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 2011

„Einführung in die objektorientierte Programmierung“

Nachklausur am 11.2.2012

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 2012 statt.

Hagen, den 22.2.2012

Im Auftrag

Musterlösung

Hinweise zur Bearbeitung

1. Prüfen Sie die Vollständigkeit Ihrer Unterlagen. Die Klausur umfasst auf insgesamt 17 Seiten:
   - 1 Deckblatt
   - Diese Hinweise zur Bearbeitung
   - 10 Aufgaben auf Seite 3-15
   - Zwei zusätzliche Seiten 16 und 17 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 45 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!

Aufgabe 1: Aliase & Objektgeflechte (10 Punkte)

Gegeben sei folgendes Java-Programm:

```
public class Blumenbeet {

    public static void main(String[] args) {

        Pflanze neuePflanze = new Pflanze();

        Pflanze rose = neuePflanze;
        rose.benenne("Rose");

        Pflanze tulpe = neuePflanze;
        tulpe.benenne("Tulpe");

        Pflanze efeu = neuePflanze;
        efeu.benenne("Efeu");

        rose.gießen();
        tulpe.gießen();
        efeu.gießen();
    }
}

class Pflanze {
    String name;

    public void benenne(String name) {
        this.name = name;
    }

    public void gießen() {
        System.out.println(name + " waechst.");
    }
}
```

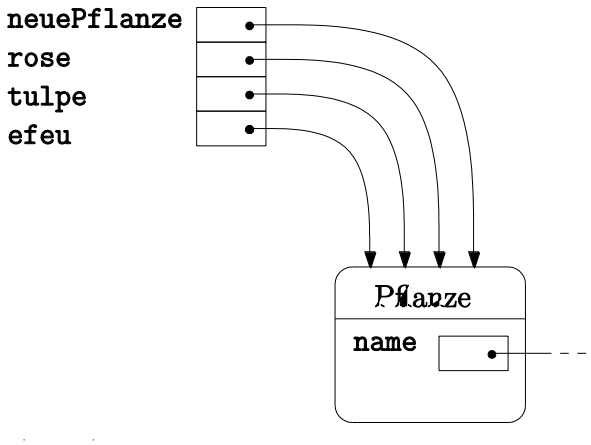
Welche Ausgabe erzeugt die `main`-Methode?

```
Efeu waechst.
Efeu waechst.
Efeu waechst.
```

(Fortsetzung der Aufgabe auf folgender Seite)

(Fortsetzung von Aufgabe 1)

Zeichnen Sie ein Objektgeflecht, das mindestens alle `Pflanze`-Objekte zeigt, die bei Ausführung der `main`-Methode entstanden sind.



Erklären Sie mit Hilfe Ihres gezeichneten Objektgeflechts und unter Verwendung der Begriffe „Alias“, „Referenz“ und „Objekt“, wie die Ausgabe zustande gekommen ist.

Bei Aufruf der `main`-Methode wird ein `Pflanze`-Objekt und eine Referenz `neuePflanze` auf dieses erzeugt. Diese Referenz wird anschließend den neu deklarierten Variablen `rose`, `tulpe` und `efeu` zugewiesen. Alle vier Referenzen zeigen auf dasselbe Objekt und sind somit Aliase (zueinander). Entsprechend wirkt sich das Benennen einer Pflanze auf Zugriffe über alle Referenzen aus.

Aufgabe 2: Subtyping (10 Punkte)

Gegeben sei folgendes Programm:

```
class Tier {
    private String tierart;

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

    public void gibLaut() {
        if (tierart.equals("Ente"))
            System.out.println("Nak! Nak!");
        else if (tierart.equals("Baer"))
            System.out.println("Brumm!");
        else
            throw new RuntimeException("Unbekanntes Tier!");
    }
}

class Main {
    public static void main(String[] args) {
        Tier ente = new Tier("Ente");
        Tier baer = new Tier("Baer");
        ente.gibLaut();
        baer.gibLaut();
    }
}
```

Welche Ausgabe erzeugt die `main`-Methode?

```
Nak! Nak!
Brumm!
```

Die obige Implementierung der Tiere wirft mehrere Probleme auf. So muss z.B. die Fallunterscheidung in der `gibLaut()`-Methode jedes Mal angepasst werden, wenn eine neue Tierart hinzukommen soll. Auch wäre es wünschenswert, auf die `RuntimeException` komplett verzichten zu können und stattdessen schon den Compiler sicherstellen zu lassen, dass keine unbekannten Tiere verwendet werden.

