NAME DES DOZENTEN: Prof. Dr. H.-W. Sehring

NORDAKADEMIE
HOCHSCHULE DER WIRTSCHAFT

Lösungsblatt

Musterartefakt: Einführung in die OO-Programmierung

A23

QUARTAL: II/2024

Dauer: 60 Minuten Anzahl Seiten **ohne** Deckblatt: 14 **Datum:** 01.01.1750

Hilfsmittel: keine.

1. (3 Punkte) Was sind korrekte Initialisierungen (vom Compiler akzeptierte Initialisierungen) von Instanzen von Collection Klassen?
   Bewertung: Richtig = +,5; Falsch = -0,5; Min = 0

   ○ `private ArrayList<int> numbers = new ArrayList<>();`

   ○ `private ArrayList<String> buchstaben = new HashMap<>();`

   √ **`private List<String> buchstaben = new ArrayList<String>();`**

   √ **`private ArrayList<String> buchstaben = new ArrayList<>();`**

   ○ `private ArrayList<String> buchstaben = new HashMap<String>();`

   ○ `private ArrayList<Integer> numbers = new HashMap<Integer>();`

2. (4 Punkte) Welche der folgenden Aussagen über Interfaces, Klassen und abstrakten Klassen sind richtig?

   √ **Interfaces und abstrakte Klassen haben gemeinsam, dass keine Instanzen von ihnen erstellt werden können.**

   ○ Interfaces können Exemplarvariablen definieren, abstrakte Klassen nicht.

   ○ Interfaces definieren keine Typen, dass gilt nur für abstrakte Klassen.

   ○ Nur Klassen können von Interfaces erben, abstrakte Klassen nicht.

   √ **Interfaces definieren Typen.**

   √ **Das Exemplar einer Klasse, die von einer abstrakten Klasse erbt, die wiederum ein bestimmtes Interface X implementiert, kann einer Variablen zugewiesen werden, die mit dem Typ des Interfaces X deklariert ist.**

   √ **Abstrakte Klassen müssen nicht alle Methoden der von ihnen implementierten Interfaces implementieren.**

   ○ Interfaces erlauben keine Mehrfachvererbung.

   ○ Klassen erweitern (extends) und implementieren nicht (implements) Interfaces. Sie implementieren lediglich abstrakte Methoden abstrakter Vorfahren.

3. (5 Punkte) Vervollständigen Sie den Text

   Eine Schnittstelle beinhaltet die

   Lösung:
   Methodensignaturen

   der Methoden, die eine die Schnittstelle

   Lösung:
   implementierende

   Klasse zur Verfügung stellen muss. Schnittstellen unterstützen, im Gegensatz zu Klassen,

   Lösung:
   Mehrfachvererbung

   . Schnittstellen können bei der Deklaration von Variablen als

   Lösung:
   Typ

   verwendet werden. Es gilt als guter Stil gegen Schnittstellen zu programmieren – der Quellcode wird dadurch unabhängig von der jeder konkreten

   Lösung:
   Implementierung

   der Schnittstelle. Eine Schnittstelle kann von jeder

   Lösung:
   Klasse

   implementiert werden, muss es aber nicht. Die

   Lösung:
   Polymorphie

   erlaubt es jeden Subtyp einer Schnittstelle dort zu verwenden wo die Schnittstelle als

Lösung:
Typ

angegeben ist. Schnittstellen helfen das Grundprinzip der

Lösung:
losen Kopplung

zwischen Bestandteilen eines Programms zu befolgen. Schnittstellendefinitionen werden nicht mit dem Schlüsselwort `class`, sondern mit dem Schlüsswort

Lösung:
interface

, eingeleitet.

4. (6 Punkte) Gegeben seien folgendes Interface und folgende Klassen:

```
public interface IDoSomething{
  void doSomething();
}

public class DoSomething implements IDoSomething{
  public void doSomething(){
    System.out.println("Working hard");
  }
}

public class DoSomethingElse implements IDoSomething{
  public void doSomething(){
    System.out.println("Working even harder");
  }
}
```

Welche der Anweisungsfolgen sind richtig?

Kreuzen Sie für jede Anweisungsfolge entweder *richtig* oder *falsch* an.

- Für jede richtig gesetzte Markierung erhalten Sie einen Punkt (+1).
- Falsch oder nicht markierte Aussagen geben 0 Punkte.

