Grundlagen der Informatik 1
WS 2008/09

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

Übung 10 - Lösungsvorschlag Version: 1.0 12.01.2009

---

1 Mini Quiz

Kreuze die korrekten Aussagen an.

Vererbung

- ☐ Eine Klasse kann gleichzeitig von mehreren Klassen erben. zurückgeben.
- ☐ Eine Unterklasse kann immer *alle* geerbten Methoden überschreiben.
- ☐ Vererbung wird in Java durch das Schlüsselwort *inherit* durchgeführt.
- ☒ Aufrufe über **super** werden statisch gebunden.
- ☒ Aufrufe über **this** werden dynamisch gebunden.

Abstrakte Klassen und Interfaces

- ☒ Eine Klasse kann gleichzeitig mehrere Interfaces implementieren.
- ☐ In jeder nicht abstrakten Klasse muss immer mindestens ein Konstruktor implementiert werden.
- ☐ *Interfaces* enthalten nur Methoden und Konstanten, aber keine Attribute.
- ☐ *Abstrakte Klassen* deklarieren nur Attribute und Methoden, aber keine Konstanten.

2 Fragen

1. Ein wichtiges Konzept der Objektorientierung ist die Vererbung. Welche Vorteile ergeben sich dadurch?

   Lösungsvorschlag:
   Vererbung bringt eine ganze Reihe an Vorteilen für Programmierer:

   - *Wir können bereits implementiertes Verhalten in abgeleiteten Klassen wiederverwenden, ohne das Rad neu zu erfinden. Somit lässt sich hier unnötige Programmierarbeit und damit Kosten (Zeit und / oder Geld) einsparen.*
   - *Einfachere Wartung! Fehlerkorrekturen oder Änderungen der vererbten Funktionalität finden nur in der zentralen Oberklasse statt.*

- *Durch die Vererbung können Beziehungen zwischen Klassen erstellt und Klassenhierarchien entwickelt werden. Jede Klasse ist für eine ganz bestimmte Funktionalität verantwortlich, und durch die Vererbungshierarchie geben Klassen ihre Funktionalität an andere Klassen weiter.*

Hinweis: *Die Rede ist von Klassen,* nicht *von Objekten. Objekte haben mit Vererbung an sich nichts zu tun. Die Vererbung erfolgt ausschließlich über die Definition von Klassen. Ob Klassen Eigenschaften und Methoden selber definieren oder erben, ist für Objekte erstmal unerheblich— Hauptsache, die benötigten Eigenschaften und Methoden sind vorhanden.*

2. Erklären Sie mit eigenen Worten den Begriff *Inkrementelles Programmieren*.

Lösungsvorschlag:
Unter Ausnutzung der Wiederverwendbarkeit lässt sich viel Aufwand sparen, indem Klassen von anderen Klassen erben und somit bereits vorhandene Funktionalität nutzen. Zusätzlich können dann Spezialisierungen und Erweiterungen vorgenommen werden, wobei diese wieder Basis für eine Vererbung sind. Kurz gesagt bedeutet inkrementelles Programmieren, dass man vorhandene Funktionalität nutzt und nur das neu implementiert, was unterschiedlich ist.

3. Worin liegen die Unterschiede zwischen *Interfaces* und *abstrakten Klassen*? Wann macht es Sinn, diese zu nutzen? Welches der beiden Konzepte unterstützt ganz besonders Reusability (Wiederverwendung)?

Lösungsvorschlag:
Ein Interface *deklariert nur Dienste und definiert damit einen gemeinsamen Typ für alle das Interface implementierenden Klassen. Die Dienste selbst müssen in jeder Klasse implementiert werden. Ein Interface kann aber* **keine** *Implementierung enthalten, also kein gemeinsames Verhalten für die dieses Interface implementierenden Typen vorgeben. Dies ist das Prinzip der* funktionalen Abstraktion*: was, nicht wie. Allgemein sagt man: von einer Klasse erben, ein Interface implementieren.*

Eine abstrakte Klasse gibt wie ein Interface einen gemeinsamen Typ für ihre Erben vor. In einer abstrakten Klasse **können** *allerdings Methoden implementiert werden, so dass eine abstrakte Klasse ihren Erben ein teilweise gemeinsames Verhalten vorgibt, sie unterstützt also Wiederverwendung.*

Das Prinzip der Schablone-Operation (template method) ist nützlich, wenn eine Klasse Verhalten implementieren soll, das zu einem Teil für alle Erben gleich ist, zu einem anderen Teil in allen Erben ähnlich ist aber nicht identisch. Der gemeinsame Teil wird dann in der abstrakten Klasse implementiert. Der individuelle Teil wird als abstrakt deklariert und muss in den Erben implementiert werden.