(Fortsetzung der Aufgabe auf folgender Seite)

(Fortsetzung von Aufgabe 3)

Implementieren Sie obiges Programm noch einmal, wobei Sie die angesprochenen Probleme lösen, indem Sie `Tier` als abstrakte Klasse implementieren und `Ente` und `Baer` 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 ente = new Ente();
        Tier baer = new Baer();
        ente.gibLaut();
        baer.gibLaut();
    }
}
```

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

class Ente extends Tier {
    public void gibLaut() {
        System.out.println("Nak! Nak!");
    }
}

class Baer extends Tier {
    public void gibLaut() {
        System.out.println("Brumm!");
    }
}
```

Aufgabe 3: Überschreiben und Überladen (10 Punkte)

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.

Damit eine Methode eine andere Methode überschreibt, müssen die nicht generischen Parametertypen beider Methode identisch sein.	☑ Wahr	☐ Falsch
Eine abstrakte Methode als `private` zu deklarieren kann sinnvoll sein.	☐ Wahr	☑ Falsch
Beim dynamischen Binden prüft der Compiler zur Laufzeit die Typen der Methodenparameter.	☐ Wahr	☑ Falsch
Für die virtuelle Maschine ist im Bytecode klar ersichtlich, welche Methoden sich überschreiben.	☑ Wahr	☐ Falsch
Feldzugriffe werden statisch gebunden, es ist dem Compiler also immer bekannt, an welches Feld eine Feldreferenz bindet.	☑ Wahr	☐ Falsch
Der Compiler löst Methoden-Überladung mit einem Most-Specific-Algorithmus auf.	☑ Wahr	☐ Falsch
Die virtuelle Maschine löst Methoden-Überschreibung mit einem Most-Specific-Algorithmus auf.	☐ Wahr	☑ Falsch
Bei überschriebenen Methoden kann es passieren, dass zur Laufzeit keine passende Methode gefunden wird, so dass eine Exception geworfen wird.	☐ Wahr	☑ Falsch
Mit dem Schlüsselwort `final` kann ein Programmierer verhindern, dass eine Methode überschrieben wird.	☑ Wahr	☐ Falsch
Mit dem Schlüsselwort `final` kann ein Programmierer verhindern, dass andere Klassen von einer bestimmten Klasse ableiten.	☑ Wahr	☐ Falsch

Aufgabe 4: Polymorphie (10 Punkte)

Gegeben sind eine Klasse `Super` und ihre Subklasse `Sub`:

```
class Super {}
class Sub extends Super {}
```

Geben Sie für folgende Zuweisungen an, ob diese zu einem Kompilierfehler, Laufzeitfehler (in Form einer Exception) oder zu keinem Fehler führen.

`Sub test1 = (Super) new Super();`	Kompilierfehler
`Super test2 = (Super) new Sub();`	Kein Fehler
`Super test3 = (Sub) new Super();`	Laufzeitfehler
`Super test4 = (Sub) new Sub();`	Kein Fehler
`Sub test5 = (Sub) new Super();`	Laufzeitfehler
`Sub test6 = (Super) new Sub();`	Kompilierfehler

Zählen Sie alle im Kurs genannten Arten von Polymorphie auf:

- Parametrische Polymorphie
- Beschränkt parametrische Polymorphie
- Subtyp-Polymorphie
- Ad-hoc-Polymorphie

Aufgabe 5: Ausnahmebehandlung (10 Punkte)

Es soll eine Klasse in Java implementiert werden, welche ein Warenlager repräsentiert. Um ein Entnehmen von Waren aus einem leeren Lager zu verhindern, soll gegebenenfalls eine Ausnahme in Form einer `LagerIstLeerException` ausgelöst werden, sobald dieser Fall eintritt.

