Aufgabe 1: Nette Kleinigkeiten (8 Punkte)

a) Bestimmen Sie das Ergebnis des im folgenden Programmcode durch ** gekennzeichneten Ausdrucks und begründen Sie Ihre Antwort. **(2 Punkte)**

```
public class TestVonAusdruecken {
  public static void main(String[] args) {
    int x = 256;
    ** long y = (x - 255.59) * 15 < 0 ? 12000L : x - 12256L;
  }
}
```

Lösung:
y hat nach Auswertung von
`long y = (x - 255.59) * 15 < 0 ? 12000L : x - 12256L;`
den Wert -12000,
da wegen x = 256 (Initialisierungswert bei der Deklaration von x) `x - 255.59 > 0` ist und somit auch `(x - 255.59) * 15.`
Daher wird der zweite Ausdruck nach dem Fragezeichen ausgewertet:
256 - 12256 =-12000.

b) Sind die markierten Zuweisungen von Werten an die Attribute und Variablen ax, v2, ay korrekt? Begründen Sie Ihre Antwort. **(3 Punkte)**

```
class C {
  final int ax, ay = 9;
  C(){
    ax = 7;  // korrekt?
    m();
  }
  void m(){
    final int v1 = 4848, v2 = -3;
    v2 = 0;  // korrekt?
    ay = 34; // korrekt?
    ...
  }
}
```

Lösung:
Unveränderliche blocklokale Variablen müssen an der Deklarationsstelle initialisiert werden, unveränderliche Attribute entweder an der Deklarationsstelle oder in den Konstruktoren.

```
class C {
  final int ax, ay = 9;
  C(){
    ax = 7;  // KORREKT: Initialisierung im Konstruktor
             // und nicht an Deklarationsstelle
    m();
  }
```

```
  void m(){
    final int v1 = 4848, v2 = -3;
    v2 = 0;  // UNZULAESSIG: v2 ist unveraenderlich
    ay = 34; // UNZULAESSIG: ay ist unveraenderlich
    ...
  }
}
```

c) Sind die folgenden Zuweisungen zulässig? Geben Sie eine kurze Begründung Ihrer Antwort an. **(3 Punkte)**
(`Person` ist in dem Beispiel ein Klassentyp, `Student` ist Subtyp von `Person`.)

```
Object   ovar = new String[3];
Person[]  pfv = new Student[2];
Student[] sfv = new Person[2];
```

Lösung:

```
Object   ovar = new String[3];
Person[]  pfv = new Student[2];
Student[] sfv = new Person[2];  // unzulaessig
```

Die ersten beiden der folgenden Zuweisungen sind zulässig; die dritte wird vom Übersetzer nicht akzeptiert, denn:
Jeder Feldtyp ist ein Subtyp von `Object`. Eine Variable vom Typ `Object` kann also sowohl (Referenzen auf) Objekte beliebiger Klassentypen speichern, als auch (Referenzen auf) Felder. Darüber hinaus ist ein Feldtyp $CS[]$ genau dann ein Subtyp eines anderen Feldtyps $CT[]$, wenn CS ein Subtyp von CT ist.

Aufgabe 2: Gemischtes (11 Punkte)

a) Ist der folgende Programmausschnitt korrekt?
Begründen Sie Ihre Antwort. **(4 Punkte)**

```
(1)  String[] strfeld = new String[2];
(2)  Object[] objfeld = strfeld;
(3)  objfeld[0] =  new Object();

(4)  int strl   =  strfeld[0].length();
```

Lösung:

```
(1)  String[] strfeld = new String[2];
(2)  Object[] objfeld = strfeld;
(3)  objfeld[0] =  new Object();      // Laufzeitfehler!!!
                                      // ArrayStoreException
(4)  int strl   =  strfeld[0].length();
```

Die Zuweisung in Zeile (2) ist korrekt, da `String` ein Subtyp von `Object` ist, also `String[]` ein Subtyp von `Object[]`. Solange man nur lesend auf das von `objfeld` referenzierte Feld zugreift, entsteht aus der Zuweisung auch kein Problem, da jedes Objekt, das man aus dem Feld ausliest, vom Typ `String` ist, also auch zum Typ `Object` gehört.

Anders ist es, wenn man wie in Zeile (3) eine Komponente des Felds verändert, beispielsweise indem man ihr ein neues Objekt der Klasse `Object` zuweist. Aus Sicht der Variablen `objfeld`, die eine Referenz vom Typ `Object[]` hält, ist nach wie vor alles in Ordnung. Aus Sicht der Variablen `strfeld`, die eine Referenz auf dasselbe Feldobjekt besitzt, stimmt die Welt aber nicht mehr: Die Benutzer der Variablen `strfeld` glauben eine Referenz auf ein Feld zu haben, in dem ausschließlich String-Objekte eingetragen sind (oder `null`). Nach der Zuweisung in Zeile (3) speichert die nullte Komponente aber eine Referenz auf ein Objekt der Klasse `Object`. Der scheinbar harmlose Aufruf der Methode `length` aus Klasse `String` in Zeile (4) würde damit zu einer undefinierten Situation führen: Objekte der Klasse `Object` besitzen keine Methode `length`.

