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 2010

„Einführung in die objektorientierte Programmierung“

Nachklausur am 12.2.2011

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											

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

O Sie haben die Klausur leider nicht bestanden. Für den nächsten Versuch wünschen wir Ihnen viel Erfolg. Die nächste Klausur findet im Sommersemester 2011 statt.

Hagen, den 23.2.2011

Im Auftrag

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 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: Schleifen (10 Punkte)

Ergänzen Sie folgendes Java-Programm, so dass die Main-Methode die übergebenen Parameter nacheinander mittels `System.out.println` insgesamt viermal ausgibt. Jeweils einmal mit einer **while-Schleife**, einer **do-Schleife** und den **beiden Varianten** der **for-Schleife**.

```
public class Loops {
	public static void main(String[] args) {

		// mit while-Schleife über args iterieren
```

```
		int i = 0;
		while (i < args.length) {
			System.out.println(args[i++]);
		}
```

```
		// mit do-Schleife über args iterieren
```

```
		int j = 0;
		if (args.length > 0)
			do {
				System.out.println(args[j++]);
			} while (j < args.length);
```

```
		// mit erster Variante der for-Schleife über args iterieren
```

```
		for (int k = 0; k < args.length; k++) {
			System.out.println(args[k]);
		}
```

```
		// mit zweiter Variante der for-Schleife über args iterieren
```

```
		for (String arg : args) {
			System.out.println(arg);
		}
```

```
	}
}
```

Aufgabe 2: Abstrakte Klassen & Interfaces (10 Punkte)

Nennen Sie **zwei** Gemeinsamkeiten von abstrakten Klassen und Interfaces

- Abstrakte Klassen und Interfaces deklarieren Typen.
- Weder eine abstrakte Klasse noch ein Interface lassen sich instanziieren.

Nennen Sie **einen** Vorteil abstrakter Klassen gegenüber Interfaces

- Abstrakte Klassen können im Gegensatz zu Interfaces auch nicht-abstrakte Methoden implementieren, also ein Teilverhalten einer Klasse vorgeben.

Nennen Sie **einen** Vorteil von Interfaces gegenüber abstrakten Klassen

- Interfaces können in beliebiger Anzahl von einer Klasse implementiert werden. Während eine Klasse also maximal eine Superklasse haben kann, können ihr mit Hilfe von Interfaces darüber hinaus noch beliebig viele weitere direkte Supertypen gegeben werden.

Welche **drei** Compilerfehler erhalten Sie für folgendes Programm?

```
interface I { void m(); }
abstract class A { void n(); }
class B extends A implements I {
      void n(){}
      abstract void k();
}
```

- Die Klasse `B` muss die Methode `m` implementieren oder muss abstrakt sein.
- Die Methode `n` in `A` muss entweder abstrakt deklariert sein oder mit einem Methodenrumpf versehen werden.
- Die Methode `k` in `B` darf entweder nicht abstrakt deklariert werden oder `B` muss abstrakt sein.

Aufgabe 3: Polymorphie (10 Punkte)

Im folgenden Codebeispiel sind **vier Arten der Polymorphie** zu finden. Benennen Sie die vier Arten der Polymorphie und beschreiben Sie kurz, wo diese im Programm erkennbar werden.

```
class Polymorph<T extends Exception> {
      public void doSomethingWith(String s) {
            List<String> list = new LinkedList<String>();
            list.add(s);
            list.get(0).charAt(2);
      }
      public void doSomethingWith(T t) {
            List<T> list = new LinkedList<T>();
            list.add(t);
            list.get(0).printStackTrace();
      }
      public void doSomethingWith(Double d) {
            List list = new LinkedList();
            list.add(d);
            System.out.println(((Double)list.get(0)).isNaN());
      }
}
```

Die erste der drei Methoden von `Polymorph` gibt ein Beispiel für **parametrische Polymorphie**. Die Liste wird mit Typparameter `String` deklariert, so dass die `String`-Methode `charAt` auf einem entnommenen Listenelement aufgerufen werden kann.

Die zweite Methode der Klasse Polymorph gibt ein Beispiel für **beschränkt parametrische Polymorphie**. Hier kann durch die obere Typschranke `Exception` ein Element entnommen werden und die Methode `printStackTrace` aufgerufen werden, unabhängig davon, welche Arten von Ausnahmen in der Liste liegen.

