P. Thiemann, L. Fennell Sommersemester 2017

Programmieren in Java

`http://proglang.informatik.uni-freiburg.de/teaching/java/2017/`

rainfall

Niederschlagsmessungen

Woche 10 Aufgabe 1/4

Herausgabe: 2017-07-04
Abgabe: 2017-07-22

Achtung: beachten Sie unbedingt die allgemeinen Hinweise zur Abgabe auf der Homepage.

Project `rainfall`
Package `rainfall`
Klassen

Main
`public static List< Optional<Double> > rainfall(List<Double> readings)`

Die Funktion `rainfall` berechnet durchschnittliche Niederschlagsmengen aus einer Liste `readings`. Die Liste `readings` enthält Serien von Messergebnisse als `double`-Werte (≥ 0). Eine Serie wird durch den speziellen Markierungswert -999 abgeschlossen. Wenn in den Serien andere negative Werte auftauchen, so sind das Messfehler; diese sollen ignoriert werden. Messwerte, die nach der letzten Serie auftreten, sollen ebenfalls verworfen werden.

Das Ergebnis von `rainfall` ist eine Liste von `Optional<Double>` Werten, die für jede Serie in `readings` den Durchschnitt angeben, sofern dieser existiert. Die Klasse `java.util.Optional` ist in der Java-API Dokumentation beschrieben.

`https://docs.oracle.com/javase/8/docs/api/java/util/Optional.html`

Sie wird verwendet wenn, wie hier, eine Funktion oder Methode nicht immer Ergebnisse liefern kann, aber auch nicht gleich mit einer `Exception` abbrechen soll. Die Klasse `Optional` enthält dazu zwei Factorymethoden:

1. `Optional.of` wird verwendet, wenn ein Ergebnis berechnet wurde. Das Ergebnis kann dann mit `Optional.get` abgerufen werden.

2. `Optional.empty` wird verwendet, wenn kein Ergebnis berechnet wurde. `Optional.get` wirft in diesem Fall eine `NoSuchElementException`.

Ob ein `Optional` Wert ein Ergebnis enthält, kann mit `Optional.isPresent` geprüft werden.

Zu dieser Aufgabe gibt es keine weiteren Hinweise.

Beispieltests

```
package rainfall;

import org.junit.Test;

import java.util.Arrays;
import java.util.List;
import java.util.Optional;

import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertTrue;

public class ExampleTests {

    @Test
    public void rainfallTest1() {
        List<Optional<Double>> readings =
                Main.rainfall(Arrays.asList(
                        3.0, 0.0, -1.0, 7.0, -999.0, -999.0, 5.0,
                        3.0, -999.0, -1.0));
        List<Optional<Double>> expected = Arrays.asList(
                Optional.of(3.333), Optional.empty(), Optional.of(4.0));
        assertRainfallEquals(expected, readings, 0.001);
    }

    public static void assertRainfallEquals(List<Optional<Double>> expected,
                                            List<Optional<Double>> actual,
                                            double delta) {
        String listMessage = " Expected: " + expected + "\n Actual: " + actual;
        assertTrue("Sizes differ. \n" + listMessage,
                   expected.size() == actual.size());
        for (int i = 0; i < expected.size(); i++) {
            String message = "Difference at index " + i + "\n" + listMessage;
            Optional<Double> expectedElem = expected.get(i);
            Optional<Double> actualElem = actual.get(i);
            assertTrue(expectedElem.isPresent() == actualElem.isPresent());
            if (actualElem.isPresent()) {
                assertEquals(message,
                             expectedElem.get(), actualElem.get(), delta);
            }
        }
    }


}
```

P. Thiemann, L. Fennell Sommersemester 2017

Programmieren in Java

`http://proglang.informatik.uni-freiburg.de/teaching/java/2017/`

sessions2

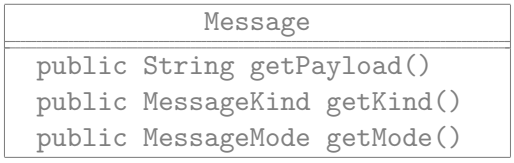
Kommunikationsprotokolle (Teil 2)