Dies widerspricht einem der zentralen Ziele der Typisierung objektorientierter Programme. Die vom Übersetzer durchgeführte Typprüfung soll nämlich insbesondere garantieren, dass Objekte nur solche Nachrichten erhalten, für die sie auch eine passende Methode besitzen. Java löst die skizzierte Problematik durch einen Kompromiss. Um einerseits Zuweisungen wie in Zeile (2) zulassen zu können und andererseits undefinierte Methodenaufrufe zu vermeiden, „verbietet“ es Zuweisungen wie in Zeile (3). Da man ein solches Verbot im Allgemeinen aber nicht statisch vom Übersetzer abtesten kann, wird zur Laufzeit eine Ausnahme erzeugt, wenn einer Feldkomponente ein Objekt von einem unzulässigen Typ zugewiesen wird; d.h. eine solche Zuweisung terminiert dann abrupt.

b) Gegeben ist der Schnittstellentyp `Lastwagen` mit den direkten Supertypen `Fahrbar` und `Fahrzeug` und den Methoden `fahren`, `getName`, `getFahrgestellnummer`, `isOldtimer`,

`hatMautConsole`.
Geben Sie die Deklaration aller drei Schnittstellen an und ordnen Sie die Methoden sinnvoll den einzelnen Schnittstellen zu. Überlegen Sie sich für alle Methoden eine sinnvolle Signatur. **(5 Punkte)**

Lösung:

```
interface Fahrbar {
  void fahren();
}

interface Fahrzeug {
  String  getName();
  int     getFahrgestellnummer();
  boolean isOldtimer( int datum );
}

interface Lastwagen extends Fahrzeug, Fahrbar {
  boolean hatMautConsole();
}
```

c) Definieren Sie eine Klasse `LinkedList`, die den Parametertyp `ET` hat, der durch die Schnittstellen `Druckbar` und `Konvertierbar` nach oben beschränkt ist. **(2 Punkte)**

Lösung:

```
public class LinkedList<ET extends Druckbar & Konvertierbar> {
}
```

Aufgabe 3: Aufzählungstypen (8 Punkte)

Gegeben ist die folgende Klasse Wochentag.

```
public class Wochentag {
  private int tag;

  public Wochentag() {
    tag = 0;
  }
  public void setTag(int i) throws KeinTagException {
  if (i < 0 || i > 6)
    throw new KeinTagException();
  else
     tag = i;
  }
  public int getTag() {
    return tag;
  }
  public void naechsterTag() {
    tag = (tag + 1) % 7;
  }
  public void vorhergehenderTag() {
    ...
  }
  public String toString() {
    switch(tag) {
      case 0: return "Montag";
      case 1: return "Dienstag";
      case 2: return "Mittwoch";
      case 3: return "Donnerstag";
      case 4: return "Freitag";
      case 5: return "Samstag";
      case 6: return "Sonntag";
      default: return "ERROR";
    }
  }
}

class KeinTagException extends Exception {
}
```

Implementieren Sie die Klasse `Wochentag` jetzt als Aufzählungstyp. Dabei soll das private Attribut `tag` seinen Namen in `aktuellerTag` ändern und seinen Typ in `Wochentag`. Die Signaturen der Methoden `setTag` und `getTag` sollen wie folgt abgeändert werden:

```
  public void setTag(Wochentag tag) { ... }
  public Wochentag getTag() { ... }
```

Beantworten Sie auch folgende Fragen:

1. Warum braucht die Methode `setTag` jetzt keine Ausnahme mehr zu werfen?
2. Was kann man zur Implementierung der Methode `toString` sagen?

Lösung:

Die Klasse `Wochentag` kann wie folgt als Aufzählungstyp implementiert werden. Dabei wurde das ehemalige Attribut `tag` ersetzt durch das Attribut `aktuellerTag`.

```
enum Wochentag {

  Montag,
  Dienstag,
  Mittwoch,
  Donnerstag,
  Freitag,
  Samstag,
  Sonntag;

  private Wochentag aktuellerTag = Wochentag.Montag;

  public void setTag(Wochentag tag) {
    aktuellerTag = tag;
  }

  public Wochentag getTag() {
    return aktuellerTag;
  }

  public void naechsterTag() {
    Wochentag[] tage = Wochentag.values();
    int i = aktuellerTag.ordinal();
    if (i == tage.length - 1) {
      aktuellerTag = tage[0];
    }
    else {
      aktuellerTag = tage[i + 1];
    }
  }

  public void vorhergehenderTag() {
    ...
  }
}
```

Die gestellten Fragen können wie folgt beantwortet werden:

1. Die Methode `setTag` braucht jetzt keine Ausnahme mehr zu werfen, da ihr Parametertyp `Wochentag` garantiert, dass ihr als aktuelle Parameter nur gültige Werte übergeben werden.

2. Die Methode `toString` braucht nicht mehr selbst implementiert zu werden, da jeder Aufzählungstyp diese Methode bereits mitbringt.