Die dritte Methode gibt ein Beispiel für **Subtyp-Polymorphie**. Zwar können beliebige Subtypen von `Object` in die Liste eingefügt werden, allerdings müssen diese, um ihre über `Object` hinausgehenden Funktionalitäten nutzen zu können, nach Entnahme auf den passenden Typ gecastet werden.

Die drei Methoden gemeinsam geben ein Beispiel für **Ad-hoc-Polymorphie**. Dies ist ein anderer Begriff für Überladung und bezeichnet die Möglichkeit, mehrere Methoden mit gleichem Bezeichner, aber unterschiedlichen Parametertypen innerhalb einer Klassenhierarchie deklarieren zu können.

Aufgabe 4: static & innere Klassen (10 Punkte)

Statischen Methoden fehlt ebenso wie statischen inneren Klassen eine Eigenschaft, die den nichtstatischen Methoden bzw. nichtstatischen inneren Klassen implizit gegeben ist. Welche ist das?

Nichtstatische Methoden und nichtstatische innere Klassen haben eine implizite Referenz (bei Methoden auch impliziter Parameter genannt) auf das *this-Objekt* (auch *self-Objekt* genannt).

Geben Sie ein Java-Programm an, welches (neben beliebigen weiteren Deklarationen) eine nichtstatische Methode und eine nichtstatische innere Klasse enthält und die Verwendung dieser impliziten Eigenschaft zeigt. Genauer: Ihr Programm soll zu Compilerfehlern führen, wenn Ihre nichtstatische Methode oder Ihre nichtstatische innere Klasse mit dem Modifizierer `static` versehen wird. Markieren Sie die kritischen Stellen, die die Compilerfehler auslösen.

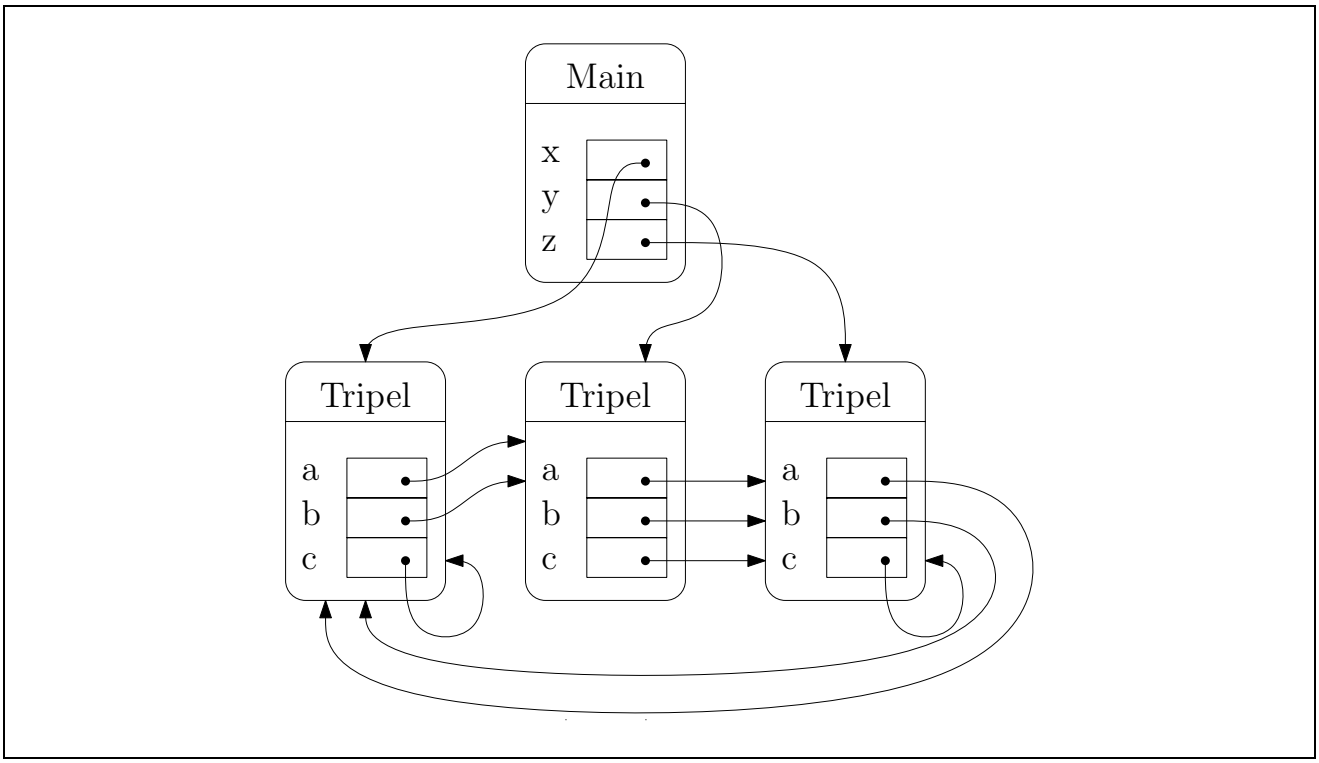
```
/*
 * In Java wird auf den impliziten Parameter bzw. die implizite
 * this-Referenz mit dem Schlüsselwort 'this' zugegriffen. Die
 * beiden 'this'-Aufrufe sind die kritischen Stellen, die zu
 * Compilerfehlern führen würden.
 */

class Outer {

      int i;

      void m(){
            i++;   // XXX (dasselbe wie 'this.i++;')
      }

      class Inner {
            int outerI = Outer.this.i; // XXX
      }
}
```