Ergänzen Sie das folgende Programm an den vorgesehenen Stellen um solch eine Ausnahmebehandlung:

```
class Lager {
   int lagerstand = 0;

   public void lagere(int anzahl) {
      lagerstand = lagerstand + anzahl;
   }

   public void entnehme(int anzahl)  throws LagerIstLeerException  {

      if (lagerstand - anzahl < 0)
         throw new LagerIstLeerException();

      lagerstand = lagerstand - anzahl;
   }
}

class LagerIstLeerException  extends Exception  {
}

public class LagerTest {

   public static void main(String[] args) {
      Lager lager = new Lager();
      lager.lagere(3);

      try {

         lager.entnehme(2);

      } catch (LagerIstLeerException e) {
         // Hier weitere Ausnahmebehandlung

      }

   }
}
```

Aufgabe 6: Threads (10 Punkte)

Gegeben sei das folgende Programm:

```
class Wert {
    int wert;
    Wert(int wert) {
        this.wert = wert;
    }
}

class Tausche extends Thread {
    Wert a,b;

    Tausche(Wert a, Wert b) {
        this.a = a;
        this.b = b;
    }
    public void run() {
        int h = a.wert;
        a.wert = b.wert;
        b.wert = h;
    }
    public static void main(String[] args) {
        Wert x = new Wert(0);
        Wert y = new Wert(1);
        Tausche tom = new Tausche(x,y);
        Tausche jerry = new Tausche(y,x);
        tom.start();
        jerry.start();
    }
}
```

Argumentieren Sie, warum nach Beendigung beider Threads – also dem zweifachen Tauschen der Werte – diese nicht zwangsläufig wieder ihre Ursprungsbelegung haben.

Da nicht gewährleistet ist, dass die Ausführung der `run`-Methode unterbrechungsfrei geschieht und die Werte von `a` und `b` währenddessen durch den jeweils anderen Thread verändert werden können, ist ein zweimaliger Tausch beider Werte nicht gesichert.

Führt `tom` zunächst

```
int h = a.wert;
```

aus und wird dann durch `jerry` unterbrochen, welcher die `run`-Methode komplett durchläuft, bevor `tom` fortfährt, haben bei Beendigung beider Threads sowohl `x` als auch `y` den Wert `0`.

(Fortsetzung der Aufgabe auf folgender Seite)

(Fortsetzung von Aufgabe 6)

Erklären Sie, warum es keine gute Idee ist, das Problem durch folgende Änderung in der `run`-Methode zu beheben,

```
public void run() {
    synchronized (a) {
        synchronized (b) {
            int h = a.wert;
            a.wert = b.wert;
            b.wert = h;
        }
    }
}
```

insbesondere, bei Beachtung der beiden Zeilen

```
Tausche tom = new Tausche(x,y);
Tausche jerry = new Tausche(y,x);
```

Es besteht nun die Gefahr einer Verklemmung. Wenn `tom` unterbrochen wird, nachdem er den Monitor `a` (alias `x`) betreten hat und nun `jerry` den Monitor `a` (alias `y`) betritt, warten anschließend beide Threads auf die Freigabe des jeweils anderen Monitors durch den jeweils anderen Thread.

Welche Lösung schlagen Sie vor, damit keines der beiden erwähnten Probleme auftritt?

Man kann ein neues Objekt `lock` einführen und auf dieses synchronisieren.

```
static Object lock = new Object();

public void run() {
    synchronized (lock) {
        int h = a.wert;
        a.wert = b.wert;
        b.wert = h;
    }
}
```

Aufgabe 7: Objektorientierte Modellierung (10 Punkte)

Implementieren Sie fünf Typen `Auto`, `Boot`, `Fahrzeug`, `MitReifen` und `Schubkarre`, so dass die folgenden Bedingungen erfüllt sind:

- `Boot` ist Subtyp von `Fahrzeug`
- `Auto` ist Subtyp von `Fahrzeug`
- `Auto` ist Subtyp von `MitReifen`
- `Schubkarre` ist Subtyp von `MitReifen`
- `Boot` ist **kein** Subtyp von `MitReifen`
- `Schubkarre` ist **kein** Subtyp von `Fahrzeug`
- Es kann **keine** Instanzen von `Fahrzeug` und von `MitReifen` geben

