Klausur, 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 05.04.2018

Wiederholungsklausur

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 **28** Seiten erhalten haben.
- Sie können maximal **150** Pinguine erreichen. Erreichen Sie mindestens **60** Pinguine, 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.

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

1	2	3	4	5	6	7	8	Σ

..... Erstkorrektor

..... Zweitkorrektor

Aufgabe 1 Multiple Choice / Grundlagen [15 Pinguine]

Kreuzen Sie in den folgenden Multiple-Choice-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 Pinguinzahl bei anderen Teilaufgaben aus.

1. Welche der folgenden Wörter gehören **nicht** zu den reservierten Schlüsselwörtern in Java?

   - ☐ int
   - ☒ allocate
   - ☐ while
   - ☐ else
   - ☒ let

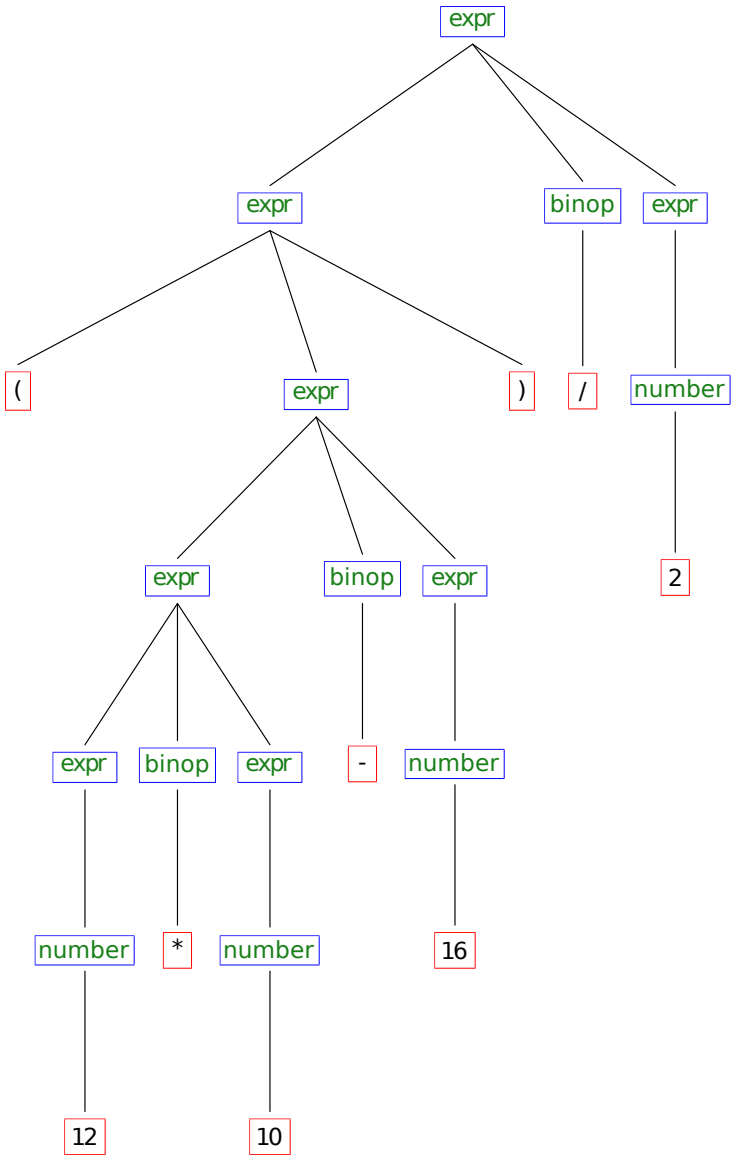
2. Geben Sie die folgenden Zahlen jeweils hexadezimal, oktal und binär an.
   Beispiel: `19 = 0x13 = 023 = 0B0001_0011`

   (a) `50 =` `0x32 = 062 = 0B0011_0010`

   (b) `1022 =` `0x3FE = 01776 = 0B0011_1111_1110`

   (c) `261 =` `0x105 = 0405 = 0B0001_0000_0101`

3. Welche der folgenden Statements enthalten Teilausdrücke, zu denen der gezeigte Syntaxbaum passt?



- ☐ `int a = 13*10 - 2;`
- ☒ `int b = (12*10 - 16) / 2;`
- ☒ `int c = (12*10 - 16) / 2 + a;`
- ☐ `int d = 12*10 - 16 / 2;`
- ☐ `int e = a + a;`

4. Die `int`-Variable `x` habe vor jeder der folgenden Teilaufgaben den Wert 1. `y` sei eine Variable vom Typ `String`. Zu welchem Wert evaluieren die folgenden Java-Ausdrücke?

   (a) `5 / 3 + 1.0`: `2.0`

   (b) `5 / (3 + 1.0)`: `1.25`

   (c) `5 - 1 + "Schlauin" + 9 + 9`: `"4Schlauin99"`

   (d) `y = x++ + "Doofuin" + --x`: `"1Doofuin1"`

   (e) `x = (1 == 1 ? 42 : ++x)`: `42`

5. Zeichnen Sie zu den Ausdrücken 4d und 4e der letzten Teilaufgabe jeweils den Auswertungsbaum.

6. Betrachten Sie den folgenden Code:

```
1  static void f(int[] a) {
2    a[0] = a[0] + 99;
3    a = new int[42];
4    a[0] = 42;
5  }
6
7  static void g() {
8    int[] a = new int[12];
9    a[1] = 3;
10   f(a);
11   // Markierung
12 }
```

Welche Aussagen sind am markierten Programmpunkt korrekt?

