Grundlagen der Informatik 1
WS 2008/09

Prof. Mühlhäuser, Dr. Rößling, Melanie Hartmann, Daniel Schreiber
http://proffs.tk.informatik.tu-darmstadt.de/gdi1

Übung 9 - Lösungsvorschlag Version: 1.0 15. 12. 2008

---

1 Mini Quiz

Objeke und Klassen

- ☐ Java und Scheme sind objektorientierte Sprache.
- ☒ Ein Objekt kann Attribute und Methoden enthalten.
- ☒ Ein Objekt ist eine Instanz einer Klasse.
- ☐ Eine Klasse instanziiert für gewöhnlich ein Objekt.

Compiler vs. Interpreter

- ☐ Scheme und Java sind beides Maschinensprachen.
- ☒ Java nutzt eine Virtual Machine (VM).
- ☐ Java -Programme sind nur lauffähig auf dem Maschinentyp, auf dem sie kompiliert wurden.
- ☐ Byte-Code Programme sind schneller als Maschinenprogramme.
- ☒ Ein Interpreter führt ein Programm einer bestimmten Programmiersprache direkt aus.

Packages

- ☒ Packages sollen Klassen bündeln, die im Hinblick auf Zuständigkeit zusammengehören.
- ☒ Mit einer Wild Card (*) kann man auf alle Unterklassen eines Packages zugreifen.
- ☐ Eine Klasse kann mehreren Packages angehören.

2 Fragen

1. Erklären Sie in eigenen Worten folgende Begriffe: **Klasse**, **Objekt** und **Instanz**. Geben Sie ein Beispiel für jeden Begriff an.

   Lösungsvorschlag:

   Klassen *...sind in der objektorientierten Programmierung ein abstrakter Oberbegriff für die Beschreibung der gemeinsamen Struktur und des gemeinsamen Verhaltens von Objekten. Klassen können*

Attribute (Eigenschaften) und Methoden (z.B. Operationen) enthalten. Ebenso kann eine Klasse von einer anderen Klasse erben.

Objekte *...sind in der objektorientierten Programmierung ein konkretes Exemplar einer Klasse. Jedes Objekt hat eine eigene Identität und wird durch den Konstruktor initialisiert. Objekte werden auch als* **Instanzen** *einer Klasse bezeichnet.*

Beispiel: Baum steht für eine Klasse, die konkrete Eiche vor dem Fenster hingegen ist ein Objekt oder eine Instanz der Baumklasse.

2. Was ist ein Konstruktor? Wofür wird dieser benötigt? Wie sieht ein minimaler Konstruktor aus?

Lösungsvorschlag:

Ein Konstruktor dient zur Erzeugung eines Objektes einer Klasse. Der Konstruktor hat immer den gleichen Namen wie die Klasse, zu der er gehört. Der Konstruktor initialisiert Attribute des zu erstellenden Objektes oder führt Methoden aus, die in ihm angegeben sind, um das Objekt zu initialisieren.

```
1 public Klassenname()
2 {
3   // minimaler Konstruktor
4 }
```

3. Nennen Sie die wichtigsten Methoden der Assert Klasse, die man bei einem typischen JUnit Testcase benötigt.

Lösungsvorschlag:

- *assertTrue(boolean)*
- *assertFalse(boolean)*
- *assertSame(Object, Object)*
- *assertNotSame(Object, Object)*
- *assertEquals(primitive Type, primitive Type) beide Parameter müssen vom gleichen Java Datentyp (z.B. int, char, float, etc...) sein*

4. Nennen und beschreiben Sie die vier Phasen der Übersetzung (Kompilierung) eines Programms.

Lösungsvorschlag:

- **Lexikalische Analyse***: Zerlegung des Quellprogramms in eine Folge von Worten.*
- **Syntaktische Analyse***: Testet, ob das Quellprogramm den Syntaxregeln der Quellsprache entspricht. Wortfolgen (Tokens) werden in gültige Sätze zusammengefasst.*
- **Semantische Analyse***: Test, ob alle benutzten Namen deklariert wurden und ihrem Typ entsprechend verwendet werden etc.*
- **Code-Generierung***: Erzeugung des Zielprogramms.*

3 Modellierung und Test eines virtuellen Autos

1. Schreiben Sie eine Klasse `Car` zur Repräsentation von Autos, die folgende Anforderungen erfüllen soll:

   - Ein Auto hat einen Namen vom Typ *String* und einen Kilometerstand (*mileage*) vom Typ *double*. Beide Attribute sollten *private* sein.
   - Der Konstruktor soll eine Zeichenkette als Parameter erhalten, die den Namen des Autos angibt. Der Konstruktor soll den Namen des Autos setzen und den Kilometerstand auf 0.0 setzen.
   - Schreiben Sie die Getter-Methoden **public double** getMileage() und **public** String getName(), um auf die Attribute der Klasse `Car` zuzugreifen.
   - Schreiben Sie die Methode **void** drive(**double** distance), die eine Distanz in Kilometern als Argument erhält und den Kilometerstand entsprechend aktualisiert.
   - Vergessen Sie nicht die Verwendung von JavaDoc-Kommentaren für alle Elemente, die nach außen sichtbar sind!

