Sammlung
21     vor.next = vor.next.next; // Entferne einen Knoten
22     return true;
23 }
```

11. SU Mo 07.12.09

A. Wiederholung

B. Organisation

Strukturen von Sammlungen (Fortsetzung)

Schrittzahlen für die Operationen Einfügen, Suchen pos., Suchen neg., Entfernen bei unsortierten und sortierten verketteten Listen:

Struktur der Sammlung	Operation	Anz. Schritte in einem schlimmsten Fall	Anz. Schritte im Durchschnitt
verkettete Liste (unsortiert)	Einfügen	1	1
	Suchen pos.	n	n/2
	Suchen neg.	n	n
	Entfernen	Suchen + 1	Suchen + 1
verkettete Liste (sortiert)	Einfügen	Suchen + 1	Suchen + 1
	Suchen pos.	n	n/2
	Suchen neg.	n	n/2
	Entfernen	Suchen + 1	Suchen + 1

Welchen Vorteil haben *sortierte* verkettete Listen im Vergleich zu *unsortierten* verketteten Listen? (Nur das *Suchen neg.* geht schneller).

Welchen Vorteil haben *unsortierte* verkettete Listen im Vergleich zu *sortierten* verkettete Listen? (Das *Einfügen* geht schneller).

Welchen *Vorteil* hat eine *sortierte Reihung* im Vergleich zu einer *verketteten Liste*? (Das *Suchen pos.* und *neg.* geht viel schneller)

Welchen *Nachteil* hat eine *sortierte Reihung* im Vergleich zu einer *verketteten Liste*? (Das Einfügen braucht immer n Schritte statt im Durchschnitt n/2, Reihungen sind aus Beton und nicht aus Gummi)

Kann man die Vorteile von *sortierten Reihungen* (binäres Suchen) und *verketteten Listen* (gummiartige Flexibilität) nicht kombinieren?

Ja: Mit binären Bäumen !

Def.: Ein *binärer Baum* ist ein Leerbaum oder er besteht aus einem Knoten, an dem zwei *binäre Bäume* hängen (linker und rechter Unterbaum).

Diese Definition ist rekursiv!

Ab jetzt ist mit *Baum* immer so ein *binärer Baum* gemeint.

Begriff der *Tiefe* eines Baums anhand von Beispielen erläutern.

Zusammenhang zwischen der *Tiefe* eines Baums und der maximalen *Anzahl seiner Knoten* behandeln.

Ein Baum der Tiefe t enthält höchstens wie viele Knoten?

Ein Baum mit n Knoten hat eine Tiefe von mindestens?

Dazu 2 Tabellen (t: Tiefe eines Baumes, n: Anzahl der Knoten des Baumes):

t	max n	n	min t
1	1	1	1
2	3	2-3	2
3	7	4-7	3
4	15	8-15	4
...
t	$2^t - 1$	n	$\lfloor \log_2(n) \rfloor + 1$

Die Klammern $\lfloor \dots \rfloor$ sollen bedeuten: Zur nächst kleineren oder gleichen ganzen Zahl abrunden.

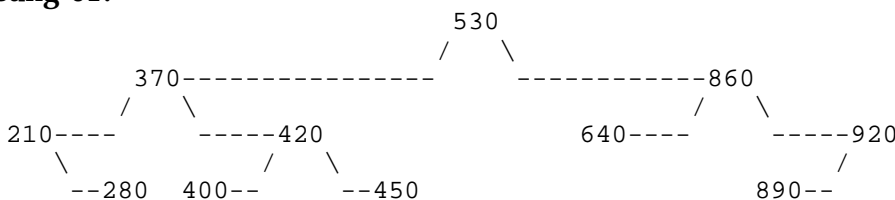
Angenommen, jeder Knoten enthält einen *Schlüssel-Wert* (z.B. eine Zahl oder einen String). Wann ist ein Baum (nach diesen Schlüsseln) *sortiert*?

Def.: Ein Baum ist *sortiert*, wenn für jeden Knoten K gilt: Der Schlüssel von K ist größer als alle Schlüssel im linken Unterbaum von K und kleiner als alle Schlüssel im rechten Unterbaum von K.

Diese Definition auswendig lernen! Auf den ersten Blick sieht sie evtl. einfacher aus als sie ist.

Aufgabe-01: Fügen Sie die folgenden Schlüssel (in der hier angegebenen Reihenfolge) in einen (anfänglich leeren) sortierten binären Baum ein: 530, 370, 420, 860, 210, 640, 920, 890, 280, 450, 400

Lösung-01:



Zur Entspannung: Ein Blatt Papier 50 Mal halbieren und stapeln

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

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

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

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

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

- 1 mm 10 Schichten
- 1 m 10 000 Schichten
- 1 km 10 000 000 Schichten (d.h. 10^7 Schichten)
- $10^{15} / 10^7$ ist gleich 10^8 gleich 100 Millionen km

Wiederholungsfragen, 12. SU, Mo 14.12.09

1. Fügen Sie die folgenden Schlüssel (in der hier angegebenen Reihenfolge) in einen (anfänglich leeren) *sortierten binären Baum* ein: 530, 370, 420, 860, 210, 640, 920, 890, 280, 450, 400
2. Ebenso für die Schlüssel 210, 280, 370, 400, 420, 450, 530, 640, 860, 890, 920
3. Ebenso für die Schlüssel 920, 210, 890, 280, 860, 370, 640, 400, 530, 420, 450
4. *Wie viele Knoten* enthält ein binärer Baum der Tiefe 12 höchstens (ungefähr)?
5. Welche *Tiefe* muss ein Baum mit 4 Millionen Knoten mindestens haben?
6. Was muss für die Knoten und Schlüssel eines binären Baumes gelten, damit er *sortiert* ist?

Rückseite der Wiederholungsfragen, 12 SU, Mo 14.12.09

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 Trillionen 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 überalle *komplementäre* Nukleotide gegenüberstehen, etwa so:

```
Kette 1: A.C.G.T. ...
          | | | |
Kette 2: T.G.C.A. ...
```

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 . Eine Straße von Stadt A nach Stadt B sieht etwa so aus:

A-nach-B: ATGCTAGAAA AATCCATGCG

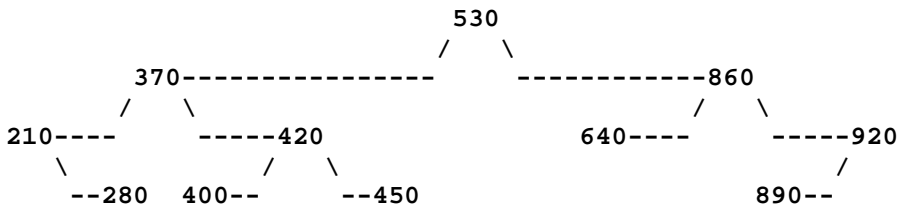
Ein solches Molekül kann sich mit dem Komplement von Stadt B verbinden wie folgt:

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

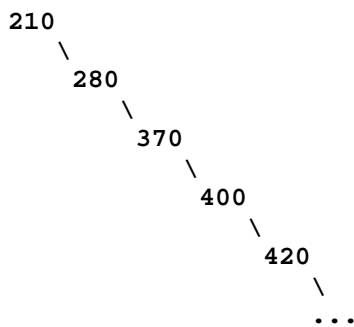
Dieses Molekül kann sich mit einer Straße B-nach-C verbinden etc.

Antworten zu den Wiederholungsfragen, 12. SU, Mo 14.12.09

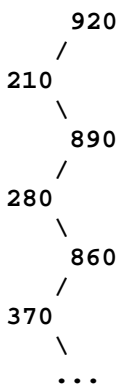
1. Fügen Sie die folgenden Schlüssel (in der hier angegebenen Reihenfolge) in einen (anfänglich leeren) *sortierten binären Baum* ein: 530, 370, 420, 860, 210, 640, 920, 890, 280, 450, 400



2. Ebenso für die Schlüssel 210, 280, 370, 400, 420, 450, 530, 640, 860, 890, 920



3. Ebenso für die Schlüssel 920, 210, 890, 280, 860, 370, 640, 400, 530, 420, 450



4. Wie viele Knoten enthält ein binärer Baum der Tiefe 12 höchstens (ungefähr)?

Ungefähr 4 Tausen.

5. Welche Tiefe muss ein Baum mit 4 Millionen Knoten mindestens haben?

Die Tiefe 22 (2^{20} ist 1 Million, 2^2 ist 4)

6. Was muss für die Knoten und Schlüssel eines binären Baumes gelten, damit er sortiert ist?

Für jeden Knoten K muss gelten: Der Schlüssel von K ist größer als alle Schlüssel im linken Unterbaum von K und kleiner als alle Schlüssel im rechten Unterbaum von K.

12. SU Mo 14.12.09**A. Wiederholung****B. Organisation****Suchen in einem binären Baum**

Einen Baum mit Schlüsseln A, B, C, ... in den Knoten anzeichnen und daran das Suchen demonstrieren.

Wie viele Schritte braucht man beim Suchen in einem Baum mit n Knoten? (So viel wie die Tiefe des Baumes angibt, d.h. im schlimmsten Fall etwa $\log_2(n)$).

Wann ist ein Baum besonders gut zum Suchen geeignet und wann ist er besonders schlecht?

Ein weicher Begriff: Ein Baum mit n Knoten gilt als *balanciert*, wenn seine Tiefe nicht größer als $2 * \log_2(n)$ ist.

Einfügen und balancieren: Man kann beim Einfügen von Knoten in einen binären Baum "bestimmte Tricks" anwenden die bewirken, dass der Baum immer balanciert bleibt (egal in welcher Reihenfolge die Schlüssel eingefügt werden, siehe Wiederholungsfragen).

Entfernen: Das Entfernen eines Knotens aus einem binären Baum erfordert ebenfalls einen etwas komplizierteren Algorithmus, den wir hier nicht besprechen.

Zur Entspannung: Mit DNS-Molekülen algorithmische Probleme lösen

siehe Rückseite der Wiederholungsfragen

Das Papier über Hash-Tabellen austeilen und besprechen.

Wiederholungsfragen, 13. SU, Mo 21.12.09

1. Welche *Tiefe* hat ein binärer Baum mit 8 Milliarden Knoten mindestens?
2. Welche *Tiefe* hat ein binärer Baum mit 8 Milliarden Knoten höchstens?
3. *Wie viele Knoten* enthält ein Baum der Tiefe 14 höchstens (ungefähr)?
4. *Wie viele Knoten* enthält ein Baum der Tiefe 14 mindestens?
5. Was ist eine *Hash-Tabelle*? Geben Sie die im letzten seminaristischen Unterricht (SU) behandelte Kurzdefinition wieder.
6. Was macht eine *allgemeine Hash-Funktion* (Parameter? Ergebnis?)?
7. Sei `ht` eine Hash-Tabelle, in der wir `JButton`-Objekte sammeln wollen. Was macht eine spezielle Hash-Funktion für `ht` (d.h. eine auf die Hash-Tabelle `ht` zugeschnittene Hash-Funktion)?
8. Schreiben Sie eine Methode entsprechend der folgenden Spezifikation:

```
1 static public void gibAus(ArrayList<Integer> ali) {
2     // Gibt die Komponenten der Sammlung ali zur Standardausgabe aus,
3     // mit pln, eine Komponente pro Zeile
4     // (natuerlich mit einer for-each-Schleife!).
5     ...
6 }
```

9. Schreiben Sie eine Methode entsprechend der folgenden Spezifikation:

