Benjamin Schürmann

Auswertung vom 12.06.2012

Benjamin Schürmann

1. Übersetzen der Java-Klassen

Dieser Abschnitt enthält etwaige Fehlermeldungen oder Warnungen des Übersetzers während des Übersetzungsvorgangs. Falls Sie hier keine Meldungen finden, kann dies bedeuten, dass Ihre hochgeladene Lösung keine Java-Dateien enthält oder alle Klassen sich fehlerfrei übersetzen ließen.

```
Note: C:\Temp\TestBench\opr7\submissions\19566\PushBackFolge.java uses unchecked or
unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

Benjamin Schürmann

2. Formale Prüfung

Dieser Abschnitt enthält das Ergebnis der formalen Prüfung. Es wird geprüft, ob alle in der Aufgabe geforderten Klassen mit allen geforderten Methoden vorhanden sind.

2.1. Übersicht der Klassen und Schnittstellen

❶	❷	❸	❹	❺	❻	Klasse oder Schnittstelle	Fehlerhinweis
✔						Schnittstelle Zahlenfolge	
✔						Klasse EndlicheFolge	
✔						Klasse FibonacciFolge	
✔						Klasse Mischfolge	
✔						Klasse PushBackFolge	
✔						Klasse EindeutigeFolge	

Legende: Die Klasse oder Schnittstelle ...

❶	ist vorhanden und ohne formale Fehler.
❷	ist nicht vorhanden oder nicht compilierbar.
❸	ist vorhanden, hat aber formale Fehler. Sofern es sich um formale Fehler in Methoden handelt, finden Sie Details dazu in der Methodenübersicht.
❹	enthält Fehler bzgl. der darin definierten symbolischen Konstanten.
❺	enthält Fehler bzgl. ihrer Oberklasse.
❻	enthält Fehler bzgl. der implementierten Schnittstellen.

2.2. Übersicht der Methoden

❶	❷	❸	❹	❺	Klasse	Methode	Fehlerhinweis
✔					Zahlenfolge	public abstract int naechstes()	
✔					Zahlenfolge	public abstract boolean hatNaechstes()	
✔					EndlicheFolge	public EndlicheFolge(int[])	
✔					EndlicheFolge	public int naechstes()	
✔					EndlicheFolge	public boolean hatNaechstes()	
✔					FibonacciFolge	public FibonacciFolge()	
✔					FibonacciFolge	public int naechstes()	
✔					FibonacciFolge	public boolean hatNaechstes()	
✔					Mischfolge	public Mischfolge(Zahlenfolge, Zahlenfolge)	
✔					Mischfolge	public int naechstes()	
✔					Mischfolge	public boolean hatNaechstes()	
✔					PushBackFolge	public PushBackFolge(Zahlenfolge)	
✔					PushBackFolge	public int naechstes()	

Benjamin Schürmann

✔					PushBackFolge	public boolean hatNaechstes()	
✔					PushBackFolge	public void schreibeZurueck(int)	
✔					EindeutigeFolge	public EindeutigeFolge(Zahlenfolge)	
✔					EindeutigeFolge	public int naechstes()	
✔					EindeutigeFolge	public boolean hatNaechstes()	

Legende: Die Methode ist ...

❶	vorhanden und ohne formale Fehler.
❷	nicht vorhanden.
❸	vorhanden, hat aber nicht die geforderten Modifikatoren. Die von Ihnen deklarierten Modifikatoren stehen im Fehlerhinweis.
❹	vorhanden, hat aber nicht den geforderten Ergebnistyp. Der von Ihnen deklarierte Ergebnistyp steht im Fehlerhinweis.
❺	zusätzlich von Ihnen definiert.

Benjamin Schürmann

3. Funktionale Prüfung

Dieser Abschnitt enthält das Ergebnis der funktionalen Prüfung. Es wird geprüft, ob alle in der Aufgabe geforderten Methoden für bestimmte Testdaten die erwarteten Ergebnisse liefern. Fehlermeldungen finden Sie in diesem Abschnitt auch, wenn geforderte Klassen oder Methoden fehlen. In diesem Fall scheitert bereits der Aufruf der Methoden.