Eine abstrakte Methode ist also ein Versprechen, dass diese Methode irgendwann in der Typhierarchie implementiert wird. Die Implementierung der abstrakten Klasse kann daher auf den abstrakten Teil zugreifen, als wäre er bereits implementiert. Da dies aber erst in den Erben geschieht, kann von der abstrakten Klasse kein Objekt erzeugt werden, sondern erst von einer Klasse, die nicht abstrakt ist.

Ein weiterer Unterschied zwischen einer (abstrakten) Klasse und einem Interface ist, dass in Java eine Klasse immer nur von einer Klasse erben kann*, aber* beliebig viele Interfaces implementieren darf*. Damit ermöglichen Interfaces den Schritt hin zur einer Mehrfachvererbung.*

4. Was versteht man unter dem Begriff des Information Hiding?

Lösungsvorschlag:
Information Hiding (Geheimnisprinzip) ist eine Design-Idee. Details eines Programm-Moduls werden versteckt, was zur Folge hat, dass die Komplexität von Anwendungen reduziert wird. Die Kopplung

zwischen Modulen wird reduziert. Die versteckte Implementierung kann geändert werden, ohne die Klienten anzupassen bzw. unwirksam zu machen.

Hinweis: *Prägen Sie sich einfach folgendes ein: Blackbox = Information Hiding.*

5. Vererbungs-Wissenstest: Streichen Sie falsche Antworten, so dass die Aussagen wahr werden.
   - Es gibt eine / keine Basisklasse, von der alle Java-Klassen direkt oder indirekt abgeleitet sind.
   - Subklasse ist ein anderer Ausdruck für Unterklasse / Oberklasse.
   - Superklasse ist ein anderer Ausdruck für Unterklasse / Oberklasse.
   - Eine Klasse kann / kann nicht Superklasse für mehrere Subklassen sein. Eine Klasse kann / kann nicht Subklasse mehrerer Superklassen sein.

   Lösungsvorschlag:

   - *Es gibt* **eine** *Basisklasse, von der alle Java-Klassen direkt oder indirekt abgeleitet sind (java.lang.Object).*
   - *Subklasse ist ein anderer Ausdruck für* **Unterklasse**.
   - *Superklasse ist ein anderer Ausdruck für* **Oberklasse**.
   - *Eine Klasse* **kann** *Superklasse für mehrere Subklassen sein. Eine Klasse* **kann nicht** *Subklasse mehrerer Superklassen sein.*

3 The Final Countdown

Für die folgende Aufgabe werden Kentnisse über das Schlüsselwort **final** vorausgesetzt. Machen Sie sich bitte damit vertraut und erklären Sie kurz die Auswirkungen von **final** bei der Anwendung auf:

- Attribute und Methodenparameter
- Methoden
- Klassen

Infos zu diesem Schlüsselwort finden Sie hier:

- `http://de.wikipedia.org/wiki/Java-Syntax` (deutsch)
- `http://renaud.waldura.com/doc/java/final-keyword.shtml` (englisch)

Lösungsvorschlag:
Das Schlüsselwort **final** *verhält sich wie folgt:*

- *Bei* Attributen *wird eine Konstante erschaffen, da ein "final" Attribut nicht mehr änderbar ist. Das gleiche gilt bei Methodenparametern: an diese kann in der Methode kein Wert zugewiesen werden.*
- *Eine "final" markierte Methode darf von Erben nicht überschrieben werden.*
- *Eine "final" markierte Klasse darf analog "nicht von Erben überschrieben werden" - praktisch darf von dieser Klasse nicht mehr geerbt werden.*