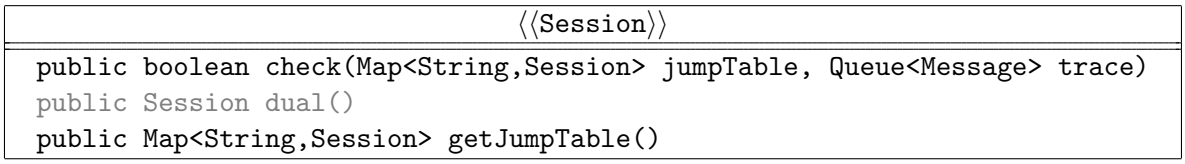
Woche 10 Aufgabe 2/4

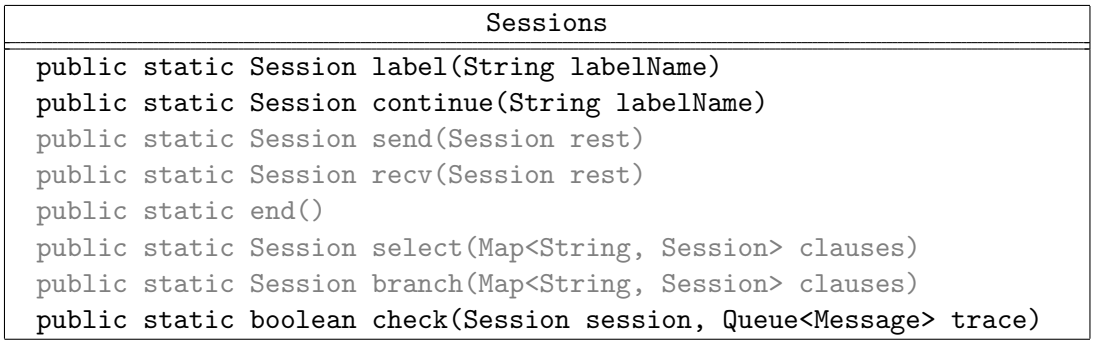
Herausgabe: 2017-07-04
Abgabe: 2017-07-22

Achtung: beachten Sie unbedingt die allgemeinen Hinweise zur Abgabe auf der Homepage.

Project `sessions2`
Package `sessions2`
Klassen







Dies ist der zweite Teil der Aufgabenreihe `sessions`. Ihre Aufgabe ist es hier, das `Session` Interface aus `w09/sessions` durch zwei weitere Klassen zu erweitern, so dass auch Protokolle mit Sprüngen dargestellt werden können.

Um Sprünge in Protokollen zu verstehen, betrachten wir nochmals das Beispiel des Bookshop-Kommunikationsprotokolls aus `w09/sessions`:

start ***Sende*** eines der folgenden Kommandos: "ADD_ITEM", "CHECKOUT"

1. Hast Du "ADD_ITEM" ***gesendet***, ***sende*** den Namen des Buches, das in den Einkaufskorb soll. ***Weiter*** bei **start**.
2. Hast Du "CHECKOUT" ***gesendet***, ***sende*** erst Deine Kreditkartennummer und danach Deine Adresse. Anschließend ***beende*** die Verbindung.

Das Protokoll hat einen Sprung, was sich in der Formulierung „Weiter bei ..." äußert. Dieser Sprung hat einen *Eintrittspunkt* oder auch **Label**, der hier durch **start** gekennzeichnet ist. Die Protokoll-Anweisung „Weiter bei *label*" ist eine *Sprunganweisung*, die das Protokoll (bzw die Prüfung eines Traces gegen das Protokoll) beim Eintrittspunkt *label* fortfahren lässt.

Um solche Sprünge zu implementieren sollen zwei neue Klassen dem `Session`-Interface hinzugefügt werden: `Label` für Eintrittspunkte und `ContinueWith` für Sprunganweisungen. Außerdem erhält die `Session`-Methode `check` zusätzlich zum Trace noch eine Map von `String`s zu `Session`s, die Labels auf Eintrittspunkte abbildet. Diese Map heißt *Sprungtabelle* und soll zu Beginn der Prüfung eines Trace gegen ein Protokoll einmalig berechnet werden. Dazu fügen wir dem `Session`-Interface die Methode `getJumpTable` hinzu.

