Klausur 24 Februar, Fragen und Antworten

Einführung in die Informatik 1 (IN0001) (Technische Universität München)

Einführung in die Informatik 1

Prof. Dr. Harald Räcke, Prof. Dr. Felix Brandt, J. Kranz, A. Reuss 24.02.2018

Klausur

Vorname	Nachname
Matrikelnummer	Unterschrift

Angabe A

- Füllen Sie die oben angegebenen Felder aus.
- Schreiben Sie nur mit einem dokumentenechten Stift in schwarzer oder blauer Farbe.
- Verwenden Sie kein „Tipp-Ex“ oder Ähnliches.
- Die Arbeitszeit beträgt **120** Minuten.
- Prüfen Sie, ob Sie **27** Seiten erhalten haben.
- Sie können maximal **150** Punkte erreichen. Erreichen Sie mindestens **60** Punkte, bestehen Sie die Klausur.
- Als Hilfsmittel ist nur ein beidseitig handgeschriebenes DIN-A4-Blatt zugelassen.
- Sie dürfen Ihre Lösungen auf die Angabe oder das Klausurpapier schreiben. Sie dürfen die Grammatik von MiniJava von der Klausur trennen.

vorzeitige Abgabe um ..... Hörsaal verlassen von ..... bis .....

1	2	3	4	5	6	7	8	Σ

..... Erstkorrektor

..... Zweitkorrektor

Aufgabe 1 Multiple Choice [15 Punkte]

Kreuzen Sie in den folgenden Teilaufgaben jeweils die richtigen Antworten an. Es können pro Teilaufgabe keine, einige oder alle Antworten richtig sein. Die Teilaufgaben werden isoliert bewertet; Fehler in einer Teilaufgabe wirken sich also nicht auf die erreichbare Punktzahl bei anderen Teilaufgaben aus.

1. Betrachten Sie folgenden Code:

```
1 String animal = getAnimalType();
2 if(animal == "penguin")
3   cuddleAnimal(animal);
```

   Welche der folgenden Aussagen ist korrekt?

   - [x] Es werden die Referenzen der String-Objekte verglichen.
   - [ ] Der String `animal` wird zeichenweise mit dem Wort „penguin“ verglichen.
   - [ ] Eventuell wird `cuddleAnimal(...)` für eine Giraffe aufgerufen.
   - [x] Eventuell wird `cuddleAnimal(...)` für einen Pinguin nicht aufgerufen.
   - [ ] `cuddleAnimal(...)` wird sicher nicht für einen Pinguin aufgerufen.

2. Zu welchem Wert evaluieren die folgenden Java-Ausdrücke?

   (a) `5 / 3`: 1

   (b) `1.0 / 2`: 0.5

   (c) `"hallo" + 1 + (int)2.999`: hallo12

   (d) `5 + 2 + "hallo" + 'a'`: 7halloa

   (e) `3 % 2 + 1 == 1 ? 2 : 3`: 3

3. Die `int`-Variable `x` habe vor jeder der folgenden Teilaufgaben den Wert 1. Zu welchem Wert evaluieren die Ausdrücke?

   (a) `x = -x++ +x`: 1

   (b) `x == +x-- -x`: `true`

4. Geben Sie die folgenden Zahlen jeweils hexadezimal, oktal und binär an.
   Beispiel: `19 = 0x13 = 023 = 0B0001_0011`

   (a) `63 =` 0x3F = 077 = 0B0011_1111

   (b) `803 =` 0x323 = 01443 = 0B0011_0010_0011

   (c) `513 =` 0x201 = 01001 = 0B0010_0000_0001

5. Betrachten Sie folgenden Code von zwei unterschiedlichen Threads. Die Variable `shared` sei hier eine statische `int`-Variable, auf die beide Threads Zugriff haben.

   Thread 1:

```
1 shared = 1;
```

Thread 2:

```
while(shared == 0)
  ;
System.out.println("Penguins are the best!");
```

Die beiden Threads werden nacheinander gestartet. Die Variable `shared` wird vor dem Start mit 0 initialisiert. Welche der folgenden Aussagen sind zutreffend?

- ☐ Die Ausgabe erfolgt an regnerischen Sonntagen immer.
- ☐ Die Ausgabe erfolgt immer, da keine Synchronisierung nötig ist.
- ☐ Die Ausgabe erfolgt nie.
- ☒ Die Ausgabe erfolgt vielleicht, da das Programmverhalten undefiniert ist.

6. Welche der folgenden Namen sind für Variablen erlaubt?

- ☐ `name!`
- ☐ `2Array`
- ☐ `(Der_Name)`
- ☐ `Pinguins*Fun`

7. Betrachten Sie den folgenden Java-Code:

```
class Example {
  int k = 1, n = 5;

  public String toString() {
    return "k:" + this.k + ", n:" + this.n;
  }
}

public class Main {
  public static void main(String[] args) {
    Example e = new Example();
    System.out.println(e);
  }
}
```

Was ist das Ergebnis der Ausführung des Codes?

- ☐ `println(e)` gibt die Referenz des `Example`-Objekts aus (z.B. `Example@7852e922`).
- ☒ `println(e)` ruft die `toString()`-Methode der Klasse `Example` auf und gibt `k:1, n:5` aus.
- ☐ `println(e)` wirft die Ausnahme `IllegalArgumentException`.
- ☐ Es tritt ein sog. *Workspace Overload* auf.