Beim Test sind keine Fehler aufgetreten.

3.1. Test der Klasse EindeutigeFolge

3.1.1. Testsituation „Leere Folge“

Die Testsituation wird in folgenden Schritten aufgebaut:

1.	Erzeuge eindeutige Folge für leere Folge.

Tests der Methode testZugriffAufLeereFolge1(FunktionstestEindeutigeFolge)

	Testfall	**Fehlerhinweis**
✔	Wende auf Zahlenfolge die Methoden hatNaechstes(), hatNaechstes() und naechstes() an.	

Tests der Methode testZugriffAufLeereFolge2(FunktionstestEindeutigeFolge)

	Testfall	**Fehlerhinweis**
✔	Wende auf Zahlenfolge 1 mal die Methode naechstes() an.	

3.1.2. Testsituation „Explizite Folge“

Die Testsituation wird in folgenden Schritten aufgebaut:

1.	Erzeuge eindeutige Folge für endliche Folge 0, 0, 0, 1, 2, 3, 3, 4, 4.

Tests der Methode testZugriffAufExpliziteFolge1(FunktionstestEindeutigeFolge)

	Testfall	**Fehlerhinweis**
✔	Wende auf Zahlenfolge die Methoden hatNaechstes(), hatNaechstes() und naechstes() an und wiederhole diese drei Aufrufe weitere 5 mal.	

Tests der Methode testZugriffAufExpliziteFolge2(FunktionstestEindeutigeFolge)

	Testfall	**Fehlerhinweis**
✔	Wende auf Zahlenfolge 6 mal die Methode naechstes() an.	

3.1.3. Testsituation „Fibonacci-Folge“

Die Testsituation wird in folgenden Schritten aufgebaut:

1.	Erzeuge eindeutige Folge für Fibonacci-Folge.

Tests der Methode testZugriffAufFibonacciFolge1(FunktionstestEindeutigeFolge)

	Testfall	Fehlerhinweis
✔	Wende auf Zahlenfolge die Methoden hatNaechstes(), hatNaechstes() und naechstes() an und wiederhole diese drei Aufrufe weitere 5 mal.	

Tests der Methode testZugriffAufFibonacciFolge2(FunktionstestEindeutigeFolge)

	Testfall	Fehlerhinweis
✔	Wende auf Zahlenfolge 6 mal die Methode naechstes() an.	

3.2. Test der Klasse EndlicheFolge

3.2.1. Testsituation „Leere Folge“

Die Testsituation wird in folgenden Schritten aufgebaut:

1.	Erzeuge leere endliche Folge.

Tests der Methode testZugriffAufLeereFolge1(FunktionstestEndlicheFolge)

	Testfall	Fehlerhinweis
✔	Wende auf Zahlenfolge die Methoden hatNaechstes(), hatNaechstes() und naechstes() an.	

Tests der Methode testZugriffAufLeereFolge2(FunktionstestEndlicheFolge)

	Testfall	Fehlerhinweis
✔	Wende auf Zahlenfolge 1 mal die Methode naechstes() an.	

3.2.2. Testsituation „Folge der Länge 4“

Die Testsituation wird in folgenden Schritten aufgebaut:

1.	Erzeuge endliche Folge 8, 5, 2, 5.

Tests der Methode testZugriffAuf4erFolge1(FunktionstestEndlicheFolge)

	Testfall	Fehlerhinweis

✔	Wende auf Zahlenfolge die Methoden hatNaechstes(), hatNaechstes() und naechstes() an und wiederhole diese drei Aufrufe weitere 4 mal.	

Tests der Methode testZugriffAuf4erFolge2(FunktionstestEndlicheFolge)

	Testfall	**Fehlerhinweis**
✔	Wende auf Zahlenfolge 5 mal die Methode naechstes() an.	

3.3. Test der Klasse FibonacciFolge

3.3.1. Testsituation „Fibonacci-Folge“

Die Testsituation wird in folgenden Schritten aufgebaut:

1.	Erzeuge Fibonacci-Folge.

Tests der Methode testZugriffAufFolge1(FunktionstestFibonacciFolge)

	Testfall	**Fehlerhinweis**
