TECHNISCHE UNIVERSITÄT MÜNCHEN
FAKULTÄT FÜR INFORMATIK

Lehrstuhl für Angewandte Informatik / Kooperative Systeme
Praktikum Grundlagen der Programmierung
Prof. Dr. A. Brüggemann-Klein, R. Palenta, J. Kranz, Dr. A. Reuss

WS 15/16
Übungsblatt 8
04.12.2015

Abgabe: 14.12.2015 (vor 5:00 Uhr)

Hinweis: Implementieren Sie alle Hausaufgaben in einem Package namens **blatt08**.

Aufgabe 8.1 (P) Vererbung

Gegeben seien folgende Java-Klassen:

```
interface I {
   public void f();
}
interface J {
   public void g();
}
class A implements I {
   public A() { System.out.println("A"); }
   public void f() { System.out.println("f() in A"); h(); }
   public void g() { System.out.println("g() in A"); }
   public void h() { System.out.println("h() in A"); }
}
class B extends A {
   public B() { super();    }
   public void h() { System.out.println("h() in B");    }
   public void g() { System.out.println("g()in B"); super.h(); }
}
class C {
   public void h() { System.out.println("h() in C"); }
}
class D extends B implements I,J {
   public void f() { System.out.println("f() in D"); super.f(); g(); }
   public void h() { System.out.println("h() in D"); }
}
```

a) Welche Anweisungen werden vom Compiler erfolgreich übersetzt? Begründen Sie Ihre Antworten!

i) `I a = new A();` ..... ☐ Richtig ☐ Falsch

ii) `A a = new A(); B b = a;` ..... ☐ Richtig ☐ Falsch

iii) `B b = new D(); J j = (D) b;` ..... ☐ Richtig ☐ Falsch

iv) `B b = new B(); I i = b;` ..... ☐ Richtig ☐ Falsch

v) `C c = new B();` ..... ☐ Richtig ☐ Falsch

vi) `D d = new B();` ..... ☐ Richtig ☐ Falsch

b) Welche Ausgaben produzieren die folgenden Anweisungen? Begründen Sie Ihre Antworten!

i) `A a = new A(); a.f();`

.....

.....

.....

.....

.....

ii) `A a = (A) new B(); a.f();`

.....

.....

.....

.....

.....

iii) `B b = new B(); b.g();`

.....

.....

.....

.....

.....

iv) `C c = new C(); c.h();`

.....

.....

.....

.....

.....

v) `D d = new D(); d.h();`

.....

.....

.....

.....

.....

vi) `D d = new D(); d.f();`

.....

.....

.....

.....

.....

Aufgabe 8.2 (P) Fabelwesen

Themengebiet: Abstrakte Klassen, Interfaces

Java erlaubt nur Einfachvererbung. Gelegentlich ist für Objekte aber eine Zugehörigkeit zu unterschiedlichen Objektgruppen wünschenswert. Hierzu dienen Schnittstellenbeschreibungen, die in Java `interface`s heißen. Klassen können mehrere solcher Schnittstellen implementieren. Die Objekte einer solchen Klasse können dann von Variablen vom Typ einer der

implementierten Schnittstellen aufgenommen werden.

a) Entwickeln Sie ein (einfaches) Vererbungsmodell über Lebewesen. Dieses soll folgende Gattungen von (Fabel-)wesen mit zugehörigen Eigenschaften/Tätigkeiten enthalten:

`Hamster`:	futtern	`Vogel` :	futtern, fliegen
`Drache` :	futtern, fliegen, feuerspeien	`Monster`:	futtern, feuerspeien

Gemeinsame Eigenschaften mehrerer Gattungen sollen in Ihrem Modell in geeigneter Weise in gemeinsamen Oberklassen bzw. Schnittstellen (Interfaces) zusammengeführt und dort als Methoden spezifiziert werden.

b) Implementieren Sie die Klassen Ihres Modells. Die Methoden sollen nur Textausgaben produzieren (z.B. „Hamster futtert.“, „Drache fliegt.“).

Aufgabe 8.3 (P) Properties

Themengebiet: Properties-Dateien lesen und schreiben

In dieser Aufgabe wollen wir ein kleines Spiel programmieren und dabei den aktuellen Spielstand abspeichern. Dieser soll dann später wieder geladen werden, um das Spiel fortzuführen, als hätte es keine Unterbrechung gegeben.

Hinweis: Mit `Integer.parseInt(s)` lässt sich ein String `s`, welcher eine Ganzzahl darstellt, in die Ganzzahl umwandeln.

