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“

Klausur am 10.9.2011

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 Wintersemester 2011/2012 statt.

Hagen, den 21.9.2011

Im Auftrag

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 keinem 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 FlugVerwaltung {
   public static void main(String[] args) {
      Person unbekannt = new Person("unbekannt");
      Flugzeugbesatzung besatzung = new Flugzeugbesatzung(unbekannt,unbekannt);

      besatzung.pilot.umbenennen("Anna Becker");
      besatzung.copilot.umbenennen("Carl Daum");

      System.out.println(besatzung);
   }
}

class Person {
   private String name;
   public Person(String name) {
      this.name = name;
   }
   public void umbenennen(String neuerName) {
      this.name = neuerName;
   }
   public String toString() {
      return name;
   }
}

class Flugzeugbesatzung {
   Person pilot;
   Person copilot;

   public Flugzeugbesatzung(Person pilot, Person copilot) {
      this.pilot = pilot;
      this.copilot = copilot;
   }
   public String toString() {
      return "Pilot: " + pilot + "  Copilot: " + copilot;
   }
}
```

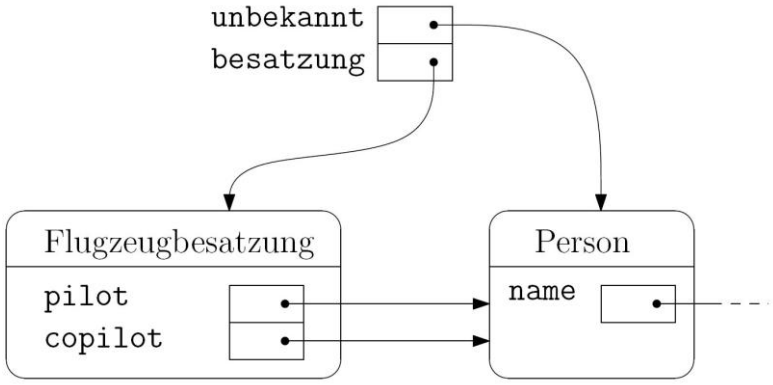
Welche (wohlmöglich unerwartete) Ausgabe erzeugt die `main`-Methode?

```
Pilot: Carl Daum  Copilot: Carl Daum
```

(Fortsetzung der Aufgabe auf folgender Seite)

(Fortsetzung von Aufgabe 1)

Zeichnen Sie ein Objektgeflecht, das mindestens alle `Person`- und `Flugzeugbesatzung`-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 `Person`-Objekt und eine Referenz `unbekannt` auf dieses erzeugt. Diese Referenz wird anschließend dem `pilot`- und `copilot`-Attribut im neu erzeugten `Flugbesatzung`-Objekt zugewiesen. Alle drei Referenzen zeigen auf dasselbe Objekt und sind somit Aliase (zueinander). Entsprechend wirkt sich das Umbenennen dieser einen Person auf Zugriffe über alle Referenzen aus.

Aufgabe 2: Ausnahmebehandlung (10 Punkte)

Schreiben Sie ein Java-Programm, welches 4 Typen implementieren soll:
`Lager`
`LagerException`
`LagerIstVollException`
`LagerIstLeerException`

`LagerException` soll Supertyp von `LagerIstVollException` und `LagerIstLeerException` sein. Ein Lager soll die parameterlosen `void`-Methoden `liefern()` und `abholen()` implementieren und einen internen Zähler haben, der garantiert, dass die Methode `liefern()` maximal hundert mal häufiger aufgerufen wurde als die Methode `abholen()` und dass `abholen()` nicht häufiger aufgerufen wurde als `liefern()`. Tritt der erste Fall auf, soll eine `LagerIstVollException` geworfen werden, tritt der zweite Fall auf, soll eine `LagerIstLeerException` geworfen werden.

