Datum -Zeit: 05.02.2016 – Beginn 11:00 Uhr

Nettobearbeitungszeit: 120 Min

Punkteverteilung

	Punkte	Erreicht
Teil A		
1.	10	
2.	5	
3.	10	
4.	10	
5.	14	
6.	10	
7.	10	
Teil A Gesamt	**69**	
Teil B		
B-1	**Summe (45)**	
1.	12	
2.	9	
3.	2	
4.	11	
5.	4	
6.	7	
B-2	**Summe (16)**	
1.	2	
2.	1	
3.	13	
B-3	**Summe (16)**	
1.	10	
2.	6	
Teil B Gesamt	***77***	

120 Punkte entsprechen der Note 15.

Viel Erfolg! ☺

Teil A: Verständnisfragen

1. Es gibt in Java drei Sprachbestandteile, die nicht objektorientiert sind. (**10Pkt**)
   a. Nennen Sie diese Bestandteile.

   b. Erklären Sie warum die Eigenschaften unter a. nicht den OO-Prinzipien entsprechen. (Stichpunkte)

2. Wie heißt der jeweilige Mechanismus, der in Java die folgenden Zuweisungen erlaubt? (**5Pkt**)

   a. `Double d1 = 13.7;`

   b. `double d2 = d1;`

   c. `d2 = 4*2;`

   d. `Collection<Integer> ci = new ArrayList<Integer>();`

   e. `List<Integer> li = (List)ci;`

3. Markieren Sie die nachfolgenden Aussagen mit **w**, wenn es sich um eine korrekte Aussage und mit **f**, wenn es sich um eine falsche Aussage handelt. (**10 Pkt**)
   a. Methoden mit Sichtbarkeit **package private** werden von allen ableitenden Klassen gesehen. ( )
   b. Private Instanz-Variablen können von Objekten der Klassen des gleichen Packages gelesen werden. ( )
   c. ***this*** in der Bedeutung eines Methodenaufrufs gibt es nur im Konstruktor. ( )
   d. ***this*** in der Verwendung als Objektreferenz gibt es nur in Instanz-Methoden, die nicht Konstruktoren sind. ( )
   e. ***this*** in statischen Methoden ist eine Referenz auf das Klassenobjekt. ( )
   f. Alle Klassen in Java müssen mindestens einen Konstruktor haben. ( )
   g. Alle Instanz-Variablen werden statisch gebunden. ( )
   h. Nur ***public*** und ***protected*** Methoden werden dynamisch gebunden. ( )
   i. Immutable Objekte sind unique. ( )
   j. Der statische Typ bestimmt, welche Methodenaufrufe für ein Objekt erlaubt sind. ( )

4. Gegeben das Interface ***ColorModel*** sowie ein Auszug aus den Implementierungen für das Interface: ***AbstractColorModel,*** ***RGBColor*** und ***HSVColor***. ***RGBColor*** stellt Farben als Kombination von Rot-, Grün-, Blau-Werten dar. ***HSVColor*** stellt Farben als Kombination von Hue (Farbwert), Saturation (Sättigung), und Brightness (Helligkeit) dar. Farbdarstellungen im RGB-Modell und HSV-Modell können verlustfrei ineinander umgerechnet werden. (**10 Pkt**)
   a. Ergänzen Sie Klasse ***RGBColor*** um einen **Kopier-Konstruktor**, dessen Parameter einen möglichst allgemeinen aber hinreichend einschränkenden Typ hat. (**3Pkt**)
   b. Implementieren Sie den Kopier-Konstruktor ohne vorhandenen Source-Code zu duplizieren. (**3Pkt**)
   c. Ergänzen Sie die ***equals***-Methode in der Klasse ***AbstractColorModel*** um eine möglichst allgemeine aber hinreichend einschränkende Typ-Prüfung und überführen Sie das Argument der ***equals***-Methode in einen geeigneten Typ. (**4Pkt**)

```
public interface ColorModel {
    // für das RGB Farbmodell
    public double getRed();
    public double getGreen();
    public double getBlue();
    // für das HSV Farbmodell;
    public double getHue();
    public double getSaturation();
    public double getBrightness();

}

public class RGBColor extends AbstractColorModel implements ColorModel {

    private double red;
    private double green;
    private double blue;

    public RGBColor(double red, double green, double blue) {
       this.red = red;
       this.green = green;
       this.blue = blue;
    }









    // ... Implementierung der Interface-Methoden ausgelassen
}

public class HSVColor extends AbstractColorModel implements ColorModel {
    private double hue;
    private double saturation;
    private double brightness;

    public HSVColor(double hue,double saturation, double brightness){
       this.hue = hue;
       this.saturation = saturation;
       this.brightness = brightness;
    }
   // ... Implementierung der Interface-Methoden ausgelassen
}
```

```
public abstract class AbstractColorModel implements ColorModel {
    @Override
    public int hashCode() {
        final int prime = 31;
        int result = 1;
        result = prime * result + Double.hashCode(getRed());
        result = prime * result + Double.hashCode(getGreen());
        result = prime * result + Double.hashCode(getBlue());
        return result;
    }

    @Override
    public boolean equals(Object obj) {
       if (this == obj)
              return true;
       if (obj == null)
              return false;









       return (Double.compare(getRed(), other.getRed()) == 0)
              && (Double.compare(getGreen(), other.getGreen()) == 0)
              && (Double.compare(getBlue(), other.getBlue()) == 0);
    }
}
```