✔	Wende auf Zahlenfolge die Methoden hatNaechstes(), hatNaechstes() und naechstes() an und wiederhole diese drei Aufrufe weitere 44 mal.	

Tests der Methode testZugriffAufFolge2(FunktionstestFibonacciFolge)

	Testfall	**Fehlerhinweis**
✔	Wende auf Zahlenfolge 45 mal die Methode naechstes() an.	

3.4. Test der Klasse Mischfolge

3.4.1. Testsituation „Leere Mischfolge“

Die Testsituation wird in folgenden Schritten aufgebaut:

1.	Erzeuge Mischfolge für zwei leere endliche Folgen.

Tests der Methode testZugriffAufLeereFolge1(FunktionstestMischfolge)

	Testfall	**Fehlerhinweis**
✔	Wende auf Zahlenfolge die Methoden hatNaechstes(), hatNaechstes() und naechstes() an.	

Tests der Methode testZugriffAufLeereFolge2(FunktionstestMischfolge)

	Testfall	Fehlerhinweis
✔	Wende auf Zahlenfolge 1 mal die Methode naechstes() an.	

3.4.2. Testsituation „Leere und nicht-leere gemischt“

Die Testsituation wird in folgenden Schritten aufgebaut:

1.	Erzeuge Mischfolge für leere Folge und endliche Folge 2, 5, 8, 15.

Tests der Methode testZugriffAuf4erFolge1(FunktionstestMischfolge)

	Testfall	Fehlerhinweis
✔	Wende auf Zahlenfolge die Methoden hatNaechstes(), hatNaechstes() und naechstes() an und wiederhole diese drei Aufrufe weitere 4 mal.	

Tests der Methode testZugriffAuf4erFolge2(FunktionstestMischfolge)

	Testfall	Fehlerhinweis
✔	Wende auf Zahlenfolge 5 mal die Methode naechstes() an.	

3.4.3. Testsituation „Explizite und Fibonacci-Folge gemischt“

Die Testsituation wird in folgenden Schritten aufgebaut:

1.	Erzeuge Mischfolge für Fibonacci-Folge und endliche Folge -1, 0, 1, 5, 50.

Tests der Methode testZugriffAufFibonacciFolge1(FunktionstestMischfolge)

	Testfall	Fehlerhinweis
✔	Wende auf Zahlenfolge die Methoden hatNaechstes(), hatNaechstes() und naechstes() an und wiederhole diese drei Aufrufe weitere 17 mal.	

Tests der Methode testZugriffAufFibonacciFolge2(FunktionstestMischfolge)

	Testfall	Fehlerhinweis
✔	Wende auf Zahlenfolge 18 mal die Methode naechstes() an.	

3.5. Test der Klasse PushBackFolge

3.5.1. Testsituation „Leere Folge“

Die Testsituation wird in folgenden Schritten aufgebaut:

1.	Erzeuge PushBack-Folge für leere Folge.

Tests der Methode testZugriffAufLeereFolge1(FunktionstestPushBackFolge)

	Testfall	Fehlerhinweis
✔	Wende auf Zahlenfolge die Methoden hatNaechstes(), hatNaechstes() und naechstes() an.	

Tests der Methode testZugriffAufLeereFolge2(FunktionstestPushBackFolge)

	Testfall	Fehlerhinweis
✔	Wende auf Zahlenfolge 1 mal die Methode naechstes() an.	

3.5.2. Testsituation „Folge der Länge 4“

Die Testsituation wird in folgenden Schritten aufgebaut:

1.	Erzeuge PushBack-Folge für leere Folge.
2.	Schreibe 8 zurück.
3.	Schreibe 5 zurück.
4.	Schreibe 2 zurück.
5.	Schreibe 5 zurück.

Tests der Methode testZugriffAuf4erFolge1(FunktionstestPushBackFolge)

	Testfall	Fehlerhinweis
✔	Wende auf Zahlenfolge die Methoden hatNaechstes(), hatNaechstes() und naechstes() an und wiederhole diese drei Aufrufe weitere 4 mal.	

Tests der Methode testZugriffAuf4erFolge2(FunktionstestPushBackFolge)

	Testfall	Fehlerhinweis
✔	Wende auf Zahlenfolge 5 mal die Methode naechstes() an.	

4. Checkstyle-Prüfung