Aufgabe 4: Autoboxing (8 Punkte)

a) Sind die Basisdatentypen Subtypen von `Object`? **(1 Punkte)**
Was versteht man unter Autoboxing? **(3 Punkte)**

Lösung:

Die Basisdatentypen sind keine Subtypen von `Object`.
Ab der Java-Version 5.0 gibt es das sogenannte *Autoboxing*, das das automatische Verpacken von Werten in Wrapperobjekte sowie das automatische Entpacken aus Wrapperobjekten unterstützt. Man unterscheidet beim Autoboxing das *Boxing* und das *Unboxing*. Boxing konvertiert einen Basisdatentyp in seinen zugehörigen Wrappertyp und Unboxing konvertiert einen Wrappertyp in seinen zugehörigen Basisdatentyp.
Kommt in einem Programm z.B. ein Ausdruck `e` vom Typ `int` vor, wo ein Ausdruck vom Typ `Integer` erwartet wird, wird dieser Ausdruck automatisch in `new Integer(e)` konvertiert.
Umgekehrt, kommt im Programm ein Ausdruck `e` vom Typ `Integer` vor, wo ein Ausdruck vom Typ `int` erwartet wird, wird `e` automatisch nach `e.intValue()` konvertiert.

b) Ist der folgende Programmausschnitt korrekt?
Begründen Sie Ihre Antwort. **(4 Punkte)**

```
      // Version ab Java 5.0
      List<Integer> ints = new ArrayList<Integer>();
(*)   ints.add(1);
(**)  int n = ints.get(0);
```

Lösung:

Die Anweisungen in Zeile (*) und (**) sind korrekt, denn:
Zeile (*) zeigt den Vorgang des Boxings: Die `add`-Methode erwartet einen Parameter vom Typ `Integer`. Tatsächlich übergeben wird aber der Wert 1. Dieser wird automatisch umgesetzt nach `new Integer(1)`.
Zeile (**) demonstriert den umgekehrten Fall. `n` ist eine Variable vom Typ `int`, die `get`-Methode liefert aber einen Wert vom Typ `Integer`, der jetzt automatisch in einen `int`-Wert ausgepackt wird.

Aufgabe 5: Beobachter (10 Punkte)

Ergänzen Sie im folgenden Programmausschnitt in der Klasse `Fluss` die fehlenden Programmteile.

```
/*****************************************************************************/
/********                  Klasse Wasserbueffelbesitzer                  *********/
/*****************************************************************************/
/** Wenn der Wasserstand zu niedrig ist, kleinergleich 0,3 m, brauchen die Bueffel   **/
/** zusaetzliches Wasser.                                                            **/
/** Sobald die Wiesen ueberflutet werden, sollten die Bueffel in Sicherheit gebracht **/
/** werden, also bei einem Wasserstand zwischen 5 und 7 m.                           **/
/** Ab 7 m muessen die Bueffel schwimmen, was sie aber nur eine Weile aushalten.     **/
/** Daher sollten sie dann dringend aus dem Fluss gerettet werden.                   **/
/*****************************************************************************/
class Wasserbueffelbesitzer implements Beobachter{

  public void fallen(double wasserhoehe){
    if (wasserhoehe <= 0.3) {
      System.out.println("Wasserstand = " + wasserhoehe);
      System.out.println(" Achtung Bueffelbesitzer: Zu wenig Wasser im Fluss");
      System.out.println(" Bueffel zusaetzlich traenken!");
      System.out.println("");
    }
  }

  public void steigen(double wasserhoehe) {
    if ((wasserhoehe >= 5) && (wasserhoehe < 7)){
      System.out.println("Wasserstand = " + wasserhoehe);
      System.out.println(" Achtung Bueffelbesitzer: Wiesen ueberschwemmt");
      System.out.println(" Bueffel in den Stall bringen");
      System.out.println("");
    }
    else {
      if (wasserhoehe >=7){
        System.out.println("Wasserstand = " + wasserhoehe);
        System.out.println("Achtung: Bueffelbesitzer: Hochwasser");
        System.out.println("Bueffel aus Wasser retten!");
        System.out.println("");
      }
    }
  }
}



interface Beobachter {
  void steigen(double wasserhoehe);
  void fallen(double wasserhoehe);
}


/*****************************************************************************/
```

```
/**                          Klasse Fluss                                        **/
/*****************************************************************************/
/** Fuer diese Aufgabe benoetigen wir als reine Flussinformationen nur die     **/
/** Wasserhoehe. Auf sie greifen wir mit der Methode 'getHoehe' zu.            **/
/** Die Beobachter muessen verwaltet werden. Dazu werden sie in einer Liste namens **/
/** beobachterListe abgelegt. Dies geschieht mit der Methode 'anmelden-Beobachter'. **/
/** In der Methode 'setHoehe' werden die Beobachtungsmethoden 'steigen' und 'fallen'**/
/** fuer die jeweiligen Beobachter passend zum Flussverhalten aufgerufen.      **/
/*****************************************************************************/
import java.util.*;

public class Fluss{
  private double wasserhoehe; /** Wasserstand **/
  /*************************************************************/
  /**** Bitte ergaenzen!                                   ****/
  /**** Legen Sie eine Beobachter-Liste an;                ****/
  /**** verwenden Sie dabei das Konzept der generischen Typen. ****/
  /*************************************************************/

  /** Fluss-Konstruktor **/
  Fluss(double h){
    wasserhoehe = h;
    /*************************************************************/
    /**** Bitte ergaenzen!                                    ****/
    /**** Erzeugen Sie eine Beobachter-Liste.                 ****/
    /*************************************************************/
  }

  /*************************************************************/
  /**** Bitte ergaenzen!                                 ****/
  /**** Definieren Sie eine Methode 'anmeldenBeobachter',   ****/
  /**** die Beobachter zur Beobachterliste hinzufügt.       ****/
  /*************************************************************/
  /** Ermittlung des Wasserstandes **/
  public double getHoehe(){
    return wasserhoehe;
  }

  /** Setzen des neuen Wasserstandes                **/
  /** Vergleich zwischen altem und neuem Wasserstand **/
  void setHoehe( double neueHoehe ){
    double alteHoehe = wasserhoehe;
    wasserhoehe = neueHoehe;
    /*****************************************************************/
    /**** Bitte ergaenzen!                                       ****/
    /**** Fuehren Sie eine Iteration ueber die Liste der Beobachter    ****/
    /**** mittels der for-each Schleife durch, wobei Sie abhaengig     ****/
    /**** von der Wasserhoehe die Methoden 'steigen' und 'fallen' fuer ****/
    /**** jeden Beobachter aufrufen                              ****/
    /*****************************************************************/
  }
}
```

