Aufgaben für das Fach Programmieren 2

Inhaltsverzeichnis

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

Im Laufe des Semesters werden 9 Tests geschrieben. Bei jedem Test können Sie maximal 15 Punkte erreichen. Am Ende des Semesters werden die Punkte Ihrer 7 besten Tests addiert und aus der Summe (Maximalwert: 7 * 15 gleich 105) 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 7 der 9 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 7 der Tests teilgenommen haben.

Wenn Ihre Übungsnote besser als 5,0 ist, dürfen Sie an der Klausur am Ende des WS10/11 (und evtl. an der Nachklausur kurz vor dem SS11) teilnehmen. In der Klausur können Sie maximal 100 Punkte erreichen. Aus der Anzahl Ihrer Klausurpunkte wird eine *Klausurnote* berechnet (ebenfalls nach der obigen Tabelle).

Die Fragen in den 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* die Tests geschrieben werden und welche Aufgaben wann (d.h. vor welchem Test) gelöst werden sollen.

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 (in Zweifelsfällen zu Ihren Gunsten). 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

Aufgabe 1: Ein kleiner Quiz über den Stoff des ersten Semesters (PR1)

Kopieren Sie die folgenden Fragen in einen Editor oder ein Textverarbeitungssystem. Beantworten Sie dann jede Frage möglichst kurz und genau und schreiben Sie jede Antwort unmittelbar hinter die zugehörige Frage (denn wenn sie nur die Antworten angeben, z.B. "42", fällt es einigen Lesern vielleicht schwer, sich an die zugehörige Frage zu erinnern).

Es ist *nicht* verboten, während der Bearbeitung dieser Aufgabe das Buch "Java ist eine Sprache" aufzuschlagen.

1. Das Entwickeln und Ausführen von Programmen kann man als ein *Rollenspiel* auffassen. Nennen Sie die drei wichtigsten *Rollen* und geben Sie zu jeder Rolle an, welche *charakteristischen Tätigkeiten* ihr zugeordnet sind.

2. Nennen Sie fünf wichtige ("die fünf wichtigsten") *Grundkonzepte* von Programmiersprachen.

3. Was ist eine *Variable*? Erläutern Sie die wichtigsten Eigenschaften einer Variable. Was kann der Programmierer bzw. der Ausführer mit einer Variablen alles machen?

4. Was ist ein *Typ*?

5. Was ist ein *Modul*?

6. Was ist eine *Klasse*?

7. Was für Dinge darf man in einer *abstrakten* Klasse vereinbaren?

8. Was für Dinge darf man in einer *Schnittstelle* vereinbaren?

9. Es gibt unüberschaubar viele Programmiersprache mit zahllosen Befehlen darin. Im Grunde genommen gibt es aber nur *3 Arten* von Befehlen (die der Programmier dem Ausführer geben kann). Wie heißen diese 3 Arten von Befehlen auf Deutsch? Auf Englisch?

10. Vereinbaren Sie (in Java) eine Variable namens `otto` vom Typ `int` mit dem Anfangswert `17`. Wie übersetzt man diese Vereinbarung (diesen Befehl des Programmierers an den Ausführer) ins Deutsche?

11. Befehlen Sie dem Java-Ausführer einen Wert zu berechnen, der um 3 größer ist als das Quadrat von `otto`. Wie übersetzt man diesen Befehl ins Deutsche?

12. Befehlen Sie dem Java-Ausführer, er solle den Wert, der um 3 größer ist als das Quadrat von `otto`, der Variablen `otto` zuweisen. Wie übersetzt man diesen Befehl ins Deutsche?

13. In Java kann man (wie in den meisten Programmiersprachen) eine Folge von Befehlen zu einem *Unterprogramm* (oder: zu einer *Methode*) zusammenfassen und mit einem Namen versehen. Welche *Vorteile* hat das typischerweise?

14. Es gibt *zwei Arten von Unterprogrammen* (oder: von *Methoden*). Wie heißen Unterprogramme der einen (bzw. der anderen) Art?

15. Was für eine *Art von Befehl* (siehe 9.) ist es, wenn man ein Unterprogramm der einen Art (bzw. der anderen Art) *aufruft*?

16. Die in einer Klasse vereinbarten *Elemente* kann man auf *drei verschiedene Weisen* in Gruppen einteilen. Geben für jede der drei Weisen an,
- nach welchem *Kriterium* die Elemente eingeteilt werden und
- in *wie viele* und *welche* Gruppen die Elemente eingeteilt werden.

Ihre Antworten sollen etwa folgende Form haben:
Nach Farbe, 3 Gruppen: rote, grüne, blaue Elemente.

17. In Java unterscheidet man *zwei Arten von Typen*. Wie heißen Typen der einen bzw. der anderen Art? Nennen Sie je drei Typen der einen bzw. der anderen Art als Beispiele.

18. In Java unterscheidet man *drei Arten von Referenztypen.* Wie heißen Typen der drei Arten? Nennen Sie von jeder Art mindestens zwei Beispiele.

19. In Java unterscheidet man (wie in den meisten Programmiersprachen) *zwei Arten von Anweisungen*. Wie heißen Anweisungen der einen bzw. der anderen Art? Nennen Sie je drei Anweisungen der einen bzw. der anderen Art als Beispiele.

20. Woraus besteht ein Java-Programm (oder: Was sind die größten sinnvollen Teile eines Java-Programms)?

21. Welche Klassen gehören zu einem Java-Programm namens `Otto`?

22. Wann werden die einzelnen Klassen eines Java-Programms geladen?

23. Was ist ein *Sammlungsobjekt*? Vereinbaren Sie zwei Sammlungsobjekte als Beispiele.

24. Was ist ein *Grabo-Objekt*? Vereinbaren Sie zwei Grabo-Objekte als Beispiele.

25. Was ist ein *Behälterobjekt*? Vereinbaren Sie zwei Behälterobjekte als Beispiele.

26. Ein Grabo-Programm wird von (mind.) 2 sequentiellen Ausführern (oder: Fäden, engl. threads) ausgeführt. Wie heißen diese Fäden?