Schauen Sie sich nun bitte folgenden Java Code an. Beheben Sie sämtliche eingebauten Fehler, um den Code lauffähig zu machen. Wenden Sie Ihr neu erworbenes Wissen an, um zu erklären, weshalb die Vererbung nicht so funktioniert, wie sie hier ursprünglich gedacht war.
Hinweis: Für die Bearbeitung dieser Aufgabe sollten Sie bitte *keine* Computer-Unterstüzung benutzen. Denken Sie dran: In der Klausur werden Sie ebenfalls keine Hilfsmittel wie Eclipse einsetzen können...

```
1  public final class A {
2    private int value1 = 0, value2 = 0;
3
4    private final int value3 = 0;
5
6    private int getValue1() {
7      return value1;
8    }
9
10   private int getValue2() {
11     return value2;
12   }
13
14   private void setValue1(int newValue1) {
15     value1 = newValue1;
16   }
17
18   private void setValue2(int newValue2) {
19     value2 = newValue2;
20   }
21
22   public final void changeValue3(final int newValue3) {
23     value3 = 0;
24   }
25 }
26
27 class B extends A {
28   public void changeValue3(final int newValue3) {
29     value3 = newValue3;
30   }
31
32   public static void main(String args[]) {
33     B obj = new B();
34
35     obj.setValue1(30);
36     obj.setValue2(3);
37     obj.value3 = 2008;
38
39     String result = "The final exam for GdI / ICS 1 will be on " +
         getValue1() + "."
40     + getValue2() + "." + obj.value3
41     + ". Please keep this in mind!";
42
43     System.out.println(result);
44   }
45 }
```

Was müssen Sie darin ändern, um die folgende Ausgabe zu erhalten?
"The final exam for GdI / ICS 1 will be on 30.3.2008. Please keep this in mind!"

Lösungsvorschlag:

Zeile 1 *Zunächst muss das Schlüsselwort* **final** *entfernt werden, sonst darf Klasse B nicht von Klasse A erben (die entsprechende Fehlermeldung erscheint in Zeile 27).*

Zeile 4 *Das* final *vor dem Attribut* value3 *ist ebenfalls zu entfernen, da sonst die Zuweisung in beiden Methoden* changeValue3 *(Zeile 23 und 29) sowie die explizite Zuweisung in Zeile 37 scheitert.*

Zeile 6, 10 *Die Methoden* getValue1(), getValue2() *dürfen auch nicht* private *sein, da sonst der Zugriff in Ziele 39 und 40 scheitert.*

Zeile 14, 18 *Das Gleiche gilt für die Methoden* setValue1(int), setValue2(int) *in Zeile 14 bzw. 18, auf die in Zeile 35 bzw. 36 zugegriffen wird.*

Zeile 22 *Die Methode darf nicht als* final *markiert sein, da sie sonst nicht im Erben überschrieben werden dürfte (Zeile 28-30). Der Parameter darf aber weiterhin "final" sein.*

Zeile 39, 40 *Die direkten Zugriffe auf* getValue1(), getValue2() *sind innerhalb der* main*-Methode auf das passende Objekt abzubilden: es muss ein* `obj.` *vorangestellt werden, wie schon bei* obj.value3 *zu sehen ist.*

Die Markierung der Methodenparamater als final *(Ziele 22 und 28) ist legal und darf stehen bleiben.*
Die korrekte Lösung sieht dann wie folgt aus:

```
1  class A {
2    private int value1 = 0, value2 = 0;
3
4    int value3 = 0;
5
6    int getValue1() {
7      return value1;
8    }
9
10   int getValue2() {
11     return value2;
12   }
13
14   void setValue1(int newValue1) {
15     value1 = newValue1;
16   }
17
18   void setValue2(int newValue2) {
19     value2 = newValue2;
20   }
21
22   public void changeValue3(final int newValue3) {
23     value3 = newValue3;
24   }
25 }
26
27 public class B extends A {
28   public void changeValue3(final int newValue3) {
29     value3 = newValue3;
30   }
31
32   public static void main(String args[]) {
33     B obj = new B();
34
```

```
35      obj.setValue1(30);
36      obj.setValue2(3);
37      obj.value3 = 2008;
38
39      String result = "The final exam for GdI / ICS 1 will be on " + obj.
          getValue1() + "."
40          + obj.getValue2() + "." + obj.value3
41          + ". Please keep this in mind!";
42
43      System.out.println(result);
44    }
45  }
```

