Einführung in die Informatik 1

Prof. Dr. Alexander Pretschner, J. Kranz, G. Hagerer 19.02.2019

Klausur

Vorname	Nachname
Merlin	Rehbinder
Matrikelnummer	Unterschrift
idk lol	

- 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 **14** Seiten erhalten haben.
- Sie können maximal **150** Punkte erreichen.
- Als Hilfsmittel ist nur ein beidseitig handgeschriebenes DIN-A4-Blatt zugelassen.
- Sie dürfen Ihre Lösungen auf die Angabe oder das Klausurpapier schreiben.
- Zugelassene Sprachen für Antworten sind Deutsch und Englisch.

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

1	2	3	4	5	6	7	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  void f(int[] array) {
2    array[0] = 19;
3    array = new int[] {42};
4  }
5
6  void g() {
7    int[] array = new int[1];
8    f(array);
9  }
```

Welche der folgenden Aussagen sind korrekt?

- ☐ Die Variable `array` hat den Wert `null`.
- ☒ Die Variable `array` wird beim Aufruf kopiert, da in Java stets *Call-by-Value* stattfindet.
- ☐ Die Variable `array` wird beim Aufruf nicht kopiert, da in Java bei Referenztypen *Call-by-Reference* stattfindet.
- ☒ Das Array `array` enthält nach dem Aufruf von `f` (also am Ende von Methode `g()`) ein Element mit dem Wert 19.
- ☐ Das Array `array` enthält nach dem Aufruf von `f` (also am Ende von Methode `g()`) ein Element mit dem Wert 42.

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

(a) `10 / 3`: 3

(b) `1.0 * 2`: 2.0

(c) `1 + 2 + "pingu" + 1 + true`: "3pingu1true"

(d) `(1 > 0 ? ("1" + "1") : "0") + 9 + 10`: "11910"

3. Betrachten Sie folgenden Code:

```
1  class Stack<T> extends List<T> {
2    // ...
3  }
```

Welche der folgenden Aussagen sind korrekt?

- ☐ Die Modellierung eines Stacks als Liste ist gut, weil `List<T>` alle für einen Stack benötigten Methoden anbietet.
- ☐ Die Modellierung eines Stacks als Liste ist schlecht, weil man Stacks nicht auf Listen aufbauen kann.

- [x] Die Modellierung eines Stacks als Liste ist schlecht, weil Listen Operationen anbieten, die auf Stacks ungültig sind.
- [x] Die Modellierung eines Stacks als Liste ist schlecht, weil interne Implementierungsdetails des Stacks nach außen sichtbar werden.
- [x] Der Typparameter des Stacks wird hier an die Liste weitergereicht.
- [ ] Der Typparameter `T` in `List<T>` ist nicht definiert, daher kompiliert der Code nicht.

4. Welche der folgenden Aussagen zum Thema Threads sind korrekt?

- [ ] Die Nutzung von mehreren Threads ist allgemein nur dann sinnvoll, wenn mehrere Prozessoren oder Prozessor-Cores zur Verfügung stehen.
- [x] Teilen sich Threads Objekte, müssen Zugriffe auf diese stets synchronisiert werden.
- [ ] Threads führen ihre `run()`-Methode wiederholt aus, bis das Programm beendet wird.
- [ ] Alle Threads verfügen über einen gemeinsamen Stack.
- [x] Alle Threads verfügen über einen gemeinsamen Heap.
- [ ] Synchronisation wird nur bei statischen Variablen benötigt, weil nur diese zwischen Threads geteilt werden können.

5. Betrachten Sie den folgenden Java-Code:

