FernUniversität in Hagen

FAKULTÄT FÜR
MATHEMATIK UND INFORMATIK

Lehrgebiet Programmiersysteme
Prof. Dr. Friedrich Steimann

(Name, Vorname)

(Straße, Nr.)

(PLZ) (Wohnort)

(Land, falls außerhalb Deutschlands)

Kurs 1618 SS 2012

„Einführung in die objektorientierte Programmierung“

Nachklausur am 09.02.2013

Dauer: 3 Std., 10 – 13 Uhr

Lesen Sie zuerst die Hinweise auf der folgenden Seite!

Matrikelnummer: | | | | | | | | **Geburtsdatum:** | | . | | . | | |

Klausurort: _____

Aufgabe	1	2	3	4	5	6	7	8	9	10	Summe
habe bear-beitet											
maximal	10	10	10	10	10	10	10	10	10	10	100
erreicht											
Korrektur											

○ Herzlichen Glückwunsch, Sie haben die Klausur bestanden. Note: .....

○ Sie haben die Klausur leider nicht bestanden. Für den nächsten Versuch wünschen wir Ihnen viel Erfolg. Die nächste Klausur findet im Sommersemester 2013 statt.

Hagen, den 20.2.2013

Im Auftrag

Musterlösung

© 2013 FernUniversität in Hagen

Hinweise zur Bearbeitung

1. Prüfen Sie die Vollständigkeit Ihrer Unterlagen. Die Klausur umfasst auf insgesamt 18 Seiten:
   – 1 Deckblatt
   – Diese Hinweise zur Bearbeitung
   – 10 Aufgaben auf Seite 3-16
   – Zwei zusätzliche Seiten 17 und 18 für weitere Lösungen

2. Füllen Sie jetzt bitte zuerst das Deckblatt aus:
   – Name, Vorname und Adresse
   – Matrikelnummer, Geburtsdatum und Klausurort

3. Schreiben Sie Ihre Lösungen mit Kugelschreiber oder Füllfederhalter (*kein Bleistift*) direkt in den bei den jeweiligen Aufgaben gegebenen, umrahmten Leerraum. Benutzen Sie auf keinen Fall die Rückseiten der Aufgabenblätter. Versuchen Sie, mit dem vorhandenen Platz auszukommen; Sie dürfen auch stichwortartig antworten. Sollten Sie wider Erwarten nicht mit dem vorgegebenen Platz auskommen, benutzen Sie bitte die beiden an dieser Klausur anhängenden Leerseiten. **Es werden nur Aufgaben gewertet, die sich auf dem offiziellen Klausurpapier befinden.** Eigenes Papier ist nur für Ihre persönlichen Notizen erlaubt.

4. Kreuzen Sie die bearbeiteten Aufgaben auf dem Deckblatt an. Schreiben Sie unbedingt *auf jedes Blatt* Ihrer Klausur Ihren Namen und Ihre Matrikelnummer, auf die Zusatzblätter auch die Nummer der Aufgabe.

5. Geben Sie die gesamte Klausur ab. Lösen Sie die Blätter nicht voneinander.

6. Es sind *keine Hilfsmittel* zugelassen.

7. Lesen Sie vor der Bearbeitung einer Aufgabe den *gesamten* Aufgabentext sorgfältig durch.

8. Es sind maximal 100 Punkte erreichbar. Wenn Sie mindestens 50 Punkte erreichen, haben Sie die Klausur bestanden.

9. Sie erhalten die korrigierte Klausur zurück, zusammen mit einer Bescheinigung für das Finanzamt und ggf. dem Übungsschein.

10. Legen Sie jetzt noch Ihren Studierendenausweis und einen amtlichen Lichtbildausweis bereit, dann kann die Arbeit beginnen. Viel Erfolg!