27. Skizzieren Sie eine typische Situation, in der der Programmierer eines Grabo-Programms einen weiteren Faden (oder: sequentiellen Ausführer, thread) starten sollte.

28. Wie werden Fäden ausgeführt? Langatmig? Nebensächlich? Weitläufig? Wettläufig? Parallel zueinander? Nebenläufig zueinander? Zeitlich abhängig voneinander?

29. Um in Java einen Faden zu programmieren kann man entweder *eine Klasse beerben* oder *eine Schnittstelle implementieren.* Wie heißen diese Klasse und diese Schnittstelle?

30. Welches ist die wichtigste positive Eigenschaft der Programmiersprache Java?

Aufgabe 2: Zufallswege auf PoxelPanels

Ein *Poxel* ist ein rechteckiger Bereich auf einem Bildschirm, dem man eine (Hintergrund-) Farbe geben und in das man ein ASCII-Zeichen (in einer Vordergrund-Farbe) zeichnen kann.

Ein *PoxelPanel* ist ein Grabo-Objekt (engl. GUI object), welches auf dem Bildschirm als *Rechteck von Poxeln* dargestellt wird. Seine Größe (z.B. 60 x 40 Poxel oder 17 x 54 Poxel etc.) kann man bei seiner Erzeugung festlegen. Die Klasse `PoxelPanel.java` finden Sie im Archiv `DateienFuer-Pr2.zip`.

Ein *Zufallsweg* beginnt an einem bestimmten Punkt und besteht aus *Schritten*. Die Länge und Richtung von jedem einzelnen Schritt werden zufällig gewählt.

Teilaufgabe 2.1. Schreiben Sie ein Programm namens `RandomWalk01`, welches einen Zufallsweg zu einem PoxelPanel ausgibt.

Jeder Schritt dieses Weges soll
um `dX` viele Poxel in x-Richtung und
um `dY` viele Poxel in y-Richtung weiterführen ("d" wie "delta").
Die Werte für die `int`-Variable `dX` sollen bei jedem Schritt *zufällig* gewählt werden und zwischen `-mDX` und `+mDX` liegen, wobei `mDX` ("maximales Delta in x-Richtung") eine `int`-Konstante ist. Entsprechendes soll auch für die Variable `dY` und die Konstante `mDY` gelten. Geben Sie den Konstanten `mDX` und `mDY` Werte wie z.B. `2` oder `3` etc.

Sie sollen verhindern, dass ein Zufallsschritt aus dem PoxelPanel *hinausführt* und dazu (durch Methoden namens `nextA`, `nextB` bzw. `nextC`) drei verschiedene Strategien realisieren:

Rand-Strategie-A ("auf der anderen Seite weitermachen"):
Wir machen unser PoxelPanel *unbegrenzt*, indem wir uns den linken und den rechten Rand (und ebenso den oberen und den unteren Rand) *zusammengeklebt* denken (nur denken! Bitte auf keinen Fall Klebstoff auf die Bildschirme der Laborrechner auftragen!). Dann gilt:
Für die x-Richtung: *Nach* dem *letzten* Poxel einer Zeile kommt wieder das *erste* Poxel dieser Zeile, und umgekehrt (*vor* dem ersten das letzte).
Für die y-Richtung: *Nach* dem *letzten* (untersten) Poxel einer Spalte kommt wieder das *erste* (oberste) Poxel dieser Spalte, und umgekehrt (*vor* dem obersten das unterste).

Rand-Strategie-B ("am Rand spieglen"):
Falls ein Schritt über einen Rand hinausgeht, *spiegeln* wir ihn an diesem Rand.

Rand-Strategie-C ("Schritte wählen bis einer paßt"):
Falls ein Zufallsschritt aus dem PoxelPanel hinausführen würde, verwerfen wir ihn und wählen einen neuen Schritt. Das wiederholen wir, bis wir zufällig einen akzeptablen Schritt gefunden haben.

Folgende drei Maßnahmen sollen es dem Benutzer vor dem Bildschirm ermöglichen, den Zufallsweg "als Weg zu sehen" oder sogar "zu erleben":

Maßnahme 1, Verzögerung:
Lassen Sie ihr Programm nach jeder Ausgabe zu einem Poxel ein paar Millisekunden schlafen (mit Hilfe der Klassenmethode `sleep` in der Klass `Thread`). Was passiert, wenn Sie ihr Programm 0 Millisekunden schlafen lassen?

Maßnahme 2, Reihenfolge erkennbar machen:
Geben Sie zu den einzelnen Poxeln die Buchstaben 'A', 'B', 'C', ... 'Z', 'A', 'B', 'C', ... 'Z', 'A', 'B' ... aus (da viele Benutzer mit der Reihenfolge dieser Zeichen gut vertraut sind).

Maßnahme 3, Gruppen von Schritten erkennbar machen:
Wechseln Sie die Farbe ihrer Ausgaben nicht bei jedem Schritt, sondern z.B. nach jeweils 20 oder 30 Schritten. Dadurch werden auf dem Bildschirm "Gruppen von Schritten" erkennbar.

Teilaufgabe 2.2.: Schreiben Sie ein Programm namens `RandomWalk02`. Es soll *ein* PoxelPanel zeichnen und *mehrere* Fäden (engl. threads) erzeugen und starten. Jeder Faden soll einen anderen Zufallsweg auf das PoxelPanel zeichnen und für alle Schritte dieses Weges *eine* bestimmte Farbe verwenden, die sich von den Farben der anderen Fäden unterscheidet (z.B. Faden 1 verwendet `Color.WHITE`, Faden 2 verwendet `Color.RED` etc.).

In den Dateien `RandomWalk01.java` und `RandomWalk02.java` (im Archiv `DateienFuerPr2.zip`) finden Sie "Skelette" der betreffenden Programme, von denen Sie ausgehen und die Sie vervollständigen können.

Tips zu Themen wie Zufallszahlen, Umgang mit Farben, Erzeugen von Fäden etc. gibt es in den Übungen.

