Check und Prepare

Lösungsvorschläge

TECHNISCHE UNIVERSITÄT DARMSTADT

Autoren: Nhan Huynh und Darya Nikitina
Fachbereich: Informatik
Übungsblatt: 07
Version: 20. November 2021
Semesterübergreifend

V1 Theoriefragen

★★★

V1.1 Grundlegendes

1. Wie hängen die Begriffe `throws` und `throws` zusammen? Wo wird was verwendet?
2. Ist es sinnvoll, eigene Exceptionklassen zu definieren? Welche Vorteile ergeben sich hieraus?
3. Nennen Sie die Methoden der `Assert`-Klasse, die Sie zum Testen bei einem typischen JUnit Testcase in der Vorlesung kennengelernt haben, und beschreiben Sie kurz deren Verwendung.

Lösungsvorschlag:

1. `throws` wird im Methodenkopf verwendet, um zu signalisieren, dass eine Methode eine Exception dieser Klasse wirft.
   `throws` hingegen wird in der Methode verwendet, um eine Exception mit `throw new Exception()` zu werfen (dabei kann statt Exception auch eine von Exception abgeleitete Klasse stehen). Falls eine `Exception` geworfen wird, so wird die momentane Ausführung der Methode unterbrochen. `Exception` bieten die Möglichkeit, wenn etwas Ungewöhnliches im Programm geschieht, diese Tatsache von der Hauptlogik eines Programms zu trennen. Durch eine `Exception` kann dadurch die Ursache des Fehlers vor allem schneller gefunden werden.
2. Bei eigenen Exception-Klassen können Informationen an den Konstruktor übergeben werden, was die Fehlerbehandlung erleichtern kann und zum Beispiel für eine detaillierte Ausgabe der Probleme sorgt.

   Ein weiterer Grund für eigene Exception-Klassen ist, dass der Name damit häufig schon die wesentliche Information enthalten kann, z.B. der Name `ArrayIndexOutOfBoundsException` sagt zusammen mit dem Call-Stack eigentlich schon alles, was man über den Fehler wissen muss.
3. - `assertEquals`(erwarteter Wert, tatsächlicher Wert, (ggf. Abweichung)):
     Die Methode liefert einen Fehler, wenn sich die beiden Werte unterscheiden (bzw. wenn sie sich um mehr als die Abweichung unterscheiden)
   - `assertTrue`(Prädikat/boolescher Wert):
     Die Methode liefert einen Fehler, wenn false zurückliefert wird

- `assertThrows`(Exceptionname.class, Executable Lambda-Ausdruck):
  Die Methode liefert einen Fehler, wenn die Exception nicht geworfen wird

  Dazu ein paar Hintergrundinformationen zu Executable. Einmal das Interface von Executable:

```
1 public interface Executable {
2
3   void execute() throws Throwable;
4 }
```

  - Jetzt noch der formale Aufbau, wie man einen Executable Lambda-Ausdruck definieren könnte:

```
Executable example = ()-> {Operationen;};
```

Information: Mehr Informationen zu `Assertions` finden Sie unter dem folgenden Verweis:

https://junit.org/junit5/docs/5.0.1/api/org/junit/jupiter/api/Assertions.html

V1.2 Wahr oder falsch?

Welche der folgenden Aussagen ist wahr?

(A) Auf einen `try`-Block muss immer mindestens ein `catch`-Block folgen.

(B) Wenn Sie eine Methode schreiben, die eine Exception auslösen könnte, müssen Sie diesen riskanten Code mit einem `try/catch`-Block umgeben.

(C) Auf einen `try`-Block können beliebig viele verschiedene `catch`-Blöcke folgen.

(D) Eine Methode kann nur eine einzige Art von Exception werfen.

(E) Die Reihenfolge der `catch`-Blöcke ist grundsätzlich gleich gültig.

(F) Laufzeit (Runtime)-Exceptions müssen gefangen oder deklariert werden.

(G) Es darf kein Code zwischen dem `try`-Block und dem `catch`-Block geschrieben werden.

(H) Eine Methode wirft eine Exception mit dem Schlüsselwort `throws`.