Lösungsvorschlag 1

1. 2 Punkte
   - 1 Punkt für die ersten beiden (alles oder nichts)
   - 0.5 Punkte für keine Giraffe
   - 0.5 Punkte für die letzten beiden (alles oder nichts)
2. 5 Punkte, ein Punkt pro Teilaufgabe
3. 1 Punkt, 0.5 pro Teilaufgabe
4. 3 Punkte, ein Punkt pro Teilaufgabe
   - 0.5 Punkte Abzug pro Fehler
   - 0.5 Punkte bei (nicht-trivialen) konsistent falschen Zahlen ($\rightsquigarrow$ Studenten, die sich bei einer Umrechnung vertan haben und dann davon ableiteten, erhalten noch einen halben Punkt)
5. 2 Pünktchen, alles oder nichts
6. 1 Punkt, alles oder nichts
7. 1 Pinguin, alles oder nichts

Festlegungen:

- Subskripte etc. statt Java-Schreibweise sind bei den Zahlenbasen ebenfalls ok
- Umrechnung in nur eine Basis: insg. 0.5P bei zwei Teilaufgaben, 1P bei drei richtigen
- Strings ohne Anführungsstriche sind bei den Ausdrücken ok

Aufgabe 2 Mut zur Lüge [15 Punkte]

Vervollständigen Sie die Implementierung unten. Die Methode `main` wandelt die übergebenen Strings einzeln in Zahlen um und gibt diese aus. Zur Umwandlung in eine Zahl werden alle Zeichen entfernt, die keine Ziffern (0 bis 9) sind, und der dadurch erhaltene String als Zahl ausgegeben. Beispiel: `new String[]{"s0r0sinu38in08h", "20_18x", "3+7/6", "42"}` als Parameter ergibt als Ausgabe die Zahlen 3808, 2018, 376 und 42.

Hinweis: Für einen String `s` liefert `s.charAt(i)` das Zeichen an Index `i`, wobei das erste Zeichen an Index 0 steht, und `s.length()` liefert die Anzahl Zeichen in `s`. Nehmen Sie an, dass jeder der übergebenen Strings mindestens eine Ziffer enthält.

Schreiben Sie in jede Box genau einen Ausdruck!

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

  // Schleife über alle Argumente
  for (int i = 0;  i < args.length ; i++) {

    // String:
    String current = args[i];

    // Zahl, in die dieser umgewandelt wird:
    int number = 0 ;

    // Schleife über alle Zeichen im String
    for(
      int tmp = current.length();
      tmp != 0 ;
      tmp = tmp - 1)
    {
      char c = current.charAt( current.length() - tmp );
      // Prüfen, ob Ziffer:
      if ( c >= ZERO && c <= NINE ) {
        // Zahlenwert aktualisieren:
        number = number * 10 + c - '0';
      }
    }
    // Zahl ausgeben:
    System.out.print(" " + number);
  }
}
```

Bewertung:

1., 2., 3. Box je 2 Punkte,
4., 5., 6. Box je 3 Punkte.

Bemerkung:

5. Box: `c >= '0' && c <= '9'`

Lösungsvorschlag 2

Festlegungen:

Block	Fehler	Abzug
1	`s.length`	1
	`i <` fehlt	1
	`<=`	0.5
	`i< args[].length`	1
	args.size()	0.5
2	Wenn `!= 0`	2
3	-1	1.5
	1	1.5
4	`tmp`	3
	`tmp - 1`	2
	`args.length - tmp`	2
	zusätzliche -1	1
5	`c.equals`	1
	`c.isDigit()`	1
	`c.isNumber()`	1
	`Character.isNumber()`	0.5
	`||` statt `&&`	2
	`0` statt `'0'`	1
	`"0"` statt `'0'`	1
	falsche ASCII	0.5
	`c==1|2|3|4 ...`	3
	`c=='1'||'2'||'3'||...`	2
	`c=='0'|'1'|'2'|...`	3
	`0' <= c <= '9'`	1
	Falsche Variable sonst richtig	2
	= statt ==	2
6	`number`	2
	`number + ...`	2

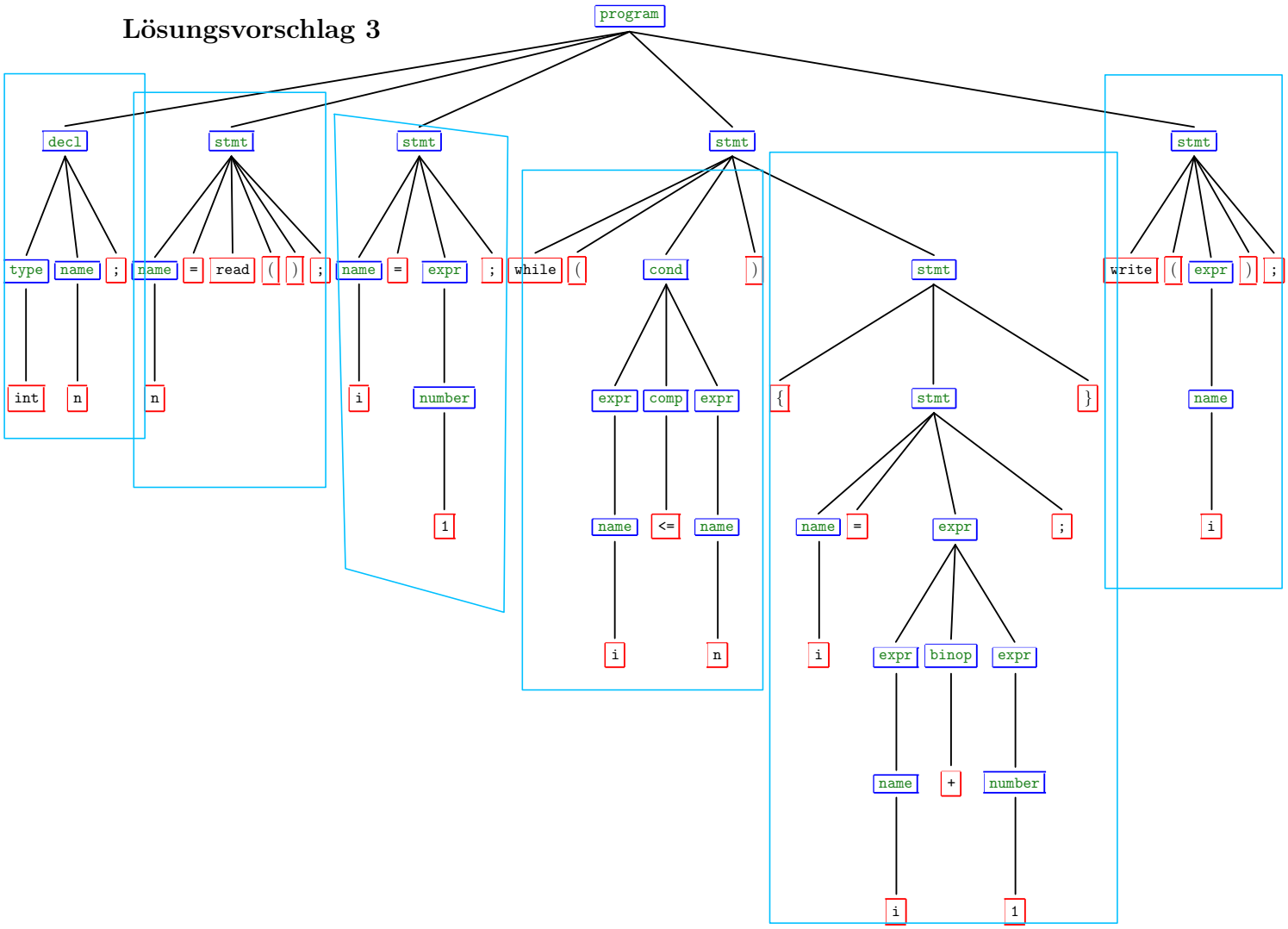
Aufgabe 3 Sonntagsbäume [15 Punkte]

Zeichnen Sie für den folgenden Code den Syntaxbaum entsprechend der Grammatik von MiniJava. Die Grammatik finden Sie auf der letzten Seite.