Lösungsvorschlag:

```
1  /**
2   * The class Car represents cars with a name and a mileage.
3   *
4   * @author Oren Avni / Guido Roessling
5   * @version 1.1
6   */
7  public class Car {
8
9    // the following two attributes are private and thus
10   // need no JavaDoc comment
11   private String name;
12
13   private double mileage;
14
15   /**
16    * Creates a new car with the given name and a mileage of 0.0
17    *
18    * @param carName the name of the new car
19    */
20    public Car(String carName) {
21      name = carName;
22      mileage = 0.0;
23    }
24
25    // Public methods
26
27    /**
28    * Drives the car for the distance passed, updates the mileage
29    *
30    * @param distance the distance the car was driven
31    */
32    public void drive(double distance) {
33      mileage = mileage + distance;
34    }
```

```
35
36      /**
37      * returns the current mileage of the car
38      *
39      * @return the current mileage of this car
40      */
41      public double getMileage() {
42        return mileage;
43      }
44
45      /**
46      * returns the name of this car
47      *
48      * @return the name of this car
49      */
50      public String getName() {
51        return name;
52      }
53  }
```

2. Schreiben Sie nun einen minimalen JUnit-Testcase, der die beiden folgenden Tests abdecken soll:

   - Test des Konstruktors (der Erzeugung des Objekts) und der Methode String getName(). Dazu soll nach dem Anlegen des Objektes dessen Name einmal mit dem tatsächlichen und einem mit einem falschen Namen überprüft werden. Benutzen Sie hier bitte sowohl *assertFalse(String message, boolean condition)* als auch *assertEquals(String expected, String actual)* und übergeben Sie für *assertFalse* einen passenden String als (mögliche) Fehlermeldung.
   - Test der Methoden **void** drive(**double** distance) und **double** getMileage(). Dazu soll das Auto erst 74.3 km "fahren", der Kilometerstand überprüft werden, dann soll das Auto weitere 26.8 km "fahren" und der Kilometerstand erneut überprüft werden.

Lösungsvorschlag:

```
1   import static org.junit.Assert.assertEquals; // to compare two values
2   import static org.junit.Assert.assertFalse; // testing  wrong results
3
4   import org.junit.Before;
5   import org.junit.Test;
6
7   /**
8    * Minimal test framework for the "Car" class using JUnit 4.4+
9    *
10   * @author Oren Avni / Guido Roessling
11   * @version 1.0
12   */
13  public class CarTest {
14    /* Our car instance, created anew for each test */
15    private Car myCar;
16
17    /**
18    * Initialize the car. Do this once before each test.
19    */
```

```
20    @Before
21    public void initCar() {
22      myCar = new Car("Skoda Fabia");
23    }
24
25    /**
26    * Check if the car construction was done properly by checking
27    * the assigned name
28    */
29    @Test
30    public void validateConstruction() {
31      String msg = "Error in comparing car name";
32      assertFalse(msg, myCar.getName().equals("Maserati Quattroporte S"
          ));
33      assertEquals("Skoda Fabia", myCar.getName());
34      assertEquals(0.0, myCar.getMileage());
35    }
36
37    /**
38    * Check if the drive method behaves as expected.
39    * Done by driving a certain distance, validating the mileage,
40    * then driving another distance and re-validating the mileage.
41    */
42    @Test
43    public void validateDriving() {
44      myCar.drive(74.3);
45      assertEquals(74.3, myCar.getMileage());
46
47      myCar.drive(26.8);
48      assertEquals(101.1, myCar.getMileage());
49    }
50  }
```

4 Objekte und deren Analyse

Bitte lesen Sie zunächst die gegebene Java-Klasse und beantworten Sie anschließend die einzelnen Fragen.

```
1  public class P {
2
3    private Long method1(Long x, Long y) {
4      if (y == 1)
5        return x;
6      return x + method1(x, y - 1);
7    }
8
9    private Long method2(Long x, Long y, Long z) {
10     z = y - 1;
11     if (y == 1)
12       return x;
13     return method1(x, method2(x, y - 1, z));
14   }
15 }
```

- Wieviele primitive Datentypen werden in der Klasse `P` verwendet?

Lösungsvorschlag:
Überhaupt keine! Sämtliche Variablen werden durch sogenannte Wrapper-Klassen deklariert. Kleine Eselsbrücke: primitive Datentypen beginnen in Java mit einem Kleinbuchstaben - z. B. int, char, byte, long und float. Bitte beachten Sie: String ist eine Klasse und kein *primitiver Datentyp.)*

Bis auf die primitiven Datentypen sind alle anderen Datentypen in Java ausschließlich Objekte!

- Was würde folgender Aufruf innerhalb der (in der Klasse P nicht angegebenen) *main-Methode* als Ergebnis liefern?

```
1  /**
2   * Run the program using method2...
3   * @param args command-line arguments (ignored).
4   */
5  public static void main(String[] args) {
6     P p = new P();
7
8     Long a = new Long(2),
9          b = new Long(5),
10         c = new Long(a - b);
11
12    System.out.print("Result: " + p.method2(a, b ,c));
13 }
```

Lösungsvorschlag:
Als Ergebnis erhalten wir hier: $2^5 = 32$. *Es wird also "Result: 32" ausgegeben.*

- Berechnen Sie auch das Ergebnis für a=4, b=3 und c=2.

Lösungsvorschlag:
Das Ergebnis ist $4^3 = 64$.