4 Wer vererbt wem was...?

Vollziehen Sie bitte nach, wie das folgende Programm arbeitet. Geben Sie die jeweilige Ausgabe des Aufrufs *Z.test()* hinter den Kommentaren der Methode test() in der Klasse Z an.
Es ist *nicht* Ziel dieser Aufgabe, dass Sie Eclipse starten, den Code einfügen und das Ergebnis einfach abzulesen!

```
1   class X {
2     int a = 4;
3
4     int get() {
5       return a;
6     }
7   }
8
9   class Y extends X {
10    static int a = 7;
11
12    int get() {
13      return a;
14    }
15
16    static void set(int x) {
17      a = x;
18    }
19
20    static void set(char c) {
21      a = 2 * c;
22    }
23  }
24
25  class Z extends Y {
26    static int b = 3;
27
28    int get() {
29      return b + a;
30    }
31
32    static int get(X x) {
33      return x.a;
34    }
35
36    static void set(int i) {
```

```
37      a = 3 * i;
38     }
39
40    static void set(X x, int i) {
41      a = i;
42    }
43
44    static void test() {
45      Z z = new Z();
46
47      System.out.println(Y.a); // --------
48      System.out.println(get(z)); // --------
49
50      Z.set('c' - 'a' - 1); // ASCII Werte: 'c' = 99, 'a' = 97
51      System.out.println(get(z)); // --------
52      System.out.println(z.get()); // --------
53
54      Y y = z;
55      Y.set(2);
56      System.out.println(z.get() + 1); // --------
57      Z.set(y, 0);
58      System.out.println(y.get() + 4); // --------
59    }
60
61    public static void main(String args[]) {
62      Z.test();
63    }
64  }
```

Lösungsvorschlag:
Das Attribut a *ändert insgesamt dreimal seinen Wert, wobei die Zeilenangaben für den Wert* nach *Ausführung des entsprechenden Befehls stehen:*

- *Von Zeile 45-48 hat* a *den Wert 7.*
- *Nach Ausführung von Zeile 50 bis einschließlich Zeile 55 hat* a *den Wert 3.*
- *In Zeile 56 und 57 hat* a *den Wert 2.*
- *Nach Ausführung von Zeile 58 ist der Wert von* a *0.*

Damit ergeben sich folgende Ergebnisse für den Aufruf von Z.test()*:*

Zeile 47 *7 - das ist der Wert des Klassenattributs* a *der Klasse Y (Zeile 10).*

Zeile 48 *4 - Der Aufruf von* get(z) *nutzt die Methode in Zeile 32. Hier wird explizit auf die "Sicht von X" verengt, und aus dieser Sicht hat das Attribut den Wert 4 (aus Zeile 2).*

Zeile 52 *4 - Der Aufruf von* set *nutzt die Methode in Zeile 36, um das Attribut a auf den Wert 3 zu setzen. Durch den Zugriff auf die Sichtbarkeit von* X *wird hier aber der Wert 4 genutzt.*

Zeile 53 *6 -* z.get() *ist die Methode in Zeile 28. Geliefert wird die Summe aus dem Wert des Klassenattributs* b *mit Wert 3 (siehe Zeile 26) und dem aktuellen Wert des Attributs* a *von Z, das in Zeile 52 als Seiteneffekt auf 3 gesetzt wurde.*

Zeile 57 *6 - Es wird erneut die Methode* z.get() *aus Zeile 28 genutzt. Am Ende des Befehls in Zeile 56 haben die Instanz- und Klassenattribute von Y und Z alle den Wert 2. Die Berechnung ist also* (b + a) $+1 = 3 + 2 + 1 = 6$.

Zeile 59 *7 - In Zeile 58 wird die Methode* set(X, int) *in Zeile 40 aufgerufen. Durch die Verengung auf* X *"sieht" die Methode den Wert* a = 4 *aus X, überschreibt diesen aber - und damit auch den für Y und Z - mit dem Wert 0 (dem Argument* i*). Ab nun ist also* a == 0*.*

Hier eine Übersicht, welchen Wert a *nach Ausführung der jeweiligen Codezeile hat:*

Zeile	z.a	y.a
45	*7*	
47	*7*	
48	*7*	
50	*3*	
52	*3*	
53	*3*	
55	*3*	
56	*2*	*2*
57	*2*	*2*
58	*0*	*0*
59	*0*	*0*

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 Fr, 23.01.2009, 16:00.