Die neuen Fabrikmethoden `Sessions.label` und `Sessions.continueWith` konstruieren neue `Label` und `ContinueWith` Instanzen.

1. `label(String labelName, Session body)` kennzeichnet einen Eintrittpunkt. Das Vorkommen eines `Labels` in einer Session ermöglicht `ContinueWith` an diese Stelle zu springen. Eine `Label`-Session akzeptiert einen Trace, wenn ihr `body` den Trace akzeptiert. `Label` entfernt keine `Message`s aus dem Trace.

2. `continueWith(String label)` akzeptiert dann, wenn der Eintrag für `label` in der Sprungtabelle akzeptiert. `ContinueWith` entfernt keine `Message`s aus dem Trace.

Die Methode `getJumpTable` soll die Sprungtabelle einer Session berechnen. Die Schlüssel der Tabelle ergeben sich aus den Namen der `Labels` in der Session, und die dazugehörigen `Session`-Werte aus den Bodies dieser `Label`. Die Sprungtabelle soll eindeutig sein: Gibt es in der `Session` mehrere Labels mit gleichem Labelnamen wirft `getJumpTable` eine `IllegalArgumentException`.

Mit diesen Erweiterungen können wir das Bookshop-Protokoll als `Session` darstellen:

```
1   /**
2    * Create a bookshop session for the client.
3    */
4   Session bookShopClient() {
5       // construct the select map
6       Map<String,Session> selectMap = new HashMap<>();
7       // in case ADD_ITEM we send the book title and continue at the label "start"
8       selectMap.put("ADD_ITEM", Sessions.send(Sessions.continueWith("start")));
9       // in case CHECKOUT we send the credit card number, send the
10      // address, and then end the session
11      selectMap.put("CHECKOUT", Sessions.send(Sessions.send(Sessions.end())));
12
13      // the entire session is an entry point "start" that contains the "select".
14      return Sessions.label("start", Sessions.select(selectMap));
15  }
```

Ihr Auftrag: Implementieren Sie die oben beschriebenen Erweiterungen. Dies bedeutet:

1. Fügen Sie die Methode `getJumpTable` zum `Session`-Interface hinzu. Implementieren sie diese für die bereits bestehenden `Session`-Klassen.
2. Fügen Sie der `check` Methode des `Session`-Interface den Parameter `jumpTable` hinzu und passen Sie die alten `Session`-Klassen entsprechend an.
3. Implementieren Sie die neuen `Session`-Klassen `Label` und `ContinueWith`, sowie die entsprechenden Fabrikmethoden in `Sessions`. Bei `ContinueWith.check` kann es passieren, dass eine Label nicht in der Sprungtabelle vorkommt; werfen Sie in diesem Fall eine `IllegalArgumentException`.
4. Implementieren Sie außerdem die statische Methode

   `Sessions.check(Session session, Queue<Message> trace)`.

   Sie soll `trace` gegen `session` mit der Sprungtabelle von `session` prüfen.

Bonusaufgabe (1P): Definieren Sie in `Sessions` ein Protokoll in der Methode

```
public static Session getLoopingProtocoll()
```

und einen zugehörigen Trace in der Methode

```
public static LinkedList<Message> getLoopingTrace()
```

so dass die beiden Methoden selber immer terminieren, aber der Aufruf

```
check(getLoopingProtocoll(), getLoopingTrace())
```

nie terminiert. (Dabei sollen Sie natürlich nicht irgendwelche Klassen „sabotieren", indem sie künstliche Endlosschleifen einbauen, o.ä.).

Zusammenfassende Hinweise:

- Während der Ausführung von `check` ändert sich die Sprungtabelle nicht; sie muss vorher berechnet werden.
- Wenn die Labelnamen in einer Session nicht eindeutig sind, wirft `getJumpTable` eine `IllegalArgumentException`.
- Wenn das Label einer Sprunganweisung in der Sprungtabelle nicht vorkommt, wirft `check` eine `IllegalArgumentException`
- Wenn ein Trace von der Bookshop-Session aus `w09/sessions` akzeptiert wurde, so wird er auch von der oben gezeigten Bookshop-Session mit Sprüngen akzeptiert.
- Die Sprungtabelle ist „global"; jeder Eintrittspunkt ist prinzipiell von jedem Punkt durch eine entsprechende Sprunganweisung erreichbar. Insbesondere kann man in einem Protokoll „nach vorne" springen, oder in den anderen Arm einer Verzweigungen springen.