```java
class Beispiel {
  int a = 1, b = 11, c;
  Beispiel() {
    c = a + b;
  }
  public static void main(String[] args) {
    Beispiel b = new Beispiel();
    System.out.println("" + b);
  }
}
```

Was ist das Ergebnis der Ausführung des Codes?

- [x] `println("" + b)` gibt die Referenz des `Beispiel`-Objekts aus (z.B. `Beispiel@7852e922`).
- [ ] `println("" + b)` ruft die `toString()`-Methode der Klasse ~~`Example`~~ Beispiel auf und gibt `c=12` aus.
- [ ] `println("" + b)` gibt `a = 1, b = 11, c = 12` aus.
- [ ] `println("" + b)` wirft die Ausnahme `NullPointerException`.
- [ ] Das Programm kompiliert nicht.

6. Welche der folgenden Namen sind für Variablen unzulässig?

- [x] `büber`
- [x] `0unter`

- [x] %Rechts1%
- [x] Linkspinguin:innen

7. Gegeben sei folgender Code:

```java
public class Threads2 {
  static String shared = "1";
  static Runnable thread1 = new Runnable() { public void run() {
    shared += "1";
  }};
  static Runnable thread2 = new Runnable() { public void run() {
    System.out.println(shared);
  }};
  public static void main(String[] args) {
    new Thread(thread1).start();
    new Thread(thread2).start();
  }
}
```

Welche der folgenden Aussagen sind zutreffend?

- [ ] Ob eine Ausgabe erfolgt, ist abhängig von der Ausführungsreihenfolge der Threads.
- [x] Das Programm gibt entweder 1 oder 11 aus.
- [ ] Die Ausgabe erfolgt nur, wenn ein Pinguin zuschaut.
- [ ] Das Programm gibt möglicherweise 2 aus.

8. Welche der folgenden Aussagen über die Java Stream-API sind korrekt?

- [ ] Streams sind das Gleiche wie Listen.
- [ ] Streams sind das Gleiche wie Arrays.
- [x] Streams verwenden anonyme Funktionen.
- [x] Streams benötigen sehr viel Arbeitsspeicher.

Aufgabe 2 Mut zur Lücke [15 Punkte]

Vervollständigen Sie die Implementierung unten.

Die Methode `main()` wandelt eine auf der Kommandozeile übergebene Hexadezimalzahl in eine Dezimalzahl um. Eine Hexadezimalzahl wird in der Regel mit vorangestelltem `0x` dargestellt. Es gibt pro Stelle 16 mögliche Ziffern (statt 10 im Dezimal- bzw. 2 im Binärsystem), wobei die Ziffern 10 bis 15 durch die Buchstaben a bis f dargestellt werden. Ein Beispiel für eine Hexadezimalzahl ist `0xb1f`.

Eine Hexadezimalzahl lässt sich in eine Dezimalzahl umwandeln, indem man jede Ziffer mit ihrer entsprechenden *Wertigkeit* multipliziert und die Ergebnisse aufaddiert. Die i-te Ziffer von rechts (Zählung ab 0) muss dementsprechend mit 16^i multipliziert werden. Um unsere Beispielzahl `0xb1f` ins Dezimalsystem umzuwandeln, müssen wir also $15 * 16^0 + 1 * 16^1 + 11 * 16^2$ (dabei entspricht `f` der Dezimalzahl 15 und `b` der Dezimalzahl 11) berechnen, was 2847 ergibt.

Ein Aufruf plus Ausgabe des Java-Programms sieht wie folgt aus:

```
$ java Main 0xb1f
Input: b1f
Output: 2847
```

Füllen Sie die Boxen aus.

```
import java.util.Arrays;
import java.util.List;

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

  String hexNumber = args[0].replace("0x" , "");
  List<Character> chars = Arrays.asList('a', 'b',  'c', 'd', 'e', 'f' );
  int number  = 0 ;

  for (int i = hexNumber .length() - 1, j = 0; i >= 0; i--, j++) {
    char ch = Character.toLowerCase(hexNumber.charAt(i));
    int stelle = (int ) Math.pow(16 , j);  // pow() berechnet x^y
    if (Character.isDigit(ch ))
      number += stelle  * Integer.parseInt(Character.toString(ch));
    else if (chars.contains(ch ))
      number  += stelle * (number  + chars.lastIndexOf(ch));
  }