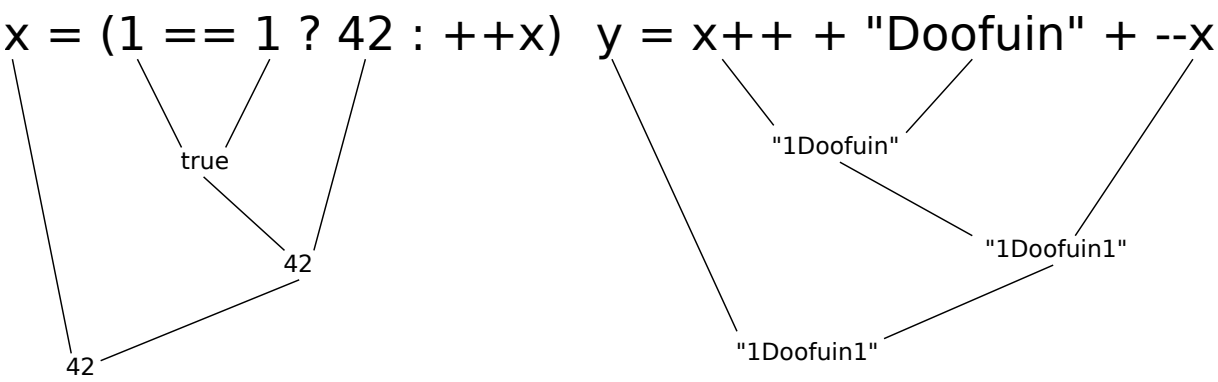
- ☐ Das Array `a` hat an Position `0` den Wert 42.
- ☐ Das Array `a` hat an Position `0` den Wert 0.
- ☒ Das Array `a` hat an Position `0` den Wert 99.
- ☐ Das Array `a` hat an Position `1` den Wert 0.
- ☒ Das Array `a` hat an Position `1` den Wert 3.
- ☐ Das Programm kompiliert nicht, da auf die uninitialisierte Array-Position 0 in `f` zugegriffen wird.
- ☐ Das Programm wirft beim Aufruf der Methode `f` eine Exception vom Typ `java.lang.DoubleVariableException`, weil die Variable `a` mehrfach deklariert ist.
- ☒ Die Variable `a` wird beim Aufruf von `f` kopiert.

7. Welche Aussagen zu Wrapperklassen sind zutreffend?

- ☐ Wrapperklassen umschließen wertvolle Objekte, um sie gegen den zerstörerischen Zugriff durch Computerviren zu schützen. Wrapperklassen werden daher insbesondere seit dem vermehrten Auftreten von Erpressungstrojanern häufiger verwendet.
- ☒ Objekte der Klasse `Integer` sind unveränderlich.
- ☐ Wrapperklassen wurden eingeführt, als klar wurde, dass Basistypen wie `int` ein Designfehler von Java darstellen. Aufgrund ihrer höheren Effizienz sind sie den Basistypen stets zu bevorzugen.
- ☒ Wrapperklassen werden im Kontext von generischen Typen verwendet, da generische Typparameter nicht mit Basistypen belegt werden können.

Lösungsvorschlag 1

Auswertungsbäumchen:



Bewertung:

1. 1 Pinguin; noch 0.5 Pinguine, wenn ein Kreuz vergessen wurde (alle anderen Fälle 0 Pinguine)
2. 3 Pinguine, ein Pinguin pro Teilaufgabe; beachte:
   - 0.5 Pinguine Abzug pro Fehler
   - 0.5 Pinguine bei (nicht-trivialen) konsistent falschen Zahlen ($\rightsquigarrow$ Studenten, die sich bei einer Umrechnung vertan haben und dann davon ableiteten, erhalten noch einen halben Pinguin)
3. 1 Pinguin, 0.5 Pinguine falls genau ein richtiges Kreuz, sonst nichts
4. 5 Pinguine, ein Pinguin pro Teilaufgabe
5. 2 Pinguine, 1 Pinguin pro Teilaufgabe; 0.5 Pinguine Abzug pro Fehler, insgesamt 0.5 Pinguine Abzug für vergessene Zuweisung in beiden Teilaufgaben
6. 2 Pinguine:
   - 1 Pinguin auf alle Fragen über den Wert von `a`, je 0.5 Pinguine auf den Wert von `a[0]` und `a[1]`
   - 1 Pinguin auf Rest, alles oder nichts
7. 1 Pinguin; noch 0.5 Pinguine ohne die korrekte Aussage über die Unveränderlichkeit

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
- Folgefehler von Teilaufgabe 4 (Ausdrucksauswertung) zu Teilaufgabe 5 (Auswertungsbäumchen) sind möglich.

Aufgabe 2 Eine [ ] kommt selten allein [15 Pinguine]

Vervollständigen Sie die Implementierung unten. Die Methode `sort(...)` führt den bekannten Mergesort-Algorithmus mit einer kleinen Modifikation aus: Das Array wird in drei statt zwei Teile geteilt. Die Teile werden rekursiv sortiert und anschließend die sortierten Bereiche mit der `merge(...)`-Methode gemischt.

Schreiben Sie in jede Box genau einen Ausdruck!

```
static void sort(int[] array) {
  sort(array, 0, array.length - 1); // Gesamter Bereich
}
static void sort(int[] array, int left, int right) {
  int nr = (right - left) / 3;
  if (right > left) {
    int endFirst = left + nr;
    int endSecond = right - nr;
    // 3 Teilbereiche sortieren:
    sort (array, left, endFirst);
    sort (array, endFirst + 1, endSecond);
    sort (array, endSecond + 1, right);
    // Sortierte Bereiche mischen:
    merge(array, left, endFirst, endSecond, right);
  }
}
static void merge(int[] a, int low, int first, int second, int high) {
  // 3 Laufvariablen für die Bereiche:
  int left = low;
  int mid = first + 1;
  int right = second + 1;
  // Ergebnisspeicher mit Laufvariable:
  int[] sorted = new int[1 + [high-low]];
  int x = 0;
  while ([left <= first] || mid <= second || right <= high) {
    if (left <= first
    && (mid > second || a[mid] > a[left])
    && (right > high || a[right] > a[left])) {
      sorted[x++] = a[left++];
    } else if (mid <= second
          && (right > high || [a[right] > a[mid]])) {
      sorted[x++] = a[mid++];
    } else {
      sorted[x++] = a[[right++]];
    }
  }
  // Ergebnis ins Array kopieren:
  for (int index = low; index <= high;) {
    a[index] = sorted[[index++-low]];
  }
}
```

Lösungsvorschlag 2
Bewertung: 3 Pinguine pro Lücke

Festlegungen:

1.
   - `low`: 1 Pinguin
   - `high`: 1 Pinguin
   - `low X high` (falscher Operator): 2 Pinguine
   - `left`: 1 Pinguin
2.
   - `left > first`: 2 Pinguine
   - `left <= ...`: 2 Pinguine
   - `first <= low`: 1 Pinguin
   - `low <= first`: 2 Pinguine
   - `left >= low`: 1 Pinguin
3.
   - `a[right] > a[left]`: 2 Pinguine
   - `a[mid] < a[left]`: 1 Pinguin
   - `a[mid] > a[left]`: 1 Pinguin
4.
   - `right--`: 2 Pinguine
   - `right`: 2 Pinguine
   - `high++`: 1 Pinguin
   - `left++`: 1 Pinguin
5.
   - `index++`: 2 Pinguine
   - `index`: 1 Pinguin
   - `low`: 1 Pinguin
   - `index - low`: 2 Pinguine

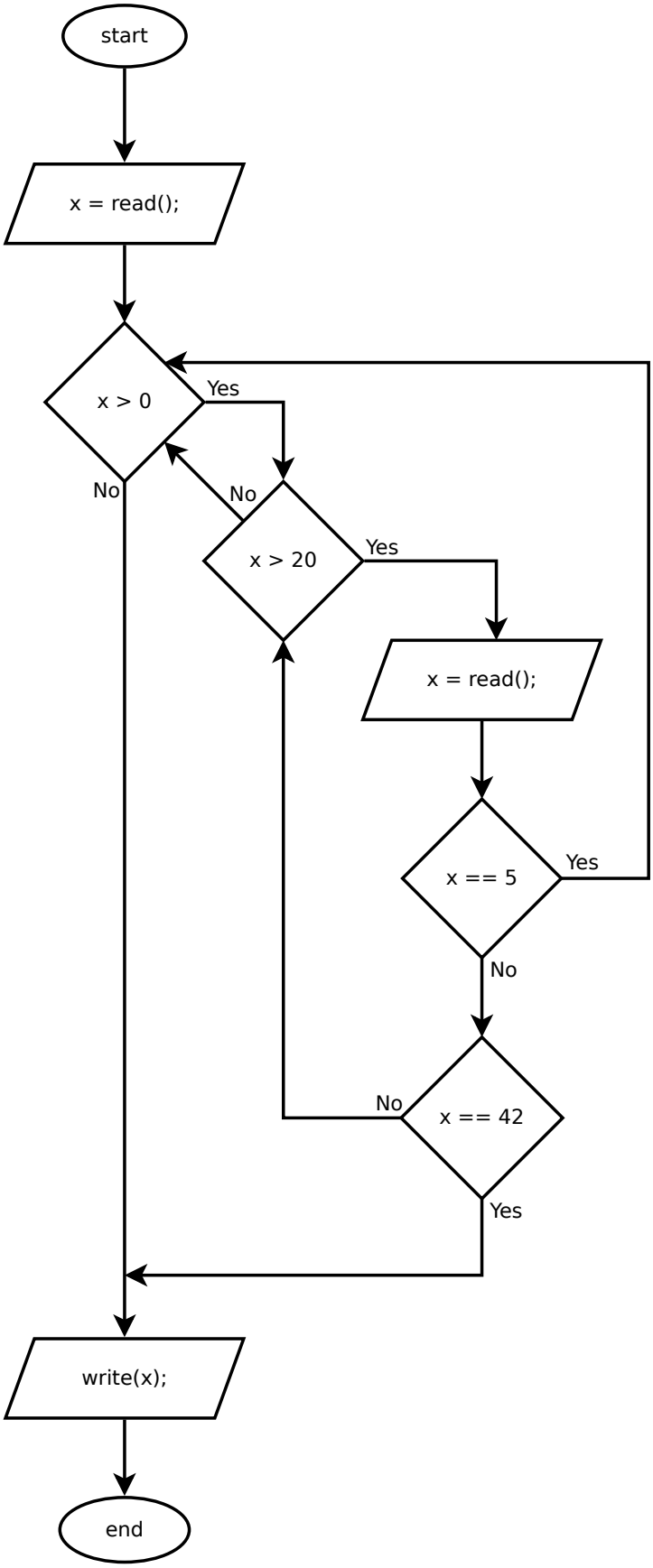
Aufgabe 3 Kontrollierter Flussgraph [15 Pinguine]

Zeichnen Sie für das folgende Programm den Kontrollflussgraphen.

```
1  int x;
2  x = read();
3
4  outer: while (x > 0) {
5    while (x > 20) {
6      x = read();
7      if(x == 5) continue outer;
8      if(x == 42) break outer;
9    }
10 }
11
12 write(x);
```

Hinweis: `continue name;` führt die Schleife mit dem Namen `name` fort; es wird also der aktelle Schleifendurchlauf von `name` abgebrochen und direkt die Schleifenbedingung wieder getestet. `break name;` verlässt die Schleife mit dem Namen `name`.

Lösungsvorschlag 3



Festlegungen:

- pro fehlendes Kästchen: 1 Pinguin Abzug
- pro fehlender Kante: 0.5 Pinguine Abzug
- pro fehlender Pfeilspitze: 0.5 Pinguine Abzug, insgesamt max. 4
- Start- bzw. Stopknoten sinnfrei benannt: 0.5 Pinguine Abzug
- Knoten für Deklaration: 1 Pinguin Abzug
- Abzweigung nicht eindeutig: 0.5 Pinguine Abzug
- Parallelogramm bzw. Rechteck egal, keine Raute bei Bedingungen: 0.5 Pinguine Abzug
- Semikolon mit oder ohne egal
- Systematisch keine Klammern bei `read()` bzw. `write(...)`: 0.5 Pinguine Abzug

Aufgabe 4 Generics + Polymorphie = ♡ [20 Pinguine]