Aufgabe 3: Ein Rechner mit Grabo nach dem MVC-Muster strukturiert

Entwickeln Sie einen "kleinen Taschenrechner", der aus 3 Java-Klassen besteht:

Die Model-Klasse implementiert die Schnittstelle `RechnerModel_I` und legt den *Typ* der Zahlen fest, mit denen gerechnet wird (`int` oder `float` oder `BigInteger` oder ... nach Ihrer Wahl).

Die View-Klasse implementiert die Schnittstelle `RechnerView_I` und realisiert die Grabo (grafische Benutzeroberfläche) des Rechners.

Die Control-Klasse verbindet die anderen beiden Klassen.

Vorgegeben sind die beiden Schnittstellen `RechnerModel_I` und `RechnerView_I` und die Control-Klasse namens `RechnerControl`. Sie sollen Gruppen bilden, die aus einer *Model-Person* und einer *View-Person* bestehen. Die Model-Person soll die Model-Klasse und die View-Person soll die View-Klasse programmieren. Beide Personen sind dafür verantwortlich, dass ihre Klassen zusammenpassen und (zusammen mit den vorgegebenen Teilen) einen funktionierenden Rechner ergeben.

Hinweis: Die Gruppenarbeit ist ein wichtiger Bestandteil dieser Aufgabe. Deshalb sollten Sie unbedingt zu allen Übungsterminen kommen und dort die Detailprobleme dieser Aufgabe besprechen und lösen.

Es folgend hier (leich gekürzt) die vorgegebenen Teile dieser Aufgabe (die Sie vollständig als Dateien im Archiv `DateienFuerPR2.zip` finden):

```
 1 //-----------------------------------------------------------------
 2 public interface RechnerModel_I {
 3   String getTypeName();
 4   // Each model (class) implementing this interface should
 5   // "work and calculate" with numbers of a certain type,
 6   // e.g. of type int or of type float or ... .
 7   // This method returns the name of this type as a String
 8   // e.g. "int" or "float" or ... .
 9
10   boolean isValidEntry(String pre);
11   // Returns true, iff there is a String suf, such that
12   // the String pre+suf can be interpreted as a number
13   // of the type expected by the implementing model.
14
15   String[] calculate(String number1, String number2);
16   // Trys to interpret number1 and number2 as numbers z1 and z2
17   // (e.g. of type int or float or ... ), calculates the 5 values
18   // z1+z2, z1-z2, z1*z2, z1/z2, z1%z2 and returns them as an
19   // array of 5 String-objects. In case of problems the String-
20   // objects may contain some error messages.
21 } // inteface RechnerModel_I
```

```
 1 //-----------------------------------------------------------------
 2 public interface RechnerView_I {
 3    // Each implementing class is expected to declare at least
 4    // - 1 submit button      (e.g. of type JButton)
 5    // - 2 input  text fields (e.g. of type JTextField)
 6    // - 5 result text fields (e.g. of type JTextField)
 7
 8    void register(ActionListener al, KeyListener kl1, KeyListener kl2) {
 9    // Registers (using the appropriate addXXXListener method)
10    // al  with the submit button,
11    // kl1 with with the first  input text field and
12    // kl2 with with the second input text field.
13
14    String getNumber1();
15    // Returns the content of the first input text field.
```

```
16    String getNumber2();
17    // Returns the content of the second input text field,
18    void setResults(String[] results);
19    // Expects the array results to be of lenght 5.
20    // Assigns each of its elements to one of the result text fields.
21 } // inteface RechnerView_I
```

```
 1 //-----------------------------------------------------------------
 2 public class RechnerControl {
 3    private RechnerModel_I aModel;
 4    private RechnerView_I  aView;
 5    // ----------------------------------------------------------------------
 6    public RechnerControl(RechnerModel_I aModel, RechnerView_I aView) {
 7       this.aModel = aModel;
 8       this.aView  = aView;
 9       // 1 listener for the submitButton:
10       ActionListener mal  = new MyActionListener();
11
12       // 2 listeners for the input text fields
13       KeyListener    mkl1 = new MyKeyListener();
14       KeyListener    mkl2 = new MyKeyListener();
15
16       // Register the 3 listeners with the view:
17       aView.register(mal, mkl1, mkl2);
18    } // Konstruktor
19    // ----------------------------------------------------------------------
20    // The following listener classes for GUI elements are declared here
21    // (and not in the view), because they call methods which are visible
22    // here, but not in the view (calculate and isValidEntry)
23    //
24    // ActionListener for the submitButton:
25    private class MyActionListener  implements ActionListener {
26       public void actionPerformed(ActionEvent e) {
27          String   s1  = aView.getNumber1();
28          String   s2  = aView.getNumber2();
29          String[] res = aModel.calculate(s1, s2);
30          aView.setResults(res);
31       }
32    }
33
34    // KeyListener for the input text fields:
35    private class MyKeyListener extends KeyAdapter {
36       String oldContent  = "";
37
38       public void keyReleased(KeyEvent ke) {
39          JTextField tf          = (JTextField) ke.getSource();
40          String     newContent  = tf.getText();
41          int        newCaretPos = tf.getCaretPosition();
42          boolean    isValid     = aModel.isValidEntry(newContent);
43
44          if (isValid) {
45             oldContent  = newContent;
46          } else {
47             // If the newContent is not valid, this is probably due to
48             // the character typed last and the following holds:
49             // 1. The caret (cursor) is positioned immediately to
50             //    the right of the faulty character.
51             // 2. This character is not present in the oldContent.
52             // 3. The caret position in the oldContent was probably
53             //    one less than it is in the newContent.
54             // If the user used "copy and paste" (instead of typing)
55             // things might be different.
```

```
56
57              // Restore the oldContent and the old caret position:
58              tf.setText         (oldContent);
59              tf.setCaretPosition(Math.max(0, newCaretPos-1));
60           }
61        } // keyReleased
62     } // class MyKeyAdapter
63     // ---------------------------------------------------------------------
64     // A method with a short name:
65     static void printf(String f, Object... v) {System.out.printf(f, v);}
66     // ---------------------------------------------------------------------
67 } // class RechnerControl
```

