

**Inhaltsverzeichnis**

Wie bekommt man eine Note für das Fach MB2-PR2?.....	2
Aufgabe 1: Eine JUnit-Testklasse entwickeln (Wiederholung aus PR1).....	3
Aufgabe 2: Mieze, eine noch ziemlich kleine Katze.....	4
Aufgabe 3: Zufallswege auf PixelPanels.....	6
Aufgabe 4: Ein Rechner mit Grabo nach dem MVC-Muster strukturiert.....	8
Aufgabe 5: Rekursion und Iteration.....	11
Aufgabe 6: Eine Collection-Klasse namens MeineSammlung entwickeln und testen.....	13
Aufgabe 7: Die toString-Methode der Klasse HashSet testen.....	15
Aufgabe 8: Eine Set-Klasse namens MyHashSet entwickeln und testen.....	16
Aufgabe 9: Formatierungen mit printf.....	17
Aufgabe 10: XML-Dokumente erstellen, einlesen, erzeugen, validieren etc.....	19
Aufgabe 11: Ein reflektives Programm.....	21

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

Alte, nicht mehr aktuelle Version. Die aktuelle Version finden Sie auf der folgenden Seite.

Im Laufe des Semesters werden 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 Punkte) wird nach der folgenden Tabelle eine *Übungsnote* berechnet:

<b>Punkte (ab):</b>	95	90	85	80	75	70	65	60	55	50	sonst
<b>Note:</b>	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 (Krankheit, keine Lust, S-Bahn-Streik, verschlafen, dringende Ferienreise wegen eines hohen Lottogewinns, ...).

Die Fragen in den Tests beziehen sich

1. auf die *Aufgaben*, die 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.

Zur Bearbeitung eines Tests haben Sie jeweils ca. 10 Minuten. Der erste Test findet am **Mo 02.04.2012** im **3. Block** (d.h. am Anfang des seminaristischen Unterrichts SU) statt. Ob die weiteren Tests ebenfalls am Anfang des SU oder am Anfang der Übungstermine geschrieben werden, wird noch bekannt gegeben.

Wenn Ihre Übungsnote besser als 5,0 ist, dürfen Sie an der **Hauptklausur** am Ende des SS12 und (falls Sie an der Hauptklausur nicht teilnehmen oder sie nicht bestehen) an der **Nachklausur** kurz vor dem WS12/13 teilnehmen. In der Hauptklausur (und ebenso in der Nachklausur) 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 (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

**Empfehlung:** Nehmen Sie an allen Tests teil, bei denen Ihnen das möglich ist. Vermeiden Sie es, schon am Anfang des Semesters einen Test zu verpassen. Rechnen Sie damit, dass Sie gegen Ende des Semesters aus nicht-vorhergesehenen Gründen (Krankheit, Einladung zu einem Besuch beim Bundespräsidenten, eine neue Zweierbeziehung, Druck durch andere Fächer, ...) einen Test "auslassen müssen".

Wie bekommt man eine Note für das Fach MB2-PR2? Die neue, aktuelle Version.

Im Laufe des Semesters werden 10 Tests geschrieben. Bei jedem Test können Sie maximal 20 Punkte erreichen. Am Ende des Semesters werden die Punkte Ihrer 7 besten Tests addiert und aus der durch 1,4 geteilten Summe (Maximalwert:  $7 * 20 / 1.4$  gleich 100 Punkte) 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 10 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 (Krankheit, keine Lust, S-Bahn-Streik, verschlafen, dringende Ferienreise wegen eines hohen Lottogewinns, ...).

Die Fragen in den Tests beziehen sich

1. auf die *Aufgaben*, die 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.

Wenn Ihre Übungsnote besser als 5,0 ist, dürfen Sie an der *Hauptklausur* am Ende des SS12 und (falls Sie an der Hauptklausur nicht teilnehmen oder sie nicht bestehen) an der *Nachklausur* kurz vor dem WS12/13 teilnehmen. In der Hauptklausur (und ebenso in der Nachklausur) 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 (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

**Empfehlung:** Nehmen Sie an allen Tests teil, bei denen Ihnen das möglich ist. Vermeiden Sie es, schon am Anfang des Semesters einen Test zu verpassen. Rechnen Sie damit, dass Sie gegen Ende des Semesters aus nicht-vorhergesehenen Gründen (Krankheit, Einladung zu einem Besuch beim Bundespräsidenten, eine neue Zweierbeziehung, Druck durch andere Fächer, ...) einen Test "auslassen müssen".