```
1 static public void gibAus(int[] ir) {
2     // Gibt die Komponenten der Reihung ir zur Standardausgabe aus,
3     // mit pln, eine Komponente pro Zeile
4     // (natuerlich mit einer for-each-Schleife!).
5     ...
6 }
```

Antworten auf die Wiederholungsfragen, 13. SU, Mo 21.12.09

1. Welche *Tiefe* hat ein binärer Baum mit 8 Milliarden Knoten mindestens?

Mindestens eine Tiefe von 33 (1 Milliarde ist 2^{30} , 8 ist 2^3 , $2^{30} * 2^3$ ist gleich 2^{33}).

2. Welche *Tiefe* hat ein binärer Baum mit 8 Milliarden Knoten höchstens?

Die Tiefe 8 Milliarden.

3. *Wie viele Knoten* enthält ein Baum der Tiefe 14 höchstens (ungefähr)?

Höchstens etwa 16 Tausend Knoten (2^{10} ist etwa 1 Tausend, 2^4 ist 16, $16 * 1$ Tausend ist 16 Tausend).

4. *Wie viele Knoten* enthält ein Baum der Tiefe 14 mindestens?

Mindestens 14 Knoten.

5. Was ist eine *Hash-Tabelle*? Geben Sie die im letzten seminaristischen Unterricht (SU) behandelte Kurzdefinition wieder.

Eine Hash-Tabelle ist eine Reihung von (verketteten) Listen.

6. Was macht eine *allgemeine Hash-Funktion* (Parameter? Ergebnis?)?

Sie berechnet aus einem Objekt eine Ganzzahl (oder: einen int-Wert).

7. Sei *ht* eine Hash-Tabelle, in der wir JButton-Objekte sammeln wollen. Was macht eine spezielle Hash-Funktion für *ht* (d.h. eine auf die Hash-Tabelle *ht* zugeschnittene Hash-Funktion)?

Sie berechnet aus einem JButton-Objekt einen Index für *ht* (d.h. eine Zahl zwischen 0 und *ht.length*-1).

8. Schreiben Sie eine Methode entsprechend der folgenden Spezifikation:

```
1 static public void gibAus(ArrayList<Integer> ali) {
2     // Gibt die Komponenten der Sammlung ali zur Standardausgabe aus,
3     // mit pln, eine Komponente pro Zeile
4     // (natuerlich mit einer for-each-Schleife!).
5
6     for (Integer n : ali) pln(n);
7 }
```

9. Schreiben Sie eine Methode entsprechend der folgenden Spezifikation:

```
1 static public void gibAus(int[] ir) {
2     // Gibt die Komponenten der Reihung ir zur Standardausgabe aus,
3     // mit pln, eine Komponente pro Zeile
4     // (natuerlich mit einer for-each-Schleife!).
5
6     for (int n : ir) pln(n);
7 }
```

13. SU Mo 21.12.09

A. Wiederholung

B. Organisation

Lösungen für einige Aufgaben im Test 6:

Wenn man ein Objekt der Klasse `Bianca` in eine Sammlung des Typs `MyHashSet` einfügt (mit der Methode `add`), soll eine Ausnahme des Typs `ClassCastException` geworfen werden. Es genügt, `Bianca` z.B. wie folgt zu vereinbaren:

```
1 class Bianca implements Serializable {}
```

Wenn man ein Objekt der Klasse `Detlef` in eine Sammlung des Typs `MyHashSet` einfügt (mit der Methode `add`), soll eine Ausnahme des Typs `IllegalArgumentException` geworfen werden. Es genügt, `Detlef` z.B. wie folgt zu vereinbaren:

```
2 class Detlef {
3     public String toString() { return "12345678901"; }
```

Was ist so toll an Modulen?

Zur Erinnerung: Ein *Modul* ist ein Behälter für Variablen (Attribute), Unterprogramme (Methoden), Typen, Module etc., der aus mindestens *zwei Teilen* besteht, einem öffentlichen (oder sichtbaren oder ungeschützten) und einem privaten (oder nicht-sichtbaren oder geschützten) Teil. Von außerhalb des Moduls kann man nur auf die Größen im öffentlichen Teil zugreifen, aber nicht auf die Größen im privaten Teil.

In Java bestehen Module sogar aus *vier Teilen*, einem öffentlichen Teil (`public`), einem halböffentlichen Teil (`protected`), einem halbprivaten Teil (paketweit erreichbar, ohne Kennzeichnung durch ein Schlüsselwort) und einem privaten Teil (`private`).

Wichtige Vorteile von Modulen:

1. Fehlersuche wird vereinbart

Angenommen, ein Programm besteht aus 100 Modulen. Im Modul `m87` wird eine `int`-Variable `anzahl` vereinbart, die eigentlich nur positive Zahlen enthalten darf. Beim Testen stellen wir fest, dass `anzahl` eine negative Zahl enthält. Wo müssen wir die falschen Befehle suchen?

Wenn `anzahl` im öffentlichen Teil von `m87` vereinbart wurde: *In allen 100 Modulen.*

Wenn `anzahl` im privaten Teil von `m87` vereinbart wurde: *Nur im Modul `m87`.*

2. Abhängigkeiten zwischen Modulen können vermindert und nachträgliche Änderungen (Verbesserungen) in einem Modul ermöglicht werden.

Größen im öffentlichen Teil eines Moduls werden in aller Regel in erstaunlich kurzer Zeit von erstaunlich vielen Kollegen benutzt. Solche Größen zu ändern (z.B. den Namen oder den Typ einer Variablen oder den Ergebnistyp einer Methode) ist dann sehr schwer oder unmöglich (weil viele Kollegen dann ihre Programme ändern müssten). Größen im privaten Teil sind viel leichter zu ändern, weil kein Kollege darauf zugreifen und sein Programm von dieser Größe abhängig machen kann.

Daten formatieren, was ist damit gemeint?

Wenn man Daten mit den Methoden `p` oder `println` ausgibt, werden sie "möglichst schlicht" ausgegeben, z.B. werden für die Zahl 123 drei Dezimalziffern ("123") ausgegeben. Bei Bruchzahlen ist die Ausgabe schon etwas schwerer vorherzusagen:

```
1 // Ausgabe
2 println(0.12345678901234567890); // 0.12345678901234568
3 println(0.12345678901234567890F); // 0.12345679
```

`double`-Werte werden mit maximal 17 Nachpunktstellen (gerundet) ausgegeben,

`float`-Werte mit maximal 8 Nachpunktstellen (gerundet).

Häufig will man Daten beim Ausgeben noch speziell formatieren, z.B. so:

- Ganzzahlen sollen in einer *bestimmten Breite* ausgegeben werden, die unabhängig von der Größe der Zahl ist, z.B. so:

```
1234567890 // Breite 10
      123 // Breite 10
    9876543 // Breite 10
```

- Die Ziffern von Ganzzahlen sollen durch *Trennzeichen* in Gruppen eingeteilt werden, um die Lesbarkeit zu verbessern,

statt 1000000000 oder 10000000000 besser
so 1.000.000.000 bzw. 10.000.000.000

- bei Bruchzahlen möchte man häufig eine bestimmte Anzahl von *Nachpunktstellen* ausgeben (und nicht immer 17 bei double und 8 bei float)

- bei allen Zahlen möchte man das Vorzeichen auf eine bestimmte Art darstellen lassen, z.B.

```
Euro 1234 // Bei pos. Zahlen: nichts
Euro -1234 // Bei neg. Zahlen: '-' davor
-----
Euro 1234 // Bei pos. Zahlen: ' ' davor
Euro -1234 // Bei neg. Zahlen: '-' davor
-----
Euro +1234 // Bei pos. Zahlen: '+' davor
Euro -1234 // Bei neg. Zahlen: '-' davor
```

Bei Bruchzahlen ganz ähnlich.

- ein Tagesdatum möchte man in einer bestimmten Form ausgeben, z.B.

21.12.09 oder 21.12.2009 oder 21. Dez 2009 oder 2009-12-21 oder ...

- häufig möchte man, dass Ganzzahlen, Bruchzahlen und Datums-Angaben nach den *Konventionen* eines bestimmten Landes oder einer bestimmten Gruppe von Menschen formatiert wird, z.B. der double-Wert 1234567.8888 wird in einigen Ländern üblicherweise so formatiert (wenn man 3 Nachpunktstellen haben möchte):

```
1.234.567,889 // In Italien und Deutschland etc.
1 234 567,889 // In Frankreich
1,234,567.889 // In England, den USA, Canada etc.
```

Eine praktische Erfahrung:

Das Programmieren *komplizierter Algorithmen* geht manchmal erstaunlich *schnell*.

Das Programmieren einer *gut aussehenden und lesbaren Ausgabe* dauert häufig überraschend *lange* und erfordert erstaunlich viele Verbesserungen und Veränderungen ("hier fehlt noch ein Pünktchen und dort ist noch ein Leerzeichen zu viel und diese Zeile ist verrutscht ...").

8.4 Methoden mit variabel vielen Parametern (S. 196)

Kein gewichtiges *Grundkonzept der Programmierung*, aber eine kleine, nützliche Einrichtung.

Zur Entspannung: Eine Parabel von Edsger W. Dijkstra

Es war einmal eine große Eisenbahngesellschaft in den USA, die jeden Tag viele Züge z.B. von New York nach San Franzisko oder von Los Angeles nach Boston fahren ließ. Eines Tages fiel einem jungen und ehrgeizigen Manager dieser Gesellschaft auf, dass ...

Wiederholungsfragen, 14. SU, Mo 04.01.10

1. Was ist ein Modul?
2. Beschreiben Sie (möglichst kurz, in Forma von Stichworten) *zwei* wichtige *Vorteile* von Modulen.
3. Wenn man in einem Programm Daten ausgibt, muss man sie vorher häufig noch *formatieren*. Was ist dabei mit "formatieren" gemeint? Beschreiben Sie (möglichst kurz) zwei Beispiele (Was für Daten sollen ausgegeben werden? Wie sollen sie formatiert werden?).
4. Was gibt der folgende Befehl aus?