Sie können das Spiel selbst programmieren oder die Vorlage `GuessTheNumber.java` von der Praktikums-Webseite nutzen, in der noch die Laden-/Speichern-Funktionalität fehlt.

a) Im Hauptprogramm soll eine „Raterunde“ gestartet werden gemäß den folgenden Regeln, die wiederholt angewendet werden, bis die Raterunde abgebrochen wird.

b) Die Spielerin soll eine Zahl zwischen 1 und 7 raten und eingeben. Wird die richtige Zahl geraten, soll eine Erfolgsmeldung ausgegeben und die Raterunde beendet werden. (Ansonsten wird weitergeraten.) Wird eine -1 eingegeben, so soll die Raterunde erfolglos abgebrochen werden.

c) *Cheat-Modus*:
   - Bei Eingabe von 0: Ausgabe des Spielstandes (Menge der geratenen Zahlen)
   - Bei erneuter Eingabe von 0: Zu ratende Zahl ausgeben.

d) Wahlweise (Eingabe von 8) soll das Spiel gespeichert werden. Die Spielerin soll hierzu dazu aufgefordert werden, einen Dateinamen anzugeben, unter dem das aktuelle Spiel gespeichert werden soll. Verwenden Sie zum Speichern `java.util.Properties` (siehe Suchmaschine Ihrer Wahl). Überlegen Sie sich ein geeignetes Format zum Speichern. Eine Möglichkeit besteht z.B. darin, allen Zahlen einen Wert zuzuordnen, der anzeigt, ob die Zahl schon geraten wurde und ob es die zu ratende Zahl ist.

e) Finden Sie die Datei, in der der Spielstand abgelegt wurde, auf Ihrem Computer (bzw. in der Entwicklungsumgebung). Schauen Sie sich den Inhalt an und versuchen Sie, ihn zu verstehen.

f) Erweitern Sie die Funktionalität zum Speichern, so dass der gesamte Zustand erfasst wird. Insbesondere soll vermerkt werden, ob die letzte Eingabe der Spielerin ein 0 war.

g) Bei Eingabe von 9 soll ein gespeichertes Spiel wieder geladen werden. Fragen Sie dazu nach dem Dateinamen, lesen Sie den Inhalt mittels `java.util.Properties` ein und führen Sie das Spiel an der unterbrochenen Stelle fort, als ob es die Unterbrechung nie gegeben hätte.

Hinweis: Java-Properties-Dateien werden voraussichtlich beim Abschlussprojekt erneut Verwendung finden.

Aufgabe 8.4 (P) equals

Themengebiet: Vererbung, Polymorphie

Im Unterschied zur *Referenzgleichheit* soll in dieser Aufgabe das Thema *strukturelle Gleichheit* vertieft werden. Gegeben sei dazu die Klasse `Point`:

```
public class Point {
  public final int x;
  public final int y;
  public Point(int x, int y) {
    this.x = x;
    this.y = y;
  }
}
```

Im Folgenden soll die korrekte Prüfung struktureller Gleichheit schrittweise implementiert werden.

a) Lesen Sie zunächst die Dokumentation von `Object.equals` unter `http://docs.oracle.com/javase/6/docs/api/java/lang/Object.html#equals(java.lang.Object)` und arbeiten Sie gemeinsam die Bedingungen heraus, die Sie erfüllen müssen, wenn Sie in Ihrer Klasse `Point` die `equals`-Methode überschreiben wollen. Handelt es sich außerdem bei der standardmäßigen Implementierung von `Object.equals` um Referenz- oder strukturelle Gleichheit? Warum?

b) Fügen Sie Ihrer Klasse `Point` nun eine Methode `boolean equals(Point p)` hinzu (**Achtung**: Die korrekte *Signatur* zum Überschreiben der `equals`-Methode ist `equals(Object)`; die „falsche“ Variante dient hier nur Übungszwecken!) und implementieren Sie diese anhand der herausgearbeiteten Bedingungen. Testen Sie, ob Ihre Implementierung diese erfüllt!

c) Betrachten Sie anschließend das folgende Codefragment:

```
Point p1 = new Point(-1, 4);
Point p2 = new Point(-1, 4);

boolean result;
Object op2 = p2;
result = p1.equals(p2);   // true
result = p1.equals(op2);  // false
```

Warum funktioniert der erste Vergleich, der zweite aber nicht? Stellen Sie der Deklaration Ihrer `equals`-Methode die Annotation `@Override` voran, um sich dieses *typische Problem* zu verdeutlichen. Korrigieren Sie Signatur und Rumpf Ihrer Methode.