**Aufgabe 1: Eine JUnit-Testklasse entwickeln (Wiederholung aus PR1)**

Ein Programmierer Peter (Name von der Redaktion verändert :-)) hat in einer Klasse namens `MyLong` eine Methode `public static String toString(long i, int radix) {...}` geschrieben. Diese Methode soll genau das Gleiche leisten, wie die entsprechende Methode in der Standardklasse `java.lang.Long` (siehe die Online-Dokumentation der Java-Standardbibliothek). Sie sollen mit der Entwicklungsumgebung **Eclipse** und **JUnit** (nach Ihrer Wahl mit **JUnit 3** oder **JUnit 4**) eine Testklasse entwickeln, die die `toString`-Methode von Peter testet (und möglichst alle darin noch enthaltenen Fehler aufdeckt).

**Auflage:** In Ihrer Testklasse sollen *keine* Methoden aus der Standardklasse `java.lang.Long` benutzt werden.

**Tipp1:** Im Archiv `DateienFuerPR2.zip` finden Sie die Datei `MyLong.java` (die ist vollständig, enthält aber möglicherweise noch Fehler).

**Tipp2:** Im Papier `TipsFuerEclipse.pdf` finden Sie (unter anderem) ein paar Hinweise, wie man mit Eclipse JUnit-Testklassen entwickeln kann.

## Aufgabe 2: Miez, eine noch ziemlich kleine Katze

Ein `Miez`-Objekt ist im wesentlichen ein **Fenster** (technisch: ein `JFrame`-Objekt) mit einem **Knopf** (einem `JCheckBox`-Objekt) und einem **Etikett** (einem `JLabel`-Objekt) darin.

In einem `Miez`-Fenster sollen **Ereignisse** der Art `mouseClicked` behandelt werden, indem der `MouseEvent`-Parameter der Methode `mouseClicked` zur Standardausgabe ausgegeben wird.

Ausserdem sollen die Koordinaten solcher `mouseClicked`-Ereignisse („Wo im Fenster hat der Benutzer mit der Maus geklickt?“) in das Etikett geschrieben werden.

**Der Clou:** Erst wenn der `JCheckBox`-Knopf AN-geklickt wird, soll beim `JFrame`-Fenster ein `MouseListener`-Objekte **angemeldet** werden (und dadurch die Beobachtung von `mouseClicked`-Ereignisse beginnen). Wenn der Knopf später AUS-geklickt wird, soll das `MouseListener`-Objekt wieder **abgemeldet** werden (und die Beobachtung von `mouseClicked`-Ereignisse wieder aufhören).

### Die Aufgabe Schritt für Schritt:

Erweitern Sie die Klasse `javax.swing.JFrame` zu einer Klasse namens **Miez**, und vereinbaren Sie in der Klasse `Miez` folgende Objektelement:

1. Ein `JCheckBox`-Objekt namens **knopf01** (Beschriftung "Clicked")
2. Ein `JLabel`-Objekt namens **etikett01** (anfangs "-----")
3. Ein `Box`-Objekt namens **kasten01** (eine horizontale Box)
4. Ein `ActionListener`-Objekt namens **knopfClickedBehandler** mit der ueblichen Methode `actionPerformed` darin.
5. Ein `MouseListener`-Objekt namens **fensterClickedBehandler** mit eine Methode `mouseClicked` darin.

Die Klasse **Miez** soll einen **Konstruktor** mit einem `String`-Parameter **titel** haben, der folgendes leistet:

1. Er ruft `super(titel)` auf.
2. Er fuegt den **knopf01** und das **etikett01** in den **kasten01** ein (mit der Methode `kasten01.add`)
3. Er fuegt den **kasten01** in das aktuelle **Miez**-Objekt ein (mit der Methode `this.add`)
4. Er meldet den **knopfClickedBehandler** beim **knopf01** an (mit der Methode `knopf01.addActionListener`)
5. Er legt fuer das aktuelle **Miez**-Objekt eine vernuenftige Groesse fest (z. B. 500 mal 300 Pixel, mit der Methode `this.setBounds`) und macht es sichtbar (mit der Methode `this.setVisible`).