Aufgabe 4: Rekursion und Iteration

Betrachten Sie die Datei `ReIter01.java` (siehe unten). Fügen Sie am Anfang der Methoden `iter01`, `iter02`, `reku03`, `reku04` und `reku05` je einen Anfangskommentar (AKO) ein, und zwar jeweils am Anfang des *Rumpfes*, d. h. nach der öffnenden geschweiften Klammer {). Dieser AKO soll die Frage „*Was macht dieses Unterprogramm?*“ möglichst einfach und verständlich beantworten (als simples Beispiel siehe das Upro `schritt01`). Hier ein paar Regeln dazu, wie man solche AKOs gestalten sollte:

Regel 1: Der AKO beginnt normalerweise mit einem Verb ("*Gibt ... aus*", "*Formatiert ...*" etc.).
Regel 2: Der AKO einer Funktion beginnt normalerweise mit "*Liefert ...*" (oder "*Gibt ... zurück*").
Regel 3: Im AKO kommt jeder *Parameter-Name* mindestens einmal vor.
Regel 4: Der AKO beschreibt, *was* das Upro leistet, nicht *wie* es diese Leistung erbringt (ob durch Aufrufe weiterer Methoden oder durch andere Befehle, ob iterativ oder rekursiv etc.).
Regel 5: Im AKO erwähnt man nichts, was schon davor ("in der ersten Zeile der Methode") steht.
Z. B. schreibt man statt "Gibt den übergebenen `String`-Parameter namens `karl_heinz` aus und schiebt unmittelbar anschließend den Bildschirmzeiger (cursor) an den Anfang der nächsten Zeile vor" kürzer "Gibt `karl_heinz` und einen Zeilenwechsel aus".

Unterprogramme mit gleicher Nr. (z. B. `iter01` und `reku01`, oder `reku03` und `iter03` etc.) sollen *exakt das Gleiche* leisten (aber auf verschiedene Weise: iterativ bzw. rekursiv). Ersetzen Sie die „unsinnigen“ Rümpfe der Upros `reku01`, `reku02`, `iter03`, `iter04` und `iter05` entsprechend.

```
 1 // Datei ReIter01.java
 2 ...
 3 class ReIter01 {
 4    // ---------------------------------------------------------------------
 5    static private void schritt01(char char01) {
 6       // Gibt char01 zur Standardausgabe aus.
 7       p(char01);
 8    } // schritt01
 9    // ------------------------------------------------------------ Aufgabe 01
10    static public void iter01(final int N) {
11       for (int i=1; i<=N; i++) {
12          schritt01('X');
13       }
14       pln();
15    } // iter01
16    // ------------------------------------------------------------ Loesung 01
17    static public void reku01(final int N) {
18       // Ersetzen Sie diesen "unsinnigen" Rumpf
19       pln("reku01, unsinniges Ergebnis!");
20    } // reku01
21    // ------------------------------------------------------------ Aufgabe 02
22    static public void iter02(final int N) {
23       int n = N;
24       while (n > 0) {
25          char ziff = (char) (n%2 + '0');
26          schritt01(ziff);
27          n = n/2;
28       }
29       pln();
30    } // iter02;
31    // ------------------------------------------------------------ Loesung 02
32    static public void reku02(final int N) {
33       // Ersetzen Sie diesen "unsinnigen" Rumpf
34       pln("reku02, unsinniges Ergebnis!");
35    } // reku02
```

```
36    // ------------------------------------------------------- Aufgabe 03
37    static public void reku03(final int N) {
38       char ziff = (char) (N%2 + '0'); // "Rechteste" Binaerziffer
39       if        (N <  0) {
40          schritt01('-');
41          reku03(-N);
42       } else if (N == 0 || N == 1) {  // Hoechstwertige Ziffer
43          schritt01(ziff);
44          return;
45       } else {
46          reku03(N/2);
47          schritt01(ziff);
48       }
49    } // reku03
50    // ------------------------------------------------------- Loesung 03
51    static public void iter03(final int N) {
52       // Ersetzen Sie diesen "unsinnigen" Rumpf
53       p("iter03, unsinniges Ergebnis!");
54    } // iter03
55    // ------------------------------------------------------- Aufgabe 04
56    static public String reku04(String s) {
57       if (s.length() <= 1) {
58          return s;
59       } else {
60          char   letzt     = s.charAt   (   s.length()-1);
61          String ohneLetzt = s.substring(0, s.length()-1);
62          return letzt + reku04(ohneLetzt);
63       }
64    } // reku04
65    // ------------------------------------------------------- Loesung 04
66    static public String iter04(String s) {
67       // Ersetzen Sie diesen "unsinnigen" Rumpf
68       return "iter04, unsinniges Ergebnis!";
69    } // iter04
70    // ------------------------------------------------------- Aufgabe 05
71    static public int reku05(int[] r1, int[] r2) {
72       return reku05a(r1, r2, 0);
73    }
74
75    static private int reku05a(int[] r1, int[] r2, int index) {
76       if (index >= r1.length || index >= r2.length) {
77          return r1.length - r2.length;
78       } else if (r1[index] != r2[index]) {
79          return r1[index] - r2[index];
80       } else {
81          return reku05a(r1, r2, index+1);
82       }
83    } // reku05a
84    // ------------------------------------------------------- Loesung 05
85    static public int iter05(int[] r1, int[] r2) {
86       // Ersetzen Sie diesen "unsinnigen" Rumpf
87       return -999999999; // unsinniges Ergebnis
88    } // iter05
89    // ---------------------------------------------------------------------
90    static public void main(String[] _) {
91       ...
92    } // main
93    // ---------------------------------------------------------------------
94 } // class ReIter01
```

Eine "unvollständige" Datei `ReIter01.java` finden Sie am üblichen Ort. *Testen* Sie Ihre Lösung, indem Sie geeignete Befehle in die `main`-Methode schreiben und ausführen lassen.

Aufgabe 5: Eine Collection-Klasse namens MeineSammlung entwickeln und testen

Als "Rohmaterial" für diese Aufgabe sind zwei Dateien vorgegeben (`MeineSammlung.java` und `MeineSammlungJut.java`).