```
1  int n;
2
3  n = read();
4  i = 1;
5
6  while (i <= n) {
7    i = i + 1;
8  }
9
10 write(i);
```

Handelt es sich bei dem Code um ein korrektes MiniJava-Programm (Begründung!)? Falls nein, wieso ist es dennoch möglich, den Syntaxbaum zu zeichnen?

Lösungsvorschlag 3



Das Programm kompiliert nicht, da die Variable `i` nicht deklariert wurde. Den Syntaxbaum kann man dennoch zeichnen, da die Deklariertheit einer Variablen eine Kontextbedingung ist, die sich mit **kontextfreien** Grammatiken nicht ausdrücken lässt (3 Punkte auf den Text, davon 1.5 Punkte darauf, dass das Programm nicht kompiliert plus Begründung).

Gesammelte Bewertungsfestlegungen:

- Fehlender oder falscher Knoten/Label: -0.5
- Fehlende oder falsche Verbindung -0.5
- Fehlt ein Knoten, kein Extraabzug wegen ebenfalls fehlender Kanten
- Kante an falscher Stelle (Reihenfolge) -0.5
- Gemeinsamer Kantenanteil wird akzeptiert, solange Reihenfolge OK
- program fehlt -0.5
- = braucht eigenen Knoten, sonst -0.5
- read(); und while(zusammenfassbar
- stmt fehlt, {, } vorhanden -0.5
- stmt fehlt, {, } fehlen -1
- <, = statt <= -0.5
- Falsche Bezeichnung von Nichtterminalen -0.5 (Formfehler)
- Max. -1 insgesamt pro Formfehler

Pro blau eingerahmtem Teilbaum-Block maximal 2 Punkte Abzug.

Aufgabe 4 Polymorphie [20 Punkte]

Betrachten Sie den folgenden Code:

```
1  public class Poly {
2    static class A {
3      void f(C c) { System.out.println("A.f(C)"); c.f(this); c.g(this); }
4      static void g() { System.out.println("A.g()"); }
5    }
6
7    static class B extends A {
8      void f(D d) { System.out.println("B.f(D)"); d.f(this);
9                    ((C)d).g(this); }
10     void f(C c) { System.out.println("B.f(C)"); c.f(this); c.g(this); }
11     static void g() { System.out.println("B.g()"); }
12   }
13
14   static class C {
15     void f(A a) { System.out.println("C.f(A)"); }
16     static void g(A a) { System.out.println("C.g(A)"); }
17     static void g(B b) { System.out.println("C.g(B)"); }
18   }
19
20   static class D extends C {
21     void f(A a) { System.out.println("D.f(A)"); }
22     void f(B b) { System.out.println("D.f(B)"); }
23     static void g(A a) { System.out.println("D.g(A)"); }
24     static void g(B b) { System.out.println("D.g(B)"); }
25   }
26
27   public static void main(String[] args) {
28     A a = new A(); B b = new B(); C c = new C(); D d = new D();
29
30     a.f(d); // Aufruf 1
31     a.g();  // Aufruf 2
32     a = b;
33     c = d;
34     a.f(c); // Aufruf 3
35     a.f(d); // Aufruf 4
36     b.f(c); // Aufruf 5
37   }
38 }
```

Geben Sie für jeden der markierten Aufrufe die Ausgabe an. Sollte einer der Aufrufe nicht kompilieren oder einen Laufzeitfehler erzeugen, so geben Sie dies ebenfalls an.

Lösungsvorschlag 4

1.

```
A.f(C)
D.f(A)
C.g(A)
```

2.