Aufgabe 5: Objektgeflechte & Serialisierung (10 Punkte)

Zeichnen Sie das Objektgeflecht, das bei Beendigung der Main-Methode entstanden ist.

```
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();
      }
}
```



Geben Sie **einen** Grund an, warum Serialisierung eine nichttriviale Aufgabe ist. Erläutern Sie dies anhand Ihrer Zeichnung.

Bei der Serialisierung eines Objektgeflechts müssen Zyklen erkannt werden, damit jedes Objekt nur einmal serialisiert wird. In obigem Beispiel referenziert `x.a` das Objekt `y`, `y.a` referenziert `z`, `z.a` referenziert `x`. `x` soll aber kein zweites Mal serialisiert werden.

Aufgabe 6: AWT (10 Punkte)

Gegeben sei folgendes Programm, welches in einem kleinen Fenster zwei Buttons anzeigen und bei Anklicken eines der beiden Buttons die Methode `buttonPressed` ausführen soll.

```
public class MyApplication extends java.awt.Frame {
      Button button1 = new Button("Button 1!");
      Button button2 = new Button("Button 2!");
      public MyApplication() {
            setSize(200,124);
            setTitle("My Application");
            add(button1);
            add(button2);
      }
      private void buttonPressed() {
            System.out.println("Button pressed!");
      }
      public static void main(String[] args) {
            MyApplication app = new MyApplication();
            app.addWindowListener(new WindowAdapter() {
                  public void windowClosing(WindowEvent e) {
                        System.exit(NORMAL);
                  }
            });
            app.setVisible(true);
      }
}
```

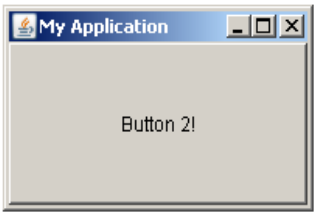


Abbildung 1:
Die resultierende Anzeige

Das Programm beinhaltet zwei Fehler. Einerseits wird nur einer der beiden Buttons angezeigt, andererseits haben Klicks auf den Button keinen Effekt.

Wo im Programm müssen welche Anweisungen ergänzt werden, damit beide Buttons sichtbar werden? (Größe und Anordnung der Buttons spielen dabei keine Rolle.) Sollten Ihnen konkrete Java-Anweisungen entfallen sein: beschreiben Sie in Worten, was fehlt.

Es fehlen Anweisungen an einen Layout-Manager, welcher die Anordnung von Komponenten eines Behälters berechnet. Beispielsweise durch Einfügen der Zeile

```
setLayout(new FlowLayout());
```

im Konstruktor von `MyApplication` würde ein Flow-Layout-Manager die beiden Buttons nebeneinander anordnen. Alternativ kann auch auf die explizite Angabe eines Layout-Managers verzichtet werden. In diesem Fall wird standardmäßig ein `BorderLayout` verwendet, für welches aber die Positionierungen der Komponenten angegeben werden müssen. Das kann beim Hinzufügen der Komponenten geschehen, indem man z.B. die beiden `add`-Aufrufe durch

```
add(button1,BorderLayout.NORTH);
add(button2,BorderLayout.SOUTH);
```

ersetzt.