`DoSomething doit = new DoSomething();` `IDoSomething doit2 = doit;`	√ **richtig** ☐ falsch
`IDoSomething doit = new DoSomething();` `DoSomething doit2 = doit;`	☐ richtig √ **falsch**
`IDoSomething doit = new IDoSomething();`	☐ richtig √ **falsch**
`IDoSomething doit = new DoSomethingElse();`	√ **richtig** ☐ falsch
`IDoSomething doit = new DoSomething();`	√ **richtig** ☐ falsch
`DoSomething doit = new DoSomethingElse();`	☐ richtig √ **falsch**

5. Betrachten Sie das folgende Code-Beispiel:

```
public abstract class Shape {
  private int posX, posY;

  protected Shape(int posX, int posY) {
    this.posX = posX;
    this.posY = posY;
  }

  public abstract int getArea();
}

public class Rectangle extends Shape {
  private int len1, len2;

  public Rectangle(int posX, int posY, int len1, int len2) {
    super(posX, posY);
    this.len1 = len1;
    this.len2 = len2;
  }

  public int getArea() {
    return len1 * len2;
  }
}

public class Square extends Rectangle {
  public Square(int posX, int posY, int len) {
    super(posX, posY, len, len);
  }

  public int getArea(int factor) {
    return factor * getArea();
  }
}
```

(5.1) (7 Punkte) Erläutern Sie die Begriffe *Überschreiben*, *Überladen* und *späte Bindung* an folgendem Code-Beispiel.

Lösung:

Beim Überschreiben wird eine ererbte Methode redefiniert, in dem eine Methodendeklaration *mit gleicher Signatur* angegeben wird. Im Beispiel wird die Methode `getArea()` von `Shape` in `Rectangle` überschrieben (genau genommen implementiert).

Beim Überladen werden zwei Methodendeklarationen mit gleichem Namen aber unterschiedlichen Parametern gemacht, im Beispiel in `Square` die ererbte Methode `getArea()` und die Methode `getArea(int)`.

Späte Bindung bedeutet, dass bei einer Nachricht an ein Objekt diejenige Methode mit der speziellsten Definition aktiviert wird. Im Beispiel würde Bei

```
Shape s = new Square(0, 0, 17);
s.getArea();
```

die Methodendefinition aus `Rectangle` gewählt.

(5.2) (4 Punkte) Überschreiben Sie die parameterlose Methode `getArea()` in der Subklasse `Square`.
Diskutieren Sie die Sinnfälligkeit des Überschreibens und der Status von `Rectangle` und `Square`.

Lösung:

Überschreiben von `getArea()`: In der Klasse `Square` ergänze

```
public int getArea() {
  return len1 * len1;
}
```

Diese Definition ändert nicht das Ergebnis. Sie stellt deutlicher als die ererbte Methode dar, dass das Quadrat nur eine Seitenlänge hat. Allerding ist die Exemplarvariable `len2` weiterhin existent, so dass diese Vererbungsbeziehung generell fragwürdig ist.

6. Das *Decorator* Entwurfsmuster ergänzt eine Methode um zusätzliches Verhalten. Dieses Entwufsmuster soll folgendermaßen Anwendung finden: Examplare einer Subklasse `BachelorStudent` der gegebenen Klasse `Student` sollen beim Erreichen des Abschlusses (Methode `graduate()`) zusätzlich ihren Hut in die Luft werfen.

```
public class Student {
  private ExaminationOfficeService examOffice;

  public void graduate() {
    examOffice.graduate(this);
  }
}

public class BachelorStudent extends Student {}
```

(6.1) (3 Punkte) Setzen Sie das spezifizierte Verhalten entsprechend um. Sie brauchen nicht den gesamten vorgegebenen Code zu wiederholen. Machen Sie im Zweifel kenntlich, wo Sie Code ändern oder ergänzen.

Lösung:

In `BachelorStudent` ergänze:

```
public void graduate() {
  super.graduate();
  new Hat().flyIntoAir();
}
```

(6.2) (2 Punkte) Begründen Sie Ihren Entwurf unter den in der Vorlesung genannten Entwurfsprinzipien.

Lösung:

Durch das Überschreiben der Methode in der Subklasse wird die Basisklasse nicht verändert; Erweiterung durch Hinzufügen, nicht durch Ändern.

Die Basisimplementierung wird weiterverwendet, keine Duplikation von Code.

Die Schnittstelle der Klassen wird nicht verändert; aller Code in Client-Klassen funktioniert weiterhin.

7. Gegeben sei folgende Schnittstellendefinition für eine Zenturie an der NORDAKADEMIE. Stellen Sie sich eine übliche Verwendung von Century-Objekten in Systemen wie dem CIS vor.

```
public interface Century {
  public Collection<FemaleStudent> getFemaleMembers();
  public Collection<MaleStudent> getMaleMembers();
  public Student getSpeaker();
  public Professor getCenturio();
}

public interface Student { String getDisplayName(); }

public interface FemaleStudent extends Student {}

public interface MaleStudent extends Student {}

public interface Professor { String getDisplayName(); }
```

(7.1) (3 Punkte) Bewerten Sie die Kopplung der Klassen.

Lösung:

Die Koppling ist eher eng. `Century` ist abhängig von `FemaleStudent`, `MaleStudent`, `Student` und `Professor`.