```
class Lager {

   private int lagerstand;
   private static final int MIN = 0;
   private static final int MAX = 100;

   void liefern() throws LagerIstVollException {
      if(lagerstand >= MAX)
         throw new LagerIstVollException();
      lagerstand++;
   }

   void abholen() throws LagerIstLeerException {
      if(lagerstand <= MIN)
         throw new LagerIstLeerException();
      lagerstand--;
   }
}

class LagerException extends Exception {}
class LagerIstLeerException extends LagerException {}
class LagerIstVollException extends LagerException {}
```

Aufgabe 3: Methodenbindung (10 Punkte)

Dynamisches Binden bei Methodenaufrufen bezeichnet den Vorgang, durch den einem im Quellcode stehenden Methodenaufruf die Methode zugeordnet wird, welche dann tatsächlich ausgeführt wird. Dieser Vorgang erfolgt bei Java in zwei Stufen.

Was erfolgt in der ersten Stufe durch den Compiler?

Durch den Compiler erfolgt die Auflösung von Überladung nach dem Most-Specific-Algorithmus. Dabei werden nur die Deklarationstypen (statischen Typen) des Empfängers des Methodenaufrufs sowie der Parameter berücksichtigt. Das Ergebnis dieses Vorgangs ist der Aufruf einer bestimmten Methode, der durch den Compiler in den Bytecode geschrieben wird.

Was geschieht zur Laufzeit?

Die Laufzeitumgebung („Virtual Machine“, VM) ermittelt den tatsächlichen Typ (dynamischer Typ) des Objekts, an das sich der Methodenaufruf richtete. Das kann ein beliebiger Subtyp des Deklarationstyps sein und in diesem Typ könnte die betreffende Methode überschrieben worden sein.
In der dem dynamischen Typ entsprechenden Klasse sucht die VM dann nach einer Methode, deren Signatur genau dem Methodenaufruf aus dem Bytecode entspricht.

Findet sie diese Methode dort nicht, schaut sie in der Superklasse nach, ggf. in deren Superklasse und so weiter. Dass sich die Methode finden muss, ist klar, sonst hätte der Compiler
den entsprechenden Methodenaufruf gar nicht erst zugelassen.

(Fortsetzung der Aufgabe auf folgender Seite)

(Fortsetzung von Aufgabe 3)

Welche Ausgabe erzeugt das folgende Java-Programm?

```
public class MethodenBinder {
   public static void main(String[] args) {
      A aa = new A();
      A ab = new B();

      aa.m(new X(), new Y());
      ab.m(new X(), new Y());
      aa.m(new Y(), new Y());
      ab.m(new Y(), new Y());

   }
}

class A {
   void m(X x, X xx) {
      System.out.println("A.m(X,X)");
   }
   void m(X x, Y y) {
      System.out.println("A.m(X,Y)");
   }
}

class B extends A {
   void m(X x, Y y) {
      System.out.println("B.m(X,Y)");
   }
   void m(Y y, Y yy) {
      System.out.println("B.n(Y,Y)");
   }
}

class X {}
class Y extends X {}
```

```
A.m(X,Y)
B.m(X,Y)
A.m(X,Y)
B.m(X,Y)
```

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 Compilierfehler, Laufzeitfehler (in Form einer Exception) oder zu keinem Fehler führen.

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

Gegeben sei folgendes Programmstück. Begründen Sie, warum der Compiler dieses zu Recht wegen fehlender Typsicherheit mit einem Compilierfehler in der zweiten Zeile versieht.

```
(01)    java.util.Stack<Sub> subStack = new java.util.Stack<Sub>();
(02)    java.util.Stack<Super> superStack = subStack;
(03)    superStack.push(new Super());
(04)    Sub sub = subStack.pop();
```

Die Zuweisung ist nicht Typsicher, da anschließend `subStack` und `superStack` auf dasselbe Objekt (also denselben `Stack`) verweisen. Somit kann in Zeile 3 über die Referenz `superStack` ein Objekt vom Typ `Super` auf den `Stack` gelegt werden und über die Referenz `subStack` in Zeile 4 wieder entnommen werden. Laut seinem deklarierten Typen implementiert das entnommene Element dann alle Methoden von `Sub` ist aber in Wirklichkeit nur vom Typ `Super`.

Aufgabe 5: (Beschränkt) Parametr. Polymorphie (10 Punkte)

Implementieren Sie eine Datenstruktur `Tupel` mit einem Konstruktor und Methoden `getLinks` und `getRechts`, welche zwei Referenzen auf Objekte aufnehmen kann und sich in folgender Art und Weise verwenden lässt:

```
class Main{
   public static void main(String[] args) {
      Tupel<String> t = new Tupel<String>("a", "b");
      String l = t.getLinks();  // "a"
      String r = t.getRechts(); // "b"
   }
}
```