5. **Varianz und generische Typen**: (**14 Pkt**)
   a. Geben Sie ein Beispiel für ungeschützte Kovarianz von Arrays in Java. (**4Pkt**)

   b. Markieren Sie in a. die Stelle, die zu einem Fehler führt. (**1Pkt**)
   c. Wann tritt dieser Fehler auf? (**1Pkt**)

   d. Bei generischen Typen gibt es in Java keine ungeschützte Kovarianz. Zeigen Sie am Beispiel, wie dies sichergestellt wird. (Markieren Sie die Stelle, die einen Compilerfehler erzeugt.) (**4Pkt**)

   e. Gegeben die statische generische Methode ***copyAIntoB***, die alle Elemente der Liste ***a*** an die Liste ***b*** anhängt. Ergänzen Sie das Typargument der Liste ***a*** in der Signatur der Methode mit einem maximalen aber korrekten Typausdruck. (Wenn z.B. ***a*** vom Typ ***List<Integer>*** und ***b*** vom Typ ***List<Number>*** ist, dann soll der Aufruf ***copyAIntoB(a,b)*** erlaubt sein.) (**2Pkt**)

```
public static <T> void  copyAIntoB(List<          > a, List<T> b){
          b.addAll(a);
}
```

   f. Gegeben die statische Methode ***countElems***, die die Anzahl der Elemente einer Collection berechnet. Ergänzen Sie das Typargument von ***aCol*** mit einem geeigneten generischen Typausdruck. (**2Pkt**)

```
public static int countElems(Collection<         > aCol) {
          return aCol.size();
}
```

6. **Statisches Binden und Effekte des Überschattens**: Gegeben die Klassen ***X*** und ***Y***, sowie eine ***main*** Methode. Welche Ausgaben erzeugt die ***main***-Methode, wenn package private Instanz-variablen statisch gebunden werden. (**10Pkt**)

```
class X {
   int val;
   public void add(int toAdd) {
          this.val += toAdd;
   }
   public int getVal() {
          return val;
   }
}

class Y extends X {
   int val;
   public Y(int value) {
          this.val = value;
   }
   @Override
   public String toString() {
          return "Y="+val;
   }
}


 public static void main(String[] args) {

   X y1 = new Y(10);
   Y y2 = new Y(100);

   System.out.println("y1");
   System.out.println(y1.val);
   System.out.println(y1.getVal());
   System.out.println(((Y)y1).val);
   System.out.println(y1);
   System.out.println();
   System.out.println("y2");
   System.out.println(y2.val);
   System.out.println(y2.getVal());
   System.out.println(((X)y2).val);
   System.out.println(y2);
   System.out.println();

   y2.add(5);
   System.out.println(y2.val);
   System.out.println(y2.getVal());
   System.out.println();

}
```

7. **Konkrete Klassen, Abstrakte Klassen, Interfaces**: Gegeben ein Interface, eine abstrakte Klasse und eine konkrete Klasse. Der Quelltext enthält eine Reihe von Fehlern, aber auch korrekte Implementierungen. Markieren Sie die korrekten Methoden mit eine großem **K** und die Fehler mit einem großen **F**. Korrigieren Sie die Fehler durch minimale Änderungen. Sie können falsche Angaben streichen und ersetzen. Es sind aber auch Ergänzungen vorzunehmen. (**10 Pkt**)

```
interface I {
    public Number getValue();
    public I c(I it);
    public I a(I n);
    public I m(I d);
    public I m(I... l);
    public I k(Double d);
}

abstract class AbstractI implements I {
    @Override
    public abstract boolean d();
    @Override
    public I m(I... ia) {
          return Stream.of(ia).reduce((acc, n1) -> acc.m(n1)).orElse(null);
    }
}

class ConcreteA extends AbstractI {
    private Integer value;

    public ConcreteA(Integer value) {
          this.value = value;
    }
    @Override
    public Integer getValue() {
          return value;
    }
    @Override
    public ConcreteA c(AbstractI it) {
          return new ConcreteA(it.getValue().intValue());
    }
    @Override
    public ConcreteA a(I it) {
          return new ConcreteA(value + it.getValue().intValue());
    }
    @Override
    public I m(I d) {
          return this.m(c(d));
    };
    @Override
    public ConcreteA m(ConcreteA ca) {
          return new ConcreteA(value * ca.value);
    }
    @Override
    public I k(double d) {
          return new ConcreteA((int) (value / d));
    }

}
```