Die Methode `public void mouseClicked(MouseEvent me){...}` im Behandlerobjekt **fensterClickedBehandler** soll das Objekt **me** zur Standardausgabe ausgeben. Ausserdem soll sie die x- und y-Koordinate im Objekt **me** ermitteln (mit den Methoden `me.getX()` und `me.getY()`) und (in lesbarer Form) in das **etikett01** schreiben (mit der Methode `etikett01.setText`)

Die Methode `public void actionPerformed(ActionEvent ae){...}` im Behandlerobjekt **knopfClickedBehandler** soll das Objekt **ae** zur Standardausgabe ausgeben.

Ausserdem soll sie folgendes leisten:

Wenn der **knopf01** "an ist" (`knopf01.isSelected()`) soll das Behandlerobjekt **fensterClickedBehandler** bei der `ContentPane` des aktuellen **Miez**-Objekts **angemeldet** werden (mit der Methode `Miez.this.addMouseListener`).

Sonst soll das angemeldete Behandlerobjekt **abgemeldet** werden (mit der Methode `Miez.this.removeMouseListener`).

Wenn die `Miez` richtig funktioniert, sollen Sie noch mindestens den Zusatz 1 realisieren (siehe unten). Die Zusätze 2 und 3 sind optional.

**Zusatz 1:** Ganz entsprechend wie `mouseClicked`-Ereignisse sollen auch `mouseDragged`-Ereignisse beobachtet werden. Das An- und Abschalten dieser Beobachtungen soll mit einem Knopf mit der Beschriftung `mouseDragged` erfolgen.

**Zusatz 2:** Ganz entsprechend sollen auch `mouseWheelMoved`-Ereignisse (an- und abschaltbar) beobachtet werden.

**Zusatz 3 (Katze):** Ganz entsprechend sollen Mausereignisse *aller acht Arten* (`mouseClicked`, `mouseEntered`, `mouseExited`, `mousePressed`, `mouseReleased`, `mouseDragged`, `mouseMoved` und `mouseWheelMoved` (einzeln an- und abschaltbar) beobachtet werden. Alle Größen, die acht Mal vorkommen (für jede Art von Ereignis einmal) sollten in Reihungen organisiert werden.

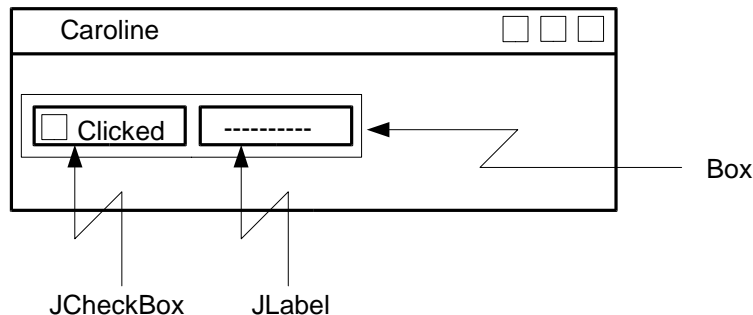
## Aufgabe 2: Miez, eine noch ziemlich kleine Katze (Version 2)

**Allgemeiner Tip:** Entwickeln Sie Programme in kleinen Schritten, von denen Sie jeden mit einer Prüfung oder einem Test abschliessen können.