Lösung:

```
/*****************************************************************************/
/********                  Klasse Wasserbueffelbesitzer                 *********/
/*****************************************************************************/
/** Wenn der Wasserstand zu niedrig ist, kleinergleich 0,3 m, brauchen die Bueffel   **/
/** zusaetzliches Wasser.                                                            **/
/** Sobald die Wiesen ueberflutet werden, sollten die Bueffel in Sicherheit gebracht **/
/** werden, also bei einem Wasserstand zwischen 5 und 7 m.                           **/
/** Ab 7 m muessen die Bueffel schwimmen, was sie aber nur eine Weile aushalten.     **/
/** Daher sollten sie dann dringend aus dem Fluss gerettet werden.                   **/
/*****************************************************************************/
class Wasserbueffelbesitzer implements Beobachter{

  public void fallen(double wasserhoehe){
    if (wasserhoehe <= 0.3) {
      System.out.println("Wasserstand = " + wasserhoehe);
      System.out.println(" Achtung Bueffelbesitzer: Zu wenig Wasser im Fluss");
      System.out.println(" Bueffel zusaetzlich traenken!");
      System.out.println("");
    }
  }

  public void steigen(double wasserhoehe) {
    if ((wasserhoehe >= 5) && (wasserhoehe < 7)){
      System.out.println("Wasserstand = " + wasserhoehe);
      System.out.println(" Achtung Bueffelbesitzer: Wiesen ueberschwemmt");
      System.out.println(" Bueffel in den Stall bringen");
      System.out.println("");
    }
    else {
      if (wasserhoehe >=7){
        System.out.println("Wasserstand = " + wasserhoehe);
        System.out.println("Achtung: Bueffelbesitzer: Hochwasser");
        System.out.println("Bueffel aus Wasser retten!");
        System.out.println("");
      }
    }
  }
}


interface Beobachter {
  void steigen(double wasserhoehe);
  void fallen(double wasserhoehe);
}

/*****************************************************************************/
/**                         Klasse Fluss                                    **/
/*****************************************************************************/
/** Fuer diese Aufgabe benoetigen wir als reine Flussinformationen nur die          **/
/** Wasserhoehe. Auf sie greifen wir mit der Methode 'getHoehe' zu.                 **/
```

```
/** Die Beobachter muessen verwaltet werden. Dazu werden sie in einer Liste namens  **/
/** beobachterListe abgelegt. Dies geschieht mit der Methode 'anmelden-Beobachter'. **/
/** In der Methode 'setHoehe' werden die Beobachtungsmethoden 'steigen' und 'fallen'**/
/** fuer die jeweiligen Beobachter passend zum Flussverhalten aufgerufen.           **/
/*********************************************************************************/
import java.util.*;

public class Fluss{
  private double wasserhoehe; /** Wasserstand **/
  private ArrayList<Beobachter> beobachterListe; /** Liste der Beobachter **/

  /** Fluss-Konstruktor **/
  Fluss(double h){
    wasserhoehe = h;
    beobachterListe = new ArrayList<Beobachter>();
  }

  /** Beobachter werden in die Beobachterliste eingetragen **/
  public void anmeldenBeobachter(Beobachter b){
    beobachterListe.add(b);
  }

  /** Ermittlung des Wasserstandes **/
  public double getHoehe(){
    return wasserhoehe;
  }

  /** Setzen des neuen Wasserstandes                 **/
  /** Vergleich zwischen altem und neuem Wasserstand **/
  void setHoehe( double neueHoehe ){
    double alteHoehe = wasserhoehe;
    wasserhoehe = neueHoehe;

    /** Iteration ueber die Liste der Beobachter mittels der   **/
    /** for-each Schleife, Aufruf der Methoden 'steigen' und   **/
    /** 'fallen' fuer jeden Beobachter                         **/
    if (wasserhoehe > alteHoehe) {
      for (Beobachter b : beobachterListe)
        b.steigen(this.getHoehe());
    }
    else {
      for (Beobachter b : beobachterListe)
        b.fallen(this.getHoehe());
    }
  }
}
```