- Hier ist noch ein Beispiel für eine erweiterte Bookshop-Session mit verschachtelten Sprüngen:

```
1   /**
2    * Create an extended bookshop session for the client. It extends
3    * bookShopClient with the possibility to correct address and creditcard
4    * number: just before ending the session, we can send the Commands CONFIRM or GO_BAC
5    * - in case of CONFIRM the session ends
6    * - in case of GO_BACK_TO_CHECKOUT, we (basically) continue with CHECKOUT
7    */
8   Session extendedBookShopClient() {
9       // construct the select map
10      Map<String,Session> selectMap = new HashMap<>();
11      // in case ADD_ITEM we send the book title and continue at the label "start"
12      selectMap.put("ADD_ITEM", Sessions.send(Sessions.continueWith("start")));
13      // in case CHECKOUT we implement the extended functionality.
14      Map<String,Session> confirmSelectMap = new HashMap<>();
15      // end on CONFIRM
16      confirmSelectMap.put("CONFIRM", Sessions.end());
17      // go to "checkout" on GO_BACK_TO_CHECKOUT
18      confirmSelectMap.put("GO_BACK_TO_CHECKOUT", Sessions.continueWith("checkout"));
19      Session checkoutBody =
20              Sessions.label("checkout",
21                      // send twice, like before, then select a confirmation command
22                      Sessions.send(
23                              Sessions.send(
24                                      Sessions.select(confirmSelectMap))));
25      selectMap.put("CHECKOUT", checkoutBody);
26
27      // the entire session is an entry point "start" that contains the "select".
28      return Sessions.label("start", Sessions.select(selectMap));
29
30  }
```