```
printf("0,03 %% ist gleich%n0,3 Promille%n");
```
5. Schreiben Sie eine Methode namens `max` mit dem Ergebnistyp `float`, die man mit einem oder mehr Parametern (z.B. 2 oder 17 oder 35 ...) vom Typ `float` aufrufen kann und die den größten ihrer Parameter als Ergebnis liefert.

Rückseite der Wiederholungsfragen, 14. SU, Mo 04.01.10

Der Anfangskommentar der Java-Quelldatei Datei XmlJStd.java
(zu finden im Archiv DateienFuerPr2/XML):

```
1 // Datei XmlJStd.java
2 /* -----
3 Dieser Modul enthaelt Methoden zum Bearbeiten von XML-Dateien und
4 benutzt dazu nur die Standardbibliothek von Java 6.
5
6 Die wichtigen Methoden:
7 lies           Liest eine XML-Datei und erzeugt daraus ein
8                 entsprechendes DOM-Document-Objekt
9                 (mit DOMImplementationLS)
10 lies2         Liest eine XML-Datei und erzeugt daraus ein
11                entsprechendes DOM-Document-Objekt
12                (mit DocumentBuilderFactory)
13 liesVD        Liest eine XML-Datei und erzeugt daraus ein
14                entsprechendes DOM-Document-Objekt.
15                Validiert die XML-Datei dabei gegen ihre DTD.
16 liesVS        Liest eine XML-Datei und erzeugt daraus ein
17                entsprechendes DOM-Document-Objekt.
18                Validiert die XML-Datei dabei gegen eine
19                XML-Schema-Datei.
20 istGuechtigVD Prueft, ob ein DOM-Document-Objekt gueltig ist
21                gegen seine DTD.
22 istGuechtigVS Prueft, ob ein DOM-Document-Objekt gueltig ist
23                gegen eine XML-Schema-Datei.
24 schreib       Schreibt ein DOM-Document-Objekt im XML-Format
25                in eine Datei
26
27 Weniger wichtige Methoden:
28 gibTextDateiAus Gibt eine beliebige Textdatei zur Standardausgabe aus.
29 gibLesbarAus   Gibt ein DOM-Document-Objekt in einer selbstgestrickten
30                Form ("als Einrueck-Baum") zur Standardausgabe aus
31 -----
32 ...
33 ----- */
```

Antworten zu den Wiederholungsfragen, 14. SU, Mo 04.01.10**1. Was ist ein Modul?**

Ein Modul ist ein Behälter für Variablen (Attribute), Unterprogramme (Methoden), Typen, Module etc., der aus mindestens zwei Teilen besteht, einem öffentlichen (oder sichtbaren oder ungeschützten) und einem privaten (oder nicht-sichtbaren oder geschützten) Teil. Von außerhalb des Moduls kann man nur auf die Größen im öffentlichen Teil zugreifen, aber nicht auf die Größen im privaten Teil.

2. Beschreiben Sie (möglichst kurz, in Forma von Stichworten) zwei wichtige Vorteile von Modulen.

2.1. Variablen im privaten Teil eines Moduls können von außerhalb des Moduls nicht (gelesen und nicht) verändert werden.

2.2. Der Programmierer eines Moduls kann z.B. den Namen oder Typ einer Variablen oder die Anzahl der Parameter eines Unterprogramms im privaten Bereich im relativ leicht ändern. Das Ändern einer Größe G im öffentlichen Teil eines Moduls ist viel schwieriger bis unmöglich, weil viele andere Programmteile von G abhängen können und dann auch geändert werden müssen.

3. Wenn man in einem Programm Daten ausgibt, muss man sie vorher häufig noch *formatieren*. Was ist dabei mit "formatieren" gemeint? Beschreiben Sie (möglichst kurz) zwei Beispiele (Was für Daten sollen ausgegeben werden? Wie sollen sie formatiert werden?).

Ganzzahlen in einer bestimmten Breite ausgeben (unabhängig von der Größe der Zahlen)

Bruchzahlen mit einer bestimmten Anzahl von Nachpunktstellen ausgeben.

Ein Tages-Datum in einer bestimmten Form ausgeben.

4. Was gibt der folgende Befehl aus?

```
printf("0,03 %% ist gleich%n0,3 Promille%n");
```

Ausgabe:

```
0,03 % ist gleich
0,3 Promille
```

5. Schreiben Sie eine Methode namens `max` mit dem Ergebnistyp `float`, die man mit einem oder mehr Parametern (z.B. 2 oder 17 oder 35 ...) vom Typ `float` aufrufen kann und die den größten ihrer Parameter als Ergebnis liefert.

```
1 float max(float f1, float... fr) {
2     float erg = f1;
3     for (float f : fr) if (f > erg) erg = f;
4     return erg;
5 }
```

14. SU Mo 04.01.10

A. Wiederholung
B. Organisation

Die Methoden formatter und printf

Das sind Methoden mit variabel vielen Parametern.

Den `printf`-Befehl muss man selbst *aktiv lernen*, indem man sich Beispiele ausdenkt und ausprobiert. Auf meiner Netzseite gibt es zwei *Applets*, mit denen man den `printf`-Befehl *ausprobieren* und seine Anwendung *üben* kann:

eTeachMePrintf (neuer, Bachelor-Arbeit eines BHT-Studenten, empfehlenswert)
PrintfApplet (älter, von mir, nur noch in seltenen Fällen empfehlenswert)

Die Grundversion des `printf`-Befehls muss man mit mind. *einem* Param vom Typ `String` aufrufen. Diesen ersten Param bezeichnet man als *Format-String* (des *printf* bzw. *format*-Befehls).

Der Formatstring darf beliebigen "normalen Text" enthalten und ausserdem *besondere Stellen*, die mit einem *Prozentzeichen* `%` beginnen. Der `printf`-Befehl gibt im Wesentlichen diesen Formatstring aus (nachdem er die besonderen Stellen besonders behandelt hat).

In einem Formatstring bezeichnet `%%` *ein* Prozentzeichen und `%n` einen *Zeilenwechsel* der benutzten Plattform (unter Unix/Linux ein LF-Zeichen, unter Mac OS ein CR-Zeichen und unter Windows ein CR-Zeichen gefolgt von einem LF-Zeichen). Meist ist `\n` nur ein schlechter Ersatz für `%n`!

Außerdem kann ein Formatstring sog. *Umwandlungsbefehle* enthalten, die mit einem Prozentzeichen beginnen und mit einem sog. *Umwandlungsbuchstaben* enden, z.B. `%d` oder `%5d` oder `%0,5d` oder `%8X` etc.

Für jeden *Umwandlungsbefehl* im Formatstring muss man (normalerweise) hinter dem Formatstring einen weiteren *zusätzlichen Parameter* angeben.

```
format( "%d Euro %d Cents%n", euros, cents );
```

Beispiel:

```
1 printf("%d Euro und %d Cents", n1, n2); // richtig
2 printf("%d Euro und %d Cents", n1, n2, n3); // falsch (harmlos)
3 printf("%d Euro und %d Cents", n1); // falsch (Ausnahme wird geworfen)
```

In Zeile 1: Im Formatstring *2 Umwandlungsbefehle*, danach *2 zusätzliche Parameter*.

In Zeile 2: *3 zusätzliche Parameter* (der dritte wird ignoriert).

In Zeile 3: *Nur 1 zusätzlicher Parameter*. Eine Ausnahme des Typs `MissingFormatArgumentException` wird geworfen.

S. 266: Struktur und Bestandteile eines Umwandlungsbefehls (in einem `printf`-Formatstring)

`%[Index][Schalter][Breite][.Genauigkeit]UBuchstabe`

Die Bestandteile in eckigen Klammern sind optional ("sie müssen nicht sein"). Lernen Sie die Vokabeln *Index*, *Schalter*, *Breite*, *Genauigkeit* und *UBuchstabe*! Sie sind nützlich, um Beschreibungen des `printf`-Befehls und Tips dazu zu verstehen (z.B. die beiden Applets auf meiner Netzseite).

S. 258, Beispiel-02: Ganzzahlen formatieren

Die hier vereinbarten Variablen `i01` bis `i03` werden auch in späteren Beispielen benutzt.

S. 260, Beispiel-04: Bruchzahlen formatieren

Zur Entspannung: Die Regel von Moore und Hardware-Tendenzen

1965 formulierte Gordon Moore (einer der Gründer der Firma Intel) eine einfache Regel: "Etwa alle 2 Jahre verdoppelt sich die Anzahl der Transistoren, die auf einen Chip passen."

Diese einfache Regel ist weder ein juristisches noch ein physikalisches Gesetz, aber als "Daumenregel" sehr nützlich. Allerdings hat man sie inzwischen neueren Entwicklungen angepasst: Anstelle von "2 Jahren" nimmt man heute etwa 18 Monate an. Offenbar gilt heute (2005):

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

Daraus folgt:

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

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

Die nächste einschneidend wichtige Hardware-Entwicklung: *Nicht-flüchtige Hauptspeicher*. Hoffnung: Kein Hochfahren (booten) mehr notwendig. Wenn man einen Rechner anschaltet, ist er nach weniger als einer Sekunde betriebsbereit (und macht genau da weiter, wo er beim Ausschalten war).

XML (eXtensible Markup Language, Erweiterbare Auszeichnungssprache)

Problem: Viele Programme geben Daten aus, die später von anderen *Programmen* weiter verwendet werden sollen.

Damit das klappt, muss das ausgebende Programm die Daten in einem bestimmten Format ausgeben, und die weiterverarbeitenden Programme müssen dieses Format kennen und "wissen, wo was steht".

Man unterscheidet zwei Arten von solchen Datenformaten: **binäre Formate** und **Text-Formate**.

Daten in einem *binären Format* belegen typischerweise *weniger Speicherplatz* (und können schneller übertragen werden) als Text-Daten, sind aber von Menschen *nicht* ohne weiteres *lesbar*.

Daten in einem *Text-Format* können von Menschen (mehr oder weniger leicht) *gelesen* werden, belegen aber typischerweise *mehr Speicherplatz* (und kosten *mehr Übertragungszeit*) als Daten in einem binären Format.

Beispiele: *Ausführbare Dateien* (unter Windows: `.exe`-Dateien) enthalten Programme (Befehlsfolgen) in einem *binären Format*. *Bytecode-Dateien* (`.class`-Dateien) enthalten Klassenvereinbarungen in einem anderen binären Format. *Skripte* (z.B. unter DOS sog. `.bat`-Dateien, unter Linux z.B. `bash`-Skripte, oder Skripte in Sprachen wie Perl oder PHP oder Tcl etc.) sind ebenfalls ausführbar, enthalten ihre Befehle aber in einem *Text-Format*.

Trend: 1 GB Speicher wird immer billiger, die Ausführung von einer Milliarde Maschinenbefehlen (1 Giga Maschinenbefehle) wird immer billiger, die Übertragung von Daten wird immer schneller. Die Vorteile von *binären Formaten* gegenüber *Text-Formaten* werden somit in immer mehr Anwendungsfällen immer *unwichtiger*. Ein starker Trend geht in Richtung von *Text-Formaten*.

Papier XML03 (1 Blatt, 4 Seiten) austeilen.

Klassische Textformate: CSV (Character Separated Values), insbesondere "ein Wert pro Zeile".

Beispiel-01: Daten zu einer Person in einem CSV-Format (Trennzeichen: Komma):

```
Jonas, Karl, 182, 85, 714430
```

Beschrieben wird hier Karl Jonas, der 182 cm groß ist, 85 kg wiegt und sich an der BHT unter der Nr. 714430 immatrikuliert hat.

Nachteile solcher Formate: Fehleranfällig:

- Wenn *ein* Wert verloren geht, werden alle nachfolgenden Werte falsch interpretiert
- Aus den Daten kann man "die richtige Reihenfolge" der einzelnen Werte nur schwer oder gar nicht rekonstruieren (ist Jonas der Vor- oder der Nachname? Ist die erste Zahl 182 das Gewicht in kg oder die Größe in cm? etc.).

Wichtige Grundidee: **Auszeichnungssprachen**

Man schreibt in eine Textdatei nicht nur die **Daten**, um die es im Kern geht, sondern zusätzlich sog. **Auszeichnungen** (engl. markups), die beschreiben, welche Daten wo stehen.

Beispiel-02: Ein Applet in der Auszeichnungs- und Programmiersprache Curl (aus Wikipedia):