5 Eine "bessere" Datenbank

In der letzten Übung haben Sie (hoffentlich) die Datenbankaufgabe programmiert, die die elementaren Funktionalitäten wie das Löschen, Einfügen oder das Finden eines Datensatzes ermöglicht.
Da Datenbanken etwas mehr können als nur diese primitiven Methoden, nutzen wir nun *Vererbung*, um die bestehende Funktionalität der Datenbank um neue Funktionalität zu erweitern. Sie können dazu Ihre eigene Datenbank aus der letzten Hausübung nutzen oder die Bibliothek datenbank.jar, die wir im GDI-Portal zur Verfügung stellen. Laden Sie nun die Vorgabendatei vom Portal und importieren Sie diese über `File -> Import... -> Archive File` in Ihr Projekt.

Hinweis: Wenn Sie Ihre eigene Lösung verwenden, implementieren Sie als zusätzliche Funktion **public** Entry getEntry(**int** pos), die den Eintrag an der entsprechenden Position in der Datenbank zurücklie fert. Diese Funktion ist in der zur Verfügung gestellten Bibliothek bereits enthalten.
Hinweis: Wie in der letzten Übung dürfen keine Vectors or Collections verwendet werden.

5.1 Implementieren des Interfaces AdditionalOperations (3 · 1.5 Punkte)

Zunächst soll das Interface `AdditionalOperations` in einer Klasse `ImprovedDatabase` in einem Package *database* implementiert werden. Diese Klasse soll von der Klasse `Database` aus der letzten Hausaufgabe bzw. dem oben erwähnten JAR erben. Achten Sie auf die korrekte Reihenfolge von

extends und *implements* im Klassenkopf. Es sollen folgende Funktionen, die Sie bereits aus der Mengenlehre kennen, zur Verfügung gestellt werden:

Gegeben zwei Mengen $A = \{a, b, c\}$ und $B = \{b, c, d, e\}$. Daraus ergibt sich

- Vereinigungsmenge: $A \cup B = \{x | x \in A \vee x \in B\} = \{a, b, c, d, e\}$
- Schnittmenge: $A \cap B = \{x | x \in A \wedge x \in B\} = \{b, c\}$
- (Relative) Komplementmenge: $A \setminus B = \{x | x \in A \wedge x \notin B\} = \{a\}$

Diese Regeln benutzen "echte" Datenbanken ebenfalls, um weitergehende Operationen darauf anzuwenden (Join, Natural Join, Equi Join).
Implementieren Sie nun die Methoden des Interfaces. Beachten Sie dabei auch die Erläuterungen:

- **public int** union(ImprovedDatabase data) verknüpft die aktuelle Datenbank und die als Argument übergebene mittels einer *Vereinigung* der Datensätze. (Faustformel: Kombination der Einträge ohne Duplikate)
- **public int** intersection (ImprovedDatabase data) arbeitet wie *union*, bildet aber die *Schnittmenge*. (Faustformel: nur gemeinsame Einträge verbleiben in der Datenbank)
- **public int** complement(ImprovedDatabase data) bildet die *Komplementmenge* durch Entfernen aller Elemente von *data* aus der aktuellen Datenbank. (Faustformel: gemeinsame Einträge aus der aktuellen Datenbank entfernen)

Alle Operationen sollen auf der *aktuellen Datenbank* arbeiten und diese entsprechend durch das Hinzufügen oder Entfernen von Datensätzen *ändern*. Als Ergebnis ist die Anzahl Datensätze nach dem Ausführen der Operationen zu liefern. Haben wir also eine Datenbank mit den Einträgen $\{a, b, c\}$ und eine mit den Einträgen $\{b, c, d, e\}$ (*data*) , so verbleiben nach Aufruf von *intersection(data)* $\{b, c\}$ in der aktuellen Datenbank und der Aufruf liefert 2 zurück.