- Was für einen Zweck erfüllt der Algorithmus, der sich aus method1(..) und method2(..) zusammensetzt? Beachten Sie die verschachtelte Rekursion!

  Hinweis: die statische Klasse `Math` die Sie kennen sollten, enthält diesen Algorithmus (wenn auch in einer leicht abgewandelter Form).

Lösungsvorschlag:
Es handelt sich bei diesen Algorithmus um `Math.pow()` *in einer abgewandelten Form, d.h. es wird* x^y *berechnet. Beachten Sie, dass* nirgendwo *eine Multiplikation auftaucht, diese wurde hier auf mehrfache Addition zurückgeführt. Analog funktioniert dies ebenfalls für die Division (hierbei jedoch durch Zurückführung auf mehrfache Subtraktion).*

- In der oberen Klasse gibt es eine überflüssige Variable. Ermitteln Sie diese und erklären Sie, warum diese nicht benötigt wird.

Lösungsvorschlag:
Die überflüssige Variable lautet z. *Auch wenn sich der Wert von* z *in* method2 *ändert, hat dies keinen Einfluss auf die Berechnung. In dem rekursiven Aufruf method2(x, y - 1, z) wird* z *nur mitgeführt, aber die eigentlichen Variablen* x *und* y *hängen in keiner Weise von* z *ab.*

5 Mehr JUnit...

In dieser Aufgabe wollen wir ein paar fortgeschrittene Techniken zum Testen mit JUnit betrachen.

1. Gegeben sei folgende Klasse *Random* mit (bewusst) unvollständiger Kommentierung:

```
1  /**
2   * Random class for GdI / ICS exercise sheet 9.
3   *
4   * @author Oren Avni / Guido Roessling
5   * @version 1.0
6   */
7  public class Random {
8
9    /**
10    *
11    *
12    * @param m
13    * @param s
14    * @return
15    */
16   public int[] doSomething(int m, int s) {
17     int i = 0;
18     int[] arr = new int[s];
19
20     while (i < arr.length) {
21       arr[i] = (int) (m * (Math.random())) + 1;
22       i++;
23     }
24
25     return arr;
26   }
27 }
```

   - Erklären Sie zunächst, was für einen Zweck die Methode *doSomething* verfolgt.

     Hinweis: `Math.random()` liefert einen zufälligen Wert $x \in [0, 1[$ zurück. (**int**)value wandelt eine Gleitkommazahl in die entsprechende Ganzzahl; (**int**)5.722 ist also die Ganzzahl 5.

     Lösungsvorschlag:
     Die Methode doSomething(m, s) *befüllt ein Array der Länge* s *mit zufälligen ganzen Zahlen, die jeweils im Intervall* $[1, m+1[= [1, m]$ *(da wir es nur mit Ganzzahlen zu tun haben) liegen.*

   - Schreiben Sie eine JUnit-Testklasse, die ein privates Attribut vom Typ *Random* anlegt und anschließend für die Ergebnisse des Aufrufes *doSomething(50, 100)* prüft, ob die einzelen Werte des Arrays die gewünschten Eigenschaften haben. Verwenden Sie für den Test *assertTrue(String messageOnFail, boolean condition)*.

   - Schreiben Sie nun einen JUnit-Testcase, der genau wie der vorherige Test arbeitet, nur für den Aufruf *doSomething(100, 2000000)*. **Bitte beachten Sie:** Der Test soll scheitern, falls die Ausführung länger als 1000 Millisekunden dauert!

     Lösungsvorschlag:

```java
import static org.junit.Assert.assertTrue;
import org.junit.Before;
import org.junit.Test;

/**
 * Test the class Random
 *
 * @author Oren Avni / Guido Roessling
 * @version 1.0
 */
public class RandomTest {

  /* the Random instance */
  private Random random ;

  /**
  * Initializes the Random instance
  */
  @Before
  public void init() {
    random = new Random();
  }

  /**
  * Test whether all values generated fall in the correct
    interval
  */
  @Test
  public void ensureCorrectValueRange() {
    int[] result = random.doSomething(50, 100);
    for (int i = 0; i < 100; i++)
      assertTrue("Value outside [1, 50]: " + result[i],
            (result[i] >= 1 && result[i] < 51));
  }

  /**
  * Test the method "doSomething" in RandomTest
  * This test will time out after 1000ms = one second.
  */
  @Test(timeout = 1000)
  public void testGiantArray() {
    // create 2.000.000 random int values in [1, 101]
    int[] result = random.doSomething(100, 2000000);
    for (int i = 0; i < 100; i++)
      assertTrue("Value outside [1, 100]",
          (result[i] >= 1 && result[i] < 101));
  }
}
```

2. Wir betrachten nun die folgende Klasse *Strange* sowie den entsprechenden JUnit-Testcase:

```java
public class Strange {
  public int getAvg(int[] array) {
    int temp = 0;

    for (int i = 0; i <= array.length; i++) {
      temp += array[i];
```

```
7       }
8
9       return (temp / array.length);
10    }
11  }
```

```
1  import static org.junit.Assert.assertEquals;
2  import org.junit.Test;
3
4  public class StrangeTest {
5    @Test
6    public void testStrange() {
7      // create Strange instance
8      Strange strange = new Strange();
9      // check value
10     assertEquals(5, strange.getAvg(new int[] { 1, 3, 5, 7, 9 }));
11   }
12 }
```

- Was passiert, wenn Sie den Test ausführen? Was müssen Sie an der Klasse *Strange* ändern, damit der Testcase durchläuft und keine Fehlermeldungen ausgibt?

