Dr. Annette Bieniusa
Mathias Weber, M. Sc.
Peter Zeller, M. Sc.

TU Kaiserslautern
Fachbereich Informatik
AG Softwaretechnik

Übungsblatt 8: Software-Entwicklung 1 (WS 2017/18)

Ausgabe: 11.12.17
Abgabe: 18.12.17

Melden Sie sich bitte im Stats-System zur Probeklausur an, falls Sie dort teilnehmen wollen. Weitere Informationen zur Probeklausur finden Sie auf der Vorlesungsseite.

Aufgabe 1 Subtyping und dynamisches Binden (8 Punkte)

```
1  interface A {
2    String m();
3  }
4
5  interface B {
6    String m();
7  }
8
9  interface C extends A {
10   String p();
11 }
12
13 interface D extends A, B {
14   String q();
15 }
16
17 class E implements B {
18   public String m() {
19     return "Em";
20   }
21   public String r() {
22     return "Er";
23   }
24 }
25 class F implements C, D {
26   public String m() {
27     return "Fm";
28   }
29   public String p() {
30     return "Fp";
31   }
32   public String q() {
33     return "Fq";
34   }
35   public String r() {
36     return "Fr";
37   }
38 }
39
40
41 public class Subtyping {
42   public static void main(String[] args) {
43
44     // ...
45
46   }
47 }
```

Betrachten Sie die folgenden Code-Beispiele, welche in der `main`-Methode in Zeile 44 eingefügt werden sollen. Entscheiden Sie jeweils, ob das Beispiel vom Java akzeptiert wird oder nicht. Falls das Beispiel akzeptiert wird, geben Sie die Ausgabe des Programms an. Andernfalls erklären Sie, warum es nicht akzeptiert wird.

a)
```
A x = new A();
System.out.println(x.m());
```

b)
```
E x = new E();
System.out.println(x.r());
```

c)
```
A x = new F();
System.out.println(x.m());
```

d)
```
A x = new F();
System.out.println(x.r());
```

e)
```
B x = new E();
B y = new F();
System.out.println(x.r() + y.r());
```

f)
```
B x = new E();
B y = new F();
System.out.println(x.m() + y.m());
```

g)
```
B x = new F();
A y = x;
System.out.println(x.m() + y.m());
```

h)
```
D x = new F();
System.out.println(x.m() + x.p());
```

Aufgabe 2 Figuren (6 Punkte)

Gegeben ist das Interface `Figur` und die Klassen `Rechteck` und `Kreis`:

```
interface Figur {

}

class Rechteck implements Figur {
    private final double x;
    private final double y;
    private final double width;
    private final double height;

    Rechteck(double x, double y, double width, double height) {
        this.x = x;
        this.y = y;
        this.width = width;
        this.height = height;
    }
}

class Kreis implements Figur {
    private final double x;
    private final double y;
    private final double radius;

    Kreis(double x, double y, double radius) {
        this.x = x;
        this.y = y;
        this.radius = radius;
    }
}
```

Schreiben Sie eine Prozedur `static double flaeche(Figur[] figuren)` in einer Klasse `Figuren`, welche ein Array von Figuren nimmt und die Gesamtfläche berechnet, die von den Figuren eingenommen wird (also die Summe der Flächen der einzelnen Figuren im Array). Erweitern Sie dazu die Klassen und das Interface um passende Methoden. Verwenden Sie keine Casts.

Aufgabe 3 Interface für Listen (4 Punkte)

Laden Sie sich die Implementierungen zu `IntList` und `IntArrayList` aus den Vorlesungs-Materialien herunter.

a) Schreiben Sie ein Interface `ListOfInt`, welches die Methoden `add`, `get`, und `size` enthält. Passen Sie die Klassen `IntArrayList` und `IntList` so an, dass sie das Interface implementieren.

b) Erstellen Sie ein Interface `IteratorOfInt`, welches die gemeinsamen Methoden der beiden Iterator-Klassen beinhaltet.

c) Erweitern Sie das Interface `ListOfInt` um die Methode `IteratorOfInt iterator()` und passen Sie die Klassen entsprechend an.

Aufgabe 4 Monitoring (12 Punkte)

Eine Monitoring Anwendung beobachtet die Entwicklung von Daten-Strömen und reagiert bei auffälligen Änderungen. In dieser Aufgabe sollen Sie ein einfaches Monitoring-System nach der folgenden Beschreibung in Java implementieren und dazu Interfaces und Klassen verwenden.

Ein `Monitor` hat eine Methode `report`, mit der ein neuer Datensatz gemeldet werden kann. Ein Datensatz besteht dabei aus einem Zeitstempel (Typ **`long`**, Zeit in Millisekunden) und einem Wert vom Typ **`double`**. Außerdem hat der Monitor Methoden, um Trigger und Alerts hinzuzufügen.