```
1 {Curl 5.0 applet}
2 {applet license="development"}
3 {text
4   color = "blue",
5   font-size = 16pt,
6   Hello World}
```

Jeweils das erste Wort nach einer öffnenden geschweiften Klammer ist besonders wichtig und legt fest, wie die nachfolgenden Elemente zu interpretieren sind.

Curl gehört einer japanischen Firma, ist vor allem in Japan und Asien populär, wird aber auch am MIT in Boston weiterentwickelt und erforscht.

Beispiel-03: Noch ein Curl-Text/Programm (aus Wikipedia)

```
1 {paragraph
2   paragraph-left-indent=0.5in,
3   {text color = "red", font-size = 12pt,
4     Hello}
5   {text color = "green", font-size = 12pt,
6     World}}
```

Die Sprache **YAML** (ursprünglich eine Abkürzung für *Yet Another Markup Language*, heute für *YAML Ain't Markup Language*) ist trotz ihres rekursiven und etwas widersprüchlichen Namens auch eine Art Auszeichnungssprache. Grundkonzepte: *Listen*, *assoziative Listen* und *Einzelwerte*.

--- leitet (innerhalb eines YAML-Dokuments) einen neuen Abschnitt ein, mit # beginnt ein Kommentar bis zum Zeilenende, mit - beginnen die Elemente einer Liste, : trennt in einer assoz. Liste einen Schlüssel von seinem Wert, | bedeutet: Zeilenumbrüche beibehalten, > bedeutet: Zeilenumbrüche nicht beibehalten.

Beispiel-04: Zwei Listen in YAML (aus Wikipedia):

```
1 --- # Favorite movies, block format
2 - Casablanca
3 - Spellbound
4 - Notorious
5 --- # Shopping list, inline format
6 [milk, bread, eggs]
```

Beispiel-05: Zwei Abbildungen (engl. maps or hashes, bestehen aus Schlüssel+Wert-Einträgen) in YAML (aus Wikipedia):

```
1 --- # Block
2 name: John Smith
3 age: 33
4 --- # Inline
5 {name: John Smith, age: 33}
```

Hier sind name und age Schlüssel und John Smith bzw. 33 die zugehörigen Werte.

Hierarchische Strukturen kann man wahlweise durch *Klammern* oder durch *Einrückung* darstellen. Im **Beispiel-05**

unter # Block muss jeder Eintrag auf einer neuen Zeile beginnen (durch *Einrückung*)

unter # Inline kann man die Darstellung der Abbildung über beliebig viele oder wenige Zeilen verteilen (durch *Klammern*).

Wiederholungsfragen, 15. SU, Mo 11.01.10

1. Wie bezeichnet man den wichtigsten (meistens ersten) Parameter eines `printf`-Befehls?
2. Was haben der `printf`- und der `format`-Befehl *gemeinsam*? Was machen sie *beide* (in einem ganz kurzen Satz und nur ganz allgemein beschrieben, ohne Einzelheiten)?
3. Wodurch unterscheidet sich der `printf`-Befehl vom `format`-Befehl? Was macht nur der `printf`-Befehl und was macht nur der `format`-Befehl?
4. Welche Strings werden von den folgenden Ausdrücken geliefert? Notieren Sie ihre Lösungen so, dass genau erkennbar ist, aus wie vielen und welchen Zeichen die Strings bestehen.

```
1 ... String.format("Sum: %5d", 123) ...
2
3 ... String.format("Sum: %-5d", 123) ...
4
5 ... String.format("Sum: %+5d", 123) ...
6
7 ... String.format("Sum: %+5d", -123) ...
8
9 ... String.format("Sum: %+-5d", 123) ...
10
11 ... String.format("Sum: %+-5d", -123) ...
12
13 ... String.format("Sum: %05d", 123) ...
14
15 ... String.format("Sum: %5d", 1234567) ...
16
17 ... String.format("Sum: %,d", 1234567) ...
```

Antworten zu den Wiederholungsfragen, 15. SU, Mo 11.01.10

1. Wie bezeichnet man den wichtigsten (meistens ersten) Parameter eines `printf`-Befehls?

Formatstring

2. Was haben der `printf`- und der `format`-Befehl *gemeinsam*? Was machen sie *beide* (in einem ganz kurzen Satz und nur ganz allgemein beschrieben, ohne Einzelheiten)?

Beide formatieren Daten.

3. Wodurch unterscheidet sich der `printf`-Befehl vom `format`-Befehl? Was macht nur der `printf`-Befehl und was macht nur der `format`-Befehl?

Der `printf`-Befehl gibt die formatierten Daten aus, der `format`-Befehl liefert die formatierten Daten als String.

4. Welche Strings werden von den folgenden Ausdrücken geliefert? Notieren Sie ihre Lösungen so, dass genau erkennbar ist, aus wie vielen und welchen Zeichen die Strings bestehen.

Die senkrechten Striche ' | ' gehören nicht zu den Ergebnis-Strings. Sie sollen Blanks am Ende erkennbar machen.

Ergebnis-Strings:

1 ... <code>String.format("Sum: %5d", 123) ...</code>	Sum: 123
2	
3 ... <code>String.format("Sum: %-5d", 123) ...</code>	Sum: 123
4	
5 ... <code>String.format("Sum: %+5d", 123) ...</code>	Sum: +123
6	
7 ... <code>String.format("Sum: %+5d", -123) ...</code>	Sum: -123
8	
9 ... <code>String.format("Sum: %+-5d", 123) ...</code>	Sum: +123
10	
11 ... <code>String.format("Sum: %+-5d", -123) ...</code>	Sum: -123
12	
13 ... <code>String.format("Sum: %05d", 123) ...</code>	Sum: 00123
14	
15 ... <code>String.format("Sum: %5d", 1234567) ...</code>	Sum: 1234567
16	
17 ... <code>String.format("Sum: %,d", 1234567) ...</code>	Sum: 1.234.567

15. SU Mo 11.01.10

A. Wiederholung

B. Organisation

Auszeichnungssprachen (markup languages), Fortsetzung

Letzte Woche haben wir schon einen kurzen Blick auf die Sprachen *Curl* und *YAML* geworfen.

Heute sollen noch *JSON* und *HTML* kurz betrachtet werden.

YAML passt besonders gut zu den Programmiersprachen Perl, Python, PHP, Ruby und Java, die mit ganz ähnlichen Datenstrukturen (Reihungen und Abbildungen) arbeiten. YAML ist eine *Obermenge* von JSON (jedes wohlgeformte JSON-Dokument ist auch ein wohlgeformtes YAML-Dokument).

JSON

Die Sprache JSON (Java Script Object Notation) wurde zusammen mit der Sprache JavaScript entwickelt, um Objekte (von objektorientierten Programmiersprachen) kompakt und doch lesbar darzustellen. Sie kann auch ganz unabhängig von JavaScript als allgemeine Auszeichnungssprache benutzt werden. Ist heute ein wichtiger Bestandteil von JavaFX.

Beispiel-06: Ein Objekt, in JSON dargestellt (aus Wikipedia):

```

1 {
2   "Kreditkarte"   : "Xema",
3   "Nummer"       : "1234-5678-9012-3456",
4   "Inhaber"      : {
5     "Name"        : "Reich",
6     "Vorname"     : "Rainer",
7     "Geschlecht" : "\"männlich\"",
8     "Vorlieben"  : [
9       "Reiten",
10      "Schwimmen",
11      "Lesen"
12    ],
13    "Alter"       : null
14  },
15  "Deckung"      : 1e+6,
16  "Währung"     : "EURO"
17 }
```

Dieses Objekt besitzt unter anderem eine *Eigenschaft*, die aus dem Schlüssel "Kreditkarte" und dem Wert "Xema" besteht. Die Eigenschaft mit dem Schlüssel "Inhaber" hat ein Objekt als Wert (dargestellt in geschweiften Klammern). Die Eigenschaft mit dem Schlüssel "Vorlieben" hat eine Reihung mit drei Komponenten als Wert (dargestellt in eckigen Klammern).

Am weitesten verbreitet ist heute (2009) immer noch die Auszeichnungssprache HTML (Hypertext Markup Language).

Beispiel-08: Struktur einer HTML-Seite

