



Grundlagen der Informatik 1

WS 2008/09

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

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

1 Mini Quiz

Objekte 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 diesem 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 betrachten.

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

```
1  /**
2   * Random class for Gdl / 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 `private` Attribut vom Typ *Random* anlegt und anschließend für die Ergebnisse des Aufrufes `doSomething(50, 100)` prüft, ob die einzelnen 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:

```

1  import static org.junit.Assert.assertTrue;
2  import org.junit.Before;
3  import org.junit.Test;
4
5  /**
6   * Test the class Random
7   *
8   * @author Oren Avni / Guido Roessling
9   * @version 1.0
10  */
11  public class RandomTest {
12
13      /** the Random instance */
14      private Random random ;
15
16      /**
17       * Initializes the Random instance
18       */
19      @Before
20      public void init() {
21          random = new Random();
22      }
23
24      /**
25       * Test whether all values generated fall in the correct
26       * interval
27       */
28      @Test
29      public void ensureCorrectValueRange() {
30          int [] result = random.doSomething(50, 100);
31          for (int i = 0; i < 100; i++)
32              assertTrue("Value outside [1, 50]: " + result[i],
33                          (result[i] >= 1 && result[i] < 51));
34      }
35
36      /**
37       * Test the method "doSomething" in RandomTest
38       * This test will time out after 1000ms = one second.
39       */
40      @Test(timeout = 1000)
41      public void testGiantArray() {
42          // create 2.000.000 random int values in [1, 101]
43          int [] result = random.doSomething(100, 2000000);
44          for (int i = 0; i < 100; i++)
45              assertTrue("Value outside [1, 100]",
46                          (result[i] >= 1 && result[i] < 101));
47      }
48  }

```

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

```

1  public class Strange {
2      public int getAvg(int [] array) {
3          int temp = 0;
4
5          for (int i = 0; i <= array.length; i++) {
6              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 Gdl1-Portal bereitgestellt. Kommentieren Sie Ihren selbst erstellten Code. Die Hausübung muss bis zum Abgabedatum im Gdl1-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 Gdl1-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 Attribut `private Entry[] entries` verfügt. Diese Klasse soll zwei unterschiedliche Konstruktoren zur Verfügung stellen:

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

- Einen Konstruktor, 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 Attribut *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.*