Soweit, so gut. Aber was, wenn nun die Klassenhierarchie erweitert wird? Gegeben sei dazu die Klasse `ColoredPoint`:

```
public class ColoredPoint extends Point {
  public final int r;
  public final int g;
  public final int b;
  public ColoredPoint(int x, int y, int r, int g, int b) {
    super(x, y);
    this.r = r;
    this.g = g;
    this.b = b;
  }
}
```

d) Überschreiben Sie analog zur Klasse `Point` die `equals`-Methode in der Klasse `ColoredPoint`. Achten Sie darauf, auch die neu hinzugekommenen Attribute zu testen. Rufen Sie zum Vergleich der Attribute der Superklasse deren eigene `equals`-Methode auf!

e) Betrachten Sie anschließend das folgende Codefragment:

```
Point p1 = new Point(-1, 4);
ColoredPoint cp1 = new ColoredPoint(-1, 4, 1, 0, 0); // x,y,r,g,b

boolean result;
result = p1.equals(cp1);     // true
result = cp1.equals(p1);     // false
```

Hier ist offensichtlich die *Symmetrie* verletzt. Dies ließe sich z.B. dadurch umgehen, dass man im Rumpf von `ColoredPoint.equals` unterscheidet, ob es sich bei dem zu vergleichenden Objekt um eine Instanz von `ColoredPoint` oder `Point` handelt, um dann abhängig davon die Objekte miteinander zu vergleichen, oder aber, indem man sich dafür entscheidet, dass Instanzen unterschiedlicher Klassen niemals gleich sein können (die genaue Klasse lässt sich durch `Object.getClass()` ermitteln).
Diskutieren Sie, warum im ersten Fall die *Transitivität* verletzt sein und welche Einschränkungen hinsichtlich der Hierarchie der zweite Fall nach sich ziehen könnte.

f) Eine mögliche Lösung ist die Definition einer Hilfsmethode `boolean canEqual(Object obj)` für *jede* Klasse C, die `equals` überschreibt. Diese Methode würde `true` oder `false` zurückgeben, je nachdem, ob es sich bei `obj` um eine Instanz der Klasse C bzw. einer Unterklasse davon handelt. Beispiel für die Klasse `ColoredPoint`:

```
public boolean canEqual(Object obj) {
  return obj instanceof ColoredPoint;
}
```

So ließe sich von `equals` aus feststellen, ob sich die Objekte in beiden Richtungen miteinander vergleichen lassen.

Definieren Sie `canEqual` analog in beiden Klassen `Point` und `ColoredPoint`. Verwenden Sie diese Methode in Ihrer Implementierung `equals` an geeigneter Stelle, um festzustellen, ob sich beide Objekte auch in umgekehrter Richtung vergleichen lassen.

Optional: Eine ausführlichere Problembeschreibung (inkl. weiterer „Baustellen“ wie bspw. `hashCode`) finden Sie unter: `http://www.artima.com/lejava/articles/equality.html`

Aufgabe 8.5 (P) Vererbung: Zusatzaufgabe

Bestimmen Sie für das folgende Programm, welche Methoden ausgeführt werden und was ausgegeben wird:

```
class A {
    public void someMethod() { System.out.println("A.someMethod()"); }
    public void someMethod(A a) { System.out.println("A.someMethod(A a)");
        }
    public void someMethod(B b) { System.out.println("A.someMethod(B b)");
        }
    public A anotherMethod() { return this; }
    public static void someStaticMethod() {
        System.out.println("A.someStaticMethod()");
    }
}

class B extends A {
    public void someMethod() { System.out.println("B.someMethod()"); }
    public void someMethod(A a) { System.out.println("B.someMethod(A a)");
        }
    public B anotherMethod() { return this; }
    public static void someStaticMethod() {
        System.out.println("B.someStaticMethod()");
```

```
    }
}

class C extends A {
    public void foo(B b) { ((A) b).someMethod(super.anotherMethod()); }
}

public class ExampleOne {
    public static void main(String[] args) {
        A a = new A();
        B b = new B();

        a.someStaticMethod();                                    // 1
        b.someStaticMethod();                                    // 2
        ((A) b).someStaticMethod();                              // 3
        A.someStaticMethod();                                    // 4
        B.someStaticMethod();                                    // 5

        a.someMethod();                                          // 6
        b.someMethod();                                          // 7
        ((A) b).someMethod();                                    // 8

        a.someMethod(a);                                         // 9
        a.someMethod(b);                                         // 10
        b.someMethod(a);                                         // 11
        b.someMethod(b);                                         // 12

        a.someMethod((A) b);                                     // 13
        b.someMethod((A) b);                                     // 14

        System.out.println(a.anotherMethod());                   // 15
        System.out.println(b.anotherMethod());                   // 16
        System.out.println(((A) b).anotherMethod());             // 17

        System.out.println(a.anotherMethod() instanceof A);      // 18
        System.out.println(a.anotherMethod() instanceof B);      // 19
        System.out.println(b.anotherMethod() instanceof A);      // 20
        System.out.println(b.anotherMethod() instanceof B);      // 21

        (new C()).foo(b);                                        // 22
    }
}
```