5.2 Verbesserter Zugriff: Interface ImprovedAccess (2.5 Punkte)

Die Klasse *ImprovedDatabase* soll nun auch das Interface *ImprovedAccess* implementieren.
Wie in jeder anderen Datenbank auch sollen Datensätze anhand einer Bedingung abfragbar sein. Wir betrachten nur den Fall, dass die Abfrage aus atomaren *Spaltennamen* besteht. Implementieren Sie dazu die folgende Methode aus dem Interface: **public** Entry[] selectXFrom(Column col, String str)
Das Ergebnis ist ein Array von Datensätzen, die in der gewählten *Spalte col* den Text *str* enthalten. Dabei soll es unerheblich sei, an welcher *Position* der Spalte der Text enthalten ist - er muss also nicht am *Anfang* oder *Ende* des Eintrags stehen. Außerdem soll *nicht* zwischen Groß- und Kleinschreibung unterschieden werden.

Hinweis: Sehen Sie in die Dokumentation der Klasse *java.lang.String* und suchen Sie nach geeigneten Methoden. Sie müssen hierbei vorhandene Methoden kombinieren.

Die zu durchsuchende Spalte wird durch den Parameter *col* angegeben. Hierbei handelt es sich um eine *Aufzählung (Enumeration)*. Sie können alle in *Column* angegebenen Werte wie Konstanten benutzen, etwa als *Column.GIVEN_NAME*.
Als Beispiel betrachten wir die folgende Datenbank:

```
1 Hans    Meiser  0170-1567952    {Roßmarkt 26, 60311, Frankfurt} Tänzer
2 Tom     Müller  0163-4574567    {Hainer Weg 12, 60486, Frankfurt}
    Clown
3 Tanja   Knechtel        0190-6666666    {Igelweg 3, 64293, Darmstadt}
    Pilotin
4 Timo    Ulbrich 0179-8888888    {Poststrasse 9, 80210, Mainz}   Klempner
5 Josh    Smithers        0180-2222222    {Wallstreet 7, 99551, Stuttgart}
     Broker
6 Karl    Reichert        0162-1111111    {Hainer Weg 42, 04504, Berlin}
    Arzt
```

Die Abfrage selectXFrom(Column.PHONE_AREA_CODE, "017"); ergibt folgende Elemente als Entry[]:

```
1 Hans    Meiser  0170-1567952    {Roßmarkt 26, 60311, Frankfurt} Tänzer
2 Timo    Ulbrich 0179-8888888    {Poststrasse 9, 80210, Mainz}   Klempner
```

Die Abfrage selectXFrom(Column.ADDRESS_STREET_NAME, "weg"); liefert folgendes Entry[]:

```
1 Tom     Müller  0163-4574567    {Hainer Weg 12, 60486, Frankfurt}
    Clown
2 Tanja   Knechtel        0190-6666666    {Igelweg 3, 64293, Darmstadt}
    Pilotin
3 Karl    Reichert        0162-1111111    {Marbachweg 42, 04504, Berlin}
    Arzt
```

Beachten Sie, dass die Suche hier sowohl Einträge mit dem Abfragewert *weg* als auch mit *Weg* zurückliefert, wie oben beschrieben.

Hinweis: wir empfehlen, eine Hilfsmethode String getContentFor(Column col, **int** index) zu schreiben. Diese Methode soll den zu der gegebenen Spalte *col* gehörenden Wert aus Zeile *index* der Datenbank liefern. In der Beispieldatenbank mit sechs Einträgen oben gäbe getContentFor(Column.GIVEN_NAME, 1) das Ergebnis *Tom*.

5.3 Sortieren der Einträge (3 Punkte)

Implementieren Sie nun die Funktion **public void** sort(Column col). Diese Methode soll die Einträge der aktuellen Datenbank aufsteigend nach der Spalte *col* sortieren.
Lesen Sie dazu in die Java-Dokumentation zur Methode myString.compareTo(a String) des Interfaces *java.lang.Comparable*, das von der Klasse String implementiert wird.
Für die Sortierung dürfen Sie einen beliebigen Algorithmus nutzen. Wir empfehlen (trotz schlechter Sortierleistung) die Nutzung einer angepassten Version des *BubbleSort* aus Übung 8, Aufgabe 4.3, da Ihnen dieser Code schon vorliegen sollte. *Benutzen Sie keine sonstigen Bibliotheken oder Klassen mit vorgefertigten Sortieralgorithmen, wie etwa java.util.Arrays.sort().*
Um Änderungen im *private Entry[] entries* durchführen zu können, verwenden Sie die bereits implementierte Methode **protected void** swap(**int** pos1, pos2).
Für Testzwecke ist die Implementierung von JUnit-Tests ebenfalls sehr sinnvoll, diese wird von Ihnen angesichts des Umfangs der Aufgabe hier aber nicht erwartet.