  System.out.println("Input: " + hexNumber);
  System.out.println("Output: " + number );

}
}
```

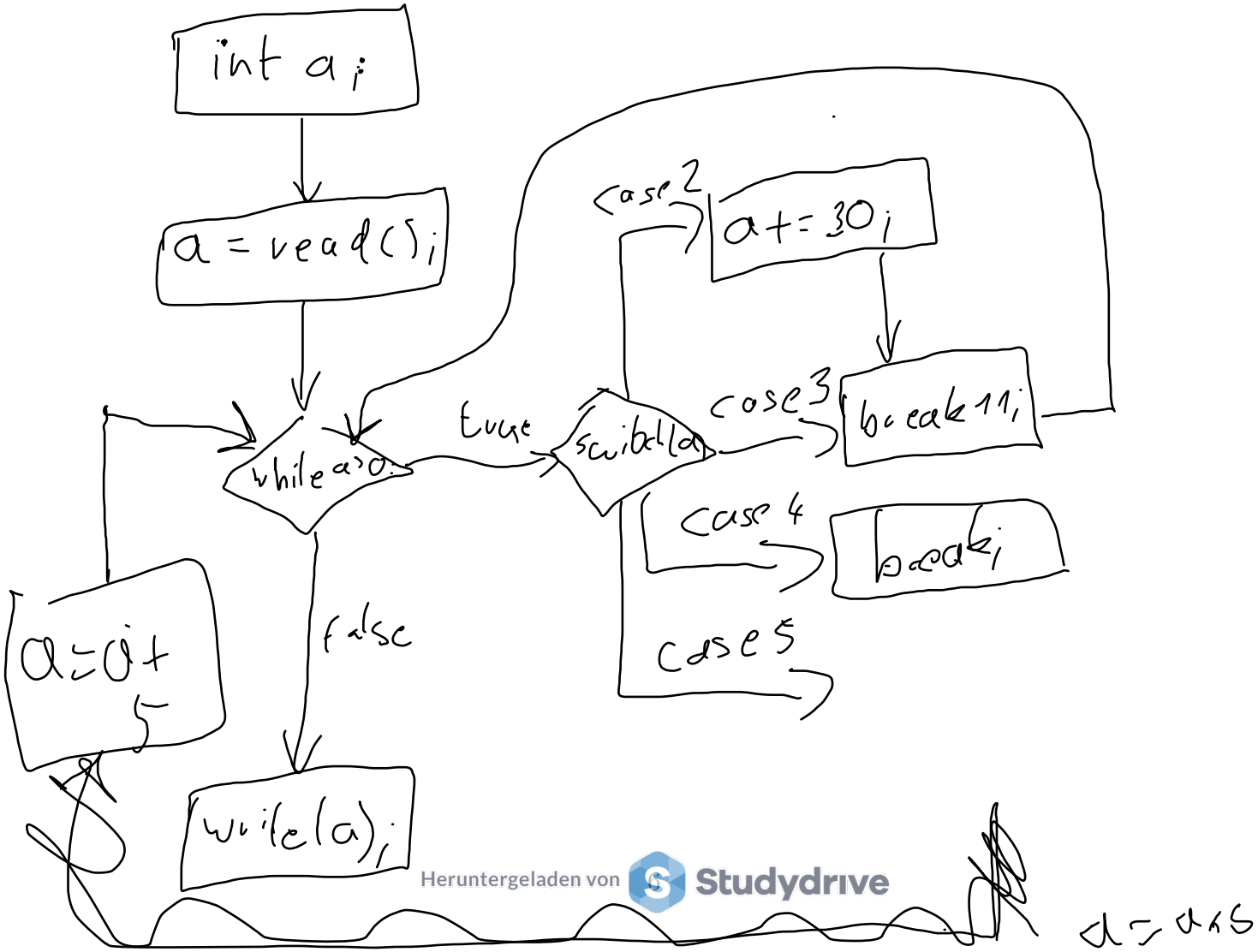
10

Aufgabe 3 Kontrollfluss [15 Punkte]

Zeichnen Sie für das folgende Programm den Kontrollflussgraphen.