Datei 1: In der Datei `MeineSammlung.java` wird eine `Collection`-Klasse namens `MeineSammlung` vereinbart. Darin sind drei Konstruktoren und einige Methoden *fertig vereinbart* (darunter alle in der Schnittstelle `Collection` als optional gekennzeichneten Funktionen wie `add`, `remove`, `addAll` etc.). Weitere Methoden, die von der Schnittstelle `Collection` vorgeschrieben werden (z.B. `size`, `isEmpty`, `toString`, `contains` etc.) sind *nur vorläufig und formal* implementiert. Wenn man eine dieser *unfertigen Funktionen* aufruft, wird nur eine Fehlermeldung ausgegeben, z.B.

```
Eine noch nicht implementierte Methode wurde aufgerufen: size
```

und ein unsinniges Ergebnis geliefert (z.B. `-999` oder `true` oder `null` etc.).

Sie sollen diese unfertigen Funktionen (eine nach der anderen) richtig implementieren und dabei (mit dem vorgegebenen Testprogramm `MeineSammlungJut`) testen.

Datei 2: Die Datei `MeineSammlungJut.java` enthält ein JUnit-Programm zum Testen der Klasse `MeineSammlung`. Dieses Testprogramm enthält 31 Test-Methoden, von denen aber anfangs 22 *auskommentiert* ("deaktiviert") sind.

Wenn man die beiden Quelldateien (`MeineSammlung.java` und `MeineSammlungJut.java`, unverändert) compiliert und das Testprogramm ausführen lässt, sollte es einen *grünen Balken* zeigen. Das liegt aber vor allem daran, dass viele Testmethoden noch *auskommentiert* sind.

Sie sollen die Klasse `MeineSammlung` testgetrieben und schrittweise fertig entwickeln. Jeder Schritt besteht aus zwei Teilschritten:

Teilschritt-1: *Aktivieren* ("entkommentarisieren") Sie in der Quelldatei `MeineSammlungJut.java` die nächste Gruppe von deaktivierten Testmethoden, z.B. die Gruppe

```
//    ts1.addTest(new MeineSammlungJut("testSize01"));
//    ts1.addTest(new MeineSammlungJut("testSize02"));
```

Die dadurch aktivierten Testmethoden `testSize01` und `testSize02` testen (wie ihre Namen andeuten) die Methode `size` in der Klasse `MeineSammlung`. Wenn Sie jetzt das Testprogramm `MeineSammlungJut` erneut compilieren und ausführen lassen, wird es einen *roten Balken* zeigen (weil die Methode `size` noch nicht richtig implementiert ist).

Teilschritt-2: Ersetzen Sie in der Quelldatei `MeineSammlung.java` den *vorläufigen und falschen* Rumpf der entsprechenden Methode (z.B. den Rumpf der Methode `size`) durch einen *richtigen Rumpf*. Lassen Sie dann das Testprogramm `MeineSammlungJut` ausführen. Wenn es einen *grünen Balken* zeigt, ist dieser Schritt fertig. Ansonsten müssen Sie den Rumpf der aktuellen Methode so lange verbessern, bis das Testprogramm grün zeigt.

Sie haben diese Aufgabe fertig gelöst, wenn Sie
1.) im Testprogramm `MeineSammlungJut.java` *alle Testmethoden* aktiviert und
2.) in der Datei `MeineSammlung.java` alle unsinnigen Methodenrümpfe durch richtige Rümpfe ersetzt haben und
3.) das Testprogramm dann einen *grünen Balken* zeigt.

Was die einzelnen Methoden (`size`, `isEmpty`, `toString`, `contains` etc.) genau machen sollen, ist in der Online-Dokumentation der Java-Standardbibliothek (Schnittstelle `Collection`) beschrieben. Lesen Sie diese Beschreibungen genau durch und besprechen Sie sie in Ihrer Arbeitsgruppe, bis Ihnen alle Einzelheiten (einschliesslich aller englischen Vokabeln in den Beschreibungen) klar sind.

Was die Methoden `toString` und `equals` machen sollen, wird in der Online-Dokumentation der Schnittstelle `Collection` nur *teilweise* festgelegt. Beim Implementieren dieser Methoden müssen Sie auch die *Anfangskommentare* (in der Datei `MeineSammlung.java`) berücksichtigen, sonst zeigt das Testprogramm `MeineSammlungJut` keinen grünen Balken an!

Hier noch ein paar Tips (darf man befolgen) und Auflagen (muss man befolgen) zu einzelnen Methoden der Klasse `MeineSammlung`:

`size`: --

`isEmpty`: Auflage: Der Rumpf darf nur eine `return`-Anweisung (und auf keinen Fall eine `if`-Anweisung) enthalten.

`toString`: Tip: Lesen Sie in der Online-Dokumentation nach (und probieren Sie aus), was die Methode `Arrays.toString` leistet.

`contains`: Tip: Achten Sie auf `null`-Werte: 1. Diese Sammlung (`this`) kann `null`-Komponenten enthalten. 2. Der Parameter `ob` dieser Methode kann gleich `null` sein.

`hasNext`: Auflage: Der Rumpf darf nur eine `return`-Anweisung (und auf keinen Fall eine `if`-Anweisung) enthalten.

`next`: Tip: Achten Sie darauf, dass in Ausnahmefällen eine Ausnahme *des richtigen Typs* (`NoSuchElementException`) geworfen wird. Ausnahmen des Typs `ArrayIndexOutOfBoundsException` sind hier (laut Online-Dokumentation zur Schnittstelle `Collection`) *nicht* erlaubt.

`containsAll`: Auflage: Benutzen Sie einen Iterator, um alle Komponenten der Sammlung `c` zu bearbeiten (und keine teureren Befehle wie z.B. `toArray`).

`equals`: Tip: Mit dem Operator `instanceof` können Sie prüfen, ob ein beliebiges *Objekt* zu einem bestimmten *Typ* gehört. Beispiel: Der Ausdruck `ob instanceof String` hat genau dann den Wert `true`, wenn `ob` ein Objekt des Typs `String` ist.