  Hinweis: Es reicht eine einzige Änderung.

  Lösungsvorschlag:
  Die einzige Änderung, die man benötigt, ist innerhalb der for-Schleife. Hier ist $<=$ *durch* $<$ *zu ersetzen. Dann tritt auch keine* java.lang.ArrayIndexOutOfBoundsException *mehr auf, die zu einem Error führte.*

- Die Methode *getAvg(int[] array)* besitzt noch einen *Intentionsfehler*. Finden Sie ihn? Wieso läuft der Testcase trotzdem fehlerfrei, nachdem Sie die erste Teilaufgabe gemeistert haben?

  Lösungsvorschlag:
  Die Methode getAvg(int[] array)*, die den Durchschnitt aller Elemente des Arrays berechnet, liefert eine Ganzzahl zurück, was für die meisten Durchschnittswerte nicht ausreicht. Da aber* $\frac{1+3+5+7+9}{5} = \frac{25}{5} = 5$ *lautet, trat der Fehler im Test nicht auf. Generell sollte die Methode so geändert werden, dass das Ergebnis als* double *zurückgegeben wird.*

Hausübung

Die Vorlagen für die Bearbeitung werden im GdI1-Portal bereitgestellt. Kommentieren Sie Ihren selbst erstellten Code. Die Hausübung muss bis zum Abgabedatum im GdI1-Portal abgegeben werden. Der Fachbereich Informatik misst der Einhaltung der Grundregeln der wissenschaftlichen Ethik großen Wert bei. Zu diesen gehört auch die strikte Verfolgung von Plagiarismus. Mit der Abgabe Ihrer Hausübung bestätigen Sie, dass Sie bzw. Ihre Gruppe alleiniger Autor des gesamten Materials sind. Falls Ihnen die Verwendung von Fremdmaterial gestattet war, so müssen Sie dessen Quellen deutlich zitieren.

Abgabe: Bis spätestens Montag, 12. 01. 2009, 16:00 Uhr

6 Implementierung einer Mini-Datenbank (13 Punkte)

Hinweis: Die nächste Hausübung (Thema: Vererbung) baut auf dieser Aufgabe auf, daher ist es ratsam, diese Aufgabe sehr sorgsam zu implementieren. Wir werden aber auch rechtzeitig eine Bei-

spielimplementierung für die nächste Übung veröffentlichen.

In dieser Aufgabe widmen wir uns dem Thema Datenbanken. Sie sollen eine Mini-Datenbank für Adressinformationen implementieren, die die wichtigsten Operationen einer normalen Datenbank unterstützt: Einfügen, Löschen und Selektieren. Einen Rahmencode für diese Aufgabe finden Sie im GdI1-Portal.

Hinweise:

- *Vergessen Sie nicht die JavaDoc-Kommentierung (***1 Punkt***)!*
- Für die Zusammenstellung der Ausgaben für *toString()* in den Teilaufgaben sollten Sie aus Performanzgründen die Klasse *java.lang.StringBuffer* nutzen. Sehen Sie dazu in der Java-Dokumentation nach (s. GdI-Portal).
- Fügen Sie geeignete Getter/Setter Methoden zu allen drei Klassen hinzu, um die Attribute lesen bzw. ändern zu können. Da dies in Eclipse automatisch möglich ist (siehe Übung 8, Aufgabe 6.6), erhalten Sie für diese Methoden keine Extrapunkte.

6.1 Datenbankeinträge (6 Punkte)

Wir benötigen die folgenden Klassen: `Address`, `PhoneNumber` und `Entry`. Diese sollen Sie nach und nach implementieren.

Anschriften: Klasse Address (2 Punkte)

Erstellen zunächst die Klasse `Address` für Anschriften. Der Konstruktor dieser Klasse erwartet die folgenden vier Parameter vom Typ *String*:

- String myStreet für den Straßennamen,
- String myNumber für die Hausnummer (String, damit auch Angaben wie "8a" möglich sind),
- String myZipCode für die Postleitzahl, sowie
- String myCity für die Stadt.

Überschreiben Sie in dieser Klasse die Methode *toString()* so, dass sie die folgende Ausgabe produziert. Achten Sie auf die genaue Formatierung, auch die Klammern {}, Kommas und Leerzeichen:

{myStreet myNumber, myZipCode myCity}

Dabei sind die Einträge durch den jeweiligen Wert zu ersetzen, zum Beispiel für die Anschrift der Fachbereichs Informatik:

{Hochschulstr. 10, 64289 Darmstadt}

Telefonnummern: Klasse PhoneNumber (1 Punkt)

Erstellen Sie nun die Klasse `PhoneNumber`, deren Konstruktor die folgenden beiden Parameter bekommt:

- String myAreaCode für die Vorwahl sowie
- String myNumber für die Durchwahl.