```
1  int a;
2  a = read();
3  l1: while(a < 10) {
4    switch(a) {
5      case 2:
6        a += 30;
7      case 3:
8        break l1;
9      case 4:
10       break;
11     case 5:
12       continue l1;
13   }
14   a = a + 5;
15 }
16 write(a);
```

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



Aufgabe 4 Polymorphie [20 Punkte]

Betrachten Sie den folgenden Code:

```
1  class A<U extends B> {
2    U u;
3    A(U u) { this.u = u; }
4    void f(U u) { System.out.println("A.f(U)");  this.u.f(u); }
5    void g(double a) { System.out.println("A.g(double)"); }
6  }
7
8  class B extends A<B> {
9    C u;
10   B(B b, C c) { super(b); u = c; }
11   void f(B u) { System.out.println("B.f(B)"); }
12   void g(double a) { System.out.println("B.g(double)"); }
13   void g(double a, int b) { System.out.println("B.g(double, int)"); }
14   void g(int a, double b) { System.out.println("B.g(int, double)"); }
15 }
16
17 class C extends B {
18   C() { super(null, null); }
19   void g(int a) { System.out.println("C.g(int)"); }
20 }
21
22 class Poly {
23   public static void main(String[] args) {
24     A<B> ab = new A<B>(new C()); B ac = new B(null, new C());
25
26     ab.u.g(42);            // Aufruf 1 B.g(double)
27     ab.u.g(42.0);          // Aufruf 2 B.g(double)
28     ab.g(42);              // Aufruf 3 A.g(double)
29     ac.u.g(11);            // Aufruf 4 C.g(int)
30     ((A<B>) ac).g(10);     // Aufruf 5 B.g(double)
31     ab.f(ac);              // Aufruf 6 StackOverflowError
32     ab.u.g(1.0, 2);        // Aufruf 7 B.g(double, int)
33     ab.u.g(1, 2);          // Aufruf 8 AmbiguityError
34     ((A<B>)ac).g(2, 1.0); // Aufruf 9 InvalidArgumentException
35   }
36 }
```

Geben Sie für jeden der markierten Aufrufe die Ausgabe an. Sollte einer der Aufrufe nicht kompilieren oder einen Laufzeitfehler erzeugen, so erklären Sie, welcher Fehler warum auftritt. Wir betrachten jeden der Aufrufe in Isolation, betrachten Sie also jeweils alle anderen Aufrufe als auskommentiert.

Hinweis: Gehen Sie bei der Wahl einer Methode systematisch vor. Erinnern Sie sich, dass der Compiler einen passenden *Overload* alleine nach dem statischen Typen auswählt. Zur Laufzeit wird dann ggf. nach dynamischem Typen in einer Unterklasse eine überschreibende Methode gewählt, sofern ihre Signatur exakt zu der vom Compiler gewählten Methode passt.

Aufgabe 5 Lambdas und Streams [30 Punkte]

In dieser Aufgabe soll die Stream-API dazu verwendet werden, Kollektionen von Elementen zu verarbeiten. Bearbeiten Sie die folgenden Teilaufgaben.

1. Gegeben sei folgender Java-Quelltext:

```java
import java.util.Arrays;
class Main {
  public static void main(String[] args) {
    int[] numbers = new int[]{5, 4, 3, 2, 1, 0};
    // TODO
  }
}
```

   Wandeln Sie das gegebene Array mit Hilfe von `Arrays.stream()` in ein Stream-Objekt um. Verwenden Sie die `map()`, `forEach()`, `sorted()`, `filter()` und Lambda-Ausdrücke auf dem Stream, um hintereinander jede Zahl zu quadrieren, die Zahlen zu sortieren und auf der Kommandozeile auszugeben. Stellen Sie zudem sicher, dass nur Zahlen größer oder gleich 10 ausgegeben werden.

2. Gegeben sei eine Klasse zur Darstellung von Punkten in einem zweidimensionalen Koordinatensystem:

```java
class Point2D implements Comparable<Point2D> {
    private double x, y, d = 0;
    public Point2D(double x, double y) {this.x = x; this.y = y;}
    public double getX() {return x;}
    public double getY() {return y;}
    public double getDistance() {return d;}
    public void setDistance(double d) {this.d = d;}
    public String toString() {return "x: "+x+", y: "+y+", d: "+d;}
    public int compareTo(Point2D point) {
        if (point.getDistance() < getDistance()) return 1;
        else if (point.getDistance() > getDistance()) return -1;
        return 0;
    }
}
```

   Gegeben sei zudem eine Methode

```java
public static Stream<String> iterateLinesFromFile(File csvFile),
```

   welche einen Stream von Strings zurückgibt, wobei die Strings den Zeilen der übergebenen CSV-Datei ohne Newline-Character, d.h. ohne `'\n'`, entsprechen.

   Implementieren Sie eine `main()`-Methode in Java, welche von der Kommandozeile als erstes Argument eine Pfadangabe zu einer CSV-Datei und als zweites Argument einen sogenannten Grenzwert als `double`-Fließkommazahl übergeben bekommt. Gehen Sie davon aus, dass die übergebene CSV-Datei eine Semikolon-separierte Liste von `x;y`-Fließkommawertepaaren ohne Kopfzeile enthält, also z.B.:

```
1.1;2.2
3.41;-2.034
0;0.0
...
```

Die `main()`-Methode soll das Resultat der Methode `iterateLinesFromFile()` verwenden, um die CSV-Datei in einen Stream von `Point2D`-Objekten umzuwandeln. Die String-Methode `split(String regex)` sowie `Double.parseDouble(String s)` können dabei hilfreich sein. Der Stream soll mittels `collect(Collectors.toList())` als Liste zwischengespeichert werden.

Die Distanz `d` von allen `Point2D`-Objekten soll mittels Streams so gesetzt werden, dass sie die Manhattan-Distanz des jeweiligen Punkts zum ersten Element der Sequenz der Punkte enthält. Die Manhattan-Distanz ist wie folgt definiert:

$$d(a, b) = |a_x - b_x| + |a_y - b_y|$$

Den Betrag können Sie mittels `Math.abs()` berechnen. Sortieren Sie schließlich die Stream-Elemente mittels `sorted()` aufsteigend nach ihrer Distanz zum ersten Element.

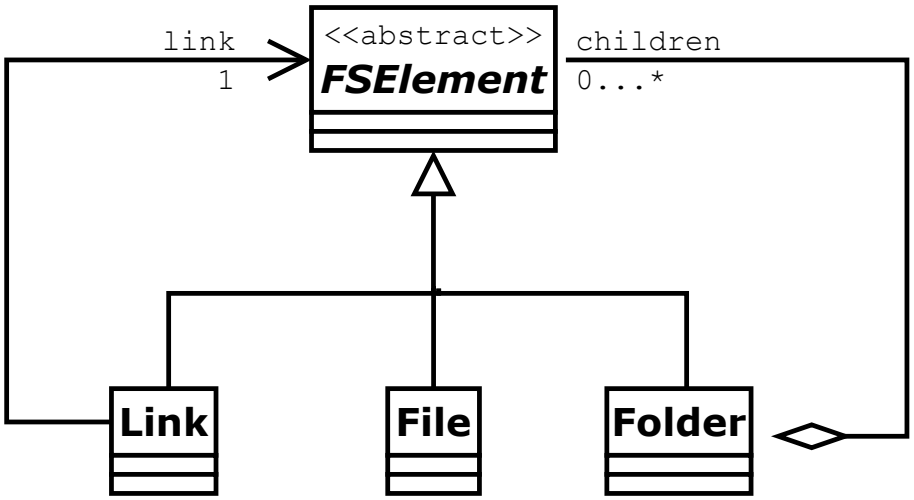
Geben Sie diejenigen Punkte mittels Stream-API auf der Kommandozeile aus, welche eine Distanz kleiner als der übergebene Grenzwert zum ersten Punkt haben.

Hinweis: Sie können davon ausgehen, dass alle ggf. notwendigen `import`-Statements bereits vorhanden sind.

Hinweis: Schleifen und Rekursion sind in der gesamten Aufgabe verboten; verwenden Sie ausschließlich die Stream-API.

Aufgabe 6 Modellierung und Rekursion [30 Punkte]

In dieser Aufgabe geht es um die Modellierung eines Dateisystems sowie der Implementierung einiger Operationen auf der resultierenden Klassenhierarchie. Dateisysteme bestehen aus Ordnern, Dateien und Links. Jedem Ordner ist bekannt, welche Elemente (`children`) er enthält; ein Ordner kann 0 oder mehr Elemente enthalten (vgl. „`0...*`“ im Klassendiagramm). Ein Link verweist auf genau einen Ordner, eine Datei oder einen anderen Link (vgl. „1“ im Klassendiagramm). Betrachten Sie dazu das folgende UML-Diagramm:



Beachten Sie, dass das Diagramm nicht vollständig ist; Sie müssen im Folgenden Methoden und Attribute hinzufügen. Sie dürfen außerdem jederzeit Hilfsmethoden und Hilfsklassen einführen.

1. Implementieren Sie die Klassenhierarchie entsprechend dem oben gezeigten UML-Diagramm. Beachten Sie, dass Datei, Ordner und Link jeweils einen Namen haben. Fügen Sie das Attribut an einer passenden Stelle in die Hierarchie ein. Der Name soll über den Konstruktor setzbar sein. Betrachten Sie als Beispiel einer Nutzung Ihrer Klassen folgenden Code, der die Dateisystem-Hierarchie aus Abbildung 1 konstruiert:

```
1  Folder home = new Folder("home", new FSElement[] {
2     new File("a"),
3     new File("b")
4  });
5  Folder etc = new Folder("etc", new FSElement[] {
6    new File("q"),
7    new Link("home", home)
8  });
9  Folder root = new Folder("root", new FSElement[] {
10   new File("test"),
11   home,
12   etc
13 });
```

   In der Abbildung werden Ordner orange, Dateien grün und Links blau dargestellt.

2. Implementieren Sie die Methode

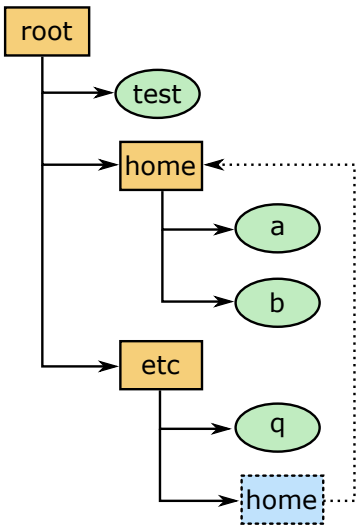


Abbildung 1: Beispiel eines Dateisystems

```
public void add(FSElement e, String path),
```

die ein Dateisystem-Element `e` in den durch `path` festgelegten Ordner in das Dateisystem einfügt. Ein Pfad besteht aus Namen von Ordnern und Links, welche durch ein `'/'`-Symbol getrennt sind. Links soll hier gefolgt werden. Um zum Beispiel die Datei *foo* in obiger Beispiel-Hierarchie in den Ordner mit dem Name *home* einzufügen, könnte man `root.add(new File("foo"), "root/home")` oder auch `root.add(new File("foo"), "root/etc/home")` aufrufen. Ihre Implementierung soll Rekursion verwenden. Ist der Pfad für die gegebene Hierarchie ungültig, so soll eine `PathException`, die von `RuntimeException` erbt (nicht von Ihnen zu implementieren), geworfen werden. **Hinweis**: Sie können vereinfachend davon ausgehen, dass jeder Link den gleichen Namen hat wie das verlinkte `FSElement`. Sie können außerdem ohne Fehlerprüfung davon ausgehen, dass eine Datei nicht mehrfach in einen Ordner eingefügt wird.

3. Implementieren Sie das Interface `Iterable<FSElement>` für die Klasse `FSElement`, um Dateisystem-Elemente iterierbar zu machen. Ihr Iterator soll dabei rekursiv sämtliche Elemente unterhalb und inklusive des Wurzelelements zurückliefern. Links soll nicht gefolgt werden.

Hinweis: Nutzen Sie die Methode `split(String regex)` der Klasse String, um einen String in ein Array von Teilstrings anhand eines Trennsymbols `regex` aufzuteilen.

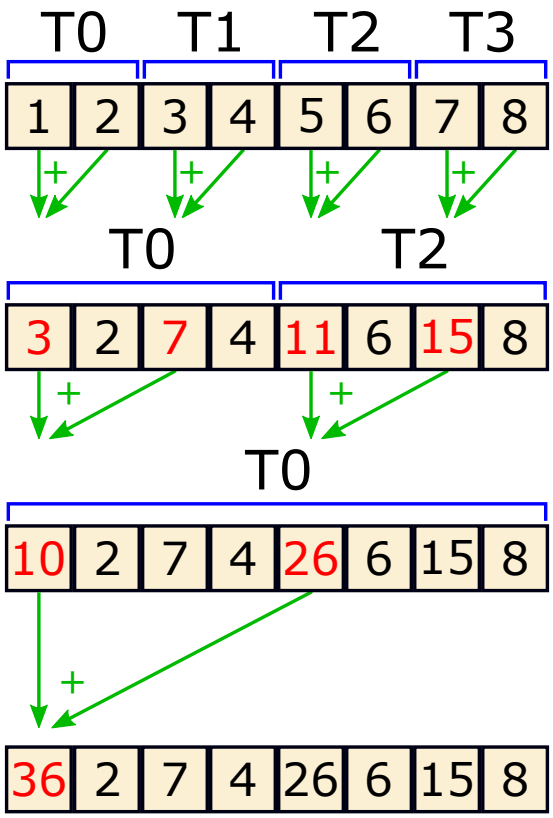