```
A.g()
```

3.

```
B.f(C)
D.f(A)
C.g(B)
```

4.

```
B.f(C)
D.f(A)
C.g(B)
```

5.

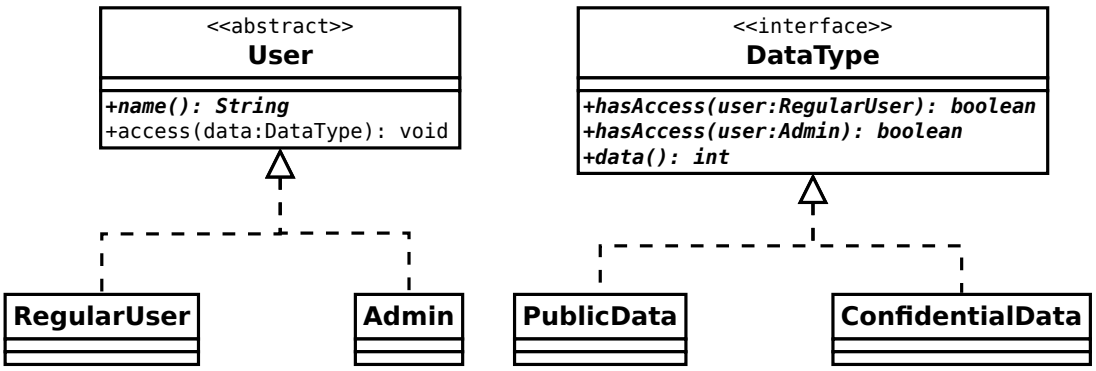
```
B.f(C)
D.f(A)
C.g(B)
```

Punkteverteilung:

- 1.25 Punkte pro Zeile, außer Teilaufgabe 2 (insg. 15 Punkte)
- 1 Punkt pro komplett richtiger Teilaufgabe (insg. 5 Punkte)

Aufgabe 5 Objekte mit Orientierung [15 Punkte]

In dieser Aufgabe geht es darum, ein System mit verschiedenen Nutzer- und Datentypen zu entwickeln. Ein Nutzer kann auf ein Datum Zugriff haben oder nicht. In unserem System kommen die folgenden Klassen bzw. Schnittstellen vor (Konstruktoren sind im Diagramm nicht enthalten):



Gehen Sie nun zunächst wie folgt vor:

1. Schreiben Sie den Java-Code für das Interface `DataType` sowie die beiden Klassen `PublicData` und `ConfidentialData` gemäß den unten beschriebenen Zugriffsregeln. Das jeweilige Datum (hier eine `int`-Zahl) wird im Konstruktor erwartet.
2. Implementieren Sie die abstrakte Klasse `User`.
   (a) Beginnen Sie mit der abstrakten Methode `name()`, die von den Unterklassen implementiert wird. Implementieren Sie außerdem die beiden Unterklassen `RegularUser` und `Admin`, die ihren Namen jeweils im Konstruktor erwarten.
   (b) Implementieren Sie die nicht-abstrakte Methode `access(...)` der Klasse `User`. Die Methode prüft zunächst, ob der jeweilige Nutzer Zugriff auf das gegebene Datenobjekt hat. Ist dies der Fall, so sollen die Daten, welche die Methode `data()` liefert, auf der Konsole ausgegeben werden; ansonsten soll eine Exception vom Typ `RuntimeException` geworfen werden, die im Konstruktor einen sinnvollen Fehlertext erhält. Implementieren Sie ggf. Hilfsmethoden.

Es sollen folgende Zugriffsregeln umgesetzt werden:

- Ein `RegularUser` hat nur Zugriff auf `PublicData`-Objekte.
- Ein `Admin` hat Zugriff auf alle Datenobjekte.

Beschreiben Sie schließlich in Worten, was bei folgendem Code passiert:

```
1 User u = new Admin("Hans Peter Wurst");
2 DataType data = new ConfidentialData(42);
3 u.access(data);
```

Hinweis: Die Verwendung von `instanceof` oder *Reflection* ist in der gesamten Aufgabe verboten.

Lösungsvorschlag 5

```java
public abstract class User {
  public abstract String name();

  protected abstract boolean hasAccess(DataType d);

  public void access(DataType d) {
    if(!hasAccess(d))
      throw new RuntimeException("Zugriff verweigert!");
    System.out.println(d.data());
  }
}
```

```java
public class RegularUser extends User {
  private String name;

  public RegularUser(String name) {
    this.name = name;
  }

  @Override
  public String name() {
    return name;
  }

  @Override
  protected boolean hasAccess(DataType d) {
    return d.hasAccess(this);
  }

}
```

```java
public class Admin extends User {
  private String name;

  public Admin(String name) {
    this.name = name;
  }

  @Override
  public String name() {
    return name;
  }

  @Override
  protected boolean hasAccess(DataType d) {
```

```
    return d.hasAccess(this);
  }
}
```

```
public interface DataType {
  int data();
  boolean hasAccess(RegularUser user);
  boolean hasAccess(Admin user);
}
```