Beispieltests:

```
1  package sessions2;
2
3  import org.junit.Test;
4
5  import java.util.*;
6
7  import static org.junit.Assert.*;
8
9  public class ExampleTests {
10     /**
11      * Create a bookshop session for the client.
12      */
13     Session bookShopClient() {
14         // construct the select map
15         Map<String,Session> selectMap = new HashMap<>();
16         // in case ADD_ITEM we send the book title and continue at the label "start"
17         selectMap.put("ADD_ITEM", Sessions.send(Sessions.continueWith("start")));
18         // in case CHECKOUT we send the credit card number, send the
19         // address, and then end the session
20         selectMap.put("CHECKOUT", Sessions.send(Sessions.send(Sessions.end())));
21
22         // the entire session is an entry point "start" that contains the "select".
23         return Sessions.label("start", Sessions.select(selectMap));
24     }
25
26     /**
27      * Create an extended bookshop session for the client. It extends
28      * bookShopClient with the possibility to correct address and creditcard
29      * number: just before ending the session, we can send the Commands CONFIRM or GO_BACK_TO
30      * - in case of CONFIRM the session ends
31      * - in case of GO_BACK_TO_CHECKOUT, we (basically) continue with CHECKOUT
32      */
33     Session extendedBookShopClient() {
34         // construct the select map
35         Map<String,Session> selectMap = new HashMap<>();
36         // in case ADD_ITEM we send the book title and continue at the label "start"
37         selectMap.put("ADD_ITEM", Sessions.send(Sessions.continueWith("start")));
38         // in case CHECKOUT we implement the extended functionality.
39         Map<String,Session> confirmSelectMap = new HashMap<>();
40         // end on CONFIRM
41         confirmSelectMap.put("CONFIRM", Sessions.end());
42         // go to "checkout" on GO_BACK_TO_CHECKOUT
43         confirmSelectMap.put("GO_BACK_TO_CHECKOUT", Sessions.continueWith("checkout"));
44         Session checkoutBody =
45                 Sessions.label("checkout",
```

```
46                                  // send twice, like before, then select a confirmation command
47                                  Sessions.send(
48                                          Sessions.send(
49                                                  Sessions.select(confirmSelectMap))));
50          selectMap.put("CHECKOUT", checkoutBody);
51
52          // the entire session is an entry point "start" that contains the "select".
53          return Sessions.label("start", Sessions.select(selectMap));
54
55      }
56
57      /**
58       * Test Session for buying at most one book against a trace buying 0
59       * book, from the view of the client.
60       *
61       * The helper functions newAddBooksTrace, addCheckoutToBooks and
62       * assertTrace are defined below.
63       */
64      @Test
65      public void bookShopClient1Trace0() {
66          Session bookShopClientSession = bookShopClient();
67
68          Queue<Message> trace = newAddBooksTrace(
69                  Collections.emptyList());
70          assertTrace(false, trace, bookShopClientSession);
71
72          trace = newAddBooksTrace(Collections.singletonList("Game of Thrones"));
73          addCheckoutToBooks(trace);
74          assertTrace(true, trace, bookShopClientSession);
75      }
76
77      /**
78       * Test Session for buying at most one book against a trace buying 1
79       * book, from the view of the client.
80       */
81      @Test
82      public void bookShopClient1Trace1() {
83          Session bookShopClientSession = bookShopClient();
84
85          Queue<Message> trace = newAddBooksTrace(
86                  Collections.singletonList("Game of Thrones"));
87          assertTrace(false, trace, bookShopClientSession);
88
89          trace = newAddBooksTrace(Collections.singletonList("Game of Thrones"));
90          addCheckoutToBooks(trace);
91          assertTrace(true, trace, bookShopClientSession);
92      }
```

```
93
94      /**
95       * Test Session for buying at most one book against a trace buying 1
96       * book, from the view of the server.
97       */
98      @Test
99      public void bookShopServer1Trace1() {
100         Session bookShopServerSession = bookShopClient().dual();
101
102         Queue<Message> trace = newAddBooksTrace(
103                 Collections.singletonList("Game of Thrones"),
104                 MessageMode.RECEIVE);
105         assertTrace(false, trace, bookShopServerSession);
106
107         trace = newAddBooksTrace(Collections.singletonList("Game of Thrones"),
108                                  MessageMode.RECEIVE);
109         addCheckoutToBooks(trace, MessageMode.RECEIVE);
110         assertTrace(true, trace, bookShopServerSession);
111     }
112
113     /**
114      * Test Session for buying a number of books.
115      */
116     @Test
117     public void bookShopClient2Trace1() {
118         Session bookShopClientSession = bookShopClient();
119         int n = 11; // number of books
120         List<String> someBookTitles = new ArrayList<>();
121         for (int i = 0; i < n; i++) {
122             someBookTitles.add("Book #" + i);
123         }
124         Queue<Message> trace = newAddBooksTrace(someBookTitles);
125         addCheckoutToBooks(trace);
126         assertTrace(true, trace, bookShopClientSession);
127     }
128
129     /**
130      * Session for 2 items against trace with two items
131      */
132     @Test
133     public void bookShop2ItemTrace2Item() {
134         Session bookShopClientSession = bookShopClient();
135
136         Queue<Message> trace = newAddBooksTrace(
137                 Arrays.asList("Game of Thrones",
138                               "Types and Programming Languages"));
139         assertTrace(false, trace, bookShopClientSession);
```

```
140
141         trace = newAddBooksTrace(
142                 Arrays.asList("Game of Thrones",
143                               "Types and Programming Languages"));
144         addCheckoutToBooks(trace);
145         assertTrace(true, trace, bookShopClientSession);
146     }
147
148     /**
149      * Create a message queue from a list of book titles by sending/receiving
150      * ADD_ITEM followed by the book title.
151      */
152     private Queue<Message> newAddBooksTrace(List<String> books, MessageMode mode) {
153         Queue<Message> result = new LinkedList<>();
154         for (String bookName : books) {
155             result.add(new Message("ADD_ITEM",
156                                    MessageKind.COMMAND,
157                                    mode));
158             result.add(new Message(bookName,
159                                    MessageKind.DATA,
160                                    mode));
161         }
162         return result;
163     }
164
165     private Queue<Message> newAddBooksTrace(List<String> books) {
166         return newAddBooksTrace(books, MessageMode.SEND);
167     }
168
169     /**
170      * Add CHECKOUT messages to the end of a message queue (send or receive).
171      */
172     private void addCheckoutToBooks(Queue<Message> trace, MessageMode mode) {
173         trace.add(new Message("CHECKOUT", MessageKind.COMMAND, mode));
174         trace.add(new Message("12345678", MessageKind.DATA, mode));
175         trace.add(new Message("Georges Koehler Allee 79",
176                               MessageKind.DATA,
177                               mode));
178     }
179     private void addCheckoutToBooks(Queue<Message> trace) {
180         addCheckoutToBooks(trace, MessageMode.SEND);
181     }
182
183     /**
184      * Assert that "session.check(trace)" == "shouldAccept" with informative
185      * error message.
186      */
```

```
187     private void assertTrace(boolean shouldAccept,
188                              Queue<Message> trace,
189                              Session session) {
190         String msg =
191                 "Trace: " + trace.toString() + "\n"
192                 + "Session: " + session.toString() + "\n";
193         if (shouldAccept) {
194             assertTrue(msg, Sessions.check(session, trace));
195         } else {
196             assertFalse(msg, Sessions.check(session, trace));
197         }
198     }
199
200 }
```

