Grundlagen der Informatik 1

Sommersemester 2011

Dr. Guido Rößling

https://moodle.informatik.tu-darmstadt.de/

Übung 9 Version: 1.0 13. 06. 2011

---

1 Mini Quiz

Objekte und Klassen

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

Compiler vs. Interpreter

- ☐ Sowohl die als auch Java sind 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 „Wildcard“ (*) beim Import kann man auf alle Unterklassen eines Packages zugreifen.
- ☐ Eine Klasse kann in Java mehreren Packages angehören.

2 Fragen

1. Erklären Sie die folgenden Begriffe in eigenen Worten: **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 sollen **private**, nicht **public**, sein.
   - Der Konstruktor soll einen String als Parameter erhalten, der 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` zugreifen zu können.
   - 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!

   Hinweis: Eclipse kann Ihnen viel von der Arbeit abnehmen, wenn Sie die Menüeinträge im Menü „Source“ etwas erkunden. Allerdings sollten Sie auch in der Lage sein, diese Aufgabe *ohne* Eclipse zu bearbeiten (Stichwort: Programmierung auf Papier in der Klausur)!

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

   - Anlegen mindestens eines „Test-Autos“ in einer mit `@Before` annotierten Methode. Die anderen Tests können dann direkt auf diesem Auto arbeiten. Denken Sie daran, dass diese Methode vor *jeder* Testmethode aufgerufen wird.
   - Nun testen Sie bitte den Konstruktor (für die Erzeugung des Objekts) und die Methode String getName(). Dazu soll nach dem Anlegen des Objektes dessen Name einmal mit dem tatsächlichen und einem mit einem falschen Namen überprüft werden sowie geprüft werden, ob der Anfangskilometerstand 0.0 beträgt. 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.
   - Testen Sie die 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  }
```

1. Wieviele primitive Datentypen werden in der Klasse `P` verwendet?
2. 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 }
```

3. Berechnen Sie auch das Ergebnis für a=4, b=3 und c=2.
4. Was für einen Zweck erfüllt der Algorithmus, der sich aus method1(..) und method2(..) zusammensetzt? Beachten Sie die verschachtelte Rekursion!

   Hinweis: Die Klasse `Math`, die Sie kennen sollten, enthält diesen Algorithmus (wenn auch in einer leicht abgewandelter Form).
5. 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 *RandomDemo* mit (bewusst) unvollständiger Kommentierung:

```
1  import java.util.Random;
2
3  /**
4   * Random class for GdI / ICS exercise sheet 9.
5   *
6   * @author Oren Avni / Guido Roessling
7   * @version 1.0
8   */
9  public class RandomDemo {
10
11   /**
12    *
13    *
14    * @param m
15    * @param s
16    * @return an int array that does something...
17    */
18   public int[] doSomething(int m, int s) {
19     int i = 0;
20     int[] arr = new int[s];
21     Random r = new Random();
```

```
22
23      while (i < arr.length) {
24        arr[i] = r.nextInt(m + 1);
25        i++;
26      }
27
28      return arr;
29    }
30  }
```

a) Was macht die Methode *doSomething*?

Hinweis: Die Methode **int** nextInt(**int** n) der Klasse java.util.Random liefert eine ganzzahlige Zufallszahl $x \in [0, n)$ zurück.

b) 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 einzelnen Werte des Arrays die gewünschten Eigenschaften haben. Verwenden Sie für den Test *assertTrue(String messageOnFail, boolean condition)*.

c) Schreiben Sie nun einen JUnit-Testcase, der genau wie der vorherige Test arbeitet, nur für den Aufruf *doSomething(100, 2000000)*. Bitte beachten Sie, dass der Test **scheitern soll**, falls die Ausführung länger als 50 Millisekunden dauert. Wenn Ihr Rechner nicht sehr schnell ist, sollte dieser Test scheitern!