Weiterhin sollen der Typ `MitReifen` eine Methode `public int anzahlReifen()` und der Typ `Fahrzeug` eine Methode `public String fahrzeugArt()` bekommen, die geeignet zu implementieren sind.

```
abstract class Fahrzeug {
   public abstract String fahrzeugArt();
}

class Boot extends Fahrzeug {
   public String fahrzeugArt() {
      return "Boot";
   }
}

interface MitReifen {
   public int anzahlReifen();
}

class Auto extends Fahrzeug implements MitReifen {
   public String fahrzeugArt() {
      return "Auto";
   }
   public int anzahlReifen() {
      return 4;
   }
}

class Schubkarre implements MitReifen {
   public int anzahlReifen() {
      return 1;
   }
}
```

Aufgabe 8: This (10 Punkte)

Schreiben Sie eine **vollständige und kompilierende Java-Klasse**, welche das Schlüsselwort `this` genau dreimal und für unterschiedliche Zwecke verwendet, und zwar…

- … einmal, um im Rumpf einer Methode das Objekt zu referenzieren, dem die Nachricht geschickt wurde.
- … einmal, um in einem Konstruktor das Objekt zu referenzieren, das gerade initialisiert wird.
- … einmal, um einen anderen Konstruktor derselben Klasse aufzurufen.

```
public class Test {

    int i;

    public Test() {
        this(0);
    }

    public Test(int i) {
        this.i = i;
    }

    int getI() {
        return this.i;
    }
}
```

Zu was führt der Gebrauch von `this` im Rumpf einer statischen Methode?

Zu einem Kompilierfehler.

Aufgabe 9: Aufzählungstypen (10 Punkte)

Ergänzen Sie die unten stehende Implementierung des Aufzählungstypen `Tier` mit den Elementen `GIRAFFE`, `HUHN` und `BLATTLAUS` und mit Methoden `istGroesserAls`, `istGleichgrossWie` und `istKleinerAls`. Dabei soll eine Giraffe größer als ein Huhn und eine Blattlaus, ein Huhn größer als eine Blattlaus, Tiere gleicher Art gleichgroß und ein Tier kleiner als ein anderes sein, wenn es nicht gleich groß oder größer als das andere ist.
Die `main`-Methode zeigt Ihnen, wie die Signaturen der Methoden zu gestalten sind.

```
public enum Tier {
```

```
   GIRAFFE, HUHN, BLATTLAUS;

   public boolean istGroesserAls(Tier anderesTier) {
      if (this == GIRAFFE)
         return anderesTier != GIRAFFE;
      else if (this == HUHN)
         return anderesTier == BLATTLAUS;
      else
         return false;
   }

   public boolean istGleichgrossWie(Tier anderesTier) {
      return this == anderesTier;
   }

   public boolean istKleinerAls(Tier anderesTier) {
      return !istGroesserAls(anderesTier)
            && !istGleichgrossWie(anderesTier);
   }
```

```
   public static void main(String[] args) {
       System.out.println(GIRAFFE.istGleichgrossWie(GIRAFFE));    // true
       System.out.println(HUHN.istGroesserAls(BLATTLAUS));        // true
       System.out.println(BLATTLAUS.istKleinerAls(BLATTLAUS));    // false
   }
}
```

Aufgabe 10: Kontrollstrukturen (10 Punkte)

Implementieren Sie die folgenden drei Methoden, die jeweils des Produkt der ganzen Zahlen $1 \cdot 2 \cdot \ldots \cdot n$ für ein $n \geq 1$ berechnen sollen, jeweils einmal mit Hilfe einer for-Schleife, einer while-Schleife und mittels Rekursion. Ungültige Eingaben (z.B. eine Eingabe $n < 1$) brauchen Sie **nicht** abzufangen.

```
public class Schleifen {

   public int forProdukt(int n) {
```

```
      int ergebnis = 1;
      for (int i = 1; i <= n; i++)
         ergebnis *= i;
      return ergebnis;
```

```
   }
   public int whileProdukt(int n) {
```

```
      int ergebnis = 1;
      while (n > 1) {
         ergebnis *= n;
         n--;
      }
      return ergebnis;
```

```
   }
   public int rekursivesProdukt(int n) {
```

```
      if (n == 1)
         return 1;
      return n * rekursiveSumme(n - 1);
```

```
   }
}
```

Zusätzlicher Platz für Ihre Lösungen

Ergänzung zu Aufgabe Nr.

Ergänzung zu Aufgabe Nr.

Zusätzlicher Platz für Ihre Lösungen

Ergänzung zu Aufgabe Nr.

Ergänzung zu Aufgabe Nr.