Aufgabe 6: Auflösen überladener Methodenaufrufe (10 Punkte)

a) Betrachten Sie folgenden Programmcode:

```
public class Objektgeflecht {
  Objektgeflecht a, b, c;

  public Objektgeflecht () {
    a = null;
    b = null;
    c = null;
  }

  public Objektgeflecht (Objektgeflecht a,
                         Objektgeflecht b,
                         Objektgeflecht c) {
    this.a = a;
    this.b = b;
    this.c = c;
  }

  public static void main (String argv[]) {
    Objektgeflecht u = new Objektgeflecht();
    Objektgeflecht v = new Objektgeflecht();
    Objektgeflecht w = new Objektgeflecht(u, v, null);

    (w.a).b = v;
    v.a = u.b;
    (u.b).c = w;
    w.c = v.c;
    u.c = (v.a).c;
    /* Markierung */
  }
}
```

Bei welchen Deklarationselementen tritt Überladung auf und bei welchen Anweisungen muss der Compiler eine Überladung auflösen?
Warum gelingt die Auflösung? **(4 Punkte)**

Lösung:

Der Konstruktorname der Klasse `Objektgeflecht` ist überladen. Der erste Konstruktor ist parameterlos, der zweite Konstruktor besitzt drei Parameter vom Typ `Objektgeflecht`. Auflösung ist nötig in den drei ersten Anweisungen der `main`-Methode, in denen der Konstruktor aufgerufen wird. Der Compiler kann die Aufrufe korrekt auflösen, da sich beide Konstruktoren in der Anzahl ihrer Parameter unterscheiden.

b) Betrachten Sie folgenden Programmcode:

```
class Haustier { ... }

class Hund extends Haustier { ... }
class Wachhund extends Hund { ... }

class Katze extends Haustier { ... }
class Leopard extends Katze { ... }

class TrautesHeim {
  void sichmoegen (Haustier a, Katze d) { ... }     // Deklaration #1
  void sichmoegen (Hund b, Haustier a) { ... }      // Deklaration #2
  void sichmoegen (Wachhund c, Katze d) { ... }     // Deklaration #3

  void SympathieTest() {
    Haustier a = new Haustier();
    Hund b = new Hund();
    Wachhund c = new Wachhund();
    Katze d = new Katze();
    Leopard e = new Leopard();

    sichmoegen(a,d);    // Aufruf-1
    sichmoegen(c,a);    // Aufruf-2
    sichmoegen(c,e);    // Aufruf-3
    sichmoegen(b,d);    // Aufruf-4
  }
}
```

Können die mit Aufruf-1 bis Aufruf-4 gekennzeichneten Methodenaufrufe erfolgreich aufgelöst werden? Wenn ja, geben Sie die jeweils zum Aufruf passende Deklaration an. Geben Sie für jeden Aufruf die Schritte an, die zur Auflösung des Aufrufs durchgeführt werden. **(6 Punkte)**

Lösung:

Allgemeines Vorgehen:
Die Auflösung überladener Methodenaufrufe wird in zwei Schritten vorgenommen.
Im ersten Schritt werden alle sichtbaren Methoden mit passendem Namen und passender Signatur bestimmt. Für die Signatur heißt das, dass die Typen der formalen Parameter (mittelbare) Supertypen der deklarierten Typen der Argumente oder — im Falle von Zahltypen — Zahltypen mit mindestens so großem Wertevorrat sein müssen. Wird in diesem Schritt genau eine solche Methode gefunden, dann wird diese aufgerufen; wenn keine solche Methode gefunden wird, wird die Übersetzung mit einem Fehler abgebrochen.
Ansonsten wird mit dem zweiten Schritt fortgefahren, dem eigentlichen most-specific-Algorithmus. Dieser streicht aus der im ersten Schritt bestimmten Menge nach und nach solche Methoden, für die es noch eine „speziellere“ Methode in der Menge gibt, d.h. eine Methode, deren Parametertypen (mittelbare) Subtypen bzw. Zahltypen mit gleichem oder kleinerem Wertevorrat sind. Bleibt nach diesem zweiten Schritt genau eine Methode übrig, so wird diese Methode aufgerufen. Ansonsten ist der Aufruf mehrdeutig,

d.h. es existiert keine im Sinne der Vererbungshierarchie speziellste Methodendeklaration. In diesem Fall kann der Aufruf zur Übersetzungszeit nicht aufgelöst werden und die Übersetzung bricht mit einem Fehler ab.

Eigentliche Lösung der Aufgabe:
Anwendung auf die vier Methodenaufrufe des Beispiels:
Die Aufrufe #1 und #2 sind nach dem ersten Schritt des obigen Verfahrens erfolgreich aufgelöst:
Aufruf #1 passt zu Deklaration #1, denn die Typen der formalen Parameter sind (mittelbare) Supertypen der deklarierten Typen der Argumente.
Aufruf #2 passt zu Deklaration #2, denn Wachhund ist Subtyp von Hund und für den zweiten Parameter stimmen die Typen überein.
Andere Deklarationen passen nicht.