Die Klasse *PhoneNumber* soll ebenfalls die String toString() Methode überschreiben. Die Ausgabe soll hier das Folgende zurückgeben:

Vorwahl−Nummer

Die Telefonnummer der TUD (Vorwahl Darmstadt, Durchwahl 160) ist damit auszugeben als

06151−160

Datensatzmodellierung: Klasse Entry (2 Punkte)

Erstellen Sie nun die Klasse `Entry`, die einen zu einer Person gehörigen Datensatz der Datenbank repräsentiert. Der Konstruktor soll die folgenden fünf Parameter erhalten und diese entsprechenden privaten Attributen zuweisen:

- String myGivenName für den Vornamen,
- String myFamilyName für den Nachnamen,
- PhoneNumber myPhoneNumber für die Telefonnummer,
- Address myAddress für die Adresse sowie
- String myJob für die Berufsbezeichnung.

Auch hier soll die Klasse die Methode: String toString() überschreiben. Das Resultat dieser Methode soll folgendes Format haben:

Vorname Name Telefonnummer Adresse Beruf

Dabei stehen die Freiräume jeweils für ein Tabulator-Leerzeichen, das in Java mit `"\t"` realisiert werden kann. Verwenden Sie die `toString()` Methoden des `Address` und `PhoneNumber` Objekts. Die Ausgabe kann also zum Beispiel wie folgt aussehen:

Markus Meier 06151−164589 {Hochschulstr. 10, 64289 Darmstadt} Student

6.2 Datenbank (7 Punkte)

Realisieren Sie nun die eigentliche Datenbank als neue Klasse `Database`, die über ein Attribute **private** Entry[] entries verfügt. Diese Klasse soll zwei unterschiedliche Konstruktoren zur Verfügung stellen:

- Einen parameterlosen Konstruktor, der das Array mit *Länge 0* anlegt.

- Einen Konstrukor, der ein Array von `Entry`-Objekten als Parameter erhält. Diese Werte sind in dem internen Attribut der Datenbank abzuspeichern.

Überprüfen Sie die Funktionalität Ihres Codes mit Hilfe der Testcases, die im Rahmencode enthalten sind. Der Testcase *muss* korrekt durchlaufen!

Implementieren Sie nun die folgenden Methoden. Die Methoden mit Ergebnistyp *boolean* sollen genau dann *true* liefern, wenn die Operation erfolgreich war. Andernfalls, etwa bei einem Zugriffsversuch auf eine ungültige Position oder nicht vorhandene Elemente, soll *false* zurückgegeben werden.

Hinweis

- Es ist ausdrücklich nicht gestattet, für diese Aufgabe *Collections* zu verwenden, also vorgefertigte Datenstrukturen wie *LinkedList, ArrayList, Vector, Stack* oder *HashSet*. Diese werden erst in Foliensatz T15 eingeführt. Nur Arrays sind als Datenstruktur erlaubt.
- **Hinweis zu den die Datenbank ändernden Methoden, die mit * markiert sind**: Die Daten der Datenbank liegen in dem privaten Attribute *entries*, das als Array nicht in der Länge verändert werden kann. Wenn eine Längenänderung fällig wird, müssen Sie stattdessen ein neues Array der "passenden" Länge anlegen und die Werte entsprechend kopieren. Verwenden Sie zum Kopieren die Methode

  System.arraycopy(Object[] src, **int** srcPos, Object[] dest, **int** destPos, **int** length)

  Diese kopiert *length* Werte ab Position *srcPos* von Array *src* in das Array *dest*, dort beginnend mit der Zielposition *destPos*. Beachten Sie, dass das Zielarray *dest* existieren und eine ausreichende Mindestlänge haben muss.

boolean addEntry(Entry e) * Hinzufügen eines Datensatzes. Wie in echten Datenbanken soll hier Redundanz vermieden werden: falls der Datensatz schon in der Datenbank existiert, soll er *nicht* erneut eingefügt werden. Die Methode liefert daher *false*, wenn der Parameter *null* ist oder schon existiert.

boolean entryExists(Entry e) prüft, ob der Datensatz *e* bereits in der Datenbank existiert. Hinweis: vergleichen Sie nicht die Objekte, sondern deren String-Repräsentierung mittels *toString()* miteinander!

boolean dropDatabase() entfernt alle Elemente aus der aktuellen Datenbank. Die Methode liefert *true*, wenn die Datenbank erfolgreich gelöscht (auf Länge 0 gebracht) wurde.

boolean deleteEntry(Entry e) entfernt den Datensatz *e* aus der Datenbank, falls er existiert. Als Ergebnis liefert die Methode *true*, falls es den Datensatz *e* gab, sonst *false*. Falls ein Datensatz gelöscht wurde, soll die Methode **void** resize(**int**) aufgerufen werden, die das Array wieder anpasst, damit keine "Lücken" (im Sinn von `null`) im Array existieren.

void resize(int pos) * passt die Datenbank nach dem Löschen des Element an Position *pos* an.

int getPos(Entry e) liefert die Position des gesuchten Datensatzes *e* in der Datenbank zurück, falls dieser existiert, sonst -1.

int getSize() gibt die Anzahl der Einträge in der Datenbank zurück.

protected void swap(int pos1, int pos2) vertauscht die Element an Position pos1 und pos2, falls beide Positionen gültig sind. *Diese Methode benötigen wir erst in Übung 10.*

String toString() Gibt eine formatierte Ausgabe der Datenbank aus. Dabei soll jeder Datensatz in einer eigenen Zeile stehen (ohne Leerzeilen). Angenommen, die Datenbank enthält drei Datensätze, dann sollte die Ausgabe wie folgt aussehen:

```
1 Hans   Meiser 0180-1234567 {Rossmarkt 26, 60311 Köln}     Tänzer
2 George Bevin  06031-2222222 {Wallstr. 7, 99551 Wien}     Bäcker
3 Karl   Heinz  030-1111111  {Hauptstr. 42, 04504 Berlin} Pilot
```

Befinden sich keine Einträge in der Datenbank oder wurde diese nicht instanziiert, so soll **exakt** die folgende Fehlermeldung zurückgegeben werden inklusive der Zeichen <>:

<Database is empty>

Hinweis: Wir testen Ihre Lösung anhand dieser Fehlermeldung, achten Sie daher bitte auf die exakt korrekte Schreibweise.