```
Starting audit...
FibonacciFolge.java:10:5: Jede Variablendeklaration muss in einer eigenen Anweisung
erfolgen.
Mischfolge.java:13:5: Jede Variablendeklaration muss in einer eigenen Anweisung
erfolgen.
Mischfolge.java:42:9: Jede Variablendeklaration muss in einer eigenen Anweisung
erfolgen.
Audit done.
```

5. Quellcode der Java-Klassen

5.1. Zahlenfolge.java

```
 1: import java.util.NoSuchElementException;
 2:
 3: /**
 4:  * Methoden die eine Zahlenfolge implementieren muss.
 5:  */
 6: public interface Zahlenfolge {
 7:
 8:     /**
 9:      * Prueft ob eine Zahlenfolge noch ein weiteres Element besitzt.
10:      * @return Liefert genau dann true, wenn noch mindestens ein Element
11:      *         vorhanden ist.
12:      */
13:     boolean hatNaechstes();
14:
15:     /**
16:      * Gibt das naechse Element zurueck.
17:      * @return naechstes Element.
18:      * @throws NoSuchElementException Kein Element mehr vorhanden.
19:      */
20:     int naechstes() throws NoSuchElementException;
21:
22: }
```

5.2. EndlicheFolge.java

```
 1: import java.util.NoSuchElementException;
 2:
 3: /**
 4:  * Diese Klasse repraesentiert eine endliche Folge, deren Werte beim Erzeugen
 5:  * explizit angegeben werden.
 6:  */
 7: public class EndlicheFolge implements Zahlenfolge {
 8:
 9:     /** interne Folge die eine endliche Folge an Zahlen speichert. */
10:     private int[] folge;
11:     /** enthaelt eine Referenz auf das momentan anktuelle Element. */
12:     private int aktuellesElement;
13:
14:     /**
15:      * Erzeugt eine neue EndlicheFolge.
16:      * @param folge Int-Array aus der die Zahlenfolge generiert wird.
17:      */
18:     public EndlicheFolge(int[] folge) {
19:         this.folge = folge;
20:         this.aktuellesElement = 0;
```

```
21:     }
22:
23:     /**
24:     * Prueft ob eine Zahlenfolge noch ein weiteres Element besitzt.
25:     * @return Liefert genau dann true, wenn noch mindestens ein Element
26:     *         vorhanden ist.
27:     */
28:     public boolean hatNaechstes() {
29:         return (this.aktuellesElement < this.folge.length);
30:     }
31:
32:     /**
33:     * Gibt das naechse Element zurueck.
34:     * @return naechstes Element.
35:     * @throws NoSuchElementException Kein Element mehr vorhanden.
36:     */
37:     public int naechstes() throws NoSuchElementException {
38:         int rueckgabeWert;
39:         try {
40:             rueckgabeWert = this.folge[aktuellesElement];
41:             aktuellesElement++;
42:         } catch (Exception ex) {
43:             throw new NoSuchElementException();
44:         }
45:         return rueckgabeWert;
46:     }
47:
48: }
```

5.3. FibonacciFolge.java

```
 1: import java.util.NoSuchElementException;
 2:
 3: /**
 4:  * Diese Klasse repraeseniert die Zahlen der Fibonacci-Folge.
 5:  */
 6: public class FibonacciFolge implements Zahlenfolge {
 7:
 8:     /** die beiden letzen Zahlen der Fibonacii-Folge um die naechste Zahl
 9:       * auszurechnen. */
10:     private int zahl0, zahl1;
11:     /** Counter nachdem der eigentliche Algorithmus startet, da die ersten
12:       * beiden Ziffern sonst nicht zu errechnen sind. */
13:     private int starteAlgorithmus;
14:
15:     /**
16:      * Erzeugt eine neue Fibonacci-Folge.
17:      */
18:     public FibonacciFolge() {
19:         this.zahl0 = 0;
```