Musterlösung

Aufgabe 1: Ausnahmebehandlung (10 Punkte)

Schreiben Sie ein gültiges (also fehlerfrei kompilierendes) Java-Programm, welches die folgenden Anforderungen erfüllt:

- Es soll eine Klasse `Bruch` implementiert werden. Ihr einziger Konstruktor erwartet zwei ganzzahlige Argumente, den `zaehler` und den `nenner`.
- Wird dem Bruch-Konstruktor als Nenner der Wert `0` übergeben, soll der Konstruktor eine ebenso zu implementierende `NennerIstNullException` werfen. `NennerIstNullException` soll **keine** `RuntimeException` sein.
- Implementieren Sie in Bruch eine `main`-Methode, die einen neuen Bruch $\frac{2}{3}$ erzeugt
- Ansonsten sind keine weiteren Methoden oder Felder in `Bruch` oder `NennerIstNullException` zu implementieren. Auch brauchen die dem Bruch-Konstruktor übergebenen Argumente keine weitere Verwendung erfahren.

```
public class Bruch {
	public Bruch(int zaehler, int nenner) throws NennerIstNullException {
		if(nenner == 0)
			throw new NennerIstNullException();
		/* ... */
	}

	public static void main(String[] args) {
		try {
			Bruch b = new Bruch(2,3);
		} catch (NennerIstNullException e) {
			/* ... */
		}
	}
}

class NennerIstNullException extends Exception { /* ... */ }
```

Musterlösung

Aufgabe 2: Klassenentwurf (10 Punkte)

Erstellen Sie eine Klasse „`KomplexeZahl`“, die die folgenden Vorgaben umsetzt:

- Eine komplexe Zahl hat die gleitkommazahligen Attribute `realteil` und `imaginaerteil`.
- Die Werte der beiden Attribute werden bei der Erzeugung eines `KomplexeZahl`-Exemplars übergeben und lassen sich danach nicht mehr ändern.
- Die Attribute sind öffentlich zugreifbar.
- In der Klasse `KomplexeZahl` ist die von `Object` geerbte Methode `toString()` so zu überschreiben, dass sie Information über Imaginär- und Realteil liefert.
- `KomplexeZahl`-Exemplare haben eine Methode `addiere`, deren einziger Parameter ein weiteres `KomplexeZahl` -Exemplar ist. Die Methode gibt ein neues `KomplexeZahl`-Exemplar zurück, welches die Summe der beiden komplexen Zahlen (Nachrichtenempfänger und Parameter) repräsentiert. (Zur Erinnerung: bei der Addition von komplexen Zahlen werden jeweils Real- und Imaginärteil miteinander addiert)

```java
public class ComplexeZahl {

    public final double realteil;
    public final double imaginaerteil;

    public ComplexeZahl(double realteil, double imaginaerteil) {
        this.realteil = realteil;
        this.imaginaerteil = imaginaerteil;
    }

    public ComplexeZahl addiere(ComplexeZahl summand) {
     double neuerRealteil = realteil + summand.realteil;
     double neuerImaginaerteil = imaginaerteil + summand.imaginaerteil;
        return new ComplexeZahl(neuerRealteil, neuerImaginaerteil);
    }

    public String toString() {
        return realteil + " + " + imaginaerteil + "*i";
    }
}
```

Musterlösung

Aufgabe 3: Subtyping und Enums (10 Punkte)

Gegeben sei folgendes Programm:

```
class Tier {

	private String tierart;

	Tier(String tierart) {
		this.tierart = tierart;
	}

	public void gibLaut() {
		if(tierart.equals("Katze"))
			System.out.println("Miau");
		else if(tierart.equals("Maus"))
			System.out.println("Fiep");
		else throw new RuntimeException("Unbekanntes Tier");
	}
}

public class Main {
	public static void main(String[] args) {
		Tier katze = new Tier("Katze");
		Tier maus = new Tier("Maus");
		katze.gibLaut();
		maus.gibLaut();
	}
}
```

Welche Ausgabe erzeugt die `main`-Methode?

```
Miau
Fiep
```

(Fortsetzung der Aufgabe auf folgender Seite)

Musterlösung

(Fortsetzung von Aufgabe 3)

