Begonnen am	Thursday, 25. June 2020, 08:30
Status	Beendet
Beendet am	Thursday, 25. June 2020, 09:30
Verbrauchte Zeit	59 Minuten 50 Sekunden
Bewertung	Bisher nicht bewertet

Frage **1**

Vollständig

Nicht bewertet

Erklärungen und Anweisungen

- 60 Minuten Zeit, insgesamt 60 Punkte!
- Schreiben Sie die geforderten Programmteile in die dafür vorgesehenen Eingabebereiche.
- Verwenden Sie für Einrückungen **KEINE Tabulatoren** sondern **Leerzeichen**!
- Bitte bestimmen Sie, ob Sie diesen praktischen Teil der Klausur **für die Vorlesung**, **für die Übung** oder **für beide** machen:

Wählen Sie eine oder mehrere Antworten:

- [x] Vorlesung
- [x] Übung

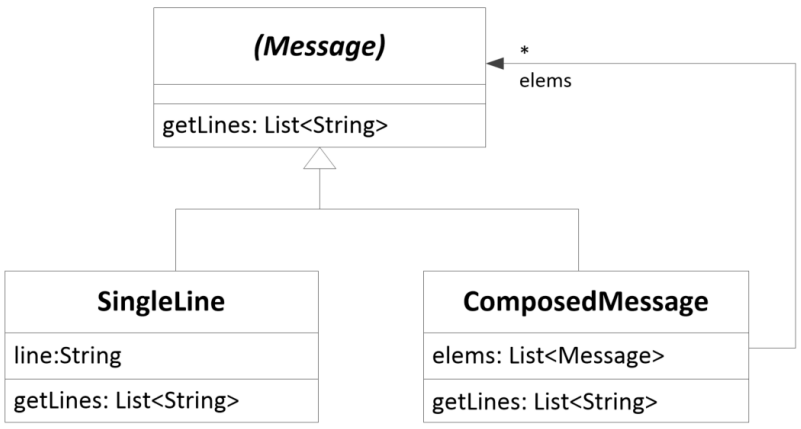
Die Antwort ist richtig.

Die richtigen Antworten sind: Vorlesung, Übung

Information

Aufgabe 1: Composite

Das folgende Klassensystem soll den hierarchischen Aufbau von Messages erlauben. *Message* stellt die abstrakte Basisklasse für Messages dar, *SingleLine* ist eine Message mit einer einzigen Zeile und *ComposedMessage* setzt sich aus einer Liste von Messages zusammen.



Message definiert eine abstrakte Methode *getLines*, welche eine flache Liste mit den Zeilen liefert.

```
public abstract class Message {
  public abstract List<String> getLines();
}
```

Folgende Anweisungen zeigen die Verwendung. Es wird eine hierarchische Message aufgebaut, dann mit der Methode *getLines* auf die Zeilen zugegriffen und diese ausgegeben:

```
Message msgFromDonald =
       new ComposedMessage(
         new SingleLine("Message from the greatest on
earth!"),
         new ComposedMessage(
           new SingleLine("I am the greatest!"),
           new SingleLine("And all others are losers."),
           new SingleLine("That's true!")
         ),
         new SingleLine("Your Donald")
       );

for (String line : msgFromDonald.getLines()) {
  System.out.println(line);
}
```

Implementieren Sie die Methode *getLines* für Klassen *SingleLine* und *ComposedMessage*.

Frage 2

Vollständig

Erreichbare Punkte: 3,00

```
public class SingleLine extends Message {

  public final String line;

  public SingleLine(String line) {
    this.line = line;
  }

  // TODO getLines

}
```

```
public List<String> getLines(){
  return new ArrayList().add(line);
}
```

Frage 3

Vollständig

Erreichbare Punkte: 5,00

```
public class ComposedMessage extends Message {

  public final List<Message> elems;

  public ComposedMessage(Message...elems) {
    this.elems = List.of(elems);
  }

  // TODO getLines

}
```

```
public List<String> getLine() {
  ArrayList<String> lines = new ArrayList[String]
  for (Message m : elems) lines.add(m.getLine());
  return lines;
}
```