Aufruf #3 ist erst nach dem zweiten Schritt erfolgreich aufgelöst. Nach dem ersten Schritt passen alle Deklarationen. Das Most-specific-Verfahren liefert dann die Methode mit der Deklaration #3. Diese ist spezieller als die Methode #1 und spezieller als #2.

Der letzte Aufruf hingegen, Aufruf #4, kann nicht erfolgreich aufgelöst werden. Im ersten Schritt werden die beiden ersten Deklarationen von `sichmoegen` als mögliche Kandidaten identifiziert. Keine dieser Deklarationen ist jedoch „spezieller“ als die jeweils andere. Folglich kann keine dieser Deklarationen im zweiten Schritt des most-specific-Verfahrens ausgeschieden werden. Der Aufruf bleibt mehrdeutig, die Übersetzung bricht mit einem entsprechenden Fehler ab.

Aufgabe 7: Ausnahmebehandlung (8 Punkte)

Gegeben ist das folgende Programm:

```
class Klausurortfehler extends Exception { }

class Matrikelnummerfehler extends Exception { }

class Nachnamenfehler extends Exception { }

class Vornamenfehler extends Exception { }

public class TryKlausuranmeldung {
  public static void main(String[] argf){
   String Klausurort, Nachname, Vorname;
   String KlausurortAuswahl = "Hamburg";
   int i, LaengeNachname, LaengeVorname;
   int Matrikelnummer;
   char x;

   try{
       Klausurort = argf[0];
       System.out.println("Klausurort = " + Klausurort);
       if (!Klausurort.equals(KlausurortAuswahl)) throw new Klausurortfehler();

       System.out.println("Matrikelnummer = " + argf[1]);
       if (argf[1].length() != 8) throw new Matrikelnummerfehler();
       Matrikelnummer = Integer.parseInt(argf[1]);

       Nachname = argf[2];
       LaengeNachname = Nachname.length();
       System.out.println("Nachname = " + Nachname);
       for (i=0; i<LaengeNachname; i=i+1) {
            x = Nachname.charAt(i);
            if ( (int)x < 65 || (int)x > 122  || ((int)x > 90 & (int)x < 97))
            throw new Nachnamenfehler();
       }

       Vorname = argf[3];
       LaengeVorname = Vorname.length();
       System.out.println("Vorname = "  + Vorname);
       for (i=0; i<LaengeVorname; i=i+1) {
            x = Vorname.charAt(i);
            if ( (int)x < 65 || (int)x > 122 || ((int)x > 90 & (int)x < 97))
             throw new Vornamenfehler();
       }
       System.out.println();
       System.out.println("Eingabe fehlerfrei");
      }
      catch (IndexOutOfBoundsException e){
            System.out.println("Eingabe nicht vollstaendig");
      }
      catch (NumberFormatException e) {
```

```
            System.out.println("Eingabe der Matrikelnummer ist keine Integer-Zahl");
      }
      catch (Klausurortfehler e) {
           System.out.println("Klausurort ist nicht Hamburg");
      }
      catch (Matrikelnummerfehler e) {
             System.out.println("Die Matrikelnummer besteht nicht aus 8 Zeichen");
      }
      catch (Nachnamenfehler e) {
             System.out.println("Nachname enthaelt nicht nur Buchstaben");
      }
      catch (Vornamenfehler e) {
             System.out.println("Vorname enthaelt nicht nur Buchstaben");
      }
    finally {System.out.println("Anmelden ist immer schwer");
    }
 }
}
```

a) Beschreiben Sie mit zwei bis zu vier Sätzen, was dieses Programm macht. **(6 Punkte)**

b) Welches Ergebnis liefert das Programm bei der folgenden Eingabe? **(2 Punkte)**

```
java TryKlausuranmeldung Hamburger 111 Wolf 110
```

Lösung:

a) Das Programm erwartet eine Eingabe, bestehend aus dem Ort Hamburg, gefolgt von einer achtstelligen Matrikelnummer, gefolgt von Namen und Vornamen der Studentin oder des Studenten. Das Programm `TryKlausuranmeldung` testet, ob die Anmeldung aus den vier geforderten Teilen besteht, ob die Matrikelnummer eine achtstellige Zahl ist und ob Name und Vorname nur Standardbuchstaben ohne deutsche Umlaute und Co. enthalten. Im Fehlerfall wird die passende Exception angestoßen und eine Fehlermeldung ausgegeben (siehe Teil b)), tritt kein Fehler auf, wird

   Klausurort = erster Eingabewert

   Matrikelnummer = zweiter Eingabewert

   Nachname = dritter Eingabewert

   Vorname = vierter Eingabewert

   und die Meldung 'Eingabe fehlerfrei' ausgegeben. Der Satz 'Anmelden ist immer schwer' wird immer ausgegeben.

b)
```
Klausurort = Hamburger
Klausurort ist nicht Hamburg
Anmelden ist immer schwer
```