Lösungsvorschlag:

```
1  /**
2   * class: Address (represents an address object)
3   *
4   * @author Oren Avni, (Tutor for GDI 1 / ICS 1 @ Winter term: 2008-2009
5   *         {TU-Darmstadt}
6   * @author Guido Roessling
7   * @category Java - Basic Object Operations
8   * @version 1.1
9   */
10 public class Address {
11   /* represents the street name */
12   private String streetName;
13
14   /* represents the house number*/
15   private String houseNumber;
16
17   /* represents the ZIP code (in German: PLZ) */
18   private String zipCode;
19
20   /* represents the city */
21   private String city;
22
23   /**
24    * creates a new Address based on the values passed in
25    *
26    * @param myStreet the street name
27    * @param myNumber the house number (String to also
28    * allow numbers such as "8a")
29    * @param myZipCode the ZIP code
30    * @param myCity the city
31    */
32   public Address(String myStreet, String myNumber,
33       String myZipCode, String myCity) {
34     streetName = myStreet;
35     houseNumber = myNumber;
36     zipCode = myZipCode;
37     city = myCity;
```

```
38    }
39
40    /**
41     * sets the street name for this address
42     *
43     * @param newStreetName the new street name for this address
44     */
45    public void setStreetName(String newStreetName) {
46      streetName = newStreetName;
47    }
48
49    /**
50     * sets the house number for this address
51     *
52     * @param newHouseNumber the new house number for this address
53     */
54    public void setHouseNumber(String newHouseNumber) {
55      houseNumber = newHouseNumber;
56    }
57
58    /**
59     * sets the ZIP code for this address
60     *
61     * @param newZipCode the new ZIP code for this address
62     */
63    public void setZipCode(String newZipCode) {
64      zipCode = newZipCode;
65    }
66
67    /**
68     * sets the city for this address
69     *
70     * @param newCity the new city for this address
71     */
72    public void setCity(String newCity) {
73      city = newCity;
74    }
75
76    /**
77     * Creates a String representation of this address
78     *
79     * @return a String representing this address
80     */
81    public String toString() {
82      // for performance reasons, we use a StringBuffer
83      StringBuffer sb = new StringBuffer(256);
84      sb.append('{').append(streetName).append(' ');
85      sb.append(houseNumber).append(", ");
86      sb.append(zipCode).append(' ').append(city);
87      sb.append('}');
88      return sb.toString();
89    }
90
91    /**
92     * returns the street name of this address
93     *
94     * @return the street name of this address
95     */
```

```
96    public String getStreetName() {
97      return streetName;
98    }
99
100   /**
101    * returns the house number of this address
102    *
103    * @return the house number of this address
104    */
105   public String getHouseNumber() {
106     return houseNumber;
107   }
108
109   /**
110    * returns the ZIP code of this address
111    *
112    * @return the ZIP code of this address
113    */
114   public String getZipCode() {
115     return zipCode;
116   }
117
118   /**
119    * returns the city of this address
120    *
121    * @return the city of this address
122    */
123   public String getCity() {
124     return city;
125   }
126 }
```

```
1  /**
2   * class: PhoneNumber (represents a phone number database entry)
3   *
4   * @author Oren Avni, (Tutor for GDI 1 / ICS 1 @ Winter term: 2008-2009
5   *         {TU-Darmstadt}
6   * @author Guido Roessling
7   * @category Java - Basic Object Operations
8   * @version 1.0, 1.1
9   */
10 public class PhoneNumber {
11   /* The phone code for the area */
12   private String areaCode;
13
14   /* the actual phone extension */
15   private String number;
16
17   /**
18    * Creates a new phone number using the area code and extension
19    *
20    * @param myAreaCode the area code
21    * @param myNumber the actual phone number
22    */
23   public PhoneNumber(String myAreaCode, String myNumber) {
24     areaCode = myAreaCode;
25     number = myNumber;
26   }
```

```
27
28    /**
29     * Creates a String representation of this phone number
30     *
31     * @return a String representing this phone number
32     */
33    public String toString() {
34      return this.areaCode + "-" + this.number;
35    }
36
37
38     /**
39      * returns the area code
40      *
41      * @return the area code
42      */
43    public String getAreaCode() {
44      return areaCode;
45    }
46
47    /**
48     * updates the area code to newAreaCode
49     *
50     * @param newAreaCode the new area code
51     */
52    public void setAreaCode(String newAreaCode) {
53      areaCode = newAreaCode;
54    }
55
56    /**
57     * returns the number
58     *
59     * @return the number
60     */
61    public String getNumber() {
62      return number;
63    }
64
65    /**
66     * updates the number to newNumber
67     *
68     * @param newNumber the new number
69     */
70    public void setNumber(String newNumber) {
71      number = newNumber;
72    }
73  }
```

```
1   /**
2    * class: Entry (represents a database-entry)
3    *
4    * @author Oren Avni, (Tutor for GDI 1 / ICS 1 @ Winter term: 2008-2009
5    *         {TU-Darmstadt}
6    * @author Guido Roessling
7    * @category Java - Basic Object Operations
8    * @version 1.0
9    */
10  public class Entry {
```

```
11  /* the given name for this database entry */
12  private String givenName;
13
14  /* the family name for this database entry */
15  private String familyName;
16
17  /* the job title for this database entry */