Information

Aufgabe 2: Funktion höherer Ordnung

Implementieren Sie nun für Messages aus Aufgabe 1 zusätzlich eine Methode *forEach*, die für jede Zeile eine Aktion ausführt. Die auszuführende Aktion wird als Parameter *action* vom Typ *Consumer<String>* übergeben:

```
public abstract class Message {
  …
  public abstract void forEach(Consumer<String> action);
}
```

Das Interface *Consumer* ist wie folgt definiert:

```
@FunctionalInterface
public interface Consumer<T> {
    void accept(T t);
}
```

Frage 4

Vollständig

Erreichbare Punkte: 2,00

```
public class SingleLine extends Message {
   public final String line;
   public SingleLine(String line) {
     this.line = line;
  }
```

```
   …
   // TODO forEach
}
```

```
public void forEach(Consumer<String> consumer) {
  consumer.accept(line);
}
```

Frage 5

Vollständig

Erreichbare Punkte: 3,00

```
public class ComposedMessage extends Message {
```

```
  public final List<Message> elems;

  public ComposedMessage(Message...elems) {
    this.elems = List.of(elems);
  }
   …
   // TODO forEach
}
```

```
public void forEach(Consumer<String> consumer) {
  for (Message m : elems) {
    for (String s : m) {
      consumer.accept(s);
    }
  }
}
```

Frage 6

Vollständig

Erreichbare Punkte: 2,00

Message *msgFromDonald* wird wie in Aufgabe 1 erzeugt:

```
Message msgFromDonald = ... // wie in Aufgabe 1
```

Verwenden Sie die Methode *forEach* nun für die Ausgabe der Zeilen der Message *msgFromDonald* auf der Konsole (mit *System.out.println*).

```
msgFromDonald.forEach(System.out::println);
```

Information

Aufgabe 3: Interfaces

Folgend ist das Interface *Scalable* mit der Methode *scale* gegeben. Die Methode soll ein *Scalable*-Objekt um den Faktor *factor* vergrößern (*factor* > 1) oder verkleinern (*factor* < 1).

```
public interface Scalable {
  void scale(double factor);
  ...
}
```

Frage 7

Vollständig

Erreichbare Punkte: 3,00

Implementieren Sie im Interface *Scalable* eine Default-Methode

- *doubled*() – die das *Scalable*-Objekt in der Größe verdoppelt

```
public interface Scalable {
   void scale(double factor);
```

```
   // TODO doubled
}
```

```
public default void doubled() {
  scale(2.0);
}
```

Frage 8

Vollständig

Erreichbare Punkte: 4,00

Die folgende Klasse *Rectangle* soll das Interface *Scalable* implementieren. Implementieren Sie das Interface *Scalable* für *Rectangle* (es sollen die Weite *width* und die Höhe *height* entsprechend vergrößert bzw. verkleinert werden):

```
public class Rectangle implements Scalable {
  private double width;
  private double height;

  public Rectangle(double width, double height) {
    this.width = width;
    this.height = height;
  }
```

```
  // TODO: Interface Scalable implementieren
}
```

```
@Override
public void scale(double factor) {
  width = width * factor;
  height = height * factor;
}
```

Information

Aufgabe 4: Streams

Gegeben ist eine Liste von Wörtern:

```
  List<String> words = List.of("This", "list", "contains",
"words", "Ann", "and", "Pat");
```

Implementieren Sie folgende Operationen für die Liste von Wörtern mit dem **Stream API** (Verwendung des Stream API ist **verpflichtend**).