Lösungsvorschlag:

(A) Ein `try`-Block kommt nie allein, sondern immer mit einem oder manchmal mehreren `catch`-Blöcken, die unmittelbar danach kommen. Die einzigen Ausnahmen sind, wenn nach einem `try`-Block ein finally-Block[1] folgt oder es sich um einen try-with-resources-Block handelt.

(B) Der riskante Code muss nicht von einem `try/catch`-Block umgeben werden, weil Exceptions auch weitergerreicht werden können. Eine weitere Ausnahme sind Runtime Exception oder die abgeleiteten Klassen, weil sie auch ohne einen `try/catch`-Block geworfen werden.

(C) Ja, solange zwischen denen kein Code steht.

(D) Eine Methode kann auch mehrere Arten von Exception werfen. Diese werden durch einen Komma getrennt.

[1] Das Schlüsselwort wurde nicht in der Vorlesung behandelt und es wird abgeraten es zu verwenden

(E) Beim Fangen mehrerer Exception sollte die Basisklasse immer zuletzt gefangen werden und eine allgemeinere Exception sollte immer nach spezifischeren Exception gefangen werden, denn die spezifischeren Exception sind von der Basisklasse abgeleitet. Würde man zuerst eine allgemeine(-re) Exception fangen, dann würde zuerst diese allgemeinere Exception gefangen werden, denn es wird immer der erste `catch`-Block ausgeführt, der zu der geworfenen Exception passt.

In anderen Worten: Die `catch`-Blöcke werden von oben nach unten durchlaufen, bis einer passt, d.h. wenn der dynamische Typ des geworfenen Exception-Objekts gleich oder Subtyp des statischen Typs im `catch`-Block ist.

(F) Müsste man all diese Exception fangen, so wäre der komplette Code mit `try`-Blöcken versehen werden und der Code wäre nicht mehr zu lesen. Deshalb müssen sie nicht gefangen oder geworfen werden und werden automatisch geworfen.

(G) Zwischen einem `try`- und `catch`-Block können Whitespaces stehen.

(H) Das Schlüsselwort `throws` steht im Methodenkopf und sagt aus, dass die Methode eine Exception werfen kann. Mit dem Schlüsselwort `throws` wird eine Exception geworfen.

V2 Try/Catch-Block

Was ist das Problem mit dem folgenden Codeausschnitt?

```
1 public static void main(String[] args) {
2   int[] arr = new int[10];
3   System.out.println(arr[77]);
4 }
```

Modifizieren Sie den Code mittels `try/catch`-Blockes um das Problem zu beheben. Im `catch`-Block soll die Fehlerbotschaft mit der Methode `System.out .print()` auf der Konsole ausgegeben werden.

Lösungsvorschlag:

Da das Array eine Größe von 10 (Index 0 bis 9) hat und man auf dem Index 77 des Arrays zugreifen möchte, wird eine `ArrayIndexOutOfBoundsException` geworfen.

```
1 public static void main(String[] args) {
2   int[] arr = new int[10];
3   try {
4   System.out.println(arr[77]);
5   } catch (ArrayIndexOutOfBoundsException e) {
6     System.out.print(e.getMessage());
7   }
8 }
```

Übungsblatt 07: Lösungsvorschläge von Nhan Huynh und Darya Nikitina

V3 Exceptions

Sehen Sie sich den folgenden Code genau an (`ExplodeException` erbt von `Exception`).

(1) Welche Ausgabe wird dieses Programm beim Aufruf der Methode `test()` liefern?