```
20:         this.zahl1 = 1;
21:         this.starteAlgorithmus = 2;
22:     }
23:
24:     /**
25:     * Prueft ob eine Zahlenfolge noch ein weiteres Element besitzt.
26:     * @return Liefert genau dann true, wenn noch mindestens ein Element
27:     *         vorhanden ist.
28:     */
29:     public boolean hatNaechstes() {
30:         return true;
31:     }
32:
33:     /**
34:     * Gibt das naechse Element zurueck.
35:     * @return naechstes Element.
36:     * @throws NoSuchElementException Kein Element mehr vorhanden.
37:     */
38:     public int naechstes() throws NoSuchElementException {
39:         int zahl2 = 2;
40:         if (starteAlgorithmus != 0) {
41:             zahl2 -= this.starteAlgorithmus;
42:             this.starteAlgorithmus--;
43:         } else {
44:             zahl2 = this.zahl1 + this.zahl0;
45:             this.zahl0 = this.zahl1;
46:             this.zahl1 = zahl2;
47:         }
48:         return zahl2;
49:     }
50:
51: }
```

5.4. Mischfolge.java

```
 1: import java.util.NoSuchElementException;
 2:
 3: /**
 4:  * Diese Klasse repraesentiert die Folge, die aus den gemeinsamen Werten zweier
 5:  * Folgen besteht. Sind diese beiden Folgen sortiert, ist die Mischfolge
 6:  * ebenfalls sortiert. (Ansonsten kann die Mischfolge die Elemente in einer
 7:  * beliebigen Reihenfolge liefern.
 8:  */
 9: public class Mischfolge implements Zahlenfolge {
10:
11:     /** Zwei PusbBackFolgen von denen immer die graessere ausgeben werden kann
12:      *  und die kleiner zurueckgeschrieben wird. */
13:     private PushBackFolge folge0, folge1;
14:
15:     /**
```

```
16:      * Erzeugt eine neue Mischfolge aus 2 Zahlenfolgen.
17:      * @param folge0 erste Folge
18:      * @param folge1 zweite Folge
19:      */
20:     public Mischfolge(Zahlenfolge folge0, Zahlenfolge folge1) {
21:         this.folge0 = (PushBackFolge) (folge0 instanceof PushBackFolge
22:                                        ? folge0 : new PushBackFolge(folge0));
23:         this.folge1 = (PushBackFolge) (folge1 instanceof PushBackFolge
24:                                        ? folge1 : new PushBackFolge(folge1));
25:     }
26:
27:     /**
28:     * Prueft ob eine Zahlenfolge noch ein weiteres Element besitzt.
29:     * @return Liefert genau dann true, wenn noch mindestens ein Element
30:     *         vorhanden ist.
31:     */
32:     public boolean hatNaechstes() {
33:         return this.folge0.hatNaechstes() || this.folge1.hatNaechstes();
34:     }
35:
36:     /**
37:     * Gibt das naechse Element zurueck.
38:     * @return naechstes Element.
39:     * @throws NoSuchElementException Kein Element mehr vorhanden.
40:     */
41:     public int naechstes() throws NoSuchElementException {
42:         int zahl0, zahl1, rueckgabeWert;
43:         if (this.folge0.hatNaechstes() && !this.folge1.hatNaechstes()) {
44:             rueckgabeWert = this.folge0.naechstes();
45:         } else if (!this.folge0.hatNaechstes() && this.folge1.hatNaechstes()) {
46:             rueckgabeWert = this.folge1.naechstes();
47:         } else if (!this.folge0.hatNaechstes() && !this.folge1.hatNaechstes()) {
48:             throw new NoSuchElementException();
49:         } else {
50:             zahl0 = this.folge0.naechstes();
51:             zahl1 = this.folge1.naechstes();
52:             if (zahl0 < zahl1) {
53:                 rueckgabeWert = zahl0;
54:                 this.folge1.schreibeZurueck(zahl1);
55:             } else {
56:                 rueckgabeWert = zahl1;
57:                 this.folge0.schreibeZurueck(zahl0);
58:             }
59:         }
60:         return rueckgabeWert;
61:     }
62:
63: }
```

5.5. PushBackFolge.java