`toArray`: (ohne Parameter) Tip: Die Klasse `Arrays` enthält 10 Methoden namens `copyOf`. Mit der vorletzten dieser Methoden (siehe die Online-Dokumentation zu `Arrays`) kann man die Methode `toArray` besonders kurz und elegant implementieren.

`toArray`: (mit Parameter) Tip: Warten Sie mit der Implementierung dieser Methode, bis sie in der Übung besprochen wurde (die Implementierung erfordert schwierige, noch nicht behandelte Befehle). Wenn Sie besonders wissbegierig sind und die Übung nicht abwarten möchten, können Sie in der Quelldatei der Klasse `ArrayList` nachschauen, wie diese `toArray`-Methode dort implementiert wurde. Wie man sich in Eclipse den Quelltext einer (Standard-) Klasse anzeigen lassen kann, wird im Papier `TipsZu-Eclipse` erläutert.

Aufgabe 6: Die toString-Methode der Klasse HashSet testen

Schreiben Sie ein JUnit-Testprogramm, welches die Methode namens `toString` in Objekten des Typs `java.util.HashSet<String>` testet.

Von den Konstruktoren und Methoden der Klasse `HashSet` soll Ihr Testprogramm nur

den Standard-Konstruktor,
die Methode `add` und natürlich
die Methode `toString`

aufrufen. Die übrigen Konstruktoren und Methoden (so sollte man sich vorstellen) sind noch ungetestet und sollen deshalb *nicht* zum Testen der Methode `toString` verwendet werden.

Tip 1: Unterschätzen Sie die Schwierigkeit dieser Aufgabe nicht.

Tip 2: Problem: Angenommen, wir haben die drei Komponenten `"ABC"`, `"BCDE"` und `"CD"` in eine Sammlung `hs` (vom Typ `HashSet<String>`) eingefügt. Dann ist jede der folgenden sechs Zeichenketten ein korrektes Ergebnis des Funktionsaufrufs `hs.toString()`:

```
"[ABC, BCDE, CD]", "[ABC, CD, BCDE]",
"[BCDE, ABC, CD]", "[BCDE, CD, ABC]",
"[CD,, ABC BCDE]", "[CD, BCDE, ABC]".
```

Die Anzahl der korrekten Ergebnisse wächst ziemlich schnell mit der Anzahl der Komponenten (bei n Komponenten gibt es n! viele korrekte Ergebnisse. Für n gleich 20 sind das mehr als 2 Trillionen Ergebnisse). Der Plan, alle korrekten Ergebnisse "durchzuprobieren" ist also schon für relativ kleine Sammlungen nicht mehr durchführbar. Aber wie kann man das Problem schneller lösen?

Tip 3: Vereinbaren Sie in Ihrem Testprogramm eine Hilfsmethode entsprechend der folgenden Spezifikation:

```
1    private void doTheTesting(String[] sr) {
2       // Aus sr wird eine Sammlung hs des Typs HashSet<String> gemacht.
3       // Das Ergebnis von hs.toString() wird getestet.
4       // Beispiele fuer korrekte Ergebnisse:
5       // "[]", "[ABC]", "[ABC, BCDE, CD]"
6       ...
7    } // doTheTesting
```

Die einzelnen Testmethoden brauchen dann nur diese Hilfsmethode mit geeigneten Parametern (des Typs `String[]`) aufzurufen.

Tip 4: Rufen Sie die Methode `doTheTesting` insbesondere mit den folgenden Parametern auf:

```
 8    doTheTesting(new String[]{});
 9    doTheTesting(new String[]{"ABC"});
10    doTheTesting(new String[]{"ABC", "BCDE", "CD"});
11    doTheTesting(new String[]{
12       "01", "02", "03", "04", "05", "06", "07", "08", "09", "10",
13       "11", "12", "13", "14", "15", "16", "17", "18", "19", "20"
14    });
15    doTheTesting(new String[]{
16       "A", "AA", "AAAA", "AAAAA", "AAAAAA", "AAAAAAA"
17    });
18    doTheTesting(new String[]{
19       "A", "]", "AA", "[", "AAA", ", , , ", "AAAA", "][", ", "
20    });
```

Die ersten vier Aufrufe sind noch relativ einfach, die letzten drei sind relativ schwierig.

Tip 5: Es ist empfehlenswert, die Übungen regelmäßig zu besuchen und dort Probleme mit anderen zu diskutieren.

Aufgabe 7: Eine Set-Klasse namens MyHashSet entwickeln und testen

1. Betrachten Sie die Schnittstelle `Collection` und die Klassen `AbstractCollection` , `AbstractSet` und `HashSet` (alle drei gehören zum Paket `java.util`). Zeichnen Sie ein möglichst einfaches Diagramm aus dem hervorgeht, wer wen erweitert bzw. implementiert.

2. In der Dokumentation zur Schnittstelle `Collection` und zu den Klassen `AbstractCollection` und `HashSet` werden u.a. anderem zwei Methoden namens `add` und `addAll` beschrieben. Welche *Ausnahmen* werden von diesen Methoden (möglicherweise) geworfen? Lesen Sie in der JavaDoc nach und tragen Sie in der folgende Tabelle an den entsprechenden Stellen Kreuzchen ein:

	interface Collection		class AbstractCollection		class HashSet	
Die Methode → wirft die ↓ Ausnahmen	`add`	`addAll`	`add`	`addAll`	`add`	`addAll`
UnsupportedOperationException						
ClassCastException						
NullpointerException						
IllegalArgumentException						
IllegalStateException						

3. In der Klasse `AbstractCollection` ist die Methode `addAll` wie folgt vereinbart:

```
1    public boolean addAll(Collection<? extends E> c) {
2       boolean modified = false;
3       Iterator<? extends E> e = c.iterator();
4       while (e.hasNext()) {
5          if (add(e.next()))
6             modified = true;
7       }
8       return modified;
9    }
```

Welche Ausnahmen wird dieses Methode höchstens werfen? Welche Spalte in der obigen Tabelle enthält also zu viele Kreuzchen?