Betrachten Sie den folgenden Code:

```
1  public class Poly {
2    static class A {
3      protected A a;
4      public A(A a) { this.a = a; }
5      public void f(A a) { System.out.println("A.f(A)");  this.a.f(a); }
6      public void f(B<A> a) { System.out.println("A.f(B)"); a.f(this.a); }
7    }
8
9    static class B<T> extends A {
10     public T t;
11     public B(Object o) { super(null); }
12     public B() { super(null); a = new C<B<T>>(this); }
13     public void f(B<A> a) { System.out.println("B.f(B<A>)"); }
14   }
15
16   static class C<T extends A> extends B<T> {
17     public T t;
18     public C(T t) { super(null); a = this; this.t = t; }
19     public void f(A a) { System.out.println("C.f(A)"); }
20     public void f(B<A> a) { System.out.println("C.f(B<A>)"); a.f(t); }
21   }
22
23   public static void main(String[] args) {
24     B<Integer> b1 = new B<Integer>();
25     B<A> b2 = new B<A>();
26     A a = new A(b1);
27     C<A> c = new C<A>(a);
28
29     a.f(b1); // Aufruf 1
30     a.f(b2); // Aufruf 2
31
32     B<C<A>> b3 = new B<C<A>>();
33     B<A> b4;
34
35     (b2 = b3).f(a);  // Aufruf 3
36     (b4 = c).t.f(a); // Aufruf 4
37   }
38 }
```

Geben Sie für jeden der markierten Aufrufe die Ausgabe an. Gehen Sie davon aus, dass nur ein Aufruf im Programm vorhanden ist; die anderen seien jeweils auskommentiert. Es kann jeweils auch ein Compiler- oder Laufzeitfehler auftreten. Geben Sie bei einem Laufzeitfehler an, wo genau bzw. wieso dieser auftritt. Begründen Sie bei *Aufruf 3* kurz das Verhalten des Java-Compilers anhand eines Beispiels.

Lösungsvorschlag 4

1.

```
A.f(A)
A.f(A)
C.f(A)
```

2.

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

3. Die Zuweisung ist nicht möglich, da generische Typen invariant sind. Stellen wir uns vor, wir hätten eine Liste von Hasen, die wir in eine Liste von Säugetieren casten:

```
ArrayList<Hase> hasen = new ArrayList<>();
ArrayList<Säugetier> säugetiere = hasen;
```

   Wäre dies erlaubt, könnte man durch `säugetiere.add(new Penguin())` einen Pinguin in eine Liste von Hasen einfügen.

4. Es tritt eine `java.lang.NullPointerException` auf, da bei Attributen generell nur der statische Typ eine Rolle spielt, mithin also das Attribut `t` aus `B` zugegriffen wird, welches nirgendwo auf einen Wert gesetzt wird.

Bewertung:

1. 1.5 Pinguine pro Zeile + 1 Pinguin wenn komplett richtig (+1P aus Begründung)
2. 1.5 Pinguine pro Zeile + 1 Pinguin wenn komplett richtig (+1P aus Begründung)
3. 3 Pinguine auf Antwort, 3 Pinguine auf Begründung + Erklärung mit Beispiel
4. 3 Pinguine (+1P aus Begründung)

Sonderregelung wegen des Ausschlusses des Beispiels generischer Listen vom klausurrelevanten Stoff: Punkte, die bei der Begründung zur 3. Teilaufgabe fehlen, verteilen sich anteilig auf die anderen Teilaufgaben. Studenten können diese Punkte also auf zwei verschiedene Arten bekommen.

Festlegungen:

Aufruf 4:

- Exception: 1 Pinguin
- NullPointer: 2 Pinguine

- + Begründung: 3 Pinguine
- Falsche Fehlerart: 0.5 Pinguine

Sonderregelung:

- 0.5 Extrapinguine pro Zeile, maximal 1 pro Teilaufgabe
- 0.5 Extrapinguine für die Exception, insgesamt 1 für `NullPointerException`

Aufgabe 5 Objektorientierte Modellierung [15 Pinguine]

In dieser Aufgabe soll die Korrektur einer Klausur objektorientiert modelliert werden. Gehen Sie entsprechend der folgenden Teilaufgaben vor.

1. Implementieren Sie zur Darstellung einer Klausuraufgabe die Klasse `Assignment`, die in zwei privaten Attributen die Maximalpunkte der Aufgabe speichert und ob in der Aufgabe Pinguine vorkommen. Das Setzen der Attribute soll über den Konstruktor der Klasse und das Abfragen über die Methoden `getPoints()` bzw. `hasPenguins()` möglich sein.

2. Definieren Sie ein Interface `Examiner` mit der Methode `checkPoints(Assignment assignment)`, die für eine gegebene Aufgabe die erreichte Punktzahl bestimmt, und implementieren Sie dieses anschließend in zwei Klassen `PenguinFriend` und `DrunkenGuy`, welche die in der Praxis am häufigsten auftretenden Arten von Klausurkorrektoren repräsentieren. Der Korrektor `PenguinFriend` gibt dabei für eine Aufgabe stets volle Punktzahl, wenn Pinguine in der Aufgabe vorkommen und ansonsten genau 5 Punkte. Der Korrektor `DrunkenGuy` weiß leider oft nicht was er tut und gibt einfach eine zufällige Anzahl an Punkten (zwischen 0 Punkte und der Maximalpunktzahl der Aufgabe).

   Hinweis: Nutzen Sie den Aufruf `new java.util.Random().nextInt(n)`, um eine Zufällige Ganzzahl z mit $0 \leq z < n$ zu erzeugen.

3. Implementieren Sie nun eine abstrakte Klasse `Exam`, die ein statisches Attribut `java.util.Map<Integer, Float> gradingScale` enthält, welches nur für die Klasse selbst sowie abgeleitete Klassen zugreifbar ist. Die `gradingScale` ordnet einer Punktzahl die entsprechende Note zu (Sie müssen diese nicht initialisieren). Die Klasse `Exam` muss außerdem eine abstrakte Methode `float examine(Examiner examiner)` definieren.

4. Implementieren Sie schließlich die konkrete Klasse `WrittenExam`, die von `Exam` erbt und diese um ein privates Array von `Assignments` erweitert, welches von einem Konstruktorargument initialisiert wird. Implementieren Sie die Methode `examine(...)` so, dass alle Aufgaben der Klausur vom übergebenen Korrektor geprüft werden und die finale Klausurnote zurückgegeben wird, welche mithilfe der `gradingScale` der Basisklasse aus der Summe der erreichten Punkte bestimmt wird.