Obiges Programm soll derart geändert werden, dass die `if`-Abfrage in der `gibLaut`-Methode durch Subtyping ersetzt wird. Implementieren Sie obiges Programm noch einmal, indem Sie `Tier` als abstrakte Klasse mit abstrakter Methode `gibLaut()` implementieren und `Katze` und `Maus` als abgeleitete Klassen hinzufügen. Mit folgendermaßen geänderter `main`-Methode soll das Programm nach wie vor dieselbe Ausgabe liefern:

```
public class Main {
    public static void main(String[] args) {
        Tier katze = new Katze();
        Tier maus = new Maus();
        katze.gibLaut();
        maus.gibLaut();
    }
}
```

```
abstract class Tier {
    public abstract void gibLaut();
}

class Katze extends Tier {
    public void gibLaut() {
        System.out.println("Miau");
    }
}

class Maus extends Tier {
    public void gibLaut() {
        System.out.println("Fiep");
    }
}
```

Musterlösung

Aufgabe 4: Subtyping und Enums II (10 Punkte)

Das Katze-Maus-Programm aus Aufgabe 3 kann ebenso auch mittels einer Enumeration gelöst werden. Implementieren Sie die Klasse `Tier` als eine `enum`, welche mittels folgender `main`-Methode weiterhin die gleiche Ausgabe liefert:

```
public class Main {
      public static void main(String[] args) {
            Tier katze = Tier.KATZE;
            Tier maus = Tier.MAUS;
            katze.gibLaut();
            maus.gibLaut();
      }
}
```

```
enum Tier {
      KATZE, MAUS;

      public void gibLaut() {
            switch (this) {
            case KATZE:
                  System.out.println("Miau");
                  break;
            case MAUS:
                  System.out.println("Fiep");
            }
      }
}
```

(Fortsetzung der Aufgabe auf folgender Seite)

Musterlösung

(Fortsetzung von Aufgabe 4)

Sie haben nun drei Varianten desselben Programms gesehen: Die if-Variante wie im Aufgabentext zu Aufgabe 3, die subtyping-Variante aus Ihrer Lösung zu Aufgabe 3 sowie die enum-Variante aus Ihrer Lösung zum ersten Teil von Aufgabe 4.

Geben Sie für jede der drei Varianten an, für wie anfällig Sie sie gegenüber Programmierfehlern halten, insbesondere, wenn ein Tier erzeugt werden soll, das nicht bekannt ist.

Die if-Variante ist sehr anfällig für Programmierfehler. Hier müssen die Strings beim Konstuktoraufruf exakt mit den Strings in der `gibLaut`-Methode übereinstimmen. Wird ein unbekanntes Tier erzeugt, wird dieser Fehler erst zur Laufzeit mit einer `RuntimeException` bekannt.

In der enum-Variante verhindert der Compiler, dass ein Tier erzeugt wird, für das keine Enum-Konstante angegeben ist. Dennoch können sich Fehler einschleichen, wenn vergessen wird, jedes Tier in der `gibLaut`-Methode zu berücksichtigen.

Die Subtyping-Variante ist vergleichsweise robust. Der Compiler verhindert die Instanziierung eines nicht deklarierten Typen. Auch gibt der Compiler einen Fehler aus, wenn vergessen wurde, die abstrakte Methode `gibLaut` zu implementieren.

Nennen Sie einen Vorteil der Subtyping-Variante bezüglich der Erweiterbarkeit um neue Tiere.

Die Subtyping-Variante kann einfacher um neue Tiere erweitert werden, ohne den bestehenden Code und die `Tier`-Klasse verändern zu müssen. Das ist praktisch, wenn die Tier-Klasse z.B. aus einer Bibliothek stammt.

Musterlösung

Aufgabe 5: Beschränkt parametrische Typen (10 Punkte)

Gegeben sei folgendes Interface `Auftrag`

```
interface Auftrag {
      public boolean hatPrioritaet();
}
```

Implementieren Sie eine Klasse `Warteschlange`, die die folgenden Bedingungen erfüllt:

- `Warteschlange` ist typparametrisierbar, wobei für den einzigen Typparameter nur Subtypen von `Auftrag` Verwendung finden dürfen.
- `Warteschlange` besitzt eine Methode `einfuegen`, welche genau ein Argument erwartet und welches vom angegebenen parametrisierten Typen ist.
- `Warteschlange` hat ein Attribut `auftraege`, welches in einer verketteten Liste alle hinzugefügten und nicht entnommenen Aufträge verwaltet. Aufträge mit Priorität stehen am Anfang der Liste, Aufträge ohne Priorität am Ende der Liste. Innerhalb des vorderen und hinteren Listenabschnittes ist die Reihenfolge beliebig.
- Eine Methode `entnehmen` entfernt das erste Element von `auftraege` und gibt es zurück.
- Hinweis: Sie dürfen die Klasse `java.util.LinkedList` mit ihren Methoden `addFirst`, `addLast` und `removeFirst` benutzen.

```
class Warteschlange<T extends Auftrag> {
    LinkedList<T> auftraege = new LinkedList<T>();

    void einfuegen(T auftrag) {
          if(auftrag.hatPrioritaet())
                auftraege.addFirst(auftrag);
          else
                auftraege.addLast(auftrag);
    }

    T entnehmen() {
          return auftraege.removeFirst();
    }
}
```

Musterlösung

Aufgabe 6: Beschränkt parametrische Typen II (10 Punkte)

Begründen Sie, warum in der vorhergehenden Aufgabe 5 der parametrische Polymorphismus tatsächlich nützlich war. Oder anders gefragt: Wo kommt es zu Problemen, wenn Sie jede Verwendung des Typparameters innerhalb Ihrer Lösung von Aufgabe 5 stets durch `Auftrag` ersetzt hätten?

Zu Problemen kommt es an den Aufrufestellen der Methode `entnehmen`. Ihr Rückgabewert wäre nicht mehr als `<T extends Auftrag>` typisiert, sondern nur noch als `Auftrag`. Möchte ein Benutzer der Warteschlange in dieser homogen Subtypen von Auftrag verwenden, kann dieses Wissen nicht in Form einer Typvariablen vorgehalten werden und es muss beim Entnehmen eines jeden Elements aus der Liste eine Typkonvertierung (cast) vorgenommen werden.

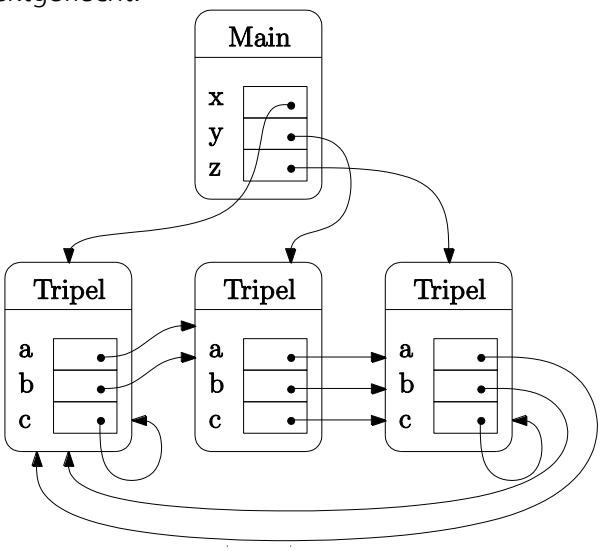
Begründen Sie, warum auch ein einfacher parametrischer Polymorphismus bei Aufgabe 5 nicht ausgereicht hätte. Oder anders gefragt: Wo kommt es zu Problemen, wenn Sie den Typen Warteschlange lediglich als `class Warteschlange<T> { … }` deklariert hätten?

Ohne die Beschränkung `<T extends Auftrag>` lässt sich auf den mit `T` typisierten Objekten (ohne Typkonvertierung) die Methode `hatPrioritaet` nicht mehr aufrufen.