4. Vereinbaren Sie eine Klasse namens `MyHashSet` entsprechend der folgenden Spezifikation:

```
/* ------------------------------------------------------------------------
Die Klasse MyHashSet ist eine direkte Unterklasse von java.util.HashSet.
In MyHashSet-Sammlungen kann man nur Objekte mit folgenden
Eigenschaften sammeln:
1. Die Objekte sind nicht Serializable.
2. Das Ergebnis ihrer toString-Methode ist hoechstens 10 Zeichen lang.
------------------------------------------------------------------------
Diese Klasse ueberschreibt nur die add-Methode. Die neue add-Methode wirft
- eine ClassCastException,
     wenn ihr Parameter Serializable ist, und
- eine IllegalArgumentException,
     wenn die String-Representation ihres Parameters laenger als 10 ist.
------------------------------------------------------------------------ */
```

5. Schreiben Sie ein JUnit-Testprogramm namens `MyHashSetJut01`, welches (nur) die `add`-Methode Ihrer Klasse `MyHashSet` testet.

Aufgabe 8: Formatierungen mit printf

Die Methoden namens `printf` bzw. `format` (in den Klassen `Formatter`, `String`, `PrintStream`, `PrintWriter` etc.) leisten weitgehend das Gleiche. Die verbindliche Beschreibung dieser Methoden findet man in der Dokumentation Klasse `java.util.Formatter`. Als Ergänzung kann man die Möglichkeiten dieser Methoden mit zwei Applets (`eTeachMePrintf` von S. Effenberg bzw. `PrintfApplet`) experimentell erkunden.

Zur Lösung dieser Aufgabe sollten Sie sich mit diesen Methoden namens `printf` alias `format` vertraut machen.

Schreiben Sie eine Klasse namens `Printf01` mit 3 Klassenmethoden darin, die den folgenden Spezifikationen entsprechen:

```
 1 class Printf01 {
 2    // ------------------------------------------------------------------------
 3    static public void gibIntsKompaktAus(int anz) {
 4       /* ---------------------------------------------------------------------
 5       Gibt folgende Daten passend fuer einen Bildschirm mit 80 Zeichen pro
 6       Zeile aus:
 7       Zwei Zeilen mit je 79 Minuszeichen.
 8       Zwischen diesen Minuszeilen genau anz viele Zufallszahlen vom Typ
 9       int, 5 Zahlen pro Zeile, lesbar formatiert wie z.B.:  -1.188.957.731
10
11       Beispiel fuer eine Ausgabe dieser Methode:
12       -------------------------------------------------------------------------------
13        -1.188.957.731  1.018.954.901    -39.088.943  1.295.249.578  1.087.885.590
14        -1.829.099.982 -1.680.189.627  1.111.887.674   -833.784.125 -1.621.910.390
15          -535.098.017 -1.935.747.844 -1.219.562.352    696.711.130    308.881.275
16        -1.366.603.797
17       -------------------------------------------------------------------------------
18       Hinweis. Zum Test sollte diese Methode auch in einem Dos-Fenster mit
19       80 Zeichen pro Zeile aufgerufen werden!
20       ------------------------------------------------------------------------ */
21       ...
22    } // gibIntsKompaktAus

23    // ------------------------------------------------------------------------
24    static public void gibDreispaltigAus(String[] sr, int[] ir, float[] fr) {
25       /* ---------------------------------------------------------------------
26       Gibt eine Fehlermeldung aus, wenn die die Reihungen sr, ir und fr
27       nicht gleich lang sind. Gibt sonst die Daten der drei Reihungen in
28       Form einer dreispaltigen Tabelle aus, z.B. so:
29
30       +----------+----+--------+
31       |Katzen    |  12|  123,45|
32       |Hunde     |   8|    3,46|
33       |Kaninchen | 130|2.734,40|
34       |Floehe    |5000|    0,01|
35       +----------+----+--------+
36
37       Die Spalten haben die Breiten 10, 4 und 8. Die Methode verlaesst sich
38       darauf, dass das ausreicht.
39       ------------------------------------------------------------------------ */
40       ...
41    } // gibDreispaltigAus
42    // ------------------------------------------------------------------------
43
```

```
44    static public void gibMoeglichstSchmalAus(int[] ir) {
45       /* -----------------------------------------------------------------
46       Die Komponenten von ir werden lesbar formatiert ausgegeben, eine Zahl
47       pro Zeile, die Einerstellen genau untereinander. Alle Zahlen werden
48       "so schmal wie moeglich" formatiert, d.h. nur so breit wie die
49       breiteste Zahl in ir es erfordert, wie in den folgenden Beispielen:
50
51       Beispiel 1: Relativ schmale Zahlen (Breite: 4)
52         17
53       -123
54        520
55          6
56
57       Beispiel 2: Relativ breite Zahlen (Breite 14)
58        1.234.567.890
59                  34
60       -1.234.567.890
61          987.654.321
62                  278
63       ----------------------------------------------------------------- */
64       ...
65    } // gibMoeglichstSchmalAus
66    // ---------------------------------------------------------------------
67    ...
68    // ---------------------------------------------------------------------
69    // Zwei Methoden mit kurzen Namen:
70    static void   printf(String f, Object... v) {System.out.printf(f, v);}
71    static String format(String f, Object... v) {
72       return String.format(f, v);
73    }
74    // ---------------------------------------------------------------------
75 } // class Printf01
```

Aufgabe 9: XML-Dokumente erstellen, einlesen, erzeugen, validieren etc.