Aufgabe 8: Ereignissteuerung (16 Punkte)

a) Woraus besteht das abstrakte GUI-Modell des AWT?
Beschreiben Sie kurz seine drei Teile. **(4 Punkte)**

Lösung:

1. Komponenten: Jedem Darstellungselement einer GUI entspricht im AWT ein Objekt vom Typ `Component`.
2. Darstellung: Es muss beschrieben werden, wie die einzelnen Komponenten auf dem Bildschirm dargestellt werden und wie die Darstellungen der Komponenten angeordnet werden sollen.
3. Ereignissteuerung: Das Fenstersystem benachrichtigt die Komponenten über Benutzereingaben und Veränderungen auf der Fensteroberfläche. Das Eintreffen derartiger Nachrichten über Benutzereingaben und Veränderungen bezeichnet man als ein *Ereignis*. Oft werden auch die Nachrichten selbst Ereignisse genannt. Die Komponenten leiten diese Nachrichten, ggf. in modifizierter Form, an sogenannte Beobachter-Objekte weiter, die die Steuerung der GUI übernehmen.

b) Was passiert, wenn an einer Komponente ein Ereignis auftritt, und wie kann eine GUI auf ein Ereignis reagieren? **(5 Punkte)**

Lösung:

1. Eine Komponente muss bekannt geben, Ereignisse welcher Ereignissorten an ihr auftreten können.
2. *Jedes* Objekt, das bestimmte Voraussetzungen erfüllt, kann sich bei Komponenten als *Beobachter* einer oder mehrerer der ihr zugeordneten Ereignissorten registrieren lassen. Im AWT werden diese Beobachter *Listener* genannt.
3. Zum Zweck des Registrierens muss eine Komponente für jede der ihr zugeordneten Ereignissorten (oder für Teilmengen davon) Methoden zur Verfügung stellen, mit Hilfe derer sich an diesen Ereignissen interessierte Beobachter bei ihr registrieren können.
4. Tritt an einer Komponente k ein Ereignis der Sorte $ESEvent$ auf, werden alle Beobachter davon benachrichtigt, die bei k für $ESEvent$ registriert sind. Die Benachrichtigung erfolgt durch Aufruf der das Ereignis identifizierenden Methode auf jedem der registrierten Beobachterobjekte. Um einen solchen Aufruf zu ermöglichen, müssen die Beobachter bestimmten Regeln genügen.

c) Implementieren Sie ein Hauptfenster, in dem eine Schaltfläche platziert ist, die beim Anklicken das Hauptfenster schließt.
Verwenden Sie bitte den Layout-Manager `FlowLayout`. **(7 Punkte)**

Lösung:

1. Lösungsvorschlag:

```
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class HauptFenster extends Frame{
    public HauptFenster() {
        setLayout(new FlowLayout());

        Button closeButton = new Button("Fenster schließen");
        closeButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                dispose();
            }
        });
        add(closeButton);

        pack();
        setLocationRelativeTo(null);
    }

    public static void main(String[] args) {
        new HauptFenster().setVisible(true);
    }
}
```

2. Lösungsvorschlag:

```
public class ButtonTest {
  public static void main(String args[]) {
    final Frame f = new BaseFrame();
    f.setLayout( new FlowLayout() );

    Button b = new Button("Fenster schliessen");
    b.addActionListener( new ActionListener() {
      public void actionPerformed( ActionEvent e ) {
        f.dispose();
      }
    } );
    f.add( b );
    f.setVisible( true );
  }
}
```

Aufgabe 9: Threads (11 Punkte)

a) Die Aktionen, die ein Thread ausführen soll, werden in einer parameterlosen Methode mit Namen `run` beschrieben. Es gibt in Java zwei Möglichkeiten, diese run-Methode zu deklarieren. Geben Sie die beiden zugehörigen Muster an. **(6 Punkte)**

Lösung:

Entweder man leitet eine neue Klasse von der Klasse `Thread` ab und überschreibt deren run-Methode oder man implementiert mit einer Klasse die Schnittstelle `Runnable`:

```
interface Runnable {  void run();  }
```

Das Muster zur Anwendung der ersten Möglichkeit sieht wie folgt aus:

```
class  MeinThread1  extends  Thread {
  ...
  public void run(){
     // Hier steht der Programmtext, der von Threads des
     // Typs MeinThread1 ausgefuehrt werden soll.
  }
}
```

Die zweite Möglichkeit, um Ausführungsstränge einzuführen, ist Folgende. Die Klasse, die einen Ausführungsstrang implementieren soll, implementiert die Schnittstelle `Runnable`. Dadurch stellt diese Klasse ihre Implementierung der run-Methode zur Verfügung:

```
class  MeinThread2  extends     WichtigeSuperklasse
                    implements  Runnable          {
  ...
  public void run(){
     // Hier steht der Programmtext, der den
     // Ausfuehrungsstrang implementiert.
     // Die Implementierung kann Attribute und Methoden
     // der Klasse WichtigeSuperklasse benutzen.
  }
}
```

b) Ergänzen Sie die Klasse `Tausche` um Codestücke, so dass wechselweiser Ausschluss der beiden Vertauschungsvorgänge gewährleistet ist und keine Verklemmungen auftreten können.
Verändern Sie dabei die Klassen `Wert` und `Main` nicht.
Begründen Sie, warum Ihre Lösung das Gewünschte leistet. **(5 Punkte)**

```
class Wert {
  int wert;

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


class Tausche extends Thread {
  Wert a;
  Wert 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;
  }
}

class Main {
  public static void main(String[] argv) {
    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();
  }
}
```