Schreiben Sie eine Grabo-Klasse namens `Miez` als Erweiterung der Klasse `javax.swing.JFrame`. Ein `Miez`-Objekt, welches man mit dem Befehl `new Miez("Caroline")` erzeugt hat, soll auf dem Bildschirm als ein Fenster der Größe 300 x 100 Pixel dargestellt werden, etwa so:



Weil `JFrame` eine *Behälterklasse* ist, ist auch jedes `Miez`-Objekt ein *Behälter* (engl. a container object). Der folgende schematische Bauplan soll deutlich machen, dass ein `Miez`-Objekt drei weitere Grabo-Objekte enthält: Ein `JCheckBox`-Objekt, ein `JLabel`-Objekt und ein horizontales `Box`-Objekt:



Das `Box`-Objekt kann man sich als rechteckiges, unsichtbares Gummiband vorstellen, in das man beliebig viele Grabo-Objekte hineintun kann. Bei einem *horizontalen* `Box`-Objekt werden diese Grabo-Objekte *nebeneinander* (von links nach rechts) angeordnet. Bei einer horizontalen `Box` werden die Grabo-Objekte untereinander (von oben nach unten) angeordnet. Mehr dazu weiter unten.

Das `JCheckBox`-Objekt soll mit dem Namen "Clicked" versehen werden. Das `JLabel`-Objekt soll anfangs den Text " ----- " (2 Blanks und 10 Minuszeichen) enthalten.

**Teilaufgabe-01:** Schreiben Sie die Klasse `Miez` so, dass ihre Objekte aussehen wie oben beschrieben.

Gehen Sie dabei wie folgt vor:

Vereinbaren Sie in der Klasse `Miez` drei *Objektattribute ohne Initialisierung*, und zwar:

eines vom Typ `JCheckBox`    namens `knopf01`  
 eines vom Typ `JLabel`        namens `etikett01`  
 eines vom Typ `Box`            namens `kasten01`

Schreiben Sie dann einen *Konstruktor* mit einem `String`-Parameter namens `titel`. Rufen Sie in diesem Konstruktor einen Konstruktor der direkten Oberklasse auf und übergeben Sie ihm den `titel` (etwa so: `super(titel);`).

Weisen Sie dann (im Konstruktor) den Attributen `knopf01`, `etikett01` und `kasten01` Referenzen auf entsprechende Objekte zu (natürlich mit Hilfe des `new`-Befehls). Ein *horizontales* `Box`-Objekt können Sie auf eine der folgenden beiden Weisen erzeugen:

```
kasten01 = new Box(BoxLayout.X_AXIS); // Möglichkeit 1: Mit new und Konstruktor
kasten01 = Box.createHorizontalBox(); // Möglichkeit 2: Mit einer Fabrik-Methode
```

Viele Programmierer finden die zweite Möglichkeit einfacher und bevorzugen sie.

Tun Sie die Objekte `knopf01` und `etikett01` mit einer Methode `add` in den `kasten01`.

Tun Sie das den `kasten01` mit einer `add`-Methode in das `this`-Objekt (d.h. in das `Miez`-Objekt, das der Konstruktor gerade initialisiert).

Initialisieren Sie dann (immer noch im Konstruktor) das `this`-Objekt mit folgenden Befehlen:

```
this.setDefaultCloseOperation(EXIT_ON_CLOSE);
this.setBounds(40, 20, 300, 100);
this.setVisible(true);
```

Lesen Sie in der Online-Dokumentation der Klasse `JFrame` nach, was diese Befehle bewirken.

Nach dem Konstruktor sollten Sie noch eine `main`-Methode vereinbaren, in der ein `Miez`-Objekt erzeugt wird.

Durch die `main`-Methode wird ihre Klasse `Miez` zu einem Programm `Miez`. Lassen Sie dieses Programm ausführen und *prüfen Sie*, ob das (in der `main`-Methode erzeugte) `Miez`-Objekt richtig aussieht.

### Teilaufgabe-02: Zwei Behandler-Objekte bei zwei Grabo-Objekten anmelden

Vereinbaren Sie in der Klasse `Miez` (als Objektelement, nicht als Klasselement) eine `ActionListener`-Klasse namens `KnopfClickedKlasse`. Die Methode `actionPerformed` in dieser Klasse soll (erstmal nur) ihren Parameter (vom Typ `ActionEvent`) zur Standardausgabe ausgeben.

Vereinbaren Sie in der Klasse `Miez` ein `KnopfClickedKlasse`-Behandler-Objekt namens `knopfClickedObjekt`.