(2) Welche Ausgabe erfolgt bei einer Änderung von Zeile 2 in `String test = “yes“;`

```
1  public void test() {
2    String test = "no";
3    try {
4      doRisky(test);
5    } catch (ExplodeException ex) {
6      System.out.println("catching ExplodeException!");
7    }
8  }
9
10 public void doRisky(String test) throws ExplodeException {
11   System.out.println("begin doRisky");
12   if (test.equals("yes")) {
13     throw new ExplodeException();
14   }
15   System.out.println("end doRisky");
16   return;
17 }
```

Lösungsvorschlag:

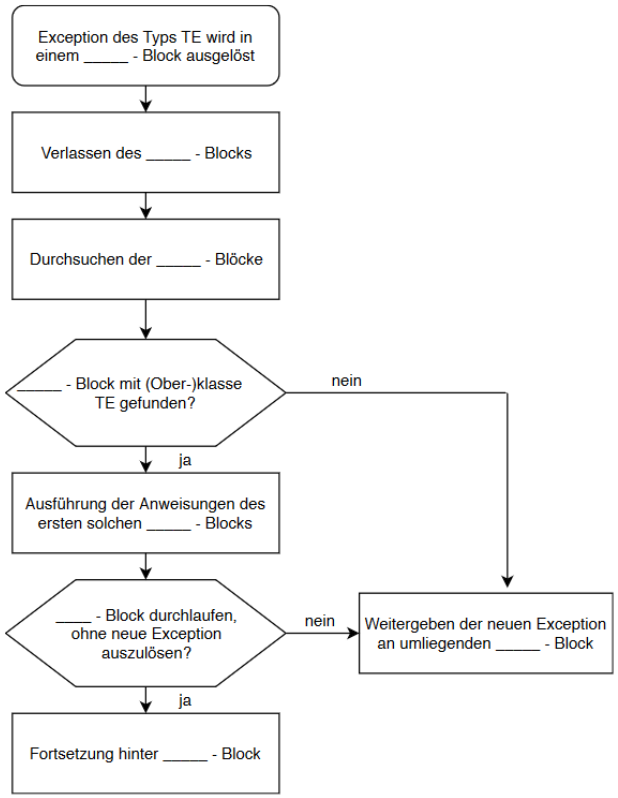
(1) Die Ausgabe wäre:
begin doRisky
end doRisky

(2) Die Ausgabe wäre:
begin doRisky
catching ExplodeException!