Ein Trigger erkennt bestimmte Änderungen in den Daten und kann damit Alerts auslösen. Jedes mal, wenn mit der `report`-Methode ein Datensatz an den Monitor gemeldet wird, benachrichtigt der Monitor alle Trigger, die ihm hinzugefügt worden sind. Die Trigger prüfen jeweils den Datensatz und wenn einer oder mehrere Trigger feuern, werden alle hinzugefügten Alerts benachrichtigt. Sie können davon ausgehen, dass die Zeitstempel in den Aufrufen der `report`-Methode immer größer werden.

Das System soll erweiterbar sein, so dass später andere Trigger und Alerts hinzugefügt werden können. Für diese Aufgabe sollen nur zwei Arten von Triggern und zwei verschiedene Alerts implementiert werden:

Ein `AboveTrigger` ist ein Trigger, der über den Konstruktor `AboveTrigger(double bound)` erstellt wird. Er wird ausgelöst, wenn ein Datensatz mit einem Wert größer als `bound` gemeldet wird.

Ein anderer Trigger ist der `DeltaTrigger`, welcher über den Konstruktor `DeltaTrigger(long t, double maxChange)` erstellt wird und ausgelöst wird, wenn ein Datensatz-Wert sich in den letzten `t` Millisekunden um mehr als `maxChange` verändert hat.

Als Alert soll es einen `TextAlert` geben, welcher über den Konstruktor `TextAlert(String message)` erstellt wird. Wenn dieser ausgelöst wird soll er die Nachricht `message` auf der Konsole ausgeben. Dabei soll in `message` der String `%t` durch den zuletzt gemeldeten Zeitstempel ersetzt werden und `%v` durch den zuletzt gemeldeten Wert (gerundet auf zwei Nachkommastellen).

Ein weiterer Alert soll der `MailAlert` sein, welcher über den Konstruktor `MailAlert(String email, String message)` erstellt wird. Dieser soll beim auslösen zuerst die `email` in einer Zeile ausgeben und danach die `message` wie der `TextAlert`.

Implementieren Sie Ihre Klassen und Interfaces so, dass sie wie in folgendem Beispiel verwendet werden können:

```
Monitor m = new Monitor();
m.addTrigger(new AboveTrigger(39));
m.addTrigger(new DeltaTrigger(4000, 1));
m.addAlert(new TextAlert("%t ms: Value changed to %v"));

m.report(new Dataset(0, 38.4));
m.report(new Dataset(1000, 38.5));
m.report(new Dataset(2000, 38.5));
m.report(new Dataset(3000, 38.6));
m.report(new Dataset(4000, 39.1)); // AboveTrigger(39) feuert
m.report(new Dataset(5000, 38.9));
m.report(new Dataset(6000, 38.5));
m.report(new Dataset(7000, 38.0)); // DeltaTrigger(10000, 1) feuert
m.addAlert(new EmailAlert("hans@example.com", "ALERT!"));
m.report(new Dataset(8000, 38.0)); // DeltaTrigger feuert erneut
m.report(new Dataset(9000, 38.0));
```

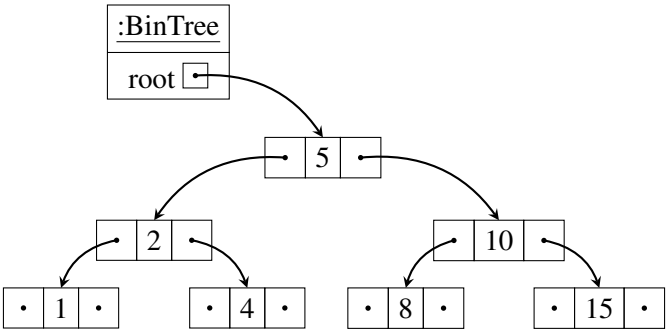
Die Ausgabe bei diesem Beispiel soll dann wie folgt aussehen:

```
4000 ms: Value changed to 39.10
7000 ms: Value changed to 38.00
8000 ms: Value changed to 38.00
hans@example.com
ALERT!
```

Aufgabe 5 Sortierte, markierte Bäume (Einreichaufgabe, 6 Punkte)

Verwenden Sie für diese Aufgabe keine Elemente aus der Java Standard-Bibliothek.

Laden Sie sich für diese Aufgabe die Dateien `BinTree.java` und `TreeNode.java` aus dem Material zur Vorlesung (Kapitel 13) herunter, welche eine Implementierung von **sortierten** Bäumen mit `int`-Markierung enthält. In dieser Aufgabe sollen Sie weitere Methoden zur Klasse `BinTree` hinzufügen.



a) Schreiben Sie eine Methode `int max()`, welche die maximale Markierung im Baum zurück gibt. Wenn der Baum leer ist soll `Integer.MIN_VALUE` zurückgegeben werden.

b) Schreiben Sie eine Methode `int size()`, welche die Anzahl der Einträge im Baum berechnet.

c) Schreiben Sie eine Methode `int[] toArray()`, welche ein aufsteigend sortiertes Array mit den Einträgen des Baums zurückgibt.