Melden Sie im Konstruktor der Klasse `Miez` das Behandler-Objekt `knopfClickedObjekt` beim Grabo-Objekt `knopf01` an (mit der Methode `addActionListener`).

*Probieren Sie aus*, ob das Behandler-Objekt `knopfClickedObjekt` richtig funktioniert (d.h. ob es im richtigen Moment die richtige Meldung zur Standardausgabe ausgibt).

Vereinbaren Sie in der Klasse `Miez` (als Objektelement, nicht als Klasselement) eine `MouseListener`-Klasse namens `FensterClickedKlasse`. Die Methode `mouseClicked` in dieser Klasse soll (erstmal nur) ihren Parameter (vom Typ `MouseEvent`) zur Standardausgabe ausgeben.

Die anderen Methoden der Schnittstelle `MouseListener` brauchen nichts zu machen.

Vereinbaren Sie in der Klasse `Miez` ein `FensterClickedKlasse`-Behandler-Objekt namens `fensterClickedObjekt`.

Melden Sie im Konstruktor der Klasse `Miez` das Behandler-Objekt `fensterClickedObjekt` beim Grabo-Objekt `this` an (mit der Methode `addMouseListener`).

*Probieren Sie aus*, ob das Behandler-Objekt `fensterClickedObjekt` richtig funktioniert (d.h. ob es im richtigen Moment die richtige Meldung zur Standardausgabe ausgibt).



**Teilaufgabe-03: Die mouseClicked-Methode aufbrezeln**

Bisher gibt die mouseClicked-Methode (in der Klasse FensterClickedKlasse) nur eine Meldung zur Standardausgabe aus.

Erweitern Sie die Methode jetzt so, dass sie aus ihrem Parameter (vom Typ MouseEvent) die x- und die y-Koordinate des betreffenden Maus-Klicks holt und in lesbarer Form in das JLabel-Objekt etikett01 schreibt (mit der Methode etikett01.setText(...)).

**Probieren Sie aus**, ob die erweiterte mouseClicked-Methode richtig funktioniert.

Machen Sie anschließend den Befehl `this.addMouseListener(...)`; im Konstruktor der Klasse Miez zu einem **Kommentar**.

**Teilaufgabe-04: Die actionPerformed-Methode aufbrezeln**

Bisher gibt die actionPerformed-Methode (in der Klasse KnopfClickedKlasse) nur eine Meldung zur Standardausgabe aus.

Erweitern Sie die Methode jetzt so, dass sie Folgendes macht:

Wenn der knopf01 **aktiviert** ist

(d.h. wenn `knopf01.isSelected()` den Wert `true` liefert), dann wird das Behandler-Objekt `fensterClickedObjekt` beim aktuellen Miez-Objekt **angemeldet**, und zwar mit dem (etwas trickreichen) Befehl:

```
Miez.this.addMouseListener(fensterClickedObjekt);
```

Wenn der knopf01 dagegen **deaktiviert** ist

(d.h. wenn `knopf01.isSelected()` den Wert `false` liefert), dann wird das Behandler-Objekt `fensterClickedObjekt` beim aktuellen Miez-Objekt **abgemeldet**, und zwar mit dem (etwas trickreichen) Befehl:

```
Miez.this.removeMouseListener(fensterClickedObjekt);
```

**Probieren Sie aus**, ob der Benutzer das "Beobachten von Ereignissen der Art mouseClicked" jetzt wirklich an- und ausschalten kann, indem er das JCheckBox-Objekt knopf01 mit der Maus aktiviert bzw. deaktiviert.

**Zusatz 1:** Ganz entsprechend wie mouseClicked-Ereignisse sollen auch mouseDragged-Ereignisse beobachtet werden. Das An- und Abschalten dieser Beobachtungen soll mit einem Knopf mit der Beschriftung mouseDragged erfolgen.

**Zusatz 2:** Ganz entsprechend sollen auch mouseWheelMoved-Ereignisse (an- und abschaltbar) beobachtet werden.

**Zusatz 3 (Katze):** Ganz entsprechend sollen Mausereignisse **aller acht Arten** (mouseClicked, mouseEntered, mouseExited, mousePressed, mouseReleased, mouseDragged, mouseMoved und mouseWheelMoved (einzeln an- und abschaltbar) beobachtet werden. Alle Größen, die acht Mal vorkommen (für jede Art von Ereignis einmal) sollten in Reihungen (engl. arrays) organisiert werden.