Aufgabe 8.6 [4 Punkte] **(H)**

Themen:

- Vererbung mit abstrakten Klassen und Interfaces
- Polymorphie

Gegeben seien folgende Java-Klassen:

```
class A {
    public void f() { System.out.println("f() in A"); }
    public void g() { System.out.println("g() in A"); f(); }
}
class B extends A {
    public void f() { System.out.println("f() in B"); }
    public void h() { System.out.println("h() in B"); f(); g(); }
}
class C extends A {
    public void h() { System.out.println("h() in C"); f(); g(); }
}
interface I {
    public void m();
}
abstract class D {
    public abstract void n(boolean b);
    public void o() { System.out.println("o() in D"); n(false); }
}
class E extends D {
    public void n(boolean b) {
        System.out.println("n() in E");
        if(b) o();
    }
    public void m() { System.out.println("m() in E"); }
}
class F extends E implements I {
    public void n(boolean b) {
        System.out.println("n() in F");
    }
}
```

a) [2] Welche der folgenden Anweisungen werden vom Compiler erfolgreich übersetzt? Begründen Sie Ihre Antwort!

i) `A a = new A();` ..... ☑ Richtig ☐ Falsch

ii) `B b = new B(); A t = b;` ..... ☑ Richtig ☐ Falsch

iii) `B b = new A();` ..... ☐ Richtig ☑ Falsch

iv) `D d = new D();` ..... ☐ Richtig ☑ Falsch

v) `D d = new F();` ..... ☑ Richtig ☐ Falsch

vi) `F f = new F(); I i = f;` ..... ☑ Richtig ☐ Falsch

b) [2] Welche Ausgaben produzieren die folgenden Anweisungen? Begründen Sie Ihre Antwort!

i) `A a = new A(); a.g();`

ii) `B b = new B(); b.h();`

iii) `C c = new C(); c.g();`

iv) `B b = new B(); A a = b; a.g();`

v) `D d = new E(); d.n(true);`

vi) `F f = new F(); f.o();`

Aufgabe 8.7 [7 Punkte] **(H) Binäre Suche mit Zwischengesichtern**

Themen:

- Interfaces (inkl. nützliche Interfaces der Standardbibliothek)
- Polymorphie
- Sortieren und Suchen

In dieser Aufgabe geht es darum, die sogenannte *binäre Suche* für beliebige Objekte zu implementieren, die das Interface `Comparable` implementieren.

a) [5] Die binäre Suche ist ein Verfahren, effizient in einer Mengen von sortierten Daten zu suchen. Wir werden in dieser Aufgabe mit Listen arbeiten. Um in einer Liste binär zu suchen, schaut man sich zunächst das mittlere Element der Liste an und vergleicht es mit demjenigen Element, das man sucht. Ist das gesuchte Element kleiner, so fährt man in der ersten Hälfte der Liste fort. Ist das gesuchte Element größer, so fährt in der zweiten Hälfte der Liste fort. Innerhalb der jeweiligen Hälften sucht man das Element erneut mit der gleichen Herangehensweise. Die binäre Sucht endet, wenn die jeweilige Teilliste leer ist oder nur ein Element enthält. In diesem Fall hat man das Element gefunden oder es befindet sich nicht in der Liste.

   Implementieren Sie eine Methode `boolean binarySearch(LinkedList<Comparable> list, Comparable value)`, welche die binäre Suche implementiert und zum Vergleich von Objekten die Methode `int compareTo(Object other)` verwendet, die sich im Interface `Comparable` befindet.

b) [1] Auf dem letzten Übungsblatt haben die die Klasse `Book` implementiert, die über eine Methode `int compareTo(Object book)` verfügt. Erweitern Sie diese Klasse um die Methode `int compareTo(Object book)` und implementieren Sie diese entsprechend der bereits vorhandenen Methode (Sie dürfen die Musterlösung des letzten Blattes ab Verfügbarkeit ebenfalls verwenden). Leiten Sie Ihre `Book`-Klasse zusätzlich vom Interface `Comparable` ab.

c) [1] Schreiben Sie ein Hauptprogramm, welches ein Element in einer sortierten Listen von Büchern und einer sortierten Liste von `Integer`-Zahlen sucht, indem es den von Ihnen implementierten Suchalgortihmus verwendet.