Frage 9

Vollständig

Erreichbare Punkte: 5,00

Schreiben Sie eine Stream-Operation, die aus den Wörtern *words* jene filtert, die mit einem Großbuchstaben beginnen und sammeln Sie diese in einer Liste. Ob ein Wort mit einem Großbuchstaben beginnt, können Sie mit *Character.isUpperCase(char c)* testen.
Verwenden Sie dazu folgende Stream-Methoden:

- *Stream<T> filter(Predicate<? super T> predicate)*
- *<R, A> R collect(Collector<? super T, A, R> collector)*

sowie die statische Methode *toList* der Klasse *Collectors*

- *Collector<T, ?, List<T>> toList()*

```
List<String> upperCaseWords = // TODO
```

```
Stream.of(words).filter(s -> Character.isUpperCas
```

Frage 10

Vollständig

Erreichbare Punkte: 5,00

Schreiben Sie eine Operation, die aus den Wörtern *words* das kürzeste Wort ermittelt.
Verwenden Sie dazu folgende Stream-Methoden:

- *Stream<T> sorted(Comparator<? super T> comparator)*
- *Optional<T> findFirst()*

wobei das Interface *Comparator* folgend definiert ist:

```
public interface Comparator<T> {
  int compare(T o1, T o2);
  …
}
```

```
Optional<String> shortestWord = // TODO
```

```
Stream.of(words).sorted(
  new Comparator<String> () {
    @Override
    int compare(String s1, String s2) {
      return s2.length() - s1.length();
```

Information

Aufgabe 5: Comparable und Comparator

Ein Klasse *Course* stellt besuchte Kurse dar. Ein *Course* hat einen Namen *name* und eine Jahreszahl *year*, die angibt, wann der Kurs besucht wurde. Die Methoden *equals* führt einen Vergleich auf Basis von *name* und *year* durch; *hashCode* ist entsprechend implementiert.

```
public final class Course implements Comparable<Course> {

  public final int year;
  public final String name;

  public Course(int year, String name) {
    this.year = year;
    this.name = name;
  }

  @Override
  public int hashCode() { return Objects.hash(name, year); }

  @Override
  public boolean equals(Object obj) {
    if (obj instanceof Course) {
       Course other = (Course) obj;
       return name.equals(other.name) && this.year ==
other.year;
    }
    return false;
  }

  // TODO: Implementierung von Interface Comparable
}
```

Frage 11

Vollständig

Erreichbare Punkte: 4,00

Das Interface *Comparable* ist folgend definiert:

```
public interface Comparable<T> {
  int compareTo(T other);
}
```

Implementieren Sie die Methode *compareTo* für die Klasse *Course*. Die Ordnung ist dabei aufsteigend nach Name und Jahr.

```
@Override
public int compareTo(Course other) {
  int r = this.name.compareTo(other.name);
  if ( r == 0) r = this.year - other.year;
  return r;
}
```

Frage 12

Vollständig

Erreichbare Punkte: 6,00

Die folgende Anweisung erzeugt ein *SortedSet* für *Course*-Objekte. Die Sortierung erfolgt dabei auf Basis der *Comparable*-Implementierung.

```
SortedSet<Course> coursesSorted = new TreeSet<Course>();
```

Um die Objekte in einem *SortedSet* nach einem anderen als durch die *Comparable*-Implementierung vorgegebenen Sortierkriterium zu sortieren, kann man das *TreeSet* mit einem *Comparator* initialisieren (Interface *Comparator* siehe Aufgabe 4).

Implementieren Sie einen *Comparator* zur Initialisierung des *TreeSet*, wobei die Sortierung der *Course*-Objekte zuerst nach dem Jahr und dann nach dem Namen erfolgen soll. Das Interface *Comparator* ist folgend definiert:

```
public interface Comparator<T> {
  int compare(T o1, T o2);
  …
}
```