Hinweis: Die Interfaces `Iterator<T>` und `Iterable<T>` seien wie folgt definiert:

```
1 interface Iterator<T> {
2   boolean hasNext();
3   T next();
4 }
5 interface Iterable<T> {
6   Iterator<T> iterator();
7 }
```

Aufgabe 7 Threads [25 Punkte]

In dieser Aufgabe geht es darum, eine parallele Reduktion zu implementieren. Sie kennen Reduktion bereits aus den Übungen von der `reduce()`-Methode des Interfaces `Stream<T>`. Unsere Reduktion soll auf `int`-Arrays arbeiten und eine Akkumulator-Funktion f als Eingabe erwarten. Sie können davon ausgehen, dass f assoziativ und kommutativ ist (das bedeutet, dass Reihenfolge und Klammerung der Operanden keine Rolle spielen). Nehmen wir als Beispiel das Array `[1; 2; 3; 4]` und die Akkumulator-Funktion `(x, y) -> x + y`. In diesem Fall ist das Ergebnis der Reduktion gleich $(((1 + 2) + 3) + 4) = 10$.

Die Reduktion soll auf mehrere Threads verteilt werden. Wir gehen im Folgenden davon aus, dass wir mit $2^k = n$ mit $k \in \mathbb{N}$ Threads beginnen und das zu reduzierende Array eine Größe von $2 * n$ hat. Wir weisen zunächst jedem Thread einen Start-Index zu, wobei der i-te Thread den Index $2 * i$ erhält. Auf diese Weise werden jedem Thread genau zwei Elemente zugewiesen. Nachdem die Threads gestartet worden sind, führt jeder Thread eine Reduktion durch und legt das Ergebnis an seinem Start-Index ab. Die Anzahl der noch zu reduzierenden Elemente verringert sich dadurch um genau die Hälfte. Die Hälfte der Threads beenden sich nun. Die andere Hälfte der Threads führt eine erneute Reduktion auf den bisherigen Reduktionsergebnissen durch. Das Verfahren wird fortgesetzt, bis nur mehr ein Element als einzelnes Reduktionsergebnis vorhanden ist. Dieses Endergebnis soll sich dann an Index 0 befinden. Das folgende Bild veranschaulicht den Ablauf für die Akkumulator-Funktion `(x, y) -> x + y` und das Eingabe-Array `[1; 2; 3; 4; 5; 6; 7; 8]`:



Gegeben sei außerdem die folgende Klasse `Reducer`.