```
public class PublicData implements DataType {
  private int data;

  public PublicData(int data) {
    this.data = data;
  }

  @Override
  public int data() {
    return data;
  }

  @Override
  public boolean hasAccess(RegularUser user) {
    return true;
  }

  @Override
  public boolean hasAccess(Admin user) {
    return true;
  }
}
```

```
public class ConfidentialData implements DataType {
  private int data;

  public ConfidentialData(int data) {
    this.data = data;
  }

  @Override
  public int data() {
    return data;
  }

  @Override
  public boolean hasAccess(RegularUser user) {
    return false;
```

```
16    }
17
18    @Override
19    public boolean hasAccess(Admin user) {
20      return true;
21    }
22
23  }
```

Bei dem gezeigten Aufruf wird zunächst dynamisch die `hasAccess()`-Methode der passenden Unterklasse von `User` aufgerufen (1. Dispatch), woraufhin wiederum dynamisch die passende `hasAccess`-Methode des Datums aufgerufen wird (2. Dispatch). Das Vorgehen erinnert an das Visitor-Pattern.

Bewertungshinweise:

1. 3 Punkte
2. (a) 3 Punkte
   (b) 7 Punkte (davon 1 Punkt für die Exception)
3. (Text-Antwort): 2 Punkte (siehe unten)

Korrekturhinweise:

- Duplikation des Code in `access()`: 2 Punkte Abzug
- Verwendung von `instanceof`: 2 Punkte Abzug (wenn doppelter Dispatch weiterhin stattfindet), sonst 4 Punkte Abzug
- Bei der Text-Antwort sind Stichpunkte ok. Wichtig ist, dass auf den doppelten Dispatch eingegangen wird (z.B. durch Verweis auf das Visitor-Pattern in geeigneter Weise)
- Wer bei der Text-Antwort nur erklärt, was der *Effekt* des Codes ist (und nicht, dass doppelter Dispatch auftritt), erhält hier einen Punkt. Der fehlende Punkt wird zusätzlich auf die korrekte Implementierung der 2 (b) gegeben.
- Die widerliche Cast-Lösung muss leider als korrekt gewertet werden, da das Verbot von Casts im Hinweis fehlt.

Festlegungen:

- UML in Parameter: -0.5 Punkte
- User-Unterklassen fehlen: 2 (a) max. 1/3 Punkte
- Double Dispatch fehlt in 2 (b): 3 Punkte Abzug
- nur Resultat bei 3: -0.5 Punkte
- `getClass()` wie `instanceof`

- Klammern und Semikolonnen egal (es sei denn systematisch)
- Rückgabetyp verändert oder fehlt: -0.5 bis -1 Punkte, je nach Klasse
- try/catch: -1 Punkt
- Exception-Text fehlt: -0.5
- Exception ohne `new`: kein Abzug
- keine RuntimeException: -1 Punkt
- `extends` generell vergessen: -1 Punkt
- `implements` generell vergessen: -1 Punkt
- `implements` und `extends` verwechselt: insgesamt -0.5 Punkte
- sinngemäße Variablenumbenennung: kein Abzug
- `abstract interface`: -0.5 Punkte
- `static interface`: -0.5 Punkte
- `static` klassenbezogen: -0.5 Punkte

Aufgabe 6 Rekursion [20 Punkte]

Lösen Sie die folgenden Teilaufgaben *rekursiv*. Es sind hier **keinerlei Schleifen** erlaubt. Es dürfen Hilfsmethoden implementiert werden, jedoch lediglich eigene Methoden verwendet werden.

1. Schreiben Sie eine Methode `static int bino(int n, int k)` zur Berechnung des Binomialkoeffizienten. Sie dürfen voraussetzen, dass $n \geq k \geq 0$ gilt (keine Fehlerbehandlung nötig). Nutzen Sie die Formel $\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$. Beachten Sie außerdem, dass für alle $n \in \mathbb{N}_0$ die Gleichung $\binom{n}{n} = \binom{n}{0} = 1$ gilt.

2. Schreiben Sie eine Methode `static int same(int[] a)`, die in einem `int`-Array nach Folgen gleicher Zahlen sucht und die Länge einer längsten solchen Folge zurückgibt. Falls das übergebene Array `null` oder leer ist, soll eine `RuntimeException` geworfen werden.

   Beispiele: Der Aufruf `same(new int[] {2, 2, 9, 4, 4, 4, 2, 4})` gibt 3 zurück, denn die längste Folge `4, 4, 4` hat die Länge 3. Für das Array `new int[] {1, 2, 2, 3, 3, 2}` ist der Rückgabewert 2, denn die längsten Folgen sind `2, 2` und `3, 3`.

3. Betrachten Sie die folgenden Java-Funktionen:

```java
class Mouse {
  static Exam createExam() {
    System.out.println("Julian, bitte mache du die Klausur!");
    return Penguin.createExam(); // Arbeit an Julian delegieren
  }
}

class Penguin {
  static Exam createExam() {
    System.out.println("Andreas, bitte mache du die Klausur!");
    return Mouse.createExam(); // Arbeit an Andreas delegieren
  }
}
```

   Stellen Sie sich vor, Sie implementieren einen Compiler für Java, der rekursive Aufrufe optimiert. Können Sie Ihren Compiler derart implementieren, dass die Aufrufe obiger Methoden optimiert werden, sodass es zu keinem Stack-Overflow kommt? Beschreiben Sie kurz die Vorgehensweise des Compilers!

Lösungsvorschlag 6

1.

```java
public static int bino(int n, int k) {
  // je 1P pro Bedingung
  if (n == k || k == 0)
    return 1; // 1P
  // je 1P pro Argument
  else return bino(n-1, k-1) + bino(n-1, k);
}
```

Bewertung: 7 Punkte (Implementierung der Fakultät gibt 1P Abzug)

2.