```
public class Tupel<T> {
   private T links;
   private T rechts;

   public Tupel(T links, T rechts) {
      this.links = links;
      this.rechts = rechts;
   }

   public T getLinks() {
      return links;
   }

   public T getRechts() {
      return rechts;
   }
}
```

(Fortsetzung der Aufgabe auf folgender Seite)

(Fortsetzung von Aufgabe 5)

Gegeben sei zusätzlich folgendes Interface `Druckbar`:

```
interface Druckbar {
  public void drucken();
}
```

Implementieren Sie **unter Verwendung von beschränkt parametrischer Polymorphie** die Klasse `Tupel` noch einmal so, dass sie für den Typparameter nur druckbare Objekte zulässt. Implementieren Sie für `Tupel` zusätzlich eine Methode `beideDrucken()`, die zuerst auf dem linken und dann auf dem rechten Objekt die Methode `drucken()` aufruft.

```
public class Tupel<T extends Druckbar> {
   private T links;
   private T rechts;

   public Tupel(T links, T rechts) {
      this.links = links;
      this.rechts = rechts;
   }

   public T getLinks() {
      return links;
   }

   public T getRechts() {
      return rechts;
   }

   public void beideDrucken() {
      links.drucken();
      rechts.drucken();
   }
}
```

Aufgabe 6: This (10 Punkte)

Für das Schlüsselwort `this` gibt es in Java mehrere Verwendungen.

Was bezeichnet `this` im Rumpf einer Methode?

`this` bezeichnet im Rumpf einer Methode das Objekt, dem die Nachricht geschickt wurde, die zur Ausführung der Methode führte, auch als "impliziter Parameter" des Methodenaufrufs bezeichnet.

Was bezeichnet `this` innerhalb eines Konstruktors?

In einem Konstruktor bezeichnet `this` das Objekt, das gerade initialisiert wird.

An welchen Stellen im Programm ist ein Aufruf `this()` möglich und was bedeutet er?

Der Ausdruck `this()` ruft einen anderen (in diesem Fall parameterlosen) Konstruktor desselben Objekts auf. Er muss der erste Ausdruck innerhalb eines Konstruktors sein.

Warum führt die Verwendung von `this` innerhalb des Rumpfs einer statischen Methode (mit Ausnahme von Konstruktoren) zu einem Compilerfehler?

`this` bezeichnet im Rumpf einer Methode das Objekt, dem die Nachricht geschickt wurde. Statische Methoden werden aber nicht auf einem konkreten Objekt, sondern nur auf einer Klasse (bzw. Typ) aufgerufen, weswegen die Verwendung von `this` hier keinen Sinn ergibt.

Aufgabe 7: Objektorientierte Modellierung (10 Punkte)

Implementieren Sie fünf Typen `Spinne`, `Fisch`, `Tisch`, `MitBeinen` und `Tier`, so dass die folgenden Bedingungen erfüllt sind:

- `Fisch` ist Subtyp von `Tier`
- `Spinne` ist Subtyp von `Tier`
- `Spinne` ist Subtyp von `MitBeinen`
- `Tisch` ist Subtyp von `MitBeinen`
- `Fisch` ist **kein** Subtyp von `MitBeinen`
- `Tisch` ist **kein** Subtyp von `Tier`
- Es kann **keine** Instanzen von `Tier` und von `MitBeinen` geben

Weiterhin soll der Typ `MitBeinen` eine Methode `public int anzahlBeine()` und der Typ `Tier` eine Methode `public String nameDerTierart()` bekommen, die geeignet zu implementieren sind.

```
abstract class Tier {
   public abstract String nameDerTierart();
}

class Fisch extends Tier {
   public String nameDerTierart() {
      return "Fisch";
   }
}

interface MitBeinen {
   public int anzahlBeine();
}

class Spinne extends Tier implements MitBeinen {
   public String nameDerTierart() {
      return "Spinne";
   }
   public int anzahlBeine() {
      return 8;
   }
}

class Tisch implements MitBeinen {
   public int anzahlBeine() {
      return 4;
   }
}
```

Aufgabe 8: Threads (10 Punkte)

In Java erzeugt man eigene Thread-Klassen durch Ableiten von der Klasse `Thread`. Umgekehrt kann in Java eine Klasse nur von einer anderen Klasse abgeleitet sein. Welche Lösung bietet einem Java an, wenn man ein `Thread`-Objekt gleichzeitig von einer anderen Klasse als `Thread` ableiten möchte?

Man kann ebenso auch das Interface `java.lang.Runnable` implementieren. Eine Instanz dieses Objekts kann dann einem `Thread` im Konstruktor übergeben und dieser kann anschließend wie gewöhnlich mit `start()` gestartet werden.

Gegeben sei folgendes Java-Programm:

```
(01)   class ThreadTest {
(02)      public static void main(String[] args) {
(03)         MeinThread t = new MeinThread();
(04)         t.start();
(05)         t.beenden();
(06)         t.stop();
(07)      }
(08)   }
(09)
(10)   class MeinThread extends Thread {
(11)      public void run() {
(12)         while(true) {
(13)            System.out.println("Thread gestartet.");
(14)            try {
(15)               Thread.sleep(500);
(16)            } catch (InterruptedException e) {
(17)               e.printStackTrace();
(18)            }
(19)         }
(20)      }
(21)      void beenden() {
(22)         try {
(23)            System.in.read();  // Enter-Taste gedrückt?
(24)         } catch (Exception e) {e.printStackTrace();}
(25)         System.out.println("Ich gehe...");
(26)      }
(27)   }
```

(Fortsetzung der Aufgabe auf folgender Seite)

(Fortsetzung von Aufgabe 8)

Während der Programmausführung werden zwei Threads erstellt. Welches ist der zuerst erstellte Thread?

Der erste Thread (T1) ist derjenige, der den Code der `main`-Methode ausführt.

Wo wird der zweite Thread gestartet?

Mit `t.start();` wird der zweite Thread (T2) gestartet.

Wo wird der erste Thread zum ersten Mal gestoppt?

Die `read()`-Anweisung in Zeile 23 innerhalb der Methode `beenden()` funktioniert wie ein Stopp-Signal: T1 wartet so lange, bis eine Zeile von der Konsole gelesen werden kann, also Return gedrückt wurde.

Wo wird der zweite Thread gestoppt?

Bei `t.stop();` in Zeile 6

Wann wird der erste Thread beendet?

Nach Beendigung der `main`-Methode.

Aufgabe 9: Aufzählungstypen (10 Punkte)

Ergänzen Sie die unten stehende Implementierung des Aufzählungstypen `ZooTier` mit den Elementen `TIGER`, `ENTE` und `SCHNECKE` und mit Methoden `istSchnellerAls`, `istGleichschnellWie` und `istLangsamerAls`. Dabei soll ein Tiger schneller als eine Ente und Schnecke sein, eine Ente schneller als eine Schnecke sein, Tiere gleicher Art gleichschnell sein und ein Tier langsamer als ein anderes sein, wenn es nicht gleich schnell oder schneller als das andere ist.
Die `main`-Methode zeigt Ihnen, wie die Signaturen der Methoden zu gestalten sind.

```
public enum ZooTier {
```

```
   TIGER, ENTE, SCHNECKE;

   public boolean istSchnellerAls(ZooTier anderesTier) {
      if (this == TIGER)
         return true;
      else if (this == ENTE)
         return anderesTier == SCHNECKE;
      else
         return false;
   }

   public boolean istGleichschnellWie(ZooTier anderesTier) {
      return this == anderesTier;
   }

   public boolean istLangsamerAls(ZooTier anderesTier) {
      return !istSchnellerAls(anderesTier)
            && !istGleichschnellWie(anderesTier);
   }
```

```
   public static void main(String[] args) {
      System.out.println(Tiger.istGleichschnellWie(Tiger));    // true
      System.out.println(Ente.istSchnellerAls(Schnecke));      // true
      System.out.println(Schnecke.istLangsamerAls(Schnecke)); // false
   }
}
```

Aufgabe 10: Kontrollstrukturen (10 Punkte)

Implementieren Sie die folgenden drei Methoden, die jeweils die Summe 0+1+2+…+n berechnen sollen: jeweils einmal mit Hilfe einer for-Schleife, einer while-Schleife und mittels Rekursion. Ungültige Eingaben (z.B. negative Zahlen) brauchen Sie **nicht** abzufangen.

```
public class Schleifen {

   public int forSumme(int n) {
```

```
      int ergebnis = 0;
      for(int i=1; i<=n; i++)
         ergebnis += i;
      return ergebnis;
```

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

```
      int ergebnis = 0;
      while (n>0) {
         ergebnis += n;
         n--;
      }
      return ergebnis;
```

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

```
      if(n==0)
         return 0;
      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.