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 }
```

a) 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.

b) 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 Lernportal Informatik bereitgestellt. Kommentieren Sie Ihren selbst erstellten Code. Die Hausübung muss bis zum Abgabedatum im Lernportal Informatik 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.
Falls Sie die Hausübung in einer Lerngruppe bearbeitet haben, geben Sie dies bitte deutlich bei der Abgabe an. Alle anderen Mitglieder der Lerngruppe müssen als Abgabe einen Verweis auf die gemeinsame Bearbeitung einreichen, damit die Abgabe im Lernportal Informatik auch für sie bewertet werden kann. Beachten Sie dazu die Hinweise bei der Aufgabenabgabe im Lernportal Informatik!
Abgabedatum: Freitag, 24. 06. 2011, 16:00 Uhr
Denken Sie bitte daran, Ihren Code hinreichend gemäß den Vorgaben zu kommentieren (Racket: Vertrag, Beschreibung und Beispiel sowie zwei Testfälle pro Funktion; Vertrag, Beschreibung und Beispiel für jede local definierte Funktion; Vertrag (ohne Namen) und kurze Beschreibung für jeden lambda-Ausdruck; Java: JavaDoc). Zerlegen Sie Ihren Code sinnvoll und versuchen Sie, wo es möglich ist, bestehende Funktionen wiederzuverwenden. Wählen Sie sinnvolle Namen für Hilfsfunktionen und Parameter.

6 Implementierung einer Tabellenkalkulation (10 Punkte)

In dieser Aufgabe werden Sie eine „Mini-Tabellenkalkulation“, das „Poor Man's Excel“[1], implementieren. Im Gegensatz zu einem „echten“ Excel ist der Funktionsumfang natürlich auf unsere Ansprüche reduziert—für viele Anforderungen aber durchaus ausreichend, wenn noch eine ansprechende GUI bereitgestellt würde.
Zu beachten: Erschrecken Sie nicht über die relativ große Anzahl Klassen: wenn Sie die Themen OO-Modellierung und Vererbung verstanden haben, ist ein Großteil der Klassen einfach zu implementieren.

6.1 Grundkonzept

Eine Tabellenkalkulation besteht normalerweise aus einer Matrix: einer Folge von Zeilen, die jeweils eine bestimmte (prinzipiell aber variable und auch erweiterbare) Anzahl von Spalten besitzen. Für unsere Aufgabe konzentrieren wir uns auf ein *eindimensionales* Feld und definieren dafür jedes Element „passend“.
In unserem „Poor Man's Excel“ besteht jede Zeile aus einem einzigen Eintrag, nämlich einem Objekt der Klasse *Entry*. Ein *Entry* wiederum besitzt genau drei anzeigbare Attribute: eine *numerischen ID*—in der Regel der Zeilennummer—, eine Beschreibung (Typ *String*) sowie den *Inhalt*, der von einem Untertyp der Klasse Value ist. Wie wir später sehen werden, können Inhalte (fast) beliebig komplex sein, werden aber *immer* bei Aufruf der Methode evaluate() zum Typ **double** ausgewertet.
Zu beachten: Alle Klassen dieser Aufgabe sollen zu dem*Package* ($\rightarrow$ T12.44-56) pme gehören.

6.2 Implementierung des Grundgerüsts (3 Punkte)

Implementieren Sie zunächst das Grundgerüst. Dieses besteht aus den folgenden Klassen und Methoden:

Klasse Entry besitzt nur die folgenden drei Methoden:

- Der Konstruktor erwartet einen String—die Beschreibung der Zelle—und ein Element vom Typ *Value* (siehe die folgende Teilaufgabe).
- Die Methode **public** String getDescription() liefert die Beschreibung der Zelle zurück.

[1] Auf Deutsch: *Excel für Arme*

- Die Methode **public** Value getValue() gibt den Wert der Zelle zurück.

Klasse PoorMansExcel besitzt die folgenden Elemente:

- Der *erste* Konstruktor erhält einen **int**-Wert und sorgt dafür, dass die Tabelle genau diese Anzahl Werte (vom Typ Entry) speichern kann.
- Der *zweite* Konstruktor bekommt ein Feld von Entry-Objekten übergeben und verwendet dieses als neue Tabelle.
- Mit **public int** getSize() kann die Größe der Tabelle abgefragt werden.
- Mit **public void** resize(**int**) kann die Tabelle auf eine neue—kleinere oder größere—Größe gebracht werden. Ist der Parameter 0 oder negativ, soll sich die vorhandene Tabelle nicht ändern.
- Mit **public boolean** update(**int**, Entry) soll der Eintrag an der übergebenen Position geändert (überschrieben) werden. Die Methode gibt bei Erfolg **true**, sonst **false** zurück.

  Zu beachten: da Tabellenkalkulationen die erste Zeile immer mit 1, nicht 0, bezeichnen, ist diese Positionsangabe entsprechend anzupassen.
- Überschreiben Sie die geerbte Methode **public** String toString(). Beachten Sie dazu die folgenden Hinweise und nutzen Sie einen StringBuilder zum Zusammenstellen der Ausgabe:
  1. Zunächst ist die Zeilenposition (beginnend mit 1 für das erste Element an Position 0 des Feldes) *rechtsbündig mit Breite 3* anzugeben; Werte kleiner 100 sind also von links „passend“ aufzufüllen.
  2. Anschließend folgt die Angabe der Beschreibung des Elements mit einer Länge von immer 60 Zeichen. Hier wird ggf. rechts („hinten“) mit Leerzeichen aufgefüllt bzw. ein zu langer Eintrag „passend gekürzt“.
  3. Als letztes folgt die Angabe des *ausgewerteten Inhalts*. Hierzu ist die im kommenden Aufgabenteil zu implementierende Methode **public double** evaluate() zu verwenden.
  4. Jede Zeile—auch die letzte—endet mit einem Zeilenvorschub "\n".

  Eine Beispielausgabe könnte also wie folgt aussehen:

```
  1 Constant                                                    3.7
  2 Investment                                                  6.0
  3 Return                                                      0.0
  4 Debt                                                        321.77
```

6.3 Implementierung der Inhalte (6 Punkte)

Implementieren Sie ein Element Value, das nur die Methode **public abstract double** evaluate() besitzt. Mit dieser Methode können *konkrete* Instanzen von Erben von Value zu einer **double** ausgewertet werden.
Implementieren Sie nun die folgenden Untertypen von Value:

- Die Klasse *ConstantValue* erhält im Konstruktor eine **double** und liefert diese als Ergebnis.
- Die Klasse *NullValue* liefert bei Evaluation immer 0.0.
- Die folgenden Klassen erhalten *zwei* Objekte vom Typ Value als Parameter im Konstruktor und nutzen diese zur Auswertung. In der folgenden Tabelle wird unterstellt, dass der erste Parameter a, der zweite b ist:

Klasse	Operation
Add	a + b
Average	Durchschnitt von a und b
Divide	a / b (0.0, wenn b 0.0 war)
Max	max(a, b)
Min	min(a, b)
Multiply	a * b
Subtract	a - b

Zu beachten: Ist einer der beiden Parameter **null**, so ist dieser durch eine Instanz von NullValue zu ersetzen, um Auswertungsprobleme zu vermeiden. Achten Sie auch bei der Division darauf (siehe Tabelle).

Zu beachten: Da alle Funktionen genau zwei Value-Parameter erwarten, kann man auch schön Funktionen „schachteln“, etwa in der Form
new Average(**new** Add(**new** ConstantValue(3.7), **new** ConstantValue(2.3)), **new** ConstantValue(1.4)). Dieser (mittelmäßig) komplexe Ausdruck sollte in der Tabelle mit dem Ergebnis 3.7 angezeigt werden.

6.4 Testen (1 Punkt)

Schreiben Sie eine Testklasse, die Ihre Quellen ausführlich testet—auch mit komplexeren Ausdrücken wie dem oben stehenden. Denken Sie daran: unsere Tests sind im Zweifelsfall eher umfangreicher als Ihre, und deren Scheitern bestimmt auch über Ihre Punktzahl—so wie Fehler beim Kunden sich später auf Ihre Bezahlung oder Anstellung auswirken können!