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 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.
2. Was ist ein Konstruktor? Wofür wird dieser benötigt? Wie sieht ein minimaler Konstruktor aus?
3. Nennen Sie die wichtigsten Methoden der Assert Klasse, die man bei einem typischen JUnit Testcase benötigt.
4. Nennen und beschreiben Sie die vier Phasen der Übersetzung (Kompilierung) eines Programms.

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!
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.

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?
- 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 }
```

- Berechnen Sie auch das Ergebnis für a=4, b=3 und c=2.
- 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).
- In der oberen Klasse gibt es eine überflüssige Variable. Ermitteln Sie diese und erklären Sie, warum diese nicht benötigt wird.

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.

- 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!

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.

- 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?

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 Beispielimplementierung 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.