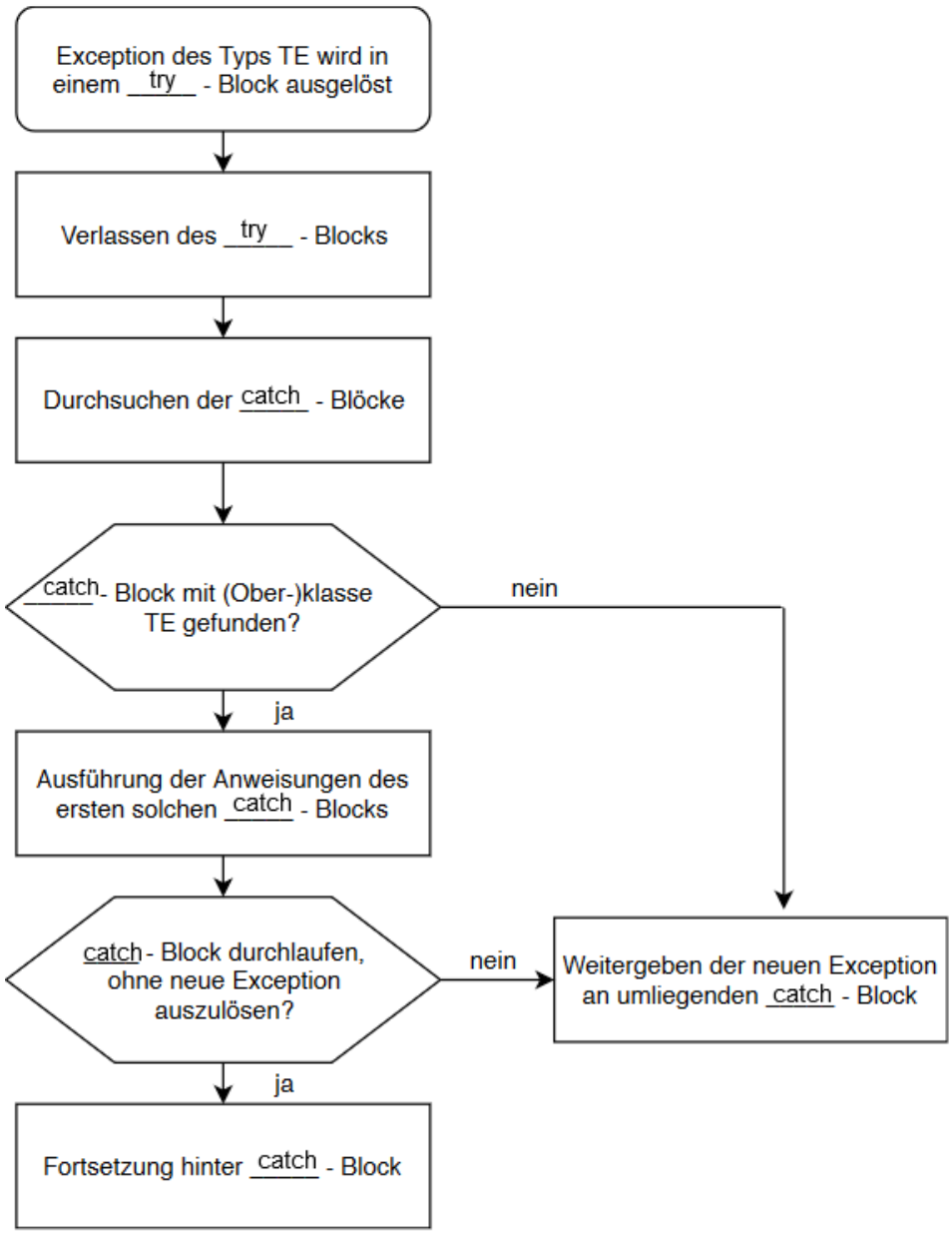
V4 Ablaufdiagramm

In dieser Aufgabe beschäftigen wir uns mit dem Auffangen einer hypothetischen `Exception` des Typs `TE`. Ergänzen Sie in nebenstehenden Ablaufdiagramm an freien Stellen, ob es sich um einen `catch`- oder `try`-Block handelt

Ein Beispiel für Ablaufdiagramme finden Sie beispielsweise hier:
https://de.wikipedia.org/wiki/Programmablaufplan#Beispiel



Lösungsvorschlag:



Übungsblatt 07: Lösungsvorschläge von Nhan Huynh und Darya Nikitina

V5 assert-Anweisungen ★☆☆

In der Vorlesung haben Sie die assert-Anweisungen kennengelernt.

1. Beschreiben Sie in eigenen Worten, was ein Assertion-Error ist.
2. Schreiben Sie den nachfolgenden Codeschnipsel kompakter mittels assert-Anweisungen!

```
1 if ((k > 0 && k + 1 <= 5) || (k % 3 == 2)) {
2   throw new AssertionError("Very bad k!");
3 }
```

3. Sie wissen, dass `assert`-Anweisungen beim Kompilieren an- oder abgeschaltet werden können mit entsprechenden Setzungen für den Compiler. Welche Vorteile ergeben sich hieraus? Warum sollten wir sie ausschalten und nicht einfach auch im realen Einsatz des Programms immer eingeschaltet mitlaufen lassen?

Lösungsvorschlag:

1. Ein Assertion-Error ist ein Fehler, der so gewichtig ist, dass er mit Fehlerbehandlung nicht mehr zu retten ist. Deshalb soll das Programm dann sofort abgebrochen werden. Ein solcher Fehler wird mit Hilfe einer `assert`-Anweisung oder mit Hilfe eines `AssertionError` geworfen.
2.

```
1 assert (k <= 0 || k + 1 > 5) && (k % 3 != 2) : "Very bad k!";
```

3. Beim Testen des Programms sind `assert`-Anweisungen sehr hilfreich, um mögliche Fehlerquellen schnell beheben zu können. Man sollte sie jedoch im realen Einsatz des Programms ausschalten, damit sich die Laufzeit des Programms nicht unnötig verzögert.

Information: assert-Anweisungen sind standardmäßig nicht angeschaltet. Um diese verwenden zu können muss der Flag `-eq` für die JVM[a] hinzugefügt werden.