### Aufgabe 3: 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.

Ein *Zufallsweg* beginnt an einem bestimmten Ort (z.B. bei einem bestimmten Poxel eines PoxelPanel's) 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. Der Zufallsweg beginnt bei einem Poxel mit zufällig gewählten Koordinaten  $x$  und  $y$ , und jeder Schritt führt zu einem zufällig gewählten Poxel in der Umgebung des vorigen Poxels.

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  $mDY$  kleine Werte wie z.B. 2 oder 3 etc.

Ihr Programm `RandomWalk01` soll 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"):

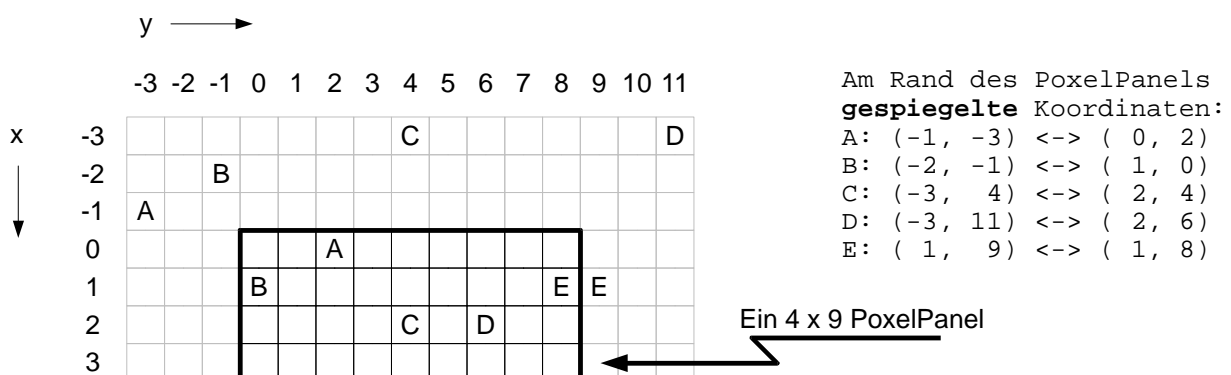
Wir machen unser PoxelPanel *unbegrenzt*, indem wir uns den linken und den rechten Rand (und ebenso den oberen und den unteren Rand) *zusammengeklebt* denken (nur denken! Bitte auf keinen Fall Klebstoff auf die Bildschirme der Laborrechner auftragen!). Dann gilt:

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 spiegeln"):

Falls ein Schritt über den Rand hinausführen würde, wird er am Rand *gespiegelt*. Im folgenden Diagramm wurden große Buchstaben A, B, ... , E am Rand eines 4 x 9 PoxelPanel's gespiegelt:



**Rand-Strategie-C ("Schritte wählen bis einer paßt"):**

Falls ein Zufallsschritt aus dem PoxelPanel hinausführen würde, würde er verworfen und ein neuer Schritt gewählt. Das wird wiederholt, bis ein akzeptabler Schritt gefunden wurde.

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 Klasse `Thread`). Was passiert, wenn Sie ihr Programm 0 Millisekunden schlafen lassen (im Vergleich zur Version *ohne* `sleep`-Befehl)?

**Maßnahme 2, Reihenfolge erkennbar machen:**

Geben Sie zu den einzelnen Poxeln die Buchstaben 'A', 'B', 'C', ... 'Z', 'A', 'B', 'C', ... 'Z', 'A', 'B' ... aus. Benutzer, die mit der Reihenfolge dieser Zeichen vertraut sind, können daran die Reihenfolge der Ausgaben (d.h. die Reihenfolge der Schritte des Zufallsweges) erkennen.

**Maßnahme 3, Gruppen von Schritten erkennbar machen:**

Wechseln Sie die Hintergrundfarbe 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.).