18  private String job;
19
20  /* the phone number for this database entry */
21  private PhoneNumber phoneNumber;
22
23  /* the address for the database entry */
24  private Address address;
25
26  /**
27   * Creates a new database entry based on the data passed in
28   *
29   * @param myGivenName the given name for this entry
30   * @param myFamilyName the family name for this entry
31   * @param myPhoneNumber the phone number object for this entry
32   * @param myAddress the address object for this entry
33   * @param myJob the occupation for this entry
34   */
35  public Entry(String myGivenName, String myFamilyName,
36      PhoneNumber myPhoneNumber, Address myAddress,
37      String myJob) {
38    givenName = myGivenName;
39    familyName = myFamilyName;
40    phoneNumber = myPhoneNumber;
41    address = myAddress;
42    job = myJob;
43  }
44
45  /**
46   * returns a String representation of this entry
47   *
48   * @return a String representing this entry
49   */
50  public String toString() {
51    String tab = "\t"; // our separator
52    StringBuffer result = new StringBuffer(256);
53    // append all values, separated by "tab"
54    result.append(givenName).append(tab).append(familyName);
55    result.append(tab).append(phoneNumber).append(tab);
56    result.append(address).append(tab).append(job);
57
58    return result.toString();
59  }
60
61  /**
62   * returns the given name for this entry
63   *
64   * @return the given name for this entry
65   */
66  public String getGivenName() {
67    return givenName;
68  }
```

```
69
70    /**
71     * updates the given name for this entry
72     *
73     * @param newGivenName the new value for the given name of this entry
74     */
75    public void setGivenName(String newGivenName) {
76      givenName = newGivenName;
77    }
78
79    /**
80     * returns the family name for this entry
81     *
82     * @return the family for this entry
83     */
84    public String getFamilyName() {
85      return familyName;
86    }
87
88    /**
89     * updates the family name for this entry
90     *
91     * @param newFamilyName the new value for the family name of this entry
92     */
93    public void setFamilyName(String newFamilyName) {
94      familyName = newFamilyName;
95    }
96
97    /**
98     * returns the phone number for this entry
99     *
100    * @return the phone number for this entry
101    */
102   public PhoneNumber getPhoneNumber() {
103     return phoneNumber;
104   }
105
106   /**
107    * updates the phone number for this entry
108    *
109    * @param newPhoneNUmber the new value for the phone number of this
         entry
110    */
111   public void setPhoneNumber(PhoneNumber newPhoneNumber) {
112     phoneNumber = newPhoneNumber;
113   }
114
115   /**
116    * returns the address for this entry
117    *
118    * @return the address for this entry
119    */
120   public Address getAddress() {
121     return address;
122   }
123
124   /**
125    * updates the address for this entry
```

```
126     *
127     * @param newAddress the new value for the address of this entry
128     */
129    public void setAddress(Address newAddress) {
130      address = newAddress;
131    }
132
133    /**
134     * returns the job for this entry
135     *
136     * @return the job for this entry
137     */
138    public String getJob() {
139      return job;
140    }
141
142    /**
143     * updates the job for this entry
144     *
145     * @param newJob the new value for the job of this entry
146     */
147    public void setJob(String newJob) {
148      job = newJob;
149    }
150  }
```

```
1   /**
2    * class: Database (represents the database, which consists of Entry
       elements)
3    *
4    * @author Oren Avni, (Tutor for GDI 1 / ICS 1 @ Winterterm: 2008-2009
5    *         {TU-Darmstadt}
6    * @author Guido Roessling
7    * @category Java - Basic Object Operations
8    * @version 1.0
9    */
10  public class Database {
11    /* the private array containing the elements */
12    private Entry[] entries;
13
14    /**
15     * initializes the database to an empty entry collection
16     */
17    public Database() {
18      entries = new Entry[0];
19    }
20
21    /**
22     * Internal service method that validates an access
23     *
24     * @param pos the position to be validated
25     * @return true if the database storage exists and position
26     * pos is valid, else false
27     */
28    protected boolean isValidAccess(int pos) {
29      // valid access: array exists and pos in [0, entries.length)
30      return (entries != null && entries.length > pos
31          && pos >= 0 && entries[pos] != null);
```

```
32    }
33
34    /**
35     * Drops the database and replaces it with one of size 0
36     *
37     * @return true if database has been deleted correctly, else false.
38     */
39    public boolean dropDatabase() {
40      entries = new Entry[0];
41      // return true if empty and exactly size 0
42      return (entries != null && entries.length == 0);
43    }
44
45
46    /**
47     * returns the element at the chosen position of null
48     * if the position is invalid
49     *
50     * @param pos the position of the Entry to be returned.
51     * @return the Entry object at the given position, if any, else null.
52     */
53  /*  public Entry getEntry(int pos) {
54      // ensure position is valid
55      if (!isValidAccess(pos))
56        return null;
57
58      // valid, so return value at this position
59      return entries[pos];
60    }*/
61
62    /**
63     * deletes the entry e if it exists and updates the database
64     *
65     * @param e the Entry which should be deleted.