Hinweis: Nutzen Sie objektorientierte Sprachmittel sinnvoll, um die Aufgabe zu lösen. Es geht bei dieser Aufgabe insbesondere um gute *Modellierung*.

Hinweis: Das Interface `java.util.Map<KeyType, ValueType>` verfügt über eine Methode `ValueType get(KeyType key)`, welche das Bild eines Elements `key` zurückliefert.

Lösungsvorschlag 5

```java
// 1, Klassenkopf: 0.5 Pinguine
public class Assignment {
  // Attribute: 0.5 Pinguine
  private int points;
  private boolean penguins;

  // Konstruktor: 1 Pinguin
  public Assignment(int points, boolean penguins) {
    this.points = points;
    this.penguins = penguins;
  }

  // Methoden: Je 1 Pinguin
  public int getPoints() { return this.points; }
  public boolean hasPenguins() { return this.penguins; }
}

//2, Ganzes Interface: 1 Pinguin
public interface Examiner {
  public int checkPoints(Assignment assignment);
}

// Klassenkopf: 0.5 Pinguine
public class PenguinFriend implements Examiner {
  // Methode: 1 Pinguin
  public int checkPoints(Assignment assignment) {
    return assignment.hasPenguins() ? assignment.getPoints() : 5;
  }
}

// Klassenkopf: 0.5 Pinguine
public class DrunkenGuy implements Examiner {
  // Methode: 1 Pinguin
  public int checkPoints(Assignment assignment) {
    return (new java.util.Random()).nextInt(assignment.getPoints() + 1);
  }
}

// 3
// Klassenkopf: 0.5 Pinguine
public abstract class Exam {
  // Abstrakte Methode: 0.5 Pinguine
  public abstract float examine(Examiner e);

  // protected: 1 Pinguin, static: 0.5 Pinguine
  protected static java.util.Map<Integer, Float> gradingScale;
}
```

```java

// 4
// Klassenkopf: 1 Pinguin
public class WrittenExam extends Exam {
  // Attribut: 0.5 Pinguine
  private Assignment[] assignments;

  // Konstruktor: 1 Pinguin
  public WrittenExam(Assignment[] assignments) {
    this.assignments = assignments;
  }

  // Methodenkopf: 0.5 Pinguine
  public float examine(Examiner e) {
    int sum = 0;
    // Schleifenkopf: 0.5 Pinguine
    for(Assignment a : assignments)
      // Schleifenkörper: 0.5 Pinguine
      sum += e.checkPoints(a);
    // Rückgabe: 0.5 Pinguine
    return Exam.gradingScale.get(sum);
  }
}
```

Festlegungen:

- Geschweifte Klammern fehlen öfters: 0.5 Pinguine Abzug
- Falscher oder fehlender Rückgabetyp: 0.5 Pinguine Abzug
- `protected` statt `private` bei Attributen: kein Abzug
- nichts statt `private` bei Attributen: 0.5 Pinguine Abzug
- Funktionsklammern bei Attributen: 0.5 Pinguine Abzug

Aufgabe 6 Die Bäuerin und die Magische Klammer [20 Pinguine]

Lösen Sie die folgenden Teilaufgaben 1 und 2 *rekursiv*. Es sind hier **keine Schleifen** erlaubt. Bei Teilaufgabe 3 ist eine Schleife erlaubt. Es dürfen Hilfsmethoden implementiert werden, jedoch lediglich eigene Methoden verwendet werden. Es dürfen insbesondere auch keine Arrays erstellt werden. Eigene Klassen sind nicht erlaubt.

1. Schreiben Sie eine Methode `static int ackerwoman(int m, int n)` zur Berechnung der Ackerfraufunktion $A(m, n)$:

$$A(m,n) = \begin{cases} n+1 & m = 0 \\ A(m-1,1) & m > 0 \wedge n = 0 \\ A(m-1, A(m, n-1)) & m > 0 \wedge n > 0 \end{cases}$$

   Die Funktion ist für alle $m, n \in \mathbb{N}_0$ definiert. Fehlerbehandlung ist nicht verlangt.

2. Implementieren Sie die Methode `static boolean isBalanced(String expr)`, welche einen gegebenen Ausdruck auf korrekte Klammerung prüft. Wir betrachten in dieser Aufgabe nur die beiden Klammertypen `'('` bzw. `')'` und `'['` bzw. `']'`. Es können außer den beiden Arten von Klammern beliebige andere Symbole im Ausdruck vorkommen, die ignoriert werden sollen. Beachten Sie die folgenden Beispiele:

   - `isBalanced("[(xXx)+]")` liefert `true`
   - `isBalanced("[other symbols]()()")` liefert `true`
   - `isBalanced("[(])")` liefert `false`
   - `isBalanced(")[](")` liefert `false`
   - `isBalanced("[Pinguin hier](")` liefert `false`

   Sie dürfen zur Lösung dieser Aufgabe zusätzlich die Methoden `charAt(int index)` und `length()` der Klasse `String` verwenden. Sie dürfen keine eigenen Strings erstellen.

3. Betrachten Sie folgende wundervolle rekursive Methode:

```
2  static int compute(int n, int a, int b) {
3    if (n < 34) {
4      return a;
5    }
6    if (n % b == 0) {
7      return compute(n - 1, a + 9, b);
8    }
9    return compute(n - 2, a - 1, b + 1);
10 }
```

   Wandeln Sie die Methode in ein nicht-rekursives Programm mit nur einer Schleife um. Wieso ist dies hier einfach möglich?