Musterlösung

Aufgabe 7: Objektgeflechte (10 Punkte)

Gegeben sei folgendes Objektgeflecht:



Implementieren Sie die Klassen `Main` und `Tripel` mit den ihnen zugehörigen Attributen. Implementieren Sie ebenso innerhalb der Klasse `Main` eine `main`-Methode, welche die im Objektgeflecht enthaltenen Objekte instanziiert und ihre zugehörigen Attribute wie angegeben initialisiert.

```
class Tripel {
    Tripel a = null;
    Tripel b = null;
    Tripel c = null;
}

class Main {
    Tripel x = new Tripel();
    Tripel y = new Tripel();
    Tripel z = new Tripel();
    void verbinde() {
        x.a = x.b = z.a = y;
        y.a = y.b = y.c = z.c = z;
        z.a = z.b = x.c = x;
    }
     public static void main(String[] args) {
        new Main().verbinde();
    }
}
```

Musterlösung

Aufgabe 8: Verschiedenes

Geben Sie an, ob die folgenden Aussagen korrekt sind. Alle Aussagen beziehen sich auf fehlerfrei kompilierende Java-Programme.

Achtung! Falsche Antworten geben Minuspunkte, eine gesamt negative Punktzahl bei dieser Aufgabe wird Ihnen aber auf null Punkte aufgerundet.

Kommunikation über Nachrichten ist asynchron.	☑ Wahr	☐ Falsch
Entfernte Methodenaufrufe zählen zu den synchronen Kommunikationsmitteln	☑ Wahr	☐ Falsch
Ein Vorteil von Stromklassen ist, dass sie relativ einfach hintereinandergeschaltet werden können.	☑ Wahr	☐ Falsch
Bei der Serialisierung ist die Ausgabe eine Menge von Klassen, die innerhalb eines Objektgeflechts erreichbar sind.	☐ Wahr	☑ Falsch
An welche von zwei (passenden) überladenen Methoden ein Methodenaufruf bindet, wird zur Laufzeit entschieden.	☐ Wahr	☑ Falsch
Es kann nur an überschriebene Methoden dynamisch gebunden werden.	☑ Wahr	☐ Falsch
Innerhalb einer statischen Methode kann niemals eine nicht-statische Methode aufgerufen werden.	☐ Wahr	☑ Falsch
In Java hat (außer `Object`) jede Klasse genau einen unmittelbaren Supertypen.	☐ Wahr	☑ Falsch
Das Programm `class X {Object o = "x";}` kompiliert fehlerfrei.	☑ Wahr	☐ Falsch
Das Programm `class X {String o = new Object();}` kompiliert fehlerfrei.	☐ Wahr	☑ Falsch

Musterlösung

Aufgabe 9: Threads (10 Punkte)

Gegeben sei das folgende Programm:

```
class Konto {
      int kontostand = 0;
}

class Überweisung extends Thread {
      private Konto konto;
      private int betrag;

      public Überweisung(Konto konto, int betrag) {
            this.konto = konto;
            this.betrag = betrag;
      }

      public void run() {
            int aktuellerKontostand = konto.kontostand;
            int neuerKontostand = aktuellerKontostand + betrag;
            konto.kontostand = neuerKontostand;
      }
}

public class Bank {
      public static void main(String[] args) {
            for (int i = 0; i < 10; i++) {
                  Konto konto = new Konto();
                  Überweisung u1 = new Überweisung(konto, 10);
                  Überweisung u2 = new Überweisung(konto, 10);
                  Überweisung u3 = new Überweisung(konto, 10);
                  u1.start();
                  u2.start();
                  u3.start();
            }
      }
}
```

Argumentieren Sie, warum Sie dieser Bank Ihr Geld nicht anvertrauen würden bzw. warum Sie sich nicht sicher sein können, dass Ihr Kontostand nach Programmausführung ~~30~~ 300 entspricht.

Die Überweisungen `u1` bis `u3` werden parallel ausgeführt. Dabei kann es passieren, dass mehrere Überweisungen erst nacheinander den aktuellen Kontostand lesen, dann nacheinander den Betrag hinzuaddieren und anschließend nacheinander den neuen Kontostand schreiben. Im schlimmsten Falle hat das Konto danach einen Kontostand von `10`.

(Fortsetzung der Aufgabe auf folgender Seite)

Musterlösung

(Fortsetzung von Aufgabe 9)

Hilft es, den Code in der `run`-Methode wie folgt mit einem `synchronized`-Block zu umfassen? Begründen Sie!

```
public void run() {
  synchronized (konto) {
    int aktuellerKontostand = konto.kontostand;
    int neuerKontostand = aktuellerKontostand + betrag;
    konto.kontostand = neuerKontostand;
  }
}
```

Ja. Diese Lösung sichert zu, dass für jedes Konto nur eine Überweisung gleichzeitig den kritischen Codeabschnitt ausführt.