(Fortsetzung der Aufgabe auf folgender Seite)

(Fortsetzung von Aufgabe 6)

Wie müssen Sie das Programm ergänzen, damit bei einem Klick auf `button1` bzw. `button2` die Methode `buttonPressed()` ausgeführt wird?

Den Buttons fehlt ein `ActionListener`, welcher auf Button-Tätigkeiten hört und auf diese hin die Methode `buttonPressed()` ausführt. Geschehen kann dies für `button1` im Konstruktor von `MyApplication` folgendermaßen.

```
button1.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
            buttonPressed();
      }
});
```

Für `button2` funktioniert es analog.

Nennen Sie drei Eigenschaften des AWT, die dieses als Programmgerüst kennzeichnen.

Die Klassen des AWT stellen ein Programmgerüst dar, denn

- Sie bilden ein System, d.h. sie wirken relativ eng zusammen.
- Mit Hilfe dieser Klassen lässt sich eine allgemeine, für viele Anwendungsprogramme relevante softwaretechnische Aufgabe lösen, nämlich die Realisierung graphischer Bedienoberflächen.
- Das System von Klassen ist erweiterbar und anpassbar.

Aufgabe 7: Methodenbindung & Ausnahmen (10 Punkte)

Methoden können in ihrer Signatur `throws`-Deklarationen enthalten. Was verlangt Ihnen der Compiler ab, wenn Sie eine solche Methode aufrufen? Gilt dies für jede Ausnahme?

Enthält eine Methodensignatur eine `throws`-Deklaration, so muss jeder Aufrufer die dort genannten Ausnahmen behandeln können. Dies kann entweder erfolgen, indem die aufrufende Methode selbst den entsprechenden Ausnahme-Typ in ihrer `throws`-Deklaration enthält oder indem der Aufruf in einem `try-catch`-Block erfolgt, der die Ausnahme abfangen kann.

Ausgenommen hiervon sind Ausnahmen, die von `Error` oder `RuntimeException` abgeleitet sind. Diese müssen nicht explizit behandelt werden.

Gegeben sei folgendes Java-Programm.

```
public class Test {
      public static void main(String[] args) {
            Object o1 = new Object();
            Object o2 = new String();
            Double d = new Double(0);
            String s = new String();
            A a = new B();
            a.m(o1);
            a.m(o2);
            a.m(d);
            a.m(s);
      }
}

class A {
      void m(String s) {System.out.println("A.m(String s)");}
      void m(Object o) {System.out.println("A.m(Object o)");}
      private void m(Double d) {System.out.println("A.m(Double d)");}
}

class B extends A {
      void m(String s) {System.out.println("B.m(String s)");}
      void m(Object o) {System.out.println("B.m(Object o)");}
}
```

(Fortsetzung der Aufgabe auf folgender Seite)

(Fortsetzung von Aufgabe 7)

Welche Ausgabe erzeugt es? Geben Sie für jeden der vier `m`-Aufrufe genau an, warum die Ausgabe wie von Ihnen vorhergesagt lautet.

(Hinweis: Hilfreiche Stichworte in Ihrer Erklärung könnten *Compiler*, *Laufzeit*, *überladen*, *überschreiben* und *nicht zugreifbar* sein).

Die Ausgabe lautet

```
B.m(Object o)
B.m(Object o)
B.m(Object o)
B.m(String s)
```

- Der erste Aufruf von `m` übergibt ein als `Object` typisierten Parameter an eine Methode `m` in `A` und bindet somit zunächst an `A.m(Object)`. Zur Laufzeit ist der Empfänger des Methodenaufrufs vom Typ `B`, so dass an die überschriebene Methode `B.m(Object o)` gebunden wird.

- Der zweite Aufruf von `m` übergibt ebenso ein als `Object` typisierten Parameter an eine Methode `m` in `A` und bindet somit auch zunächst an `A.m(Object)`. Dass zur Laufzeit ein Objekt vom Typ `String` übergeben wird, entzieht sich der Kenntnis des Compilers, so dass **nicht** an die überladene Methode `A.m(String)` gebunden wird. Zur Laufzeit ist der Empfänger des Methodenaufrufs vom Typ `B`, so dass an die überschriebene Methode `B.m(Object o)` gebunden wird.