Betrachten Sie folgende Dokumenten-Typ-Definition:

```
 1 <?xml version="1.0" encoding="UTF-8"?>
 2 <!-- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
 3    Datei stundenplan.dtd:
 4    Document Type Definition fuer die Dateien sp01.xml, sp02.xml etc.
 5 - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -  -->
 6 <!ELEMENT stundenplan  (kopf, lv+)>
 7
 8 <!ELEMENT kopf         (semester, studiengang, name?)>
 9 <!ELEMENT semester     (#PCDATA)>
10 <!ELEMENT studiengang  (MB | TB | MM | EM)>
11 <!ELEMENT name         (#PCDATA)>
12
13 <!-- Die folgenden Elemente muessen immer leer sein! -->
14 <!ELEMENT MB EMPTY>
15 <!ELEMENT TB EMPTY>
16 <!ELEMENT MM EMPTY>
17 <!ELEMENT EM EMPTY>
18
19  <!-- "lv" wie "Lehrveranstaltung -->
20 <!ELEMENT lv           (#PCDATA)>
21
22 <!ATTLIST lv
23    kuerzel ID                                        #REQUIRED
24    w_tag   (mo | di | mi | do | fr | sa | so)        #REQUIRED
25    block   (b1 | b2 | b3 | b4 | b5 | b6 | b7 | bx)   #REQUIRED
26    dauer   ( 1 |  2 |  3 | dx)                       "1"
27 >
```

Teilaufgabe 4.1.: *Erstellen* Sie (mit einem Editor) ein XML-Dokument, welches entsprechend der obigen DTD *gültig* (engl. valid) ist. Dieses Dokument soll eine Dokumenten-Typ-Deklaration enthalten, etwa so:

```
<!DOCTYPE stundenplan SYSTEM "stundenplan.dtd">
```

und mindestens 5 Lehrveranstaltungen (`lv`-Elemente) beschreiben. Speichern Sie das Dokument in einer Datei namens `sp01.xml`.

Teilaufgabe 4.2.: Programmieren Sie eine Klasse namens `LoesungXml01`, die eine `main`-Methode und eine Klassenmethode mit der folgenden Spezifikation enthält:

```
28    static org.jdom.Document createDoc() {
29       // Erzeugt ein JDOM-Document-Objekt, welches moeglichst genau dem
30       // XML-Dokument in der Datei sp01.xml entspricht (ohne dazu eine
31       // Datei einzulesen).
```

In der Datei `CreateJdom.java` findet man Beispiele dafür, wie man JDOM-Objekte *erzeugen* kann ("erzeugen" im Gegensatz zu "einlesen").

Die `main`-Methode der Klasse `LoesungXml01` soll folgendes bewirken:

4.2.1. Die Datei `sp01.xml` wird als ein DOM-Dokument (des Typs `org.w3c.dom.Document`) *eingelesen* und gegen die Dokumenten-Typ-Definition (DTD) in der Datei `stundenplan.dtd` validiert. Das DOM-Dokument und das Ergebnis der Validierung (`true` bzw. `false`) werden ausgegeben.

4.2.2. Ein JDOM-Dokument (des Typs `org.jdom.Document`) wird (mit Ihrer Methode `createDoc`, siehe oben) *erzeugt*, zum Bildschirm ausgegeben und in die XML-Datei `sp02.xml` geschrieben.

4.2.3. Die Datei `sp02.xml` wird als ein DOM-Dokument (des Typs `org.w3c.dom.Document`) *eingelesen* und gegen die Dokumenten-Typ-Definition (DTD) in der Datei `stundenplan.dtd` validiert. Das DOM-Dokument und das Ergebnis der Validierung (`true` bzw. `false`) werden ausgegeben.

In Ihrem Programm `LoesungXml01` dürfen (und sollen) Sie Methoden aus den Moduln `XmlJStd` und `XmlJdom` aufrufen. Die Quelldateien dieser Module enthalten ziemlich umfangreiche Kommentare und Erläuterungen. Diese Module, die Datei `stundenplan.dtd` und die Datei `CreateJdom.java` finden Sie am üblichen Ort.

Teilaufgabe 4.3. (freiwilliger Zusatz):

Ergänzen Sie in der Klasse `LoesungXml01` eine Methode mit der folgenden Spezifikation:

```
32    static void gibLvsAus(org.jdom.Document jdomDoc) {
33       // Das jdomDoc sollte gueltig bezueglich der DTD stundenplan.dtd
34       // sein. Gibt eine Liste aller Lehrveranstaltungen (lv-Elemente)
35       // aus (zur Standardausgabe), die im Stundenplan jdomDoc beschrieben
36       // werden.
```

Rufen Sie (in der `main`-Methode der Klasse `LoesungXML01`) diese Methode `gibLvsAus` (mit einem geeigneten Parameter) auf, so dass eine Liste von Lehrveranstaltungen zum Bildschirm ausgegeben wird.

Aufgabe 10: Ein reflektives Programm

Schreiben Sie eine Klasse namens `Reflektion04` und darin eine Klassenmethode wie folgt:

```
 1    static public int addIntAttribute(Object ob) {
 2       // Greift reflektiv auf alle Attribute A im Objekt ob zu, die
 3       // die folgenden Eigenschaften haben:
 4       // 1. A ist ein Attribut des Typs int.
 5       // 2. Im Namen von A kommt der String "i" vor.
 6       // 3. A ist ein Objektattribut (kein Klassenattribut).
 7       //
 8       // Liefert die Summe aller Attribute mit diesen Eigenschaften.
 9
10    } // addIntAttribute
```

Beispiel für eine Anwendung dieser Funktion: Sei folgende Klasse gegeben:

```
11 public class Klazz01 {
12           public    int public_oa_i1 =   1;
13           protected int protec_oa_i1 =   2;
14                     int paketw_oa_i1 =   4;
15           private   int privat_oa_i1 =   8;
16
17    static public    int public_ka_i1 = 100;
18    static protected int protec_ka_i1 = 200;
19    static           int paketw_ka_i1 = 400;
20    static private   int privat_ka_i1 = 800;
21 } // class Klazz01
```

Sei `otto` ein `Klazz01`-Objekt. Dann enthält `otto` offenbar vier Attribute, die die Bedingungen 1. bis 3. (siehe oben Zeile 4 bis 6) erfüllen und der Funktionsaufruf `addIntAttribute(otto)` sollte als Ergebnis die Zahl 15 liefern (weil 1 + 2 + 4 + 8 gleich 15 ist).

Hinweis: Die Methode `addIntAttribute` muss also auch auf *private* Attribute ihres Parameters `ob` zugreifen (was normalerweise nicht erlaubt, mit Reflexion aber möglich ist).