```

1 <html>
2   <head>
3     <title>Titel der Webseite</title>
4     <!-- Evtl. weitere Kopfinformationen -->
5   </head>
6   <body>
7     Inhalt der Webseite
8   </body>
9 </html>
```

Eine besonders wichtige, positive Eigenschaft von HTML: HTML-Seiten kann man mit sog. Links (oder: Hyperlinks, engl. hyper references, href) *miteinander verknüpfen*.

Bei der Entwicklung und Verbreitung von HTML sind leider ein paar *schwerwiegende Pannen* passiert:

1. Um das Schreiben von HTML-Dateien zu erleichtern, haben die Entwickler von HTML Regeln eingeführt, die das *Weglassen bestimmter schliessender Tags* (im obigen Beispiel z.B. `</body>`) erlaubt. Diese Regeln bewirken, dass man mit üblichen Compilerbau-Werkzeugen (`yacc` und `lex`) keinen Parser für HTML generieren kann ("weil HTML keine LR-Sprache ist").

2. Wichtige Browser prüfen HTML-Dateien nicht auf Korrektheit, sondern versuchen, möglichst viele formale Fehler zu übersehen oder "irgendwie zu interpretieren" (diese Erscheinung ist eine Folge des sogenannten *Browserkriegs*, engl. browser wars), zwischen Netscape und Microsoft). Deshalb sind sehr viele HTML-Seiten formal nicht korrekt und werden von verschiedenen Browsern verschieden dargestellt.

Hinweis: Mit dem Programm `tidy` (siehe `tidy.sourceforge.net`) kann man "schlampig geschriebene HTML-Dateien" prüfen und verbessern lassen.

Außerdem deckt HTML nur ein relativ spezielles Anwendungsgebiet ab: Die Beschreibung der Darstellung (des Layouts) von Netzseiten. HTML war nicht als ein "Text-Format für beliebige Daten" gedacht und ist dafür auch nicht geeignet.

Die Auszeichnungssprache XML (eXtensible Markup Language) wurde als ein "Text-Format für beliebige Daten" entwickelt. HTML und XML stammen beide von der älteren (und sehr komplizierten) Auszeichnungssprachen SGML (Standard Generalized Markup Language) ab.

Wichtige Unterschiede zwischen HTML und XML:

- Mit HTML beschreibt man (vor allem oder nur) die *Darstellung* von Netzseiten.
- In HTML gibt es eine *feste Anzahl von Tags* wie `<html>`, `<body>`, `<h1>` etc. (etwa 50 Stück)
- Mit XML beschreibt man erstmal die *Struktur* von Daten ("wo steht was")
- Zusätzlich kann man die strukturierten Daten auch mit Hinweisen für die Darstellung versehen.
- In XML kann der Benutzer im Prinzip *beliebig viele Tags* erfinden und ihre Bedeutung festlegen.

XHTML ist eine Variante von HTML, bei der man *keine Tags weglassen darf* und bei der die Browser Dokumente auf Korrektheit *prüfen müssen* (sonst sind sie keine *XHTML-fähigen* Browser). Jedes XHTML-Dokument ist auch ein HTML-Dokument und ein XML-Dokument.

Meine persönliche Einschätzung von XML:

1. Die Entwickler von SGML und später die von XML haben sich große Verdienste erworben, weil sie den Nutzen und die Notwendigkeit von Auszeichnungssprachen erkannt und entsprechende Sprachen entwickelt haben. Allerdings wurde SGML für die Auszeichnung wenig strukturierter Texte (Dokumentationen von Behörden, dem Militär, von Firmen etc.) entwickelt, die aus viel freiem Text bestehen, in dem man ab und zu ein Wort als ein Stichwort (für ein Stichwortverzeichnis), eine Zeile als Überschrift oder einen Abschnitt als Inhaltsverzeichnis auszeichnen möchte.

2. SGML war viel, viel zu kompliziert (was bei einem ersten Versuch, eine ganz neue *Art* von Sprache zu entwickeln, nicht verwundern sollte).

3. XML ist eine Untermenge von SGML und wird für zwei unterschiedliche Aufgaben eingesetzt:

3.1. Zum Auszeichnen relativ schwach strukturierter Texte.

3.2. Zum Serialisieren von (stark strukturierten) Programm-Ausgabedaten.

Die unterschiedlichen Anforderungen, die daraus entstehen, haben dazu beigetragen, dass XML für beide Aufgabengebiete nicht "maßgeschneidert" und zu kompliziert ist.

JSON und YAML sind sehr gut zum Serialisieren von Programm-Ausgabedaten geeignet, aber nicht zum Auszeichnen schwach strukturierter Texte.

4. Negative Eigenschaften von XML:

4.1. XML ist sehr kompliziert. Eine vollständige Beschreibung füllt ein dickes Buch. XML vollständig zu lernen dauert Monate oder Jahre, nicht Stunden oder Tage.

4.2. XML beruht nicht auf den Grundkonzepten verbreiteter Programmiersprachen (*Typen* und *Variablen*), sondern auf dem ganz anderen Konzept eines *Tags*. Tags haben mit Typen und Variablen zu tun, aber auf eine ziemlich komplizierte und schwer zu durchschauende Weise.

4.2. In XML fehlen die Grundtypen, die es in vielen Programmiersprachen gibt (int, float, char, String). In XML kann man nicht ausdrücken, dass eine bestimmte Information durch eine Ganzzahl (oder eine Bruchzahl) dargestellt werden soll.

4.3. XML ist von Menschen nur in einfachen Fällen lesbar und in vielen praktischen Fällen nicht lesbar.

4.4. Es gibt zu viele Schema-Sprachen, d.h. Sprachen, mit denen Typen von XML-Dokumenten beschrieben werden:

DTD (Document Type Definition, gehört zu XML, ist aber ziemlich schwach)

DSD (Document Structure Description, BRICS in Aarhus und AT&T)

XML-Schemen (Empfehlung des W3C [World Wide Web Consortium])

Relax (Regular Language Description for XML, ISO/IEC Technischer Bericht 22250-1)

Relax-NG (Relax New Generation, ISO/IEC 19757-2),

Schematron (ISO/IEC-Standard 19757-3:2006, eine neue Version ist unterwegs)

TREX (Tree Regular Expressions for XML)

Einige dieser Schema-Sprachen sind selbst XML-Sprachen (DSD, XML-Schema, Relax, Relax-NG, Schematron), und ihre Schemen sind entsprechen schwer lesbar und voller störender Redundanz.

Andere Schema-Sprachen bieten eine kompaktere, lesbarere Schreibweise (DTD, Relax-NG).

4. Eine gleichzeitig positive und negative Eigenschaft von XML:

Viele Menschen (darunter auch viele Manager) haben beschlossen, XML zu verwenden.

5. Möglicherweise konnten viele Manager nur deshalb vom Nutzen von XML überzeugt werden, weil XML sehr kompliziert ist. Diese Manager hätte man möglicherweise nicht vom Nutzen einer ganz einfachen (aber ähnlich nützlichen) Sprache überzeugen können.

6. Ich hoffe, dass für das Serialisieren von Programm-Ausgabedaten XML in den nächsten Jahren zunehmend durch einfachere, leichter beschreibbare, leichter lernbare, leichter handhabbare aber ausdrucksstärkere Auszeichnungssprachen ersetzt wird.

Wiederholungsfragen, 16. SU, Mo 18.01.10

1. Die Struktur eines `YAML`-Objekts kann man auf zwei unterschiedliche Weisen darstellen. Beschreiben Sie diese beiden Weisen ganz kurz.
2. Zur Auszeichnungssprache `HTML` gehören auch ein paar fragwürdige (oder: falsche) Regeln. Was erlauben diese Regeln dem Schreiber eines `HTML`-Dokuments?
3. Angenommen, Sie wollen eine `HTML`-Seite entwickeln, die von möglichst *allen Benutzern des Internets* angesehen und benutzt werden kann. Was müssen Sie beim Entwickeln einer solchen Seite beachten und machen?
4. Nennen Sie mind. einen wichtigen Unterschied zwischen `HTML` und `XML`.
5. Was ist die wichtigste *positive* Eigenschaft von `XML`?
6. Nennen Sie mind. zwei wichtige *negative* Eigenschaften von `XML`.

Rückseite der Wiederholungsfragen und Arbeitsblatt zur 16. VL, Mo 18.01.10

Angenommen, in einer Klasse K werden die folgenden Methoden vereinbart:

```

1 // Klasse K:
2
3     void    ometA(int n)      { ... }
4     String  ometB(long n, float f) { ... }
5     static void kmetA(int n)  { ... }
6     static String kmetB(long n, float f) { ... }

```

Die Ergebnistypen und die Parametertypen dieser Methoden ist hier *unwichtig*. Wichtig ist aber der Unterschied zwischen *Objektmethoden* (ometA und B) und *Klassenmethoden* (kmetA und B).

Sei weiter angenommen, dass die Klasse K in einem Programm $P01$ wie folgt *benutzt* wird:

```

7 // Programm P01:
8
9 K k01 = new K( ... );
10 K k02 = new K( ... );
11 ...
12 k01.ometA(17);
13 k02.ometA(25);
14 K.kmetA(38);
15 ...
16 String s01 = k01.ometB(120, 3.5);
17 String s02 = k02.ometB(230, 1.8);
18 String s03 = K.kmetB(250, 3.7);

```

Wenn der Programmierer die Klasse K und das Programm $P01$ dem Ausführer übergibt, dann prüft der beides und verändert dann die Klasse K und das Programm $P01$ wie folgt:

```

1 // Klasse K:
2
3     void    ometA(K k, int n)      { ... }
4     String  ometB(K k, long n, float f) { ... }
5     static void kmetA(K k, int n)  { ... }
6     static String kmetB(K k, long n, float f) { ... }

```

In der Klasse K hat jede Methode einen zusätzlichen Parameter k vom Typ K bekommen.

```

7 // Programm P01:
8
9 K k01 = new K( ... );
10 K k02 = new K( ... );
11 ...
12 ometA(k01, 17);
13 ometA(k02, 25);
14 kmetA(null, 38);
15 ...
16 String s01 = ometB(k01, 120, 3.5);
17 String s02 = ometB(k02, 230, 1.8);
18 String s03 = kmetB(null, 250, 3.7);

```

In den Aufrufen von Klassenmethoden (kmetA und kmetB) wird der zusätzliche Parameter nicht verwendet und für ihn wird immer `null` übergeben.

Statt mal die Objektmethode `k01.ometA` und mal `k02.ometA` aufzurufen wird im veränderten Programm $P01$ immer nur die *eine* Methode `ometA` aufgerufen, aber mal mit dem zusätzlichen Parameter `k01` und mal mit `k02`. Für `ometB` gilt natürlich Entsprechendes.

Statt in jedes K -Objekt eine Methode `ometA` mit 2 Parametern einzubauen, erzeugt der Ausführer in der Klasse K nur *eine* Methode `ometA` mit 3 Parametern, und der erste Parameter (vom Typ K) gibt an, welches Objekt durch die Methode jeweils bearbeitet werden soll.

Antworten zu den Wiederholungsfragen, 16. SU, Mo 18.01.10

1. Die Struktur eines `YAML`-Objekts kann man auf zwei unterschiedliche Weisen darstellen. Beschreiben Sie diese beiden Weisen ganz kurz.

Durch Einrückung oder durch Klammern.

2. Zur Auszeichnungssprache HTML gehören auch ein paar fragwürdige (oder: falsche) Regeln. Was erlauben diese Regeln dem Schreiber eines HTML-Dokuments?

Diese Regeln erlauben es, bestimmte schliessende Tags wegzulassen (z.B. die zum öffnenden Tag `<p>`).

3. Angenommen, Sie wollen eine HTML-Seite entwickeln, die von möglichst *allen Benutzern des Internets* angesehen und benutzt werden kann. Was müssen Sie beim Entwickeln einer solchen Seite beachten und machen?

Man muss beachten, dass unterschiedliche Browser bestimmte HTML-Konstrukte unterschiedlich darstellen. Deshalb muss man beim Entwickeln die Seite in mehreren Browsern ansehen, prüfen und evtl. ändern.

4. Nennen Sie mind. einen wichtigen Unterschied zwischen HTML und XML.

HTML: feste Anzahl von Tags (ca. 50), XML: Menge der Tags ist unbegrenzt

XML ist eine Metasprache, HTML ist eine (Auszeichnungs-) Sprache

HTML beschreibt das Layout eines Dokuments, XML beschreibt vor allem die Struktur.

5. Was ist die wichtigste *positive* Eigenschaft von XML?

Dass viele Menschen und Organisationen sich darauf geeinigt haben, XML zu benutzen.

6. Nennen Sie mind. zwei wichtige *negative* Eigenschaften von XML.

Die vollständige Beschreibung von XML ist sehr lang und kompliziert.

Alle Feinheiten von XML zu lernen ist sehr aufwendig.

XML-Dokumente sind von Menschen kaum lesbar.

XML beruht auf anderen Grundkonzepten als heute verbreitete Programmiersprachen.

XML soll zwei ganz unterschiedliche Probleme lösen (die Beschreibung schwach strukturierter Texte und die Beschreibung stark strukturierter Programm-Ausgaben). Diese beiden Ziele "widersprechen und behindern sich gegenseitig".

16. SU Mo 18.01.10

A. Wiederholung
B. Organisation

Die Wahrheit: Klassen sind auch Objekte (der Klasse Class)

Bisher haben wir sorgfältig zwischen einer Klasse (Einem Modul und Bauplan für Module) und einem Objekt unterschieden. Tatsächlich werden Klassen üblicherweise auch als Objekte einer speziellen Klasse namens `Class` realisiert.

Sei K eine Klasse, von der wir ein paar Objekte k_1, k_2, \dots erzeugt haben. Bisher haben wir die Klasse K und ihre Objekte etwa so beschrieben:

Modul K	Objekt k_1	Objekt k_2
Klassen-Attribut-1	Objekt-Attribut-1	Objekt-Attribut-1
Klassen-Attribut-2	Objekt-Attribut-2	Objekt-Attribut-2
...
Klassen-Methode-1	Objekt-Methode-1	Objekt-Methode-1
Klassen-Methode-2	Objekt-Methode-2	Objekt-Methode-2
...
Konstruktor-1		
Konstruktor-2		
...		

Implementiert wird das Ganze aber so, dass Objekte nur ihre *Attribute* enthalten, die *Objekt-Methoden* aber auch nur *einmal* im Modul K gespeichert werden (statt in jedem Objekt), etwa so:

Modul K	Objekt k_1	Objekt k_2
Klassen-Attribut-1	Objekt-Attribut-1	Objekt-Attribut-1
Klassen-Attribut-2	Objekt-Attribut-2	Objekt-Attribut-2
...
Klassen-Methode-1		
Klassen-Methode-2		
...		
Konstruktor-1		
Konstruktor-2		
...		
Objekt-Methode-1		
Objekt-Methode-2		
...		

Wenn es viele Objekte derselben Klasse K gibt, spart man dadurch viel Speicherplatz.

Aber: Was ist der Unterschied zwischen einer Objekt-Methode im Objekt k_1 und der gleichen Methode im Objekt k_2 ? (Die Methoden greifen auf die Attribute von k_1 bzw. k_2 zu).

Wie kann man es erreichen, dass eine Objekt-Methode mit Modul K "weiß", auf die Attribute von welchem Objekt sie zugreifen soll? (Man übergibt der Methode das betreffende Objekt, z.B. k_1 oder k_2 oder ...) als zusätzlichen Parameter. Eine Methodenaufruf des Programmierers wie