Die Verwaltung des Geschlechts sollte nicht bei der Zenturie liegen, sondern bei den Studierenden bzw. den genannten Subtypen. (Hier ist also auch die Kohäsion niedrig.) Daher könnte zumindest die Abhängigkeit von `FemaleStudent` und `MaleStudent` vermieden werden.

(7.2) (5 Punkte) Schlagen Sie einen alternativen, besseren Entwurf vor.

Lösung:

Die Methoden `getFemaleMembers()` und `getMaleMembers()` in Century sollten ersetzt werden durch

```
public Collection<Student> getMembers();
```

Je nach Anwendung kann mit den gegebenen Schnittstellen von allen Interfaces abstrahiert werden, da es nur die eine Methode `getDisplayName()` gibt. Es gibt also nur Personen in verschiedenen Rollen:

```
public interface Person {
  String getDisplayName();
}
public interface Student extends Person {}
public interface Professor extends Person {}
public interface Century {
  public Collection<Person> getMembers();
  public Collection<Person> getMaleMembers();
  public Person getSpeaker();
  public Person getCenturio();
}
```

(7.3) (3 Punkte) Erläutern Sie, inwiefern Ihre Lösung aus der vorigen Teilaufgabe “besser” ist.

Lösung:

Es wird eine losere Kopplung erreicht (je nach Grad der Umsetzung nur noch abhängig von 1 oder 2 Interfaces).

Die Kohäsion ist höher: die Verwaltung des Geschlechts liegt nicht mehr in der Zenturie

schlankeres Interface (eine Methode weniger) erlaubt höheren Grad an Wiederverwendung

8. (8 Punkte) Kreuzen Sie für jede Aussage entweder *richtig* oder *falsch* an.

   - Für jede korrekt gesetzte Markierung erhalten Sie einen Punkt (+1).
   - Bei einer falschen oder fehlenden Markierung erhalten Sie keinen Punkt für die Teilaufgabe.

   Welche der folgenden Aussagen über Konzepte der Programmiersprache Java sind richtig?

Subtyping erlaubt es, mehrere Klassen unter einer gemeinsamen Schnittstelle anzusprechen. Dabei können die Exemplarvariablen, die im Interface deklariert werden, für alle das Interface implementierenden Klassen wiederverwendet werden.	☐ richtig √ **falsch**
Eine Subklasse ist eine Klasse, die eine andere Klasse erweitert bzw. von dieser Klasse erbt. Sie erbt alle Datenfelder und Methoden von ihrer Superklasse.	√ **richtig** ☐ falsch
Eine Subklasse ist eine Klasse, die in einer anderen Klasse als Typ einer Exemplarvariablen verwendet wird.	☐ richtig √ **falsch**
Superklassen werden als „super“ bezeichnet, da sie für viele unterschiedliche Facetten einer Software zuständig sind. Weitere Spezialisierungen dieser Klasse werden in der Regel nicht mehr vorgenommen.	☐ richtig √ **falsch**
Konstruktoren einer Spezialisierung können zu einem beliebigen Zeitpunkt der Abarbeitung der Anweisungen ihres Konstruktors den Konstruktor der Generalisierung aufrufen. Wenn im Quelltext kein solcher Aufruf angegeben ist, versucht Java automatisch einen parameterlosen Aufruf einzufügen.	☐ richtig √ **falsch**
Eine Variable kann ein Objekt halten, dessen Typ entweder gleich dem deklarierten Typ der Variablen oder ein beliebiger Subtyp des deklarierten Typs ist.	√ **richtig** ☐ falsch
Eine Subklasse kann die Implementierung einer Methode überschreiben. Dazu deklariert die Subklasse eine Methode mit der gleichen Signatur wie in der Superklasse, implementiert diese jedoch mit einem anderen Rumpf. Die überschreibende Methode wird dann bei Aufrufen der Unterklasse vorgezogen.	√ **richtig** ☐ falsch
Methodenaufrufe in Java sind niemals polymorph. Derselbe Methodenaufruf kann immer nur eine bestimmte Methode aufrufen. Dies wird durch den statischen Typ der Variablen fest vorgegeben.	☐ richtig √ **falsch**

9. (2 Punkte) Erstellen Sie eine Methode, die einen Stream von `Element`s bekommt und diesen auf einen Stream von `boolean`-Objekten, entsprechend dem jeweiligen Attribut `dead`, abbildet.

```
public class Element {
  private boolean dead;
  private int iq;
  public boolean isDead() { return dead; };
  public int getIQ() { return iq; };
}
```

Lösung:

```
public Stream<E> mapToDeadAliveStream (Stream<Element> elements) {
return elements
  .map(e -> e.isDead());
}
```

Bewertung auf Basis der Konzepte/Konventionen: kleine Variablen-/Parameterbezeichner, Ergebnis (Mandatory), Map