Hinweis: Sie brauchen in dieser Aufgabe nicht auf Fehlerbehandlung zu achten; einen bestimmten Wert in einer Liste nicht zu finden, ist allerdings kein Fehler.

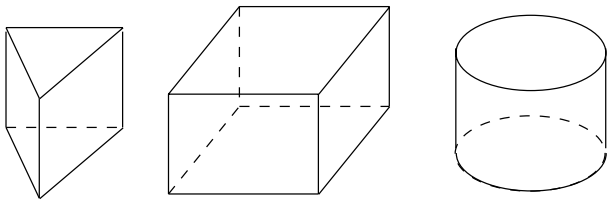
Hinweis: Sie sollten die Warnung „*Comparable is a raw type. References to generic type Comparable<T> should be parameterized*“ ignorieren.

Aufgabe 8.8 [7 Punkte] **(H)**

Themen:

- Modellierung
- Vererbung und Polymorphie

Die folgende Abbildung zeigt als Beispiele für Prismen ein Dreiecksprisma, einen Quader und einen Zylinder, die durch Verschiebung eines Dreiecks, eines Rechtecks bzw. eines Kreises entstehen:



Prismen sind durch ihre Höhe und ihre jeweilige Grundfläche bestimmt. Als Grundflächen in Frage kommen (zum Beispiel):

- *regelmäßige n-Ecke*; bestimmt durch die Anzahl der Ecken und die Seitenlänge
- *Kreise*; bestimmt durch ihren Radius
- *Rechtecke*; bestimmt durch Breite und Länge

An Operationen muss eine Grundfläche, die Berechnung von Umfang und Flächeninhalt zur Verfügung stellen; ein Prisma die Berechnung von Oberfläche und Volumen.

In dieser Aufgabe soll eine Klassenhierarchie für diese geometrischen Körper in Java implementiert werden.

a) [1] Erstellen Sie die Klasse `Grundflaeche`, als Oberklasse für alle Grundflächen. Legen Sie Basisversionen der Methoden `umfang()` und `flaeche()`, sowie eine geeignete `toString()`-Methode an.
b) [1] Erstellen Sie Klassen `Kreis`,`Rechteck` und `NEck` zur Repräsentation von Kreisen, Rechtecken und n-Ecken. Diese Klassen erweitern `Grundflaeche` um die benötigten Werte und überschreiben die Methoden zur Flächen- und Umfang-berechnung. Ergänzen Sie die `toString()`-Methode sinnvoll.
c) [1] Erstellen Sie die Klasse `Prisma` mit den benötigten Feldern und den Methoden `volumen()` und `oberflaeche()`, sowie einer geeigneten `toString()`-Methode.
d) [1] Ergänzen Sie alle Grundflächen um die Methode `istQuadrat()`, die zurückgibt, ob es sich bei der aktuellen Instanz um ein Quadrat handelt. Z.B. falls bei einem Rechteck Länge und Breite gleich sind.
e) [1] Erstellen Sie die Klasse `Quadrat`, welche durch ihre Seitenlänge eindeutig bestimmt ist, als weitere Unterklasse von `Grundflaeche` und ergänzen Sie alle Grundflächen um einem Methode `zuQuadrat()`, die, falls es sich um ein Quadrat (auch passende Rechtecke und n-Ecke) handelt, ein `Quadrat`-Object mit der entsprechenden Seitenlänge zurückgibt. Handelt es sich nicht um ein Quadrat, dann wird `null` zurückgegeben.
f) [1] Ergänzen Sie die Klasse `Prisma` um eine Methode `istWuerfel()`, die zurückgibt, ob es sich bei dem aktuellen Prisma um einen Würfel handelt.
g) [1] Testen Sie Ihren Code durch ein geeignetes Hauptprogramm, welches sämtliche Funktionalität verwendet.

Hinweise:

- Planen Sie vor dem Programmieren die Klassenhierarchie.
- Fassen Sie gleichartige Attribute und Methoden in einer geeigneten Oberklasse zusammen.
- Verwenden Sie Getter- und ggf. Setter-Methoden.
- Greifen Sie falls möglich auf bereits implementierte Methoden zurück.
- Die Fläche eines regelmäßigen n-Ecks mit Seitenlänge a ist $\frac{n*a^2}{4*\tan(\frac{\pi}{n})}$.
- Die Java-Klasse `Math` stellt in der Klassenvariablen `Math.PI` einen Wert für π sowie in der Klassenmethode `Math.tan()` die Berechnung des Tangens zur Verfügung.
- Testen Sie ihre Implementierung mit einer geeigneten Test-Klasse.