```java
  public static int same(int[] a) {
    if (a == null || a.length == 0) { // Sonderfälle
      throw new IllegalArgumentException();
    }
    return same(a, 0); // Hilfsfunktion
  }

  // Gleiche zählen
  public static int count(int[] a, int left) {
    if (left == a.length - 1 || a[left + 1] != a[left]) {// Abbruch
      return 1;
    }
    return 1 + count(a, left + 1); // Rekursiv zählen
  }

  // Hilfsfunktion
  public static int same(int[] a, int left) {
    if (left == a.length - 1) { // Abbruch
      return 1;
    }
    int compare = same(a, left + 1); // Rekursiv suchen
    int count = count(a, left);      // Zählen
    return (compare > count ? compare : count); // Maximum nehmen
  }
```

Bewertung (9 Punkte insgesamt, davon Sonderfälle:1, Zählen:4, Rest:4):

- Sonderfälle: 1 Punkt (0.5 für die Bedingung, 0.5 für das Werfen der Exception)
- Gleiche zählen.Abbruch: 2 Punkt(e)
- Gleiche zählen.Rekursion: 2 Punkt(e)
- Hilfsfunktion.Abbruch: 1 Punkt(e)

- Hilfsfunktion.Rekursion: 1 Punkt(e)
- Hilfsfunktion.Zählen: 1 Punkt(e)
- Hilfsfunktion.Maximum: 1 Punkt(e)

Bemerkung:

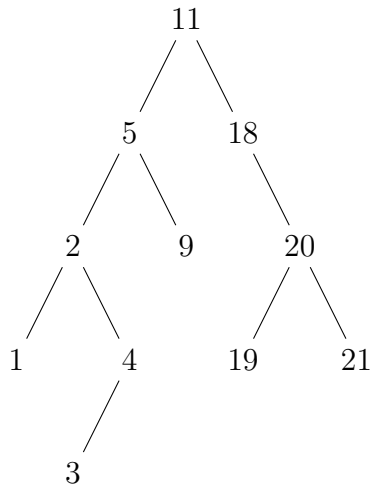
Lösungsanteile, die Schleifen enthalten, werden nicht gewertet.
Es gibt alternative Lösungen, etwa mit nur einer Hilfsfunktion. Bewertung nach Funktionalität.

3. Der Compiler kann hier die *Tail Call Optimization* anwenden, da es sich jeweils um letzte Aufrufe handelt, der Stack-Frame also nicht mehr benötigt wird.

   Bewertung: 4 Punkte, sollte eine Erzählung über Stack-Frames enthalten

Aufgabe 7 Binärsuchbaumsortierung [25 Punkte]

Das folgende Bild stellt einen binären Suchbaum mit Zahlen als Inhalt dar:



Jeder Knoten im Baum hat einen Inhalt vom Typ `Integer` sowie zwei Referenzen auf die Kindknoten (linkes Kind und rechtes Kind, im Bild jeweils eine Ebene tiefer dargestellt). Wir speichern im Suchbaum (Klasse `BST`) als Attribute zusätzlich für jedes Kind den Elternknoten. Beispiel: Das linke Kind des Knotens mit dem Inhalt 4 ist der Knoten mit dem Inhalt 3, das rechte Kind ist `null` und der Elternknoten hat den Inhalt 2.

Jeder Knoten repräsentiert zugleich den binären Suchbaum, dessen oberster Knoten (Wurzel) er ist. Beispiel: Der Knoten mit Inhalt 5 repräsentiert den gesamten Teilbaum mit den Knoten mit den Inhalten 1, 2, 3, 4, 5, 9. Alle Zahlen im vom linken Kind repräsentierten Teilbaum sind kleiner, alle Zahlen im vom rechten Kind repräsentierten Teilbaum sind größer als die Zahl im eigenen Knoten. Keine Zahl kommt doppelt vor.

Im Folgenden soll ein Iterator für die Klasse `BST` implementiert werden, mit dessen Hilfe die Zahlen aufsteigend sortiert zurückgegeben werden. Überlegen Sie sich zunächst anhand des Beispiels, welche Zahl dieser Iterator als erstes zurückgeben muss. Es ist nicht die Zahl 11, die in der Wurzel steht, sondern die 1, welche im linkesten Knoten steht. Der Iterator soll also zu Beginn diesen Knoten aufsuchen. Danach soll er jeweils **nur** aufgrund der Position des aktuellen Knotens den nächsten Knoten in aufsteigender Reihenfolge der Inhalte aufsuchen. Zum Navigieren im Baum dienen dabei einzig die Referenzen auf Elternknoten und Kindknoten.

Implementieren Sie entsprechend dieser Beschreibung für die Iteratorklasse `TreeTerator` die folgenden Iterator-Methoden:

1. `public TreeTerator(BST bst)`:

   Der Konstruktor bekommt den obersten Knoten (Wurzel) des Baums, in dem iteriert werden soll, übergeben. Dieser hat keinen Elternknoten (`bst.parent == null`).

2. `public boolean hasNext()`:

   Gibt `true` zurück, wenn eine nächste Zahl existiert, sonst `false`.

3. `public Integer next()`: Gibt die nächste Zahl zurück. Existiert diese nicht, soll eine Exception geworfen werden.

```java
public class BST {

  // Iterator als innere Klasse des Suchbaums, über den iteriert wird
  public class TreeTerator implements java.util.Iterator<Integer> {

    private BST current; // Aktuelle Position des Iterators

    public TreeTerator(BST bst) {
      // TODO Implementierung
    }
    public boolean hasNext() {
      // TODO Implementierung
    }
    public Integer next() {
      // TODO Implementierung
    }
  }

  public TreeTerator getIterator() {
    return new TreeTerator(this);
  }
  // Attribute
  private Integer content; // Inhalt
  private BST right;       // rechtes Kind
  private BST left;        // linkes Kind
  private BST parent;      // Elternknoten
  // Konstruktor
  public BST(Integer content, BST left, BST right, BST parent) {
    this.content = content;
    this.left = left;
    this.right = right;
    this.parent = parent;
  }
  // Einfügen einer Zahl
  public void insert(Integer x) { ... (Gegeben) }
}
```

Lösungsvorschlag 7