```
 1: import java.util.LinkedList;
 2: import java.util.List;
 3: import java.util.NoSuchElementException;
 4:
 5: /**
 6:  * Diese Klasse basiert auf einer Zahlenfolge und ergaenzt diese um die
 7:  * Faehigkeit, Werte „zurueckzuschreiben“.
 8:  */
 9: public class PushBackFolge implements Zahlenfolge {
10:
11:     /** List in die die Zahlen zurueckgeschrieben werden koennen. */
12:     private List<Integer> list;
13:     /** Zahlenfolge auf die zugegriffen wird. */
14:     private Zahlenfolge folge;
15:
16:     /** erzeugt eine neue PushBackFolge aus einer Zahlenfolge.
17:      * @param folge Zahlenfolge.
18:      */
19:     public PushBackFolge(Zahlenfolge folge) {
20:         this.folge = folge;
21:         this.list = new LinkedList();
22:     }
23:
24:     /**
25:     * Prueft ob eine Zahlenfolge noch ein weiteres Element besitzt.
26:     * @return Liefert genau dann true, wenn noch mindestens ein Element
27:     *         vorhanden ist.
28:     */
29:     public boolean hatNaechstes() {
30:         return !this.list.isEmpty() || this.folge.hatNaechstes();
31:     }
32:
33:     /**
34:     * Gibt das naechse Element zurueck.
35:     * @return naechstes Element.
36:     * @throws NoSuchElementException Kein Element mehr vorhanden.
37:     */
38:     public int naechstes() throws NoSuchElementException {
39:         int ruegabeWert;
40:         if (!this.list.isEmpty()) {
41:             ruegabeWert = this.list.remove(0);
42:         } else {
43:             ruegabeWert = this.folge.naechstes();
44:         }
45:         return ruegabeWert;
46:     }
47:
```

```
48:     /**
49:      * Schreibt einen Zahlenwert zurueck in die Folge.
50:      * @param wert Wert der zurueckgeschrieben werden soll.
51:      */
52:     public void schreibeZurueck(int wert) {
53:         this.list.add(wert);
54:     }
55:
56: }
```

5.6. EindeutigeFolge.java

```
 1: import java.util.NoSuchElementException;
 2:
 3: /**
 4:  * Diese Klasse basiert auf einer Zahlenfolge und repraesentiert deren Werte
 5:  * ohne doppelte Elemente. Es wird davon ausgegangen, dass die uebergebene Folge
 6:  * sortiert ist.
 7:  */
 8: public class EindeutigeFolge implements Zahlenfolge {
 9:
10:     /** interngespeicherte Folge mit PushBack-Methoden. */
11:     private PushBackFolge folge;
12:
13:     /**
14:      * Erzeugt eine Eindeutige Folge aus der uebergebenen Zahlenfolge.
15:      * @param folge Zahlenfolge.
16:      */
17:     public EindeutigeFolge(Zahlenfolge folge) {
18:         this.folge = (PushBackFolge) (folge instanceof PushBackFolge
19:                                       ? folge : new PushBackFolge(folge));
20:     }
21:
22:     /**
23:      * Prueft ob eine Zahlenfolge noch ein weiteres Element besitzt.
24:      * @return Liefert genau dann true, wenn noch mindestens ein Element
25:      *         vorhanden ist.
26:      */
27:     public boolean hatNaechstes() {
28:         return this.folge.hatNaechstes();
29:     }
30:
31:     /**
32:      * Gibt das naechse Element zurueck.
33:      * @return naechstes Element.
34:      * @throws NoSuchElementException Kein Element mehr vorhanden.
35:      */
36:     public int naechstes() throws NoSuchElementException {
37:         int rueckgabeWert;
38:         int nachestesElement;
```

```
39:
40:         if (this.folge.hatNaechstes()) {
41:             rueckgabeWert = this.folge.naechstes();
42:             nachestesElement = rueckgabeWert;
43:             while (this.folge.hatNaechstes()
44:                    && (nachestesElement == rueckgabeWert)) {
45:                 nachestesElement = this.folge.naechstes();
46:             }
47:             if (nachestesElement != rueckgabeWert) {
48:                 this.folge.schreibeZurueck(nachestesElement);
49:             }
50:         } else {
51:             throw new NoSuchElementException();
52:         }
53:         return rueckgabeWert;
54:     }
55:
56: }
```