P. Thiemann, L. Fennell Sommersemester 2017

Programmieren in Java

`http://proglang.informatik.uni-freiburg.de/teaching/java/2017/`

shapes-error

Fehlersuche: ***`shapes`***

Woche 10 Aufgabe 3/4

Herausgabe: 2017-07-04
Abgabe: 2017-07-22

Achtung: beachten Sie unbedingt die allgemeinen Hinweise zur Abgabe auf der Homepage.

Project	`shapes-error`
Package	`shapeserror`
Klassen	

Test
`@Test public void bad()`
`@Test public void good()`

Im Skelett zu dieser Aufgabe finden Sie Klassen wie sie ähnlich in der Aufgabe `w07/shapes` vorkamen. Eine Klasse entspricht dabei nicht ihrer Spezifikation.

Schreiben Sie einen Unit-Test `bad` der den Fehler aufzeigt. Korrigieren Sie außerdem den Fehler, indem Sie neue Klassen implementieren und prüfen Sie diese mit einem zu `bad` äquivalenten Test `good`. Das heißt, der Test `bad` soll in Ihrer Abgabe fehlschlagen, aber `good` soll erfolgreich sein.

Achtung: Für diese Aufgabe zählen die Tests von Jenkins **nicht**; ihr/e Tutor/in prüft die Abgabe manuell. Ist die Aufgabe komplett richtig gelöst, gibt es zwei Punkte; Abzüge liegen im Ermessen ihrer Tutorin/ihres Tutors.

P. Thiemann, L. Fennell Sommersemester 2017

Programmieren in Java

http://proglang.informatik.uni-freiburg.de/teaching/java/2017/

btree-error

Fehlersuche: `btree`

Woche 10 Aufgabe 4/4

Herausgabe: 2017-07-04
Abgabe: 2017-07-22

Achtung: beachten Sie unbedingt die allgemeinen Hinweise zur Abgabe auf der Homepage.

Project	`btree-error`
Package	`btreeerror`
Klassen	

Test
`@Test public void bad()`
`@Test public void good()`

Im Skelett zu dieser Aufgabe finden Sie Klassen wie sie ähnlich in der Vorlesung zu Binärbäumen vorkamen. Eine Klasse entspricht dabei nicht ihrer Spezifikation.

Schreiben Sie einen Unit-Test `bad` der den Fehler aufzeigt. Korrigieren Sie außerdem den Fehler, indem Sie neue Klassen implementieren und prüfen Sie diese mit einem zu `bad` äquivalenten Test `good`. Das heißt, der Test `bad` soll in Ihrer Abgabe fehlschlagen, aber `good` soll erfolgreich sein.

Achtung: Für diese Aufgabe zählen die Tests von Jenkins **nicht**; ihr/e Tutor/in prüft die Abgabe manuell. Ist die Aufgabe komplett richtig gelöst, gibt es zwei Punkte; Abzüge liegen im Ermessen ihrer Tutorin/ihres Tutors.