Lösungsvorschlag 6

- Ackerfraufunktion:

```
1  public class Ackerwoman {
2    public static int ackerwoman(int m, int n) {
3      // Test: 1 Pinguin
4      if (m == 0) // Zuweisung: -0.5 Pinguine
5        // Rückgabe: 1 Pinguin
6        return n + 1;
7      // Test: 1.5 Pinguine
8      else if (m > 0 && n == 0)
9        // Rekursiver Aufruf: 1.5 Pinguine
10       return ackerwoman(m - 1, 1);
11     else
12       // Rekursiver Aufruf: 2 Pinguine
13       return ackerwoman(m - 1, ackerwoman(m, n - 1));
14   }
15 }
```

Festlegungen:

  - Funktion gibt u.U. nichts zurück (Compilerfehler): 0.5 Punkte Abzug

- Teilaufgabe 3:

```
12 static int computer(int n, int a, int b) {
13   // Schleife: 1 Pinguin
14   // Festlegung: n > 34: -0.5 Pinguine
15   // Bedingungen vertauscht: -0.5 Pinguine
16   while (n >= 34) {
17     if (n % b == 0) {
18       // Parameter setzen: 0.5 Pinguine
19       n = n - 1;
20       a = a + 9;
21     } else {
22       // Parameter setzen: 0.5 Pinguine
23       n = n - 2;
24       a = a - 1;
25       b = b + 1;
26     }
27   }
28   return a; // return fehlt: -0.5 Pinguine
29 }
```

Die Umwandlung ist hier so einfach möglich, weil die Methode endrekursiv ist (1 Pinguin).

- Klammergedöns:

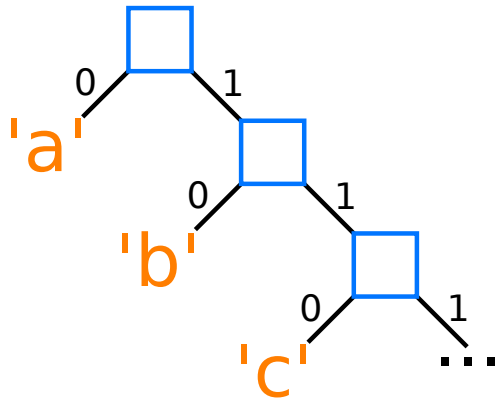
```java
public class BracketsWithoutStack {
  private static char matching(char c) {
    if (c == '(')
      return ')';
    return ']';
  }

  private static int isBalancedHelper(String expr, int from) {
    // Abbruch bei Ende: 1 Pinguin
    if (from == expr.length())
      return expr.length();
    char atFrom = expr.charAt(from);
    // Abbruch bei schließender Klammer: 1 Pinguin
    if (atFrom == ')' || atFrom == ']')
      return from;
    // Rekursiver Aufruf: 1.5 Pinguine
    int next = isBalancedHelper(expr, from + 1);
    // Fehler wird durchgereicht: 1 Pinguin
    if (next < 0)
      return -1;
    // Test auf öffnende Klammer: 1 Pinguin
    if (atFrom == '(' || atFrom == '[') {
      // Korrekte Reaktion auf öffnende Klammer: 2 Pinguin
      if (next < expr.length() && expr.charAt(next) ==
      ↪  matching(atFrom))
        next = next + 1;
      else
        return -1;
    }
    // Rekursiver Aufruf: 1.5 Pinguine
    return isBalancedHelper(expr, next);
  }

  public static boolean isBalanced(String expr) {
    // Aufruf von Hilfsfunktion, Behandlung von Ergebnis: 1
    ↪  Pinguin
    return isBalancedHelper(expr, 0) == expr.length();
  }
}
```

Festlegungen:

- Ohne Reihenfolge: maximal 3 Punkte
- Nur eine Klammer wird gezählt (`boolean`): maximal 2 Punkte

Aufgabe 7 Huffwoman-Kodierung [25 Pinguine]

Bei der sog. *Huffwoman*-Kodierung werden unterschiedlichen Buchstaben Code-Wörter zugeordnet, die nicht gleich lang sind. Man wählt nun für häufiger vorkommende Buchstaben möglichst kurze Kodierungen; auf diese Weise kann man insgesamt Speicherplatz sparen. Die Zuordnung von Symbol zu Code-Wort erfolgt durch einen binären Baum, wobei sich die Symbole an den Blättern befinden und Kodierungen durch die Pfade zu den Blättern repräsentiert werden. Betrachten Sie das folgende Beispiel:



Hier wird der Buchstabe `'c'` durch 110 dargestellt; für `'a'` benötigen wir dagegen nur ein Zeichen, nämlich 0. Es sei für die Repräsentierung des Baumes folgender Code gegeben:

```
1  interface Node { }
2
3  class Leaf implements Node {
4    private char c;
5
6    public char getC() {
7      return c;
8    }
9
10   public Leaf(char c) {
11     this.c = c;
12   }
13 }
14
15 class InnerNode implements Node {
16   // Teilbaum mit folgender '0' in der Kodierung
17   private Node succZero;
18
19   // Teilbaum mit folgender '1' in der Kodierung
20   private Node succOne;
21
22   public InnerNode(Node succZero, Node succOne) {
23     this.succZero = succZero;
24     this.succOne = succOne;
25   }
```

```
}

class Huffwoman {
  public static String decode(String encoded, Node root) {
    // Todo
  }

  public static String[] buildEncodingTable(Node root) {
    // Todo
  }

  public static String encode(String text, Node root) {
    // Todo
  }
}
```

Für sämtliche der folgenden Teilaufgaben dürfen Sie die gegebenen Klassen um Hilfsmethoden erweitern. Kennzeichnen Sie Methoden als in den gegebenen Klassen einfach wie folgt:

```
// In Klasse InnerNode:
public void meineLustigeHilfsmethode() {
}
```

Bearbeiten Sie nun folgende Teilaufgaben.

1. Implementieren Sie die Methode `decode(...)`, die als Parameter einen kodierten String s und einen Huffwoman-Baum erhält und s dekodiert; das Ergebnis soll zurückgeliefert werden. Ist die Eingabe zum Beispiel der String `"0010110"` zusammen mit dem oben abgebildeten Baum, so soll die Methode `"aabc"` zurückgeben. Erzeugen Sie eine `RuntimeException`, wenn `encoded` nicht dekodiert werden kann.

2. Implementieren Sie die Methode `buildEncodingTable(...)`, die aus einem gegebenen Huffwoman-Baum die Kodierungstabelle aufbaut. Die Tabelle ordnet jedem Kleinbuchstaben (a bis z, `'a'` ist der 0-te Buchstabe) seine jeweilige Kodierung zu.