[a] Java Virtual Machine

Übungsblatt 07: Lösungsvorschläge von Nhan Huynh und Darya Nikitina

V6 Erster Test mittels BeforeEach

In dieser Aufgabe wollen wir einen Blick auf die `BeforeEach`-Annotation von JUnit 5 werfen. Methoden mit dieser Annotation werden **vor Beginn jedes einzelnen Tests** ausgeführt!

Gegeben sei eine Bibliothek für Geometrie-Funktionen. Dabei betrachten wir nur die Funktion `triangleArea`, die den Flächeninhalt eines Dreiecks berechnet und dazu die Längen der einzelnen Seiten (`a`, `b` und `c`) als `int`-Werte übergeben bekommt.

V6.1 Setup vor jedem Test

Gegeben sei folgende Klasse `GeoLib`:

```
1 public class GeoLib {
2
3   public GeoLib() {
4   }
5 }
```

Um die Funktionen der Bibliothek verwenden zu können, muss zunächst ein Objekt vom Typ `GeoLib` erzeugt werden. Hierzu können Sie den parameterlosen Standard-Konstruktor der Klasse `GeoLib` verwenden. Schreiben Sie eine entsprechend mit JUnit-Annotationen versehene Methode namens `setup`, die in einer Testklasse steht und die für jeden Testfall eine neue Instanz von `GeoLib` in dem bereits deklarierten Attribut `geoLib` speichert.

Lösungsvorschlag:

```
1 private GeoLib g;
2
3 @BeforeEach
4 public void setup() {
5   g = new GeoLib();
6 }
```

V6.2 Testfall

Schreiben Sie mindestens drei JUnit-Tests, die überprüfen, ob die Methode `triangleArea` für verschiedene Dreiecke korrekt arbeitet. Mindestens ein Testdreieck sollte dabei auch entartet sein.

Lösungsvorschlag:

```
1  @Test
2  public void testTtiangleArea01() {
3    assertEquals(43.301, g.triangleArea(10, 10, 10), delta);
4  }
5
6  @Test
7  public void testTtiangleArea02() {
8    assertEquals(4452.572, g.triangleArea(42, 221, 213), delta);
9  }
10
11 @Test
12 public void testTtiangleArea03() {
13   assertEquals(3901146.591, g.triangleArea(4324, 3242, 2432), delta);
14 }
```

Übungsblatt 07: Lösungsvorschläge von Nhan Huynh und Darya Nikitina

V7 Fehlertypen

In der Vorlesung haben Sie kennengelernt, dass man Fehler in Programmen in zwei Kategorien einteilen kann. Es wurde unterschieden zwischen Kompilierzeitfehlern und Laufzeitfehlern.

(1) Beheben Sie im folgenden Codeausschnitt alle Kompilierzeitfehler:

```
1  public static int[] reverseArray(int[] source) throw Exception {
2    int length = source.length();
3    int[] inverted;
4    int i = 0;
5    int j = length;
6
7    try {
8      inverted = new int[length];
9
10     while (i < length) {
11       inverted[i] = source[j];
12       i--;
13       j++;
14     }
15   } catch (IndexOutOfBoundsException e) {
16     System.out.println("Caught Exception: " + e);
17   } catch (Exception) {
18     System.out.println("Caught Exception ...");
19   }
20
21   inverted = new int[length];
22   inverted = source;
23
24   return inverted;
25 }
```

(2) Was passiert generell beim Aufruf der Methode `reverseArray`? Warum kann der Code auch ohne vorhandene Kompilierzeitfehler nicht fehlerfrei ausgeführt werden? Was müsste man beheben um den Code ausführbar zu machen?

(3) Das Programm läuft zwar jetzt fehlerfrei, das Ergebnis entspricht aber noch nicht dem gewünschten Ergebnis (Array soll umgedreht werden). Beheben Sie alle fehlerhaften Stellen im Code, um das gewünschte Ergebnis zu erreichen.