**Tipp:** Im Archiv `DateienFuerPR2.zip` finden Sie folgende Dateien:

`PoxelPanel.java` (ist vollständig).

`RandomWalk01.java` (muss noch ergänzt werden)

`RandomWalk02.java` (muss noch ergänzt werden)

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

**Aufgabe 4: 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 folgen hier (leicht gekürzt) die vorgegebenen Teile dieser Aufgabe:

```

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    // Tries 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 } // interface 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 k11, KeyListener k12) {
9         // Registers (using the appropriate addXXXListener method)
10        // al with the submit button,
11        // k11 with with the first input text field and
12        // k12 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 length 5.
20 // Assigns each of its elements to one of the result text fields.
21 } // interface 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 submit button:
10        ActionListener mal = new MyActionListener();
11
12        // 2 listeners for the input text fields
13        KeyListener mk11 = new MyKeyListener();
14        KeyListener mk12 = new MyKeyListener();
15
16        // Register the 3 listeners with the view:
17        aView.register(mal, mk11, mk12);
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 submit button:
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
```

**Tipp: Im Archiv** DateienFuerPR2.zip **finden Sie folgende Dateien:**

RechnerModel\_I.java **(ist vollständig)**

RechnerView\_I.java **(ist vollständig)**

RechnerControl.java **(ist vollständig)**

## Aufgabe 5: 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“ Rumpfe 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

```

**Tipp:** Diese Datei `ReIter01.java` finden Sie auch im Archiv `DateienFuerPR2.zip`. **Testen Sie Ihre Lösung**, indem Sie geeignete Befehle in die `main`-Methode schreiben und ausführen lassen.



## Aufgabe 6: 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 Methodenrumpfe durch richtige Rumpfe 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 **Tipps** (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`: **Tipp:** Lesen Sie in der Online-Dokumentation nach (und probieren Sie aus), was die Methode `Arrays.toString` leistet.
- `contains`: **Tipp:** 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`: **Tipp:** 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`: **Tipp:** 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) **Tipp:** 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) **Tipp:** 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 `TipsZuEclipse` erläutert.

**Tipp:** Im Archiv `DateienFuerPR2.zip` finden Sie die Dateien

`MeineSammlung.java` (muss noch ergänzt werden)

`MeineSammlungJut.java` (ist vollständig, einige Befehle müssen entkommentarisiert werden)

### Aufgabe 7: 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 drei 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. Im Archiv `DateienFuerPR2.zip` finden Sie die Datei `HashSetJut01.java` (die noch ergänzt werden muss).

### Aufgabe 8: 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 unter 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 <b>Collection</b>		class <b>AbstractCollection</b>		class <b>HashSet</b>	
Die Methode → wirft die ↓ Ausnahmen	add	addAll	add	addAll	add	addAll
<b>UnsupportedOperationException</b>						
<b>ClassCastException</b>						
<b>NullPointerException</b>						
<b>IllegalArgumentException</b>						
<b>IllegalStateException</b>						

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 höchstens 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 9: 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 der 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         |Floehes   |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 10: 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-Dokument-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`) *eingelassen* 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 Modulen `XmlJStd` und `XmlJdom` aufrufen. Die Quelldateien dieser Module enthalten ziemlich umfangreiche Kommentare und Erläuterungen.

#### 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.

**Tipp:** Im Archiv `DateienFuerPR2.zip` finden Sie die Dateien

<code>stundenplan.dtd</code>	(ist vollständig)
<code>CreateDom10.java</code>	(ist vollständig)
<code>CreateJdom19.java</code>	(ist vollständig)
<code>XML\XmlJStd.java</code>	(ist vollständig)
<code>XML\XmlJdom.java</code>	(ist vollständig)
<code>XML\XmlXerces.java</code>	(ist vollständig)



**Aufgabe 11: 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_il = 1;
13     protected int protec_oa_il = 2;
14     int paketw_oa_il = 4;
15     private int privat_oa_il = 8;
16
17     static public int public_ka_il = 100;
18     static protected int protec_ka_il = 200;
19     static int paketw_ka_il = 400;
20     static private int privat_ka_il = 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  $(1 + 2 + 4 + 8)$  gleich 15 liefern.

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