Lösung:

Hier ist eine Modifikation der Klasse `Tausche`, die wechselweisen Ausschluss garantiert:

```
class Tausche extends Thread {
  Wert a;
  Wert b;


  static Object lock = new Object();


  Tausche(Wert a, Wert b) {
    this.a = a;
    this.b = b;
  }
```

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

Die Klasse `Tausche` wurde um ein statisches Attribut `lock` erweitert, das mit einem Wert vom Typ `Object` initialisiert wurde. Der Code, der die Vertauschung der Werte vornimmt, wurde mit Hilfe eines `synchronized`-Blocks in den Monitor des vom Attribut `lock` referenzierten Objekts verlagert.

Diese Lösung garantiert wechselweisen Ausschluss: Da das Attribut `lock` als statisches Attribut vereinbart wurde, verwenden beide Threads dasselbe Objekt zur Synchronisation. Wenn der erste Thread mit seinem Vertauschungsvorgang beginnt, hat er den Zugang zu dem Monitor dieses Objekts zuvor für die gesamte Dauer der Vertauschung gesperrt. Daher kann der andere Thread erst dann mit dem Vertauschen beginnnen, wenn der erste Thread seine Vertauschung vollständig durchgeführt hat.

Verklemmungen können bei dieser Lösung nicht auftreten, da nur ein einziger Monitor gesperrt wird.

Es genügt nicht, die `run`-Methode der Klasse `Tausche` als `synchronized` zu vereinbaren. Zwar müsste jeder der Threads dann vor Ausführung seiner `run`-Methode einen Monitor betreten. Da aber jeder der beiden `Tausche`-Objekte seinen eigenen Monitor besitzt, würden die beiden Threads verschiedene Monitore betreten, so dass wechselweiser Ausschluss nicht gewährleistet wäre. Aus einem ähnlichen Grunde funktioniert die oben angegebenen Lösung auch nur, weil das Attribut `lock` als statisch vereinbart wurde, so dass beide Threads denselben Monitor (nämlich den von `lock`) betreten müssen.

Aufgabe 10: Programmierung verteilter Objekte (10 Punkte)

a) Was wird im Kurs 1618 unter einem *verteilten System* verstanden? **(2 Punkte)**

Lösung:

Unter einem *verteilten System* verstehen wir eine Menge von lose gekoppelten Prozessen, d.h. von Prozessen, die unabhängig voneinander Berechnungen ausführen und miteinander kommunizieren können. Jeder Prozess arbeitet dabei ein eigenes Programm in einem eigenen Adressraum ab; insbesondere arbeiten die Prozesse im Allgemeinen parallel. Typischerweise sind verteilte Systeme räumlich, d.h. auf mehrere Rechner verteilt.

b) Sockets bilden eine Abstraktionsschicht zur Kommunikation zwischen verteilten Prozessen, die den Programmierer von einigen nicht trivialen Tätigkeiten befreit. Welche Tätigkeiten sind das? **(3 Punkte)**

Lösung:

- Zerteilen der Daten;
- Einpacken der Daten in Pakete mit Adressinformation, Verwaltungsinformation und Paketinhalt;
- eintreffende Pakete auspacken;
- Reihenfolge der eintreffenden Pakete sicherstellen;
- fehlende Pakete nachfordern;
- Pakete am Zielort wieder zusammensetzen; usw.

c) Wie sieht das grundlegende Muster zur Realisierung eines einfachen Servers aus? Geben Sie das passende Programmfragment an oder beschreiben Sie das Muster textuell. **(5 Punkte)**

Lösung:

Das grundlegende Muster zur Realisierung eines einfachen Servers sieht wie folgt aus: Der Server-Prozess schließt sich an den gewünschten Port an. Dies geschieht in Java mittels des Konstruktors der Klasse `ServerSocket` (s.u.). Dann betritt er üblicherweise eine Endlosschleife. Jeder Durchlauf durch die Schleife entspricht der Bedienung eines Clients. In der Schleife wartet der Server, dass sich ein Client an den Port anschließt. Das Warten und das nachfolgende Herstellen der Socket-Verbindung zum Client wird in Java von der Methode `accept` geleistet. Ihr Aufruf blockiert, bis sich ein Client angeschlossen hat, und liefert dann das Socket-Objekt zurück, das die Verbindung zum Client repräsentiert. Das Socket-Objekt stellt Methoden zur Verfügung, um die Ein- und Ausgabeströme zum Client zu öffnen.

Das folgende Programmfragment fasst dieses Muster zusammen:

```
ServerSocket serversocket = new ServerSocket( Portnummer );
while( true ) {
  Socket socket = serversocket.accept();
  Öffnen der Ein- und Ausgabeströme zum Client
  Kommunikation zwischen Server und Client
  socket.close();
}
```