66     * @return true if Entry has been deleted successfully, else false.
67     */
68    public boolean deleteEntry(Entry e) {
69      if (!entryExists(e))
70        return false;
71
72      // determine target position
73      int pos = getPos(e);
74
75      // delete this value...
76      entries[pos] = null;
77
78      // resize according to deleted position
79      resize(pos);
80
81      // everything OK
82      return true;
83    }
84
85    /**
86     * resizes the database after the entry at position
87     * deletedPos was removed
88     *
89     * @param deletedPos the position that was deleted
```

```
90      */
91     public void resize(int deletedPos) {
92       // create new storage, size is "our size - 1"
93       Entry[] result = new Entry[getSize() - 1];
94
95       // if we did not delete the first element,
96       // copy all elements until position "deletedPos"
97       if (deletedPos > 0) {
98         System.arraycopy(entries, 0, result, 0, deletedPos);
99       }
100
101      // if the deleted element was not the last element,
102      // copy all elements following this one
103      if (deletedPos != getSize() - 1) {
104        System.arraycopy(entries, deletedPos + 1, result,
105            deletedPos, getSize() - deletedPos - 1);
106      }
107
108      // assign as new database
109      entries = result;
110    }
111
112    /**
113     * returns true if an entry matching the parameter exists
114     * in the database
115     *
116     * @param e the Entry that should exist.
117     * @return true if the Entry exists in the database, else false.
118     */
119    public boolean entryExists(Entry e) {
120      // entry exists if its position is not -1...
121      return (getPos(e) != -1);
122    }
123
124    /**
125     * inserts an entry into the database. Note: entries that already
126     * exist will not be inserted again.
127     *
128     * @param e: the Entry that should be inserted to the database.
129     * @return true if Entry has been successfully inserted to the database
          , else
130     * false (if the value is null or already existed).
131     */
132    public boolean addEntry(Entry e) {
133      // do not insert null or existing entries
134      if (e == null || entryExists(e))
135        return false;
136
137      // create new storage space
138      Entry[] newDatabase = new Entry[getSize() + 1];
139
140      // copy all values
141      System.arraycopy(entries, 0, newDatabase, 0, getSize());
142
143      // add entry at end of database
144      newDatabase[getSize()] = e;
145
146      // update database to this
```

```
147       entries = newDatabase;
148
149       // success!
150       return true;
151     }
152
153     /**
154      * returns the position of the entry in the database or -1 if not found
155      *
156      * @param e the Entry for which the position shall be determined.
157      * @return either a valid position or -1 if the entry is not contained
           or
158      * the database is empty.
159      */
160     public int getPos(Entry e) {
161       // check if database has at least one element and entry exists
162       if (!isValidAccess(0) || e == null)
163         return -1;
164
165       // start searching...
166       int pos = 0;
167       // create String representation once
168       String ourEntry = e.toString();
169       // compare Strin representation to stored entries
170       while (pos < getSize() && !(ourEntry.equals(entries[pos].toString()))
            )
171         pos++;
172
173       // if we have not reached pos == getSize(), we found the entry
174       if (pos < getSize())
175         return pos;
176
177       // return -1 as an error marker
178       return -1;
179     }
180
181     /**
182      * returns the size of this database
183      *
184      * @return the size of this database.
185      */
186     public int getSize() {
187       if (entries == null)
188         return -1;
189
190       return entries.length;
191     }
192
193     /**
194      * Swaps the elements at the two given indices,
195      * if both indices are valid
196      *
197      * As no entries are changed, only their ordering,
198      * the database will stay consistent
199      * @param pos1 the first position to be swapped
200      * @param pos2 the second position to be swapped
201      */
202     protected void swap(int pos1, int pos2) {
```

```
203       // Both positions must be at least 0 and
204       // less than the number of elements.
205       // They should also not be identical
206       if (pos1 >= 0 && pos1 < getSize()
207           && pos2 >= 0 && pos2 < getSize()
208           && pos1 != pos2) {
209         // triangular exchange
210         Entry temp = entries[pos2];
211         entries[pos2] = entries[pos1];
212         entries[pos1] = temp;
213       }
214     }
215
216     /**
217      * Returns the complete String representation of this database
218      *
219      * @return the contents of the database as one formated String.
220      */
221
222     public String toString() {
223       if (!isValidAccess(0))
224         return "<Database is empty>";
225
226       // Use a StringBuffer...
227       StringBuffer buffer = new StringBuffer(entries.length * 128);
228
229       // iterate through the array...
230       int size = getSize(); // prevent multiple lookups
231       for (int pos = 0; pos < size - 1; pos++)
232         buffer.append(entries[pos].toString()).append("\n");
233
234       // add last entry without final return character
235       buffer.append(entries[size - 1]);
236       return buffer.toString();
237     }
238 }
```