3. Implementieren Sie die Methode `encode(...)`, die als Parameter einen String s und einen Huffwoman-Baum erhält und s kodiert; das Ergebnis soll zurückgeliefert werden. Ist die Eingabe zum Beispiel der String `"aabc"` zusammen mit dem oben abgebildeten Baum, so soll die Methode `"0010110"` zurückgeben.

   Die Eingabe-Strings bestehen hier aus Kleinbuchstaben (a bis z). Sie dürfen ohne Fehlerprüfung davon ausgehen, dass der Parameter `text` lediglich Kleinbuchstaben enthält.

Lösungsvorschlag 7

```java
public class Huffwoman {
  static class DecodeResult {
    public int next;
    public char result;

    public DecodeResult(int next, char result) {
      this.next = next; this.result = result;
    }
  }

  static interface Node {
    DecodeResult decode(String encoded, int from);
    void buildEncodingTable(String[] table, String codeWord);
  }

  static class Leaf implements Node {
    private char c;

    public char getC() {
      return c;
    }

    public Leaf(char c) {
      this.c = c;
    }

    public DecodeResult decode(String encoded, int from) {
      return new DecodeResult(from, c);
    }

    public void buildEncodingTable(String[] table, String codeWord) {
      // Einfügen an der richtigen Stelle: 3 Pinguine
      table[c - 'a'] = codeWord;
    }
  }

  static class InnerNode implements Node {
    private Node succZero;
    private Node succOne;

    public InnerNode(Node succZero, Node succOne) {
      this.succZero = succZero;
      this.succOne = succOne;
    }

    public DecodeResult decode(String encoded, int from) {
      // Fehlerbehandlung (hier + unten): je 0.5 Pinguine
```

```
48        if(from >= encoded.length())
49          throw new RuntimeException("Invalid input");
50        // Abstieg in die richtige Richtung: 4 Pinguine
51        // Festlegung: Programmierung nach Beispiel (Oer-Seite ist immer
52        // fix ein Blatt) gibt noch 1 Pinguin
53        if(encoded.charAt(from) == '0')
54          return succZero.decode(encoded, from + 1);
55        else if(encoded.charAt(from) == '1')
56          return succOne.decode(encoded, from + 1);
57        throw new RuntimeException("Invalid input");
58      }
59
60      public void buildEncodingTable(String[] table, String codeWord) {
61        // Abstiegt in beide Richtungen: 2 Pinguine
62        // Verlängerung von codeWord: 2 Pinguine
63        succZero.buildEncodingTable(table, codeWord + '0');
64        succOne.buildEncodingTable(table, codeWord + '1');
65      }
66    } // end of InnerNode
67
68    // Ganze Methode: 9 Pinguine
69    public static String decode(String encoded, Node root) {
70      String result = "";
71      int from = 0;
72      // Schrittweise Dekodierung in Schleife: 4 Pinguine
73      // Festlegung: Schleifenkopf: 1 Pinguin
74      while(from < encoded.length()) {
75        // Festlegung: 1 Pinguin pro Zeile im Körper
76        DecodeResult decodeRes = root.decode(encoded, from);
77        from = decodeRes.next;
78        result += decodeRes.result;
79      }
80      return result;
81    }
82
83    // Ganze Methode: 7 Pinguine
84    public static String[] buildEncodingTable(Node root) {
85      String[] table = new String[26];
86      // Festlegung: 1 Pinguin für an richtige Stelle schreiben alleine
87      // ohne Nutzung des Baumes
88      root.buildEncodingTable(table, "");
89      return table;
90    }
91
92    // Ganze Methode: 9 Pinguine
93    public static String encode(String text, Node root) {
94      // Bau der Tabelle: 2 Pinguin
95      String[] table = buildEncodingTable(root);
96      String result = "";
97      // Schleife: 3 Pinguin
```

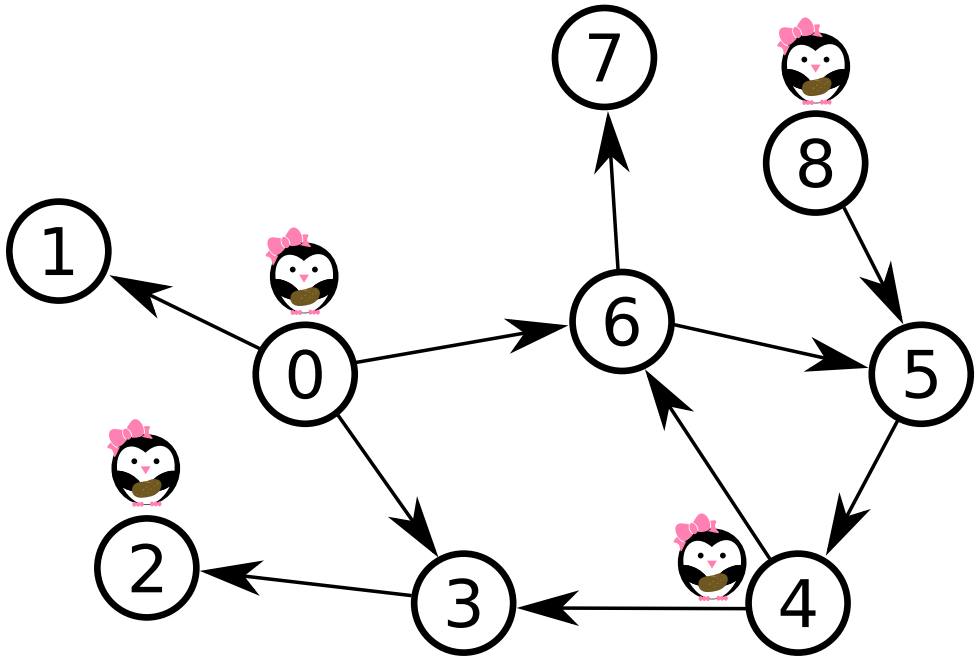
```
98      for (int i = 0; i < text.length(); i++)
99        // Aufbau von result: 4 Pinguin
100       result += table[text.charAt(i) - 'a'];
101     return result; // Festlegung: ohne return 0.5 Abzug
102   }
103 }
```

Sonstige Festlegungen:

- Bei den letzten beiden Teilaufgaben gibt es 7 Punkte für die Traversierung des Baumes. Man kann jeweils eine der Aufgaben durch die andere lösen, beide Richtungen sind erlaubt. Wird aber die dritte Teilaufgabe durch Aufruf von `buildEncodingTable(...)` und die zweite durch Aufruf von `encode(...)` gelöst, fehlen die 7 Punkte für den Ablauf des Baumes.