```
1 class Reducer extends Thread {
2   private int[] array;
```

```
3     private int start;
4     private Reducer[] reducers;
5     private BiFunction<Integer, Integer, Integer> accumulator;
6
7     public Reducer(int[] array, int start, Reducer[] reducers,
8         BiFunction<Integer, Integer, Integer> accumulator) {
9       this.array = array; this.start = start;
10      this.reducers = reducers; this.accumulator = accumulator;
11    }
12
13    public void run() {
14      // Todo
15    }
16
17    public static void reduce(int[] array,
18        BiFunction<Integer, Integer, Integer> accumulator)
19          throws InterruptedException {
20      Reducer[] workers;
21      // Todo
22    }
23
24    public static void main(String[] args) throws InterruptedException {
25      int[] array = {1, 2, 3, 4};
26      // Todo
27      System.out.println(array[0]); // Gibt 10 aus
28    }
29  }
```

Die Klasse `Reducer` implementiert einen Thread für unsere Reduktion. Sie erwartet im Konstruktor das Eingabe-Array, den Start-Index, ein Array aller beteiligten Threads sowie die Akkumulator-Funktion. Die Methode `reduce()` parallelisiert eine Reduktion, indem sie ein passendes Array von Threads erstellt, die Threads startet und auf Beendigung desjenigen Threads wartet, der die letzte Reduktionsoperation durchführt. Die `main()`-Methode soll die eingangs am Beispiel gezeigte Reduktion durchführen. Vervollständigen Sie die Methoden der Klasse `Reducer` entsprechend den mit *// Todo* gekennzeichneten Stellen.

Hinweis: Threads sollen so lange wie nötig, aber so kurz wie möglich auf die Beendigung anderer Threads warten. Im abgebildeten Beispiel warten zunächst der Thread `T0` auf die Beendigung des Threads `T1` und der Thread `T2` auf die Beendigung von Thread `T3`. Im zweiten Schritt wartet dann Thread `T0` auf den Thread `T2`.

Hinweis: Auf jeder Ebene muss ein noch aktiver Thread entscheiden, ob er sich beenden oder auf einen anderen Thread warten muss. Beachten Sie hierzu, dass sich die Anzahl der Threads pro Ebene halbiert. Ein Thread kann mithilfe dieser Tatsache sowie seines Start-Indexes entscheiden, ob er sich beenden oder auf einen Thread warten muss. Überlegen Sie sich ein passendes Entscheidungskriterium.

Hinweis: Das funktionale Interface `BiFunction<T, U, R>` repräsentiert binäre Funktionen. Eine binäre Funktion erwartet zwei Argumente (hier vom Typ `T` bzw. `U`) und gibt ein Ergebnis (hier vom Typ `R`) zurück. Die Methode `R apply(T, U)` wendet die Funktion auf zwei Parameter an.