Hilft es, die `run`-Methode mit einem `synchronized`-Schlüsselwort zu versehen? Begründen Sie!

Nein. Dies entspräche einem **synchronized** (**this**) { … } und würde entsprechend nur zusichern, dass pro Überweisung der kritische Codeblock nur einmal ausgeführt wird. Den Konflikt der Überweisungen untereinander löst diese Änderung nicht.

Musterlösung

Aufgabe 10: Überschreiben und Überladen

Gegeben sei folgendes Java-Programm:

```
public class Fuhrpark {
    public static void main(String[] args) {
        LKW lkw = new Sattelschlepper();
        PKW pkw1 = new PKW();
        PKW pkw2 = new Kleinwagen();
        Kombi kombi = new Kombi();
        Sattelschlepper sattelschlepper = new Sattelschlepper();

        Super sup1 = new Super();
        Super sup2 = new Sub();

        sup1.m(pkw2, lkw);
        sup1.m(pkw1, sattelschlepper);
        sup1.m(kombi, pkw2);
        sup2.m(pkw1, lkw);
        sup2.m(pkw1, sattelschlepper);
    }
}

class Super {
    public void m(Fahrzeug t, LKW f) {
        System.out.println("1");
    }
    public void m(Fahrzeug t1, Fahrzeug t2) {
        System.out.println("2");
    }
}

class Sub extends Super {
    public void m(PKW v, LKW f) {
        System.out.println("3");
    }
    public void m(Fahrzeug t1, LKW t2) {
        System.out.println("4");
    }
}

class Fahrzeug {}

class LKW extends Fahrzeug {}

class Sattelschlepper extends LKW {}

class PKW extends Fahrzeug {}

class Kombi extends PKW {}
class Kleinwagen extends PKW {}
```

(Fortsetzung der Aufgabe auf folgender Seite)

Musterlösung

(Fortsetzung von Aufgabe 10)

Begründen Sie, warum folgende Anweisungen zu welcher Ausgabe führen:

`sup1.m(pkw2, lkw);`	**Ausgabe:** 1 **Begründung:** Von beiden überladenen Methoden `m` in `Super` ist „1" die speziellere, weswegen der Compiler an diese bindet. Da auch der Laufzeittyp von `sup1` `Super` ist, wird „1" ausgegeben.
`sup1.m(pkw1, sattelschlepper);`	**Ausgabe:** 1 **Begründung:** Von beiden überladenen Methoden `m` in `Super` ist „1" die speziellere, weswegen der Compiler an diese bindet. Da auch der Laufzeittyp von `sup1` `Super` ist, wird „1" ausgegeben.
`sup1.m(kombi, pkw2);`	**Ausgabe:** 2 **Begründung:** Von beiden überladenen Methoden `m` in `Super` ist „2" die einzige mit überhaupt passender Signatur.
`sup2.m(pkw1, lkw);`	**Ausgabe:** 4 **Begründung:** Von beiden überladenen Methoden `m` in `Super` ist „1" die speziellere, weswegen der Compiler an diese bindet. Diese wird in `Sub` von der Methode „4" (nicht der Methode „3"!) überschrieben, weswegen die Ausgabe 4 lautet.
`sup2.m(pkw1, sattelschlepper);`	**Ausgabe:** 4 **Begründung:** Von beiden überladenen Methoden `m` in Super ist „1" die spezielle-re, weswegen der Compiler an diese bindet. Zur Laufzeit wird dann aber „1" von „4" überschrieben.

(jeweils 2 Punkte; ohne Begründung keine Punkte)

Musterlösung

Zusätzlicher Platz für Ihre Lösungen

Ergänzung zu Aufgabe Nr.

Ergänzung zu Aufgabe Nr.

Musterlösung

Zusätzlicher Platz für Ihre Lösungen

Ergänzung zu Aufgabe Nr.

Ergänzung zu Aufgabe Nr.

Musterlösung