Aufgabe 8 Kartoffeluine (Threads) [25 Pinguine]

In dieser Aufgabe geht es darum, Pinguine durch gerichtete Graphen parallel heiße Kartoffeln tragen zu lassen. Ein Graph besteht aus Knoten, die durch gerichtete Kanten miteinander verbunden sind. Die folgende Abbildung zeigt als Beispiel einen Graphen mit vier Kartoffeluininnen:



Der Graph wird durch ein Objekt einer Klasse repräsentiert, die die folgende Schnittstelle implementiert:

```
1  public interface Graph {
2    // Liefert den Knoten mit dem Index 'nodeIndex' zurück
3    public Node getNode(int nodeIndex);
4
5    // Liefert ein Array der Indices der Zielknoten von Kanten ab dem
6    // Knoten mit dem Index 'fromNode' zurück
7    public int[] getEdges(int fromNode);
8  }
```

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

```
1  public class Node {
2    // Speichert die Anzahl der Pinguine, die an diesem Knoten auf
3    // Übernahme einer Kartoffel warten
4    public int waiting;
5
6    // Speichert die Temperatur der Kartoffel zur Übergabe von einem
7    // zum nächsten Pinguin zwischen
```

```
8     public int potato;
9
10    public Node() {
11      waiting = 0;
12      potato = 0;
13    }
14  }
```

Ein Pinguin beginnt mit einer Kartoffel einer bestimmten Anfangstemperatur sowie einem Anfangsknoten, ab dem er den Graphen rekursiv durchläuft. Die verwendete Kante soll dabei jeweils zufällig gewählt werden. Befindet sich der Pinguin in einer Sackgasse, so stirbt das arme Tierchen leider an Verbrennungen (Thread beendet sich). Wann immer ein Kartoffeluin einer Kante im Graphen folgt, kühlt sich die Kartoffel durch den Wind der Bewegung um genau ein Grad ab.

Erreicht die Kartoffel die Temperatur 0 Grad, so wird die Kartoffel gegessen (keine Ausgabe nötig) und der Pinguin wartet aufopferungsvoll darauf, eine heiße Kartoffel von einem anderen Pinguin zu erhalten.

Ist die Kartoffel dagegen noch heiß (Temperatur größer als 0 Grad), prüft der Pinguin sehnsüchtig, ob am aktuellen Knoten bereits ein anderer Pinguin darauf wartet, ihm die schmerzende Last abzunehmen. In diesem Fall soll die Kartoffel übergeben werden. Der abgebende Pinguin wartet nun seinerseits am Knoten auf eine neue Kartoffel, der empfangende Pinguin läuft weiter durch den Graphen.

Implementieren Sie den Thread `Kartoffeluin`, der im Konstruktor einen Graphen, einen Anfangsknoten und die Anfangstemperatur der Kartoffel 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 nodeIndex) throws InterruptedException {
2      int[] edges = graph.getEdges(nodeIndex);
3      int edgeIndex = ...; // Zufällige Kante wählen
4      traverse(edges[edgeIndex]);
5  }
```

Hinweis: Nutzen Sie die Klasse `java.util.Random` zur Erzeugung von Zufallszahlen. Insbesondere liefert die Objektmethode `nextInt(int b)` eine Zufallszahl k mit $0 \leq k < b$. Sie dürfen der Klasse `Kartoffeluin` Attribute hinzufügen.

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 + extends Thread: 1 Pinguin
public class Kartoffeluin extends Thread {
    // Attribute + Konstruktor: 1 Pinguin
    private Graph graph;
    private int startNode;
    private java.util.Random random = new java.util.Random();
    private int potato;

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

    private void traverse(int nodeIndex) throws InterruptedException {
      Node node = graph.getNode(nodeIndex);

      // Verringern der Kartoffel-Temperatur: 1 Pinguin
      potato--;

      // Synchronisierung: 2 Pinguine
      synchronized (node) {
        boolean wait = false;
        if (potato == 0)
          // Behandlung von kalter Kartoffel: 1 Pinguine
          wait = true;
        else if (node.waiting > 0) {
          if (node.potato == 0) {
            wait = true;
            // Behandlung keine Kartoffel wartet: 2 Pinguine
            node.potato = potato;
            node.notify();
          } else {
            // Tausch bei Übergabe: 2 Pinguine
            int tmp = potato;
            potato = node.potato;
            node.potato = tmp;
            // Benachrichtigung: 1 Pinguine
            node.notify();
          }
        }
        // Warten (zusammen mit wait-Variable): 3 Pinguine
        if (wait) {
          // Warteschleife: 3 Pinguine
          node.waiting++;
          node.wait();
          while (node.potato == 0)
```

```
48              node.wait();
49            // Übernahme der Kartoffel: 2 Pinguine
50            node.waiting--;
51            potato = node.potato;
52            node.potato = 0;
53          }
54        }
55
56        int[] edges = graph.getEdges(nodeIndex);
57        // Abbruch in Sackgasse: 2 Pinguine
58        if (edges.length == 0) {
59          return;
60        }
61        // Zufällige Kante: 1 Pinguin
62        int edge = random.nextInt(edges.length);
63        traverse(edges[edge]);
64      }
65
66      // run()-Methode: 1 Pinguin
67      public void run() {
68        // Fehlerbehandlung: 1 Pinguin
69        try {
70          // Aufruf von traverse(): 1 Pinguin
71          traverse(startNode);
72        } catch (InterruptedException e) {
73          e.printStackTrace();
74        }
75      }
76    }
```