```
ob.machDies(p1, p2, p3) // ob ist ein Objekt, p1 bis p3 sind Parameter
```

wird vom Compiler umgewandelt in einen Aufruf der Form:

```
machDies(ob, p1, p2, p3)
```

Und der Einheitlichkeit wegen werden Aufrufe von *Klassenmethoden* wie etwa

```
Klas.tuJenes(p1, p2, p3) // Klas ist eine Klasse, p1 bis p3 sind Parameter
```

umgewandelt in

```
tuJenes(null, p1, p2, p3)
```

21 Reflexion (S. 515)

Ein Java-Programm kann, während es ausgeführt wird, sozusagen "in einen Spiegel schauen und sich selbst betrachten und untersuchen". Mit den Befehlen der *Java-Reflexionsschnittstelle* kann ein Programm z.B. feststellen, wie viele Methoden eine bestimmte Klasse enthält, wie die Methoden heißen und welche Parameter die Methoden haben. Außerdem kann es diese Methoden aufrufen, obwohl der Programmierer beim Schreiben des Programms den Namen der Methoden (und ihre Parameter etc.) noch gar nicht kannte.

Die Java-Reflexionsschnittstelle ist besonders nützlich um Programme zu schreiben, die Java-Programme bearbeiten sollen. Beispiele für solche Programme:

Ein Klassen-Browser. *Eingabe:* Der (volle) Name einer Klasse. *Ausgabe:* Eine genaue Beschreibung aller Elemente und Konstruktoren der Klasse.

Ein Dokumentations-Programm: *Eingabe:* Der Name einer Klasse. *Ausgabe:* Eine gut strukturierte Dokumentation aller Elemente und Konstruktoren der Klasse. Voraussetzung: Der Programmierer hat die einzelnen Elemente und Konstruktoren vernünftig kommentiert. Ein solches (kostenloses) Programm namens `javadoc` gehört zu Java-Distribution der Firma Sun.

Eine Funktion, die Objekte beliebiger Typen erzeugen kann: *Parameter:* Der Name einer Klasse (oder eines Reihungstyps). *Ergebnis:* Ein Objekt dieser Klasse (bzw. dieses Reihungstyps), bei dem alle Attribute zufällig gewählte Werte haben. Kann für Maßentests gut sein, wo man z.B. viele Objekte eines bestimmten Typs benötigt, es aber nicht wichtig ist, wie die einzelnen Objekte aussehen.

Zur Java-Reflexionsschnittstelle gehören etwa 20 Klassen und Schnittstellen. Damit kann man u.a. Programme schreiben, die Probleme der folgenden Art lösen:

Problem 1: Den Namen einer beliebigen Klasse einlesen und die Namen aller Elemente dieser Klasse ausgeben.

Problem 2: Den Namen einer beliebigen Klasse K und den Namen einer (in K vereinbarten) Methode einlesen und die Methode aufrufen.

Problem 3: Ein Objekt einer beliebigen Klasse einlesen (aus einer Datei oder vom Internet) und die Werte all seiner Attribute ausgeben.

Wenn man ein objektorientiertes Programm zur Verwaltung der BHT entwickeln will, muss man wahrscheinlich Klassen namens `StudentIn`, `ProfessorIn`, `Raum`, `Vorlesung` etc. programmieren.

Wenn man ein Programm zum Analysieren und bearbeiten von Java-Programmen entwickeln will, muss man wahrscheinlich Klassen namens `Klasse`, `Methode`, `Attribut`, `Konstruktor` etc. programmieren. Genau solche Klassen gehören zur Java-Reflexionsschnittstelle.

Bisher haben wir *Klassen* und *Objekte* sorgfältig unterschieden: Eine Klasse ist (ein Modul und) ein Bauplan für Objekte, und damit etwas *ganz anderes* als ein Objekt.

Die Wahrheit ist subtiler. Jetzt (nach fast 2 Semestern) sind Sie (hoffentlich) reif dafür:

Klassen werden (während der Ausführung eines Java-Programms) als *Objekte* der Klasse `Class` dargestellt. Ein solches `Class`-Objekt enthält alle Informationen über die betreffende Klasse. Man sagt auch: Das `Class`-Objekt *reflektiert* die betreffende Klasse.

`Class`-Objekte sind sozusagen "Der Eingang in die Welt der Reflexion".

3 Möglichkeiten, sich ein bestimmtes `Class`-Objekt zu besorgen

S. 517, Beispiel-01

Die ersten beiden Möglichkeiten (`String.class` und `strob.getClass()`) "gehen immer gut", d.h. sie werfen *nie eine Ausnahme*. Die dritte Möglichkeit (`Class.forName("...")`) "kann schief gehen" und wirft dann eine Ausnahme des Typs `ClassNotFoundException`.

Zur Entspannung:

Von der JVM (Java Virtual Machine) zur MVM (Multi Tasking Virtual Machine)

Eine JVM kann zu jedem Zeitpunkt nur *ein* Bytecode-Programm ausführen. Um *mehrere* Programme ausführen zu lassen, muss man entsprechend *viele* JVMs starten.

Nachteil 1: Um miteinander zu kommunizieren, müssen Bytecode-Programme Befehle des jeweiligen Betriebssystems verwenden (statt nur Java/Bytecode-Befehle).

Nachteil 2: Jede JVM muss bei ihrem Start zahlreiche Standardklassen laden und verifizieren (kostet Zeit) und speichern (kostet Platz). Das macht das Starten eines Java-Programms *langsam*.

Konkretes Experiment: Ein `Hallo`-Programm von einer Kommandozeile aus starten, etwa so:

```
> java -verbose Hallo01
```

An der Ausgabe erkennt man, dass etwas mehr als 300 Klassen geladen werden, bevor das Mini-Programm `Hallo01` ausgeführt wird.

Im Projekt Barcelona von Sun wurde eine MVM entwickelt, die *mehrere* Bytecode-Programme *gleichzeitig* ausführen kann. Die einzelnen Programme sind etwa so gegeneinander geschützt, wie die Prozesse eines Betriebssystems, können aber mit Java/Bytecode-Befehlen miteinander kommunizieren und Klassen gemeinsam benutzen.

Ergebnis dieses Forschungsprojekts: Verabschiedung des *JSR121-Jul-20-2005* (Java Specification Request 121) und das Java-Paket `javax.isolate` mit den Klassen `Isolate`, `Link` etc. darin. Ein `Isolate`-Objekt hat Ähnlichkeit mit einem Prozess eines Betriebssystems.

Wiederholungsfragen, 17. SU, 25.01.10**1. Betrachten Sie die folgenden Befehle:**

```
1 String str = EM.liesString();
2 ... str.substring(3, 5) ...
3 ... String.format("%s", str) ...
```

Wie werden die Methodenaufrufe in Zeile 3 und 4 vom Java-Compiler umgewandelt (damit Objektmethoden nicht wirklich in jedes Objekt eingebaut werden müssen)?

2. Die folgende Methode soll (mit `println`) den *Typ ihres Parameters* ausgeben. Wie kann man das (natürlich mit Hilfe von Reflexion) programmieren?

```
4 static public void printType(Object ob) {
5
6
7
8
9 }
```

3. Weisen Sie der Variablen `kob17` das Class-Objekt zu, welches den Typ `char` repräsentiert:

```
10 Class<?> kob17 = _____ ;
```

4. Ergänzen Sie die folgende Zuweisung so, dass der Variablen `kob18` das Class-Objekt zugewiesen wird, das den Typ *Reihung von int* (d.h. `int[]`) repräsentiert:

```
11 Class<?> kob18 = Class.forName( _____ );
```

5. Ergänzen Sie die folgende Zuweisung so, dass der Variablen `kob19` das Class-Objekt zugewiesen wird, das den Typ *Reihung von Reihung von String* (d.h. `String[][]`) repräsentiert:

```
12 Class<?> kob19 = Class.forName( _____ );
```

6. Die Klasse `Class<K>` ist generisch. Warum?

Rückseite der Wiederholungsfragen, 17. SU, 25.01.10

Die folgenden Fragen beziehen sich auf die Methoden `main`, `pruefeUndRufeAuf` und `rufeAuf` auf den Seiten 521-522 des Buches. Diskutieren Sie die Fragen in kleinen Gruppen (zu zweit oder zu dritt) und schreiben Sie die Antworten lesbar auf. Sie dürfen auch die Erläuterungen zu den Methoden lesen.

1. Fragen zur Methode `main` (Zeile 1 bis 10)

- 1.1. Warum muss in Zeile 2 `throws ClassNotFoundException` stehen? Welcher Befehl in dieser Methode könnte eine solche Ausnahme werfen?
- 1.2. Von welchem Typ ist die Variable `rm`?
- 1.3. Was liefert der Funktionsaufruf `kob.getDeclaredMethods()` als Ergebnis?
- 1.4. Was macht die `for-each`-Schleife in Zeile 9?

2. Fragen zur Methode `pruefeUndRufeAuf`

- 2.1. Was erwartet diese Methode als Parameter? Einen `int`-Wert? Nein! Ein `String`-Objekt? Nein! Sondern?
- 2.2. Von welchem Typ ist die Variable `rt`?
- 2.3. Was liefert der Funktionsaufruf `m.getParameterTypes()` (in Zeile 12) als Ergebnis?
- 2.4. Was liefert der Funktionsaufruf `m.getName()` wohl als Ergebnis?
- 2.5. Was bezeichnet der Name `Double.TYPE`?
- 2.6. Übersetzen Sie die Bedingung der `if`-Anweisung in Zeile 15 in verständliches Deutsch. Welche Eigenschaften der Methode `m` prüft diese Bedingung?

3. Fragen zur Methode `rufeAuf`

- 3.1. Was erwartet diese Methode als Parameter?
- 3.2. Womit wird die Variable `metName` initialisiert? Mit dem Namen des Programmierers? Nein! Sondern?
- 3.3. Was bewirkt der Methodenaufruf `m.invoke(null, dopa)` (in Zeile 24)?
- 3.4. Was liefert der Methodenaufruf `m.invoke(null, dopa)`, wenn das Method-Objekt `m` eine *Funktion* reflektiert?
- 3.5. Was liefert der Methodenaufruf `m.invoke(null, dopa)`, wenn das Method-Objekt `m` eine *Prozedur* reflektiert?
- 3.6. Wann wirft der Methodenaufruf `m.invoke(null, dopa)` eine Ausnahme und von welchem Typ ist diese Ausnahme?
- 3.7. Was liefert der Funktionsaufruf `m.getDeclaringClass()` als Ergebnis?
- 3.8. Was liefert der Funktionsaufruf `m.getDeclaringClass().newInstance()` als Ergebnis?
- 3.9. Was bewirkt der Methodenaufruf `m.invoke(ob, dopa)` (in Zeile 33)?
- 3.10. Was liefert der Methodenaufruf `m.invoke(ob, dopa)` wenn das Method-Objekt `m` eine *Funktion* reflektiert?
- 3.11. Was liefert der Methodenaufruf `m.invoke(ob, dopa)` wenn das Method-Objekt `m` eine *Prozedur* reflektiert?

Antworten zur Rückseite der Wiederholungsfragen, 17. SU, 25.01.10**1. Fragen zur Methode main (Zeile 1 bis 10)**

- 1.1. Warum muss in Zeile 2 `throws ClassNotFoundException` stehen? Welcher Befehl in dieser Methode könnte eine solche Ausnahme werfen? (Der Befehl `Class.forName(...)` in Zeile 6)
- 1.2. Von welchem Typ ist die Variable `rm`? (Vom Typ Reihung von Method)
- 1.3. Was liefert der Funktionsaufruf `kob.getDeclaredMethods()` als Ergebnis? (Eine Reihung von Method-Objekten. Jedes Objekt beschreibt eine in der Klasse `kob` vereinbarte Methode).
- 1.4. Was macht die `for-each`-Schleife in Zeile 9? (Sie ruft mehrmals die Methode `pruefeUndRufeAuf` auf und übergibt ihr eines der Method-Objekte `m` aus der Reihung `rm`).

2. Fragen zur Methode `pruefeUndRufeAuf`

- 2.1. Was erwartet diese Methode als Parameter? Einen `int`-Wert? Nein! Ein `String`-Objekt? Nein! Sondern? (Ein Method-Objekt)
- 2.2. Von welchem Typ ist die Variable `rt`? (Vom Typ Reihung von Class)
- 2.3. Was liefert der Funktionsaufruf `m.getParameterTypes()` (in Zeile 12) als Ergebnis? (Eine Reihung von Class-Objekten. Jedes Objekt beschreibt den Typ eines Parameters der Methode `m`)
- 2.4. Was liefert der Funktionsaufruf `m.getName()` wohl als Ergebnis? (Den Namen der Methode `m`)
- 2.5. Was bezeichnet der Name `Double.TYPE`? (Das Class-Objekt, welches den primitiven Typ `double` reflektiert)
- 2.6. Übersetzen Sie die Bedingung der `if`-Anweisung in Zeile 15 in verständliches Deutsch. Welche Eigenschaften der Methode `m` prüft diese Bedingung? (Wenn `m` genau 1 Parameter hat und dieser Parameter vom Typ `double` ist, dann ...)

3. Fragen zur Methode `rufeAuf`

- 3.1. Was erwartet diese Methode als Parameter? (Ein Method-Objekt)
- 3.2. Womit wird die Variable `metName` initialisiert? Mit dem Namen des Programmierers? Nein! Sondern? (Mit dem Namen der Methode `m`)
- 3.3. Was bewirkt der Methodenaufruf `m.invoke(null, dopa)` (in Zeile 24)? (Die Methode `m` wird als Klassenmethode mit dem Parameter `dopa` aufgerufen)
- 3.4. Was liefert der Methodenaufruf `m.invoke(null, dopa)`, wenn das Method-Objekt `m` eine *Funktion* reflektiert? (Das Ergebnis der Funktion `m`)
- 3.5. Was liefert der Methodenaufruf `m.invoke(null, dopa)`, wenn das Method-Objekt `m` eine *Prozedur* reflektiert? (Den Wert `null`)
- 3.6. Wann wirft der Methodenaufruf `m.invoke(null, dopa)` eine Ausnahme und von welchem Typ ist diese Ausnahme? (Wenn `m` keine Klassenmethode [sondern eine Objektmethode] ist, wird eine Ausnahme des Typs `NullPointerException` geworfen)
- 3.7. Was liefert der Funktionsaufruf `m.getDeclaringClass()` als Ergebnis? (Die Heimatklasse von `m`. D.h. das Class-Objekt, welches die Klasse reflektiert, in der die Methode `m` vereinbart wurde)
- 3.8. Was liefert der Funktionsaufruf `m.getDeclaringClass().newInstance()` als Ergebnis? (Ein Objekt der Heimatklasse von `m`)
- 3.9. Was bewirkt der Methodenaufruf `m.invoke(ob, dopa)` (in Zeile 33)? (Die Methode `m` wird als Objektmethode mit dem Parameter `dopa` aufgerufen und greift auf die Attribute des Objekts `ob` zu)
- 3.10. Was liefert der Methodenaufruf `m.invoke(ob, dopa)` wenn das Method-Objekt `m` eine *Funktion* reflektiert? (Das Ergebnis der Funktion `m`)
- 3.11. Was liefert der Methodenaufruf `m.invoke(ob, dopa)` wenn das Method-Objekt `m` eine *Prozedur* reflektiert? (Den Wert `null`)

Antworten zu den Wiederholungsfragen, 17. SU, 25.01.10**1. Betrachten Sie die folgenden Befehle:**

```
1 String str = EM.liesString();
2 ... str.substring(3, 5) ...
3 ... String.format("%s", str) ...
```

Wie werden die Methodenaufrufe in Zeile 3 und 4 vom Java-Compiler umgewandelt (damit Objektmethoden nicht wirklich in jedes Objekt eingebaut werden müssen)?

```
2 ... substring(str, 3, 5) ...
3 ... format(null, "%s", str) ...
```

2. Die folgende Methode soll (mit `println`) den *Typ ihres Parameters* ausgeben. Wie kann man das (natürlich mit Hilfe von Reflexion) programmieren?

```
4 static public void printType(Object ob) {
5
6     Class<?> kob = ob.getClass();
7     String nam = kob.getName();
8     println(nam);
9
10    // oder println(ob.getClass().getName());
11 }
```

3. Weisen Sie der Variablen `kob17` das Class-Objekt zu, welches den Typ `char` repräsentiert:

```
12 Class<?> kob17 = Character.TYPE;
```

4. Ergänzen Sie die folgende Zuweisung so, dass der Variablen `kob18` das Class-Objekt zugewiesen wird, das den Typ *Reihung von int* (d.h. `int[]`) repräsentiert:

```
13 Class<?> kob18 = Class.forName("[I" );
```

5. Ergänzen Sie die folgende Zuweisung so, dass der Variablen `kob19` das Class-Objekt zugewiesen wird, das den Typ *Reihung von Reihung von String* (d.h. `String[][]`) repräsentiert:

```
14 Class<?> kob19 = Class.forName("[[Ljava.lang.String;");
```

6. Die Klasse `Class<T>` ist generisch. Warum?

Damit sie Funktionen mit dem Ergebnistyp `T` enthalten kann (z.B. die Methode `newInstance`). Diese Methode hatte vor Einführung generischer Klassen den "ungenaueren" Ergebnistyp `Object`.

17. SU Mo 25.01.10

A. Wiederholung

B. Organisation:

Unterlagen, die man zur Klausur mitbringen darf: 5 Blätter, maximal DIN A 4, beliebig beschriftet.

Außerdem reichlich leere Seiten, um die Lösungen darauf zu schreiben.

Es gibt 6 Aufgaben. Verwenden Sie für *jede Lösung* ein *neues Blatt* und beschreiben Sie nur die *Vorderseiten* Ihrer Blätter (Rückseiten frei lassen).

Normalerweise passt jede Lösung auf die Vorderseite von *einem* Blatt.

Auf jedem abgegebenen Blatt sollte Ihr *Nachnahme* stehen (am besten oben rechts).

Weitere Kennzeichnungen (Ihre Matrikel-Nr, Schuhgröße etc.) sind erlaubt, aber *nicht nötig*.

Sie dürfen mit Bleistift schreiben. Schreiben Sie lesbar!

Klausur-Termin und -Ort: Mo, 08.02.10, 12 Uhr, Beuth-Saal.

Reflexion, Fortsetzung

Die Klasse `java.lang.Math` enthält einige wichtige mathematische Funktionen. Wie heißen die und was für Parameter haben sie? (Ok, die meisten von Ihnen scheinen keine oder nur sehr wenige Methoden der Klasse `Math` zu kennen. Trotzdem und ohne die Namen dieser Funktionen erstmal zu lernen könnten wir ein Programm schreiben, welches alle oder einige dieser Methoden aufruft. Natürlich mit Reflexion. Statt es zu *schreiben*, werden wir jetzt ein solches Programm *lesen*).

Wir sehen uns jetzt ein Programm an, welches alle in der Klasse `Math` vereinbarten Methoden, die genau einen `double`-Parameter erwarten, auf den Parameter-Wert `0.0` anwenden und das Ergebnis der Methode ausgeben.

In kleinen Gruppen die Fragen auf der Rückseite der Wiederholungsfragen diskutieren und beantworten. Als Unterlage müsste das Kapitel 21.2 (ab S. 520) ausreichen.

Zur Entspannung: Christian Morgenstern (1871-1914)

Das Butterbrotpapier

Ein erstaunlich aktuelles Gedicht über die Evolution.

Besonders wichtige Methoden der Klasse Class

```
getDeclaredXXX      getXXX
getDeclaredXXXs    getXXXs
```

(wobei XXX gleich `Field` oder `Method` oder `Constructor` ist):

Alle Elemente einer Klasse in vier Gruppen eingeteilt:

	public	nicht-public
in der Klasse vereinbart	VP	VN
geerbt	GP	GN

```
getDeclaredXXX (String name): Ein XXX aus VP oder VN
getDeclaredXXXs()           : Alle XXX aus VP und VN
getXXX (String name): Ein XXX aus GP oder VP
getXXXs ()                 : Alle XXX aus GP und VP
```

An die GN-Elemente kommt man nur *indirekt* dran (mit der Methode `getSuperclass`)

Wenn XXX gleich `Constructor` ist gibt es GP und GN nicht!

Mit anderen Worten: Alle Konstruktoren sind `declared`, kein Konstruktor ist *geerbt*.

Trotzdem gibt es auch für Konstruktoren 4 `get`-Methoden wie für Attribute (fields) und Methoden.

Mit `getDeclaredConstructor` kann man auf *jeden* Konstruktor zugreifen.

Mit `getConstructor` kann man nur auf *öffentliche* (public) Konstruktoren zugreifen.

Warum gibt es `getConstructor` überhaupt? Ist `getDeclaredConstructor` nicht überflüssig?

Nein, aus folgendem Grund:

Mit einem *SecurityManager* kann man reflektive Zugriffe grundsätzlich verbieten oder mehr oder weniger erlauben. Eine Möglichkeit besteht darin, den Zugriff auf *öffentliche* Elemente und Konstruktoren zu erlauben (z.B. Zugriffe mit `getConstructor`), aber Zugriffe auf *andere* Elemente und Konstruktoren zu verbieten (z.B. Zugriffe mit `getDeclaredConstructor`).

Auf jedem Computer, auf dem Java läuft, gibt es eine zentrale Datei namens `policy.java` (kurz: Die `policy`-Datei). Standardmäßig liegt die in folgendem Verzeichnis:

```
java.home\lib\security\policy.java
```

Dabei steht `java.home` für den Wert der System-Property mit dem Schlüssel "`java.home`". Deren Wert kann man z.B. mit dem Befehl

```
String jh = System.getProperty("java.home", "schade, klappt nicht");
```

lesen (natürlich nur, wenn man die Erlaubnis dazu hat). Ein typischer Wert dieser Property:

```
C:\Programme\Java\jdk1.6.0_14\jre
```

S. 524, Zeile 1, `getSuperclass`

S. 524-525

Wie viele Attribute mit einem bestimmten Namen kann es in einer Klasse geben? (1)

Wie viele Methoden mit einem bestimmten Namen kann es in einer Klasse geben? (mehrere)

Wie viele Konstruktoren mit einem bestimmten Namen kann es in einer Klasse geben? (mehrere)

Vergleichen Sie die Methoden in den Zeilen 3, 7 und 11. Warum haben diese Methoden unterschiedliche Parameter?

Wiederholungsfragen, 18. SU, Mo 01.02.10

1. In Objekten welcher Klasse gibt es Methoden namens `getField`, `getFields`, `getDeclaredField` und `getDeclaredFields`?
2. Wie viele Parameter hat die Methode `getField` und was liefert sie als Ergebnis?
3. Ebenso für `getFields`.
4. Ebenso für `getDeclaredField`.
5. Ebenso für `getDeclaredFields`.
6. Wie heißt die Datei, in der u.a. festgelegt wird, ob Java-Programme reflektiv auf private Attribute zugreifen dürfen oder nicht?
7. Angenommen `f01` ist ein `Field`-Objekt, das ein privates Attribut reflektiert. Welchen Methodenaufruf sollte man auf jeden Fall ausführen lassen, ehe man versucht, auf den Wert dieses Attributs zuzugreifen (z.B. mit der Methode `getBoolean` oder `getInt` etc.)?
8. Was reflektiert ein `Class`-Objekt? Ein *Programm*? Eine *Klasse*? Alle *Elemente* einer Klasse? Nur die *öffentlichen Elemente* und *Konstruktoren* einer Klasse? Einen *Typ*? Ein *Attribut*? Gar *nichts*?
9. Was wäre ein besserer Name für die Klasse `Class`?
10. Wann und wo findet die (Haupt-) Klausur für das Fach MB2-PR2 statt?

Antworten zu den Wiederholungsfragen, 18. SU, Mo 01.02.10

1. In Objekten welcher Klasse gibt es Methoden namens `getField`, `getFields`, `getDeclaredField` und `getDeclaredFields`?

In Objekten der Klasse `Class<T>`.

2. Wie viele Parameter hat die Methode `getField` und was liefert sie als Ergebnis?

Einen Parameter (vom Typ `String`). Sie liefert ein `Field`-Objekt, welches ein öffentliches Attribut der Klasse reflektiert.

3. Ebenso für `getFields`.

Null Parameter. Sie liefert eine Reihung von `Field`-Objekten, welche alle öffentlichen Attribute der Klasse reflektieren.

4. Ebenso für `getDeclaredField`.

Einen Parameter (vom Typ `String`). Sie liefert ein `Field`-Objekt, welches ein in der Klasse vereinbartes (d.h. nicht geerbtes) Attribut reflektiert.

5. Ebenso für `getDeclaredFields`.

Null Parameter. Sie liefert eine Reihung von `Field`-Objekten, welche alle in der Klasse vereinbarten (d.h. nicht geerbten) Attribute reflektieren.

6. Wie heißt die Datei, in der u.a. festgelegt wird, ob Java-Programme reflektiv auf private Attribute zugreifen dürfen oder nicht?

`java.policy`

7. Angenommen `f01` ist ein `Field`-Objekt, das ein privates Attribut reflektiert.

Welchen Methodenaufruf sollte man auf jeden Fall ausführen lassen, ehe man versucht, auf den Wert dieses Attributs zuzugreifen (z.B. mit der Methode `getBoolean` oder `getInt` etc.)?

`f01.setAccessible(true);`

8. Was reflektiert ein `Class`-Objekt? Ein *Programm*? Eine *Klasse*? Alle *Elemente* einer Klasse? Nur die *öffentlichen Elemente* und *Konstruktoren* einer Klasse? Einen *Typ*? Ein *Attribut*? Gar *nichts*?

Ein `Class`-Objekt reflektiert im allgemeinen einen `Typ`.

9. Was wäre ein besserer Name für die Klasse `Class`?

`Typ`

10. Wann und wo findet die (Haupt-) *Klausur* für das Fach MB2-PR2 statt?

Am Mo 08.02.10, ab 12 Uhr im Beuth-Saal (nicht in der Beuth-Halle!).

18. SU Mo 01.02.10**A. Wiederholung**

B. Organisation: Meine Übung am **Mo 08.02.10 ab 10 Uhr** im SWE-Labor findet statt, wenn ich bis **Fr 05.02.10, 12 Uhr** eine Email bekomme, in der mindestens 3 (mit Namen genannte) StudentInnen versichern, dass sie zu dieser Übung kommen möchten. Dieses Angebot ist ernst gemeint.

Haben Sie Fragen zum Stoff dieser Vorlesung?

Tips zur Klausur:

1. Nutzen Sie es aus, dass Sie als Unterlagen bis zu 5 Blättern (maximal Din A 4) mitbringen dürfen. Fertigen Sie diese Blätter *von Hand* an (das schließt die Benutzung einer Tastatur, eines Editors und eines Druckers nicht aus).
2. Bringen Sie gutes Schreibpapier (am besten kariertes) und gutes Schreibgerät mit (darunter auch einen Bleistift und einen Radiergummi). Kennzeichnen Sie schon mal ein paar Blätter mit Ihrem Nachnamen (am besten oben rechts).
Für Blätter mit sehr rauhem, ausgefranstem Rand gibt es 10 Minuspunkte!
3. Planen Sie ein, vor der Klausur die Cafeteria zu besuchen. Falls Sie dann zu spät kommen, können Sie evtl. die Cafeteria ausfallen lassen und doch noch pünktlich zur Klausur kommen.
4. Die Klausur besteht aus 6 Aufgaben. Für eine davon gibt es 25 Punkte, für die anderen je 15 Punkte. Überfliegen Sie die Aufgaben am Anfang und bearbeiten Sie dann zuerst die einfacheren, danach die schwereren. Lesen Sie jede Zeile und jedes Wort der Aufgabenstellungen!
5. Vermutlich haben Kenntnisse auf den folgenden Sachgebieten keine schädlichen Nebenwirkungen:

Rekursion

Woraus besteht der Rumpf *jeder* rekursiven Methode? (Aus einer Fallunterscheidung)
Was für Fälle müssen dadurch unterschieden werden? (Einfache und rekursive Fälle).

Reihungen und Sammlungen, wie man sie bearbeitet, wie man sie als Bojen darstellt, die Strukturen von Sammlungen, generische Sammlungsklassen.

XML (alles, was wir in der Vorlesung und in den Übungen gemacht haben)

Außerdem schadet es nichts, wenn man sich intensiv beschäftigt hat mit

- den **Aufgaben**
- den **Tests**
- den **Wiederholungsfragen**

S. 526, Zeile 15 bis 26: Weitere Objektmethoden der Klasse Class.

```
1 Object    ob01 = new    ... ;    // Irgendein Objekt
2 Class<?> kob  =        ... ;    // Irgendein Class-Objekt
3                                     // Sei T der von kob reflektierte Typ:
4 boolean   kob.isInstance(ob01); // true wenn ob01    zum Typ T gehört
5 ... kob.cast(ob01)    ...    // Castet ob01        zum Typ T
6 ... kob.newInstance() ...    // Erzeugt ein Objekt des Typs T
```

Parameterübergabe per Referenz, fehlt leider in Java:

In Java können Parameter von Methoden können nur *per Wert* übergeben werden (nicht *per Referenz*)

Mit der Parameterübergabe per Referenz kann man z.B. eine Methode wie die folgende vereinbaren:

```
1 static void rech(int w1, int w2, ref int r1, ref int r2) {
2     r1 = w1 / w2;
3     r2 = w1 % w2;
4 }
```

Ein Aufruf dieser Methode `rech` kann z.B. so aussehen:

```
5 int a1 = 6;
6
7 int e1;
8 int e2;
9
10 rech(2*a1+1, 5, e1, e2); // bewirkt: rech(13, 5, e1, e2)
```

Für die Wert-Parameter `w1`, `w2` darf man ("wie üblich") beliebige *Ausdrücke* angeben.

Für die Ref-Parameter `r1`, `r2`, die es in Java (noch?) nicht gibt, darf man nur *Variablen* angeben.

Nach dem Methodenaufruf in Zeile 10 hat die Variable `e1` den Wert (13 / 5 gleich) 2 und die Variable `e2` hat den Wert (13 % 5 gleich) 3.

Obwohl `rech` formal eine *Prozedur* ist (siehe das `void` in Zeile 1), hat sie doch Ähnlichkeit mit einer *Funktion*, die *mehrere* Ergebnisse berechnet. Diese Ergebnisse werden allerdings *nicht* (mit `return`) *geliefert*, sondern in den Ref-Parametern `r1` und `r2` (bzw. `e1` und `e2`) abgelegt.

In Java fehlt ein `goto`-Befehl.

In einem von Hand geschriebenen Programm (welches von einem Menschen gewartet werden muss) sollte man `goto`-Befehle vermeiden. Aber für ein Java-Programm ZP, das von einem Compiler aus einem Quellprogramm QP erzeugt wird, gilt diese Regel nicht. Denn in einem solche Fall wird nicht das Zielprogramm ZP gewartet, sondern nur das Quellprogramm QP. Nach jeder Änderung an QP wird mit dem Compiler eine neue Version von ZP erzeugt.

Grund zur Hoffnung: `goto` ist in Java ein reserviertes Wort (wie `class` und `if` und `while` etc.).

In Java kann man verschiedene Implementierungen einer Schnittstelle nicht unterscheiden.

Mit Hilfe von *Paketen* kann man in Java *Klassen* unterscheiden, die zufällig *gleiche Namen haben* (z.B. die Klassen `de.beuth_hochschule.fb6.Test` und `com.siemens.abt17a/Test`). Gut!

Für Schnittstellen gibt es keinen entsprechenden Mechanismus. Beispiel:

Angenommen, der FB VI der BHT und die Abteilung 17a der Firma Siemens haben beide eine Schnittstelle namens `Sendable` erfunden, die eine Funktion `String getAddress()` enthält. Inhaltlich haben die beiden Schnittstellen nichts miteinander zu tun, sie heißen nur zufällig gleich.

Jetzt wollen wir eine Klasse `K2` vereinbaren, deren Objekte beide `getAddress`-Methoden enthalten.

Hoffnungsvoll beginnen wir etwa so:

```
1 class K2 implements
2     com.siemens.abt17a.Sendable, de.beuth_hochschule.fb6.Sendable
3 {
4     public String getAddress() { ... }
5 }
```

Leider können wir in `K2` nur *eine* Funktion namens `getAddress` vereinbaren, und die gilt dann als Implementierung der Siemens-Schnittstelle und der BHT-Schnittstelle. Das ist nicht was wir wollen.

Anmerkung: In C# gibt es eine Lösung für dieses Problem, aber die ist unpraktisch kompliziert.

Viel Erfolg bei der Klausur!