Hinweis: die String-Sortierung bei den Hausnummern entspricht *nicht* der Ordnung der Zahlen: der String 513 ist "kleiner" als der String 6, da das Zeichen "5" vor dem Zeichen "6" im Zeichensatz steht. Dies muss aber in Ihrer Lösung nicht berücksichtigt werden.

Lösungsvorschlag:

```
1  package database;
2
3  /**
4   * class: BetterDatabase (represents the better database, which still
      consists
5   * from Entries)
6   *
7   * @author Oren Avni, (Tutor for GDI 1 / ICS 1 @ Winterterm: 2008-2009
8   *         {TU-Darmstadt}, Guido Roessling
9   * @category Java - Inherit, Abstraction, Interfaces, Basic Object
      Operations
10  * @version 1.0
11  */
12 public class ImprovedDatabase extends Database implements
      AdditionalOperations,
13     ImprovedAccess {
14   /**
15    * Creates a new ImprovedDatabase
16    */
17   public ImprovedDatabase() {
18     super();
19   }
20
21   // ========================================================
22   // Methods from interface "AdditionalOperations"
23   // ========================================================
24
25   /**
26    * Compute the (relative) complement of the current
27    * database and the argument.
28    *
29    * @param data The database with which the current database
30    * will be merged by the complement operator
31    * @return the number of elements in the current database after
32    * the operation has been executed
33    */
34   public int complement(ImprovedDatabase data) {
35     // safety exit if argument was null: do nothing.
36     if (data == null)
37       return getSize();
38
39     // store the size of the other database
40     int otherSize = data.getSize();
41
42     // iterate over all database items...
43     for (int pos = 0; pos < otherSize; pos++) {
44       // retrieve current item from other DB
45       Entry e = data.getEntry(pos);
46
47       // if this entry is in our current database,
48       // delete it from our database
49       if (entryExists(e))
50         deleteEntry(e);
51     }
52
53     // return the new size of the database
54     return getSize();
```

```
55    } // complement(ImprovedDatabase)
56
57    /**
58     * Compute the intersection (German "Schnittmenge") of the current
59     * database and the argument.
60     *
61     * @param data The database with which the current database
62     * will be intersected
63     * @return the number of elements in the current database after
64     * the operation has been executed
65     */
66    public int intersection(ImprovedDatabase data) {
67      // safety exit if argument was null: do nothing.
68      if (data == null)
69        return getSize();
70
71      // store the size of our database
72      int size = getSize();
73
74      // iterate over all database items...
75      for (int pos = size - 1; pos >= 0; pos--) {
76        // retrieve current item from THIS database
77        Entry e = getEntry(pos);
78
79        // if this entry is not in the other database,
80        // delete it from our database
81        if (!data.entryExists(e)) {
82          boolean isOK = deleteEntry(e);
83          if (!isOK)
84            System.err.println("....:" + e +"\n" +toString());
85          }
86      }
87
88      // return the new size of the database
89      return getSize();
90    } // intersection(ImprovedDatabase)
91
92
93    /**
94     * Compute the union (German "Vereinigung") of the current
95     * database and the argument.
96     *
97     * @param data The database with which the current database
98     * will be merged
99     * @return the number of elements in the current database after
100    * the operation has been executed
101    */
102   public int union(ImprovedDatabase data) {
103     // safety exit if argument was null: do nothing.
104     if (data == null)
105       return getSize();
106
107     // store the size of the other database
108     int otherSize = data.getSize();
109
110     // iterate over all database items...
111     for (int pos = 0; pos < otherSize; pos++) {
112       // retrieve current item from the other database
```

```
113         Entry e = data.getEntry(pos);
114
115         // if this entry is not in our current database,
116         // add it
117         if (!entryExists(e))
118           addEntry(e);
119       }
120
121       // return the new size of the database
122       return getSize();
123     } // union(ImprovedDatabase)
124
125     // ========================================================
126     // Methods from interface "AdditionalOperations"
127     // ========================================================
128
129     /**
130      * Returns the array of entries that match string str in
131      * column col.
132      *
133      * Example: selectXFrom(Column.PHONE_NUMBER, "0173") will
134      * return all entries which contain "0173" in their phone
135      * number field.
136      *