```
5     public TreeTerator(BST bst) {
6       current = bst;
7       while (current.left != null) {
8         current = current.left;
9       }
10    }
11    public boolean hasNext() {
12      return current != null;
13    }
14    public Integer next(){
15      if (current == null) {
16        throw new java.util.NoSuchElementException();
17      }
18      Integer x = current.content;
19      if (current.right != null) {
20        current = current.right; // Weiter im rechten Teilbaum
21        while (current.left != null) { // Linkesten Knoten aufsuchen
22          current = current.left;
23        }
24      } else { // Kein rechtes Kind vorhanden
25        BST whereFrom = null;
26        // Gehe so lange zum Elternknoten, bis der letzte solche
27        // Schritt von links aus erfolgt / die Wurzel erreicht ist.
28        while (null != current && current.right == whereFrom) {
29          whereFrom = current;
30          current = current.parent;
31        }
32      }
33      return x;
34    }
```

Punkteverteilung:

- Konstruktor: 4 Punkte
- hasNext(): 2 Punkte
- next():
  - Exception, falls `!hasNext()`: 2 Punkte
  - Wert zurückgeben: 3 Punkte
  - Fallunterscheidung: Rechter Teilbaum vorhanden? 5 Punkte
  - Fall 1: Rechter Teilbaum ist vorhanden: 3 Punkte
  - Fall 2: Kein rechter TB: 6 Punkte
    Nur 4 von 6 Punkten, wenn Test auf Wurzel fehlt (und deshalb eine Exception auftritt anstatt dass `current` auf `null` gesetzt wird)

Festlegungen:

- Konstruktor: maximal 4 Punkte

`current = bst`	+2P
`current` auf größtes	+1P
`current` korrekt	+2
`current` neu definiert	-
`parent` auf `null` gesetzt	-

- `hasNext()`: maximal 2 Punkte

`current.right != null && current.left != null`	0P
`parent&&right == null`	+1P
`current == null`	+1P
wenn return fehlt	-1P

- `next()`: maximal 19 Punkte

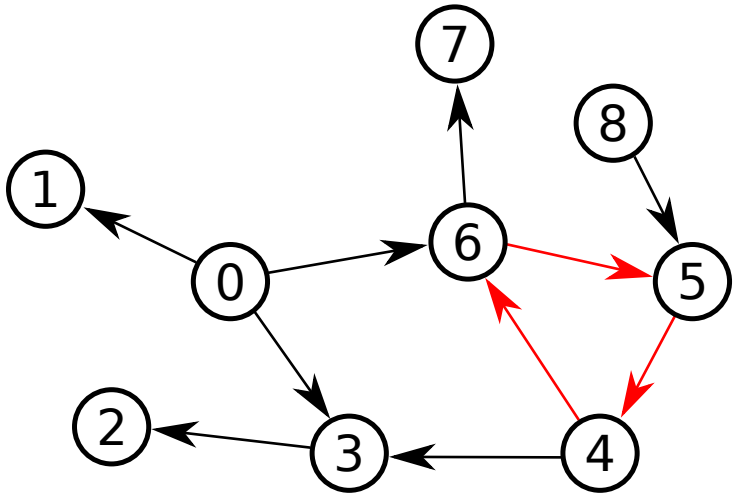
Exception bei `current == null` fliegt eine Exception	maximal 2 Punkte +2P
`if(current.right ==/!= null)`	+5P

Rechter Teilbaum inhalt
`curr = curr.right`
`while(curr.left!= null)`
`curr = curr.left`
	max 3P
	+1P
	+1P
	+1P

kein rechter Teilbaum Inhalt Test auf wurzel Geht nach oben Korrekt von rechts 1 zu früh/spät	max 6P +2P +2P +2P -1P
Return korrekter Wert Wert aus dem Baum Korrekter Knoten/Falscher Typ	max 3P +3P +2P +1

Aufgabe 8 Kreisbrecher (Threads) [25 Punkte]

In dieser Aufgabe geht es darum, in gerichteten Graphen Kreise parallel aus mehreren Threads heraus aufzubrechen. Ein solcher Graph besteht aus Knoten, die durch gerichtete Kanten miteinander verbunden sind (Start- und Zielknoten von Kanten sind stets verschieden, hierfür ist keine Fehlerbehandlung nötig). Die folgende Abbildung zeigt ein Beispiel:



Der Kreis $5 \to 4 \to 6 \to 5$ ist rot markiert. Der Graph wird durch ein Objekt repräsentiert, welches die folgende Schnittstelle implementiert:

```
1  public interface Graph {
2    /*
3     * Liefert den Knoten mit dem Index 'nodeIndex' zurück
4     */
5    public Node getNode(int nodeIndex);
6
7    /*
8     * Liefert eine Kopie der Menge der Indices der Zielknoten von
9     * Kanten ab dem Knoten mit dem Index 'fromNode' zurück
10    */
11   public java.util.Set<Integer> getEdges(int fromNode);
12
13   /*
14    * Entfernt eine Kante aus dem Graphen
15    */
16   public void removeEdge(int from, int to);
17 }
```

Jeder Knoten hat einen eindeutigen Index und wird durch ein Objekt der Klasse `Node` repräsentiert, die ebenfalls gegeben ist:

```
1  public class Node {
2    /*
```

```
3       * Speichert, ob ein Thread gerade einen Pfad durchsucht,
4       * der den Knoten enthält
5       */
6     public boolean locked;
7
8     /*
9       * Speichert die Thread-ID desjenigen Threads, der den Knoten
10      * aktuell bearbeitet oder zuletzt bearbeitet hat
11      */
12    public long threadId;
13
14    public Node() {
15      locked = false;
16      threadId = -1;
17    }
18  }
```

Ein Thread t mit ID id_t beginnt mit einem Anfangsknoten, ab dem er den Graphen rekursiv durchläuft. Für jede Kante unterscheidet er dabei die folgenden Fälle:

1. Ist die Thread-ID des Zielknotens größer als id_t, so wird die Kante ignoriert.
2. Ist der Zielknoten bereits durch den aktuellen Thread blockiert, so wird die Kante zum Zielknoten entfernt. Auf diese Weise wird ein Kreis im Graphen aufgebrochen.
3. Ansonsten wird die Kante zum rekursiven Abstieg verwendet.

Generell gilt, dass der Zielknoten, wenn die Kante nicht ignoriert wird, vom aktuellen Thread blockiert werden muss. Dazu muss der Thread u.U. warten, bis ein anderer Thread den Zielknoten verlassen hat. Es muss stets sichergestellt sein, dass keine Deadlocks auftreten oder unkoordinierte parallele Zugriffe auf Daten stattfinden.

Implementieren Sie nun den Thread `CycleBreaker`, der im Konstruktor einen Graphen und einen Anfangsknoten erwartet. Implementieren Sie neben der `run()`-Methode innerhalb dieser Klasse die Objektmethode `traverse`, indem Sie den folgenden Code beliebig erweitern:

```
1   private void traverse(int from, int to) throws InterruptedException {
2     long myId = Thread.currentThread().getId(); // ID ist stets >= 0
3
4     // Kanten ab 'to' vom Graphen erfragen; 'graph' ist Objektvariable
5     java.util.Set<Integer> edgesFromTo = graph.getEdges(to);
6
7     // Rekursiv alle Kanten ab 'to' besuchen
8     for (int dest : edgesFromTo)
9       traverse(to, dest);
10  }
```

Hinweis: Sie dürfen zur Synchronisierung ausschließlich `synchronized`-Blöcke, `wait()`, `notify()` und `notifyAll()` verwenden. Nutzen Sie keine Locks etc. aus der Java-Bibliothek.

Lösungsvorschlag 8

```java
// Klasse + Konstruktor + Vererbung von Thread / bzw. Runnable + run() vorhanden: 3 Punkte
public class CycleBreaker extends Thread {
  private Graph graph;
  private int startNode;

  public CycleBreaker(Graph graph, int startNode) {
    this.graph = graph;
    this.startNode = startNode;
  }

  private void traverse(int from, int to) throws InterruptedException {
    long myId = Thread.currentThread().getId(); // ID ist stets >= 0

    Node node = graph.getNode(to);
    boolean workWithNode = false;
    boolean cycle = false;

    // Synchronisierung von Knoten (nur zum Setzen des Locks): 3 Punkte
    synchronized (node) {
      // Warten: 2 Punkte, je einen pro Bedingung
      while (node.threadId < myId && node.locked)
        // 1 Punkt, auch wenn anderswo gewartet wird
        node.wait();
      if (node.threadId <= myId) {
        // Zyklus richtig erkennen: 3 Punkte
        cycle = node.locked && myId == node.threadId;
        workWithNode = true;
        // Blockieren des Knotens: 2 Punkt
        node.locked = true;
        node.threadId = myId;
      }
    }

    // Ignorierung von Knoten: 2 Punkte
    if (!workWithNode)
      return;

    if (cycle && from != to)
      // Zyklus entfernen: 1 Punkt
      graph.removeEdge(from, to);
    else {
      // Traversierung an der richtigen Stelle: 1 Punkt
      // Kanten ab 'to' vom Graphen erfragen; 'graph' ist Objektvariable
      java.util.Set<Integer> edgesFromTo = graph.getEdges(to);
```

```
45
46          // Rekursiv alle Kanten ab 'to' besuchen
47          for (int dest : edgesFromTo)
48            traverse(to, dest);
49        }
50
51        //  Synchronisierung für Benachrichtigung: 2 Punkt
52        synchronized (node) {
53          //  Rücksetzen des Locks: 1 Punkt
54          node.locked = false;
55          //  Benachrichtigung: 1 Punkt
56          node.notifyAll();
57        }
58      }
59
60      @Override
61      public void run() {
62        //  Error handling: 1 Punkt
63        try {
64          //  2 Punkt, zusammen mit Test in Zeile~38 bzgl. from != to
65          traverse(startNode, startNode);
66        } catch (InterruptedException e) {
67          e.printStackTrace();
68        }
69      }
70
71  }
```

Festlegungen:

- Je ein Punkt auf Klasse + Konstruktor, `extends Thread` bzw. `implements Runnable` und das Vorhandensein der `run()`-Methode
- `synchronized` bringt auch an lustigen Stellen noch einen Punkt.
- Falscher Knoten wird geblockt: -1 Punkt
- `notify()` und `notifyAll()` werden akzeptiert, dürfen auch außerhalb des `synchronized`-Blocks stehen

Die Grammatik von MiniJava

```
<program> ::= <decl>* <stmt>*

<decl>    ::= <type> <name> ( , <name> )* ;

<type>    ::= int

<stmt>    ::= ;
            | { <stmt>* }
            | <name> = <expr>;
            | <name> = read();
            | write(<expr>);
            | if (<cond>) <stmt>
            | if (<cond>) <stmt> else <stmt>
            | while(<cond>) <stmt>

<expr>    ::= <number>
            | <name>
            | (<expr>)
            | <unop> <expr>
            | <expr> <binop> <expr>

<unop>    ::= -

<binop>   ::= - | + | * | / | %

<cond>    ::= true | false
            | (<cond>)
            | <expr> <comp> <expr>
            | <bunop> (<cond>)
            | <cond> <bbinop> <cond>

<comp>    ::= == | != | <= | < | >= | >

<bunop>   ::= !

<bbinop>  ::= && | ||
```