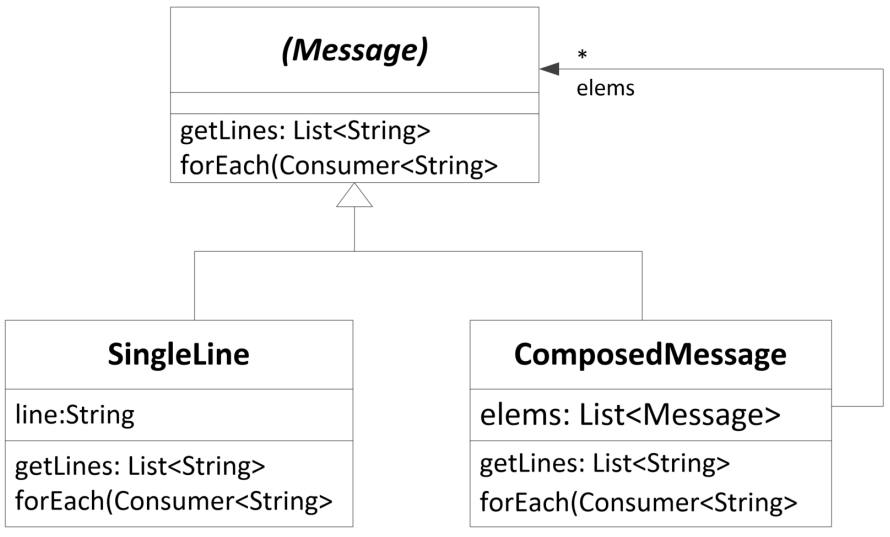
```
SortedSet<Course> coursesSortedByYear = new TreeSet<Course>(
            // TODO: Comparator-Objekt für TreeSet erzeugen
      );
```

```
new Comparator<Course> () {
  @Override
  int compare(Course c1, Course c2) {
    int r = c1.year -c2.year;
    if (r==0) r = c1.name.compareTo(c2.name);
    return r;
}
```

Information

Aufgabe 6: Visitor

Implementieren Sie einen Visitor für Messages aus Aufgabe 1.
Folgend ist zur Erinnerung nochmals die Klassenhierarchie gezeigt:



Frage 13

Vollständig

Erreichbare Punkte: 4,00

Definieren Sie ein Interface *MessageVisitor* als Visitor für Messages:

```
interface MessageVisitor {
  void visit(Message m);
}
```

Frage 14

Vollständig

Erreichbare Punkte: 2,00

Definieren Sie eine *accept*-Methode in der Klasse *Message*:

```
public abstract class Message {
  …
   // TODO: Methode accept
}
```

```
public abstract void accept(MessageVisitor v);
```

Frage 15

Vollständig

Erreichbare Punkte: 2,00

Implementieren Sie die *accept*-Methode in *SingleLine*:

```
public class SingleLine extends Message {
  …
  // TODO: Methode accept
}
```

```
public void accept(MessageVisitor v) {
  v.visit(this);
}
```

Frage 16

Vollständig

Erreichbare Punkte: 2,00

Implementieren Sie die *accept*-Methode in *ComposedMessage*:

```
public class ComposedMessage extends Message {
  …
  // TODO: Methode accept
}
```

```
public accept(MessageVisitor v) {
  v.visit(this);
}
```

Frage 17

Vollständig

Erreichbare Punkte: 6,00

Implementieren Sie nun einen Visitor *CountContainsVisitor*, der die Zeilen zählt, die einen gegeben String *text* enthalten (Verwenden Sie im Visitor ein Feld *count* zum Zählen):

```
public class CountContainsVisitor implements MessageVisitor {

  public int count = 0;
  private final String text;

  public CountContainsVisitor(String text) {
    this.text = text;
  }
```

```
  // TODO
}
```

```
public visit(Message m) {
  m.forEach(s -]] {if(s.contains(text) [count++];
}
```

Frage 18

Vollständig

Erreichbare Punkte: 2,00

Erzeugen Sie nun einen *CountContainsVisitor* für die Suche nach dem Text "*great*" und wenden Sie diesen Visitor auf die Message *msgFromDonald* an:

```
Message msgFromDonald =
       new ComposedMessage(
         new SingleLine("Message from the greatest on
earth!"),
         new ComposedMessage(
           new SingleLine("I am the greatest!"),
           new SingleLine("And all others are losers."),
           new SingleLine("That's the truth!")
         ),
         new SingleLine("Your great Donald")
       );

// TODO: CountContainsVisitor auf msgFromDonald anwenden
```

```
msgFromDonald.accep(new CountContainsVisitor("gre
```

◄ Zoom-Links

Direkt zu:

Klausur 1: Theoretischer Teil ►