- Der dritte Aufruf von m übergibt ein als `Double` typisierten Parameter an eine Methode `m` in `A`. Eine solche Methode `A.m(Double)` existiert zwar, allerdings ist sie (da private) nicht zugreifbar, weswegen an die einzige weitere passende überladene Methode `m(Object)` gebunden wird. Zur Laufzeit ist der Empfänger des Methodenaufrufs vom Typ `B`, so dass an die überschriebene Methode `B.m(Object o)` gebunden wird.

- Der vierte Aufruf von `m` übergibt ein als `String` typisierten Parameter an eine Methode `m` in `A` und bindet somit zunächst an `A.m(String)`. Zur Laufzeit ist der Empfänger des Methodenaufrufs vom Typ `B`, so dass an die überschriebene Methode `B.m(String s)` gebunden wird.

Aufgabe 8: Aufzählungstypen (10 Punkte)

Es sollen mit einem Aufzählungstyp verschiedene Tiere modelliert werden: Mäuse, Katzen und Elefanten. Ebenso soll auf den Elementen des Aufzählungstyps anhand von geeigneten Methoden feststellbar sein, ob ein Tier vor einem anderen Tier flieht, mit ihm Freundschaft schließt oder es verjagt. Elefanten verjagen Katzen, Katzen verjagen Mäuse und Mäuse verjagen Elefanten. Tiere gleicher Art schließen Freundschaft, und ein Tier flieht, wenn es verjagt wird.

Ergänzen Sie folgendes Programmfragment, damit die Main-Methode die in den Kommentaren angegebenen Ausgaben liefert.

```
public enum Tier {
```

```
    ELEFANT, KATZE, MAUS;


    public boolean schliesstFreudschaftMit(Tier tier) {
        return this == tier;
    }


    public boolean verscheucht(Tier tier) {
        switch (tier) {
        case ELEFANT:
            return this == MAUS;
        case KATZE:
            return this == ELEFANT;
        default: //case MAUS:
            return this == KATZE;
        }
    }


    public boolean wirdVerscheuchtVon(Tier tier) {
        return(tier.verscheucht(this));
    }
```

```
    public static void main(String[] args) {
        System.out.println(ELEFANT.verscheucht(KATZE));   // true
        System.out.println(ELEFANT.verscheucht(ELEFANT));    // false
        System.out.println(ELEFANT.schliesstFreudschaftMit(ELEFANT)); // true
        System.out.println(ELEFANT.schliesstFreudschaftMit(KATZE));// false
        System.out.println(ELEFANT.wirdVerscheuchtVon(MAUS)); // true
        System.out.println(ELEFANT.wirdVerscheuchtVon(ELEFANT));  // false
    }
}
```

Aufgabe 9: 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;
```

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 9)

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;
            }
      }
}
```

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;

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

Aufgabe 10: Verteilte Systeme (10 Punkte)

Der Kurstext nennt **vier** verschiedene Kommunikationsmittel für verteilte Systeme, wovon zwei primär zur asynchronen und zwei primär zur synchronen Kommunikation eingesetzt werden. Nennen Sie diese und ordnen Sie sie der asynchronen bzw. synchronen Kommunikation zu.

Asynchrone Kommunikation:

- Kommunikation über gemeinsamen Speicher
- Kommunikation über Nachrichten

Synchrone Kommunikation:

- Entfernter Prozedur- bzw. Methodenaufruf
- Spezielle Kommunikationskonstrukte

Nennen Sie **zwei** Gründe, warum sich das objektorientierte Grundmodell besonders gut für die Programmierung verteilter Systeme eignet.

- Objekte fassen Daten und Operationen zu Einheiten mit klar definierten Schnittstellen zusammen; Objekte bzw. Klassen bilden demnach weitgehend unabhängige Programmteile, die sich auf verschiedene Prozesse bzw. Rechner verteilen lassen.

- Kommunikation zwischen Objekten über Nachrichten ist integraler Bestandteil des objektorientierten Grundmodells, sodass bereits alle wesentlichen Sprachmittel für die Kommunikation im Rahmen verteilter Systeme zur Verfügung stehen.

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.