137      * @param col the column to be used for the select operation
138      * @param str the String to be matched. The matching is
139      * case-insensitive and looks for substrings, not complete
140      * matches or only those at the beginning or end.
141      * @return the array of all matching entries where column col
142      * contains the (case-insensitive) match of str "somewhere"
143      * in the column.
144      */
145     public Entry[] selectXFrom(Column col, String str) {
146       // first, convert our search String to lower case
147       String searchFor = str.toLowerCase();
148
149       // the number of elements we currently have
150       int size = getSize();
151
152       // the number of elements for our result, starts at 0
153       int resultSize = 0;
154
155       // used to mark those elements that will be copied
156       // Note: had we been allowed to use Vector or other
157       // collections, we could have stored selected
158       // values there and then copied them to our target
159       // array on completion...
160       boolean[] selectThese = new boolean[getSize()];
161
162       // iterate over all elements in our database...
163       for (int index = 0; index < size; index++) {
164         // retrieve the current element value
165         String value = getContentFor(col, index);
166
167         // the entry matches only iff:
168         // the (already lower-cased) String "searchFor" is
169         // contained (using 'indexOf') in the
170         // lower-cased current value
```

```
171         if (value.toLowerCase().indexOf(searchFor) != -1) {
172           // mark as selected
173           selectThese[index] = true;
174 
175           // increment counter for result length
176           resultSize++;
177         }
178       } // end for loop
179 
180       // Now we can declare the result Entry[] with the correct size
181       Entry[] result = new Entry[resultSize];
182 
183       // the current position in our result array
184       int resultPos = 0;
185 
186       // loop over the array (again...)
187       for (int pos = 0; pos < size; pos++) {
188         // if the entry at position pos is marked, copy it
189         if (selectThese[pos])
190           result[resultPos++] = getEntry(pos);
191       }
192 
193       // finally, return the selected result elements
194       return result;
195     } // selectXFrom(Column, String)
196 
197     /**
198      * Sorts the given array using BubbleSort, based on
199      * the selected column
200      *
201      * @param the column used for sorting the elements by
202      */
203     public void sort(Column col) {
204       // the size of the database
205       int size = getSize();
206 
207       // BubbleSort: start with index i at end
208       for (int i = size; i > 0; i--) {
209         // Start with index j at position 1
210         for (int j = 1; j < i; j++)
211           // check if entries[j - 1] > entries[j]
212           // we need to adapt this to our database...
213           // 1. Instead of accessing the entry, we use
214           //    our "getContentFor(col, pos)" method.
215           // 2. Instead of ">", we use "compareTo" and
216           //    compare the result for "> 0"
217           if (getContentFor(col, j - 1).compareTo(
218               getContentFor(col, j)) > 0) {
219             // we need to swap the values, so do so!
220             swap(j - 1, j);
221           }
222       }
223     } // sort(Column)
224 
225     /**
226      * Extracts the entry field 'col' from the given
227      * entry in the database
228      *
```

```
229     * @param col the column to be used
230     * @param index the index of the database to be used
231     */
232    private String getContentFor(Column col, int index) {
233      // ensure that the index is valid
234      if (!isValidAccess(index))
235        return "invalid";
236
237      // Retrieve the entry at the selected index
238      Entry entry = getEntry(index);
239
240      // return the proper field by switching over col
241      switch(col) {
242        case ADDRESS_CITY:
243          return entry.getAddress().getCity();
244
245        case ADDRESS_HOUSE_NUMBER:
246          return entry.getAddress().getHouseNumber();
247
248        case ADDRESS_STREET_NAME:
249          return entry.getAddress().getStreetName();
250
251        case ADDRESS_ZIP_CODE:
252          return entry.getAddress().getZipCode();
253
254        case JOB:
255          return entry.getJob();
256
257        case GIVEN_NAME:
258          return entry.getGivenName();
259
260        case FAMILY_NAME:
261          return entry.getFamilyName();
262
263        case PHONE_AREA_CODE:
264          return entry.getPhoneNumber().getAreaCode();
265
266        case PHONE_NUMBER:
267          return entry.getPhoneNumber().getNumber();
268      }
269      return "invalid";
270    } // getContentFor(Column, int)
271  }
```