Übungsblatt 07: Lösungsvorschläge von Nhan Huynh und Darya Nikitina

Lösungsvorschlag:

(1)

- Zeile 1: `throws` statt `throws`
- Zeile 2: `source.length` statt `source.length()`
- Zeile 17: `Exception e`: `e` fehlt

```
1  public static int[] reverseArray(int[] source) throws Exception {
2    int length = source.length;
3    int[] inverted;
4    int i = 0;
5    int j = length;
6
7    try {
8      inverted = new int[length];
9
10     while (i < length) {
11       inverted[i] = source[j];
12       i--;
13       j++;
14     }
15   } catch (IndexOutOfBoundsException e) {
16     System.out.println("Caught Exception: " + e);
17   } catch (Exception e) {
18     System.out.println("Caught Exception ...");
19   }
20
21   inverted = new int[length];
22   inverted = source;
23
24   return inverted;
25 }
```

(2)

- Beschreibung:
  Die Methode reverseArray hat die Funktionalität ein Array in umgekehrter Reihenfolge auszugeben. Dabei werden eine Variable, welche die Länge des Arrays zwischen speichert und zwei Laufzähler eingerichtet und das zurückzugebende Array wird deklariert. Im `try`-Block wird das zurückzugebende Array initialisiert. Die `while`-Schleife ist für das Invertieren zuständig. Wird eine `IndexOutOfBoundsException` oder jegliche andere Exception geworfen, so wird diese gefangen und eine Nachricht wird auf der Konsole ausgegeben.
- Fehler:
  Der Code wird ohne Kompilierfehler nicht durchgehen. Die Gründe dafür sind:
  a) Zeile 5: `j` soll auf das letzte Element des Arrays zeigen, aber dieses befindet sich am Index `length - 1`
  b) Zeile 12 + 13: Die Zählvariaben wurden falsch herum gesetzt und somit würde eine `IndexOutOfBoundsException` geworfen werden, da hier `j` vergrößert und `i` verkleinert wird.

```
1  public static int[] reverseArray(int[] source) throws Exception {
2    int length = source.length;
3    int[] inverted;
4    int i = 0;
5    int j = length - 1;
6
7    try {
8      inverted = new int[length];
9
10     while (i < length) {
11       inverted[i] = source[j];
12       i++;
13       j--;
14     }
15   } catch (IndexOutOfBoundsException e) {
16     System.out.println("Caught Exception: " + e);
17   } catch (Exception e) {
18     System.out.println("Caught Exception ...");
19   }
20
21   inverted = new int[length];
22   inverted = source;
23
24   return inverted;
25 }
```

(3) Das Problem ist, dass in der Zeile 21 und 22 das Ergebnis überschrieben wird, welches wir vorher berechnet haben. Um das Problem zu beheben wurden folgende Änderungen durchgeführt:

- Zeile 3: `inverted` wurde mit dem Wert `null` initialisiert, da die Methode in jedem Fall einen Wert zurückgeben muss.
- Zeile 23 und 24 wurden entfernt.

```
1  public static int[] reverseArray(int[] source) throws Exception {
2    int length = source.length;
3    int[] inverted = null;
4    int i = 0;
5    int j = length - 1;
6
7    try {
8      inverted = new int[length];
9
10     while (i < length) {
11       inverted[i] = source[j];
12       i++;
13       j--;
14     }
15   } catch (IndexOutOfBoundsException e) {
16     System.out.println("Caught Exception: " + e);
17   } catch (Exception e) {
18     System.out.println("Caught Exception ...");
19   }
20
21   return inverted;
22 }
```

Übungsblatt 07: Lösungsvorschläge von Nhan Huynh und Darya Nikitina

V8 Testen mit JUnit - Qualitätskontrolle

Wir wollen ein neues System zur Qualitätskontrolle in einer Produktionskette testen. Hierzu gibt es eine Klasse `ProductLineManagement`, die Güter (Typ `Product`) herstellt. Ihre Tests sollen nun prüfen, ob dies schnell genug und hinreichend gut erfolgt. Die Maschinen garantieren dabei immer eine Mindestqualität von 89 ($= 89\%$ der optimalen Qualität).

Die Qualität wird gemessen auf einer Skala von 0 (defekt) bis 100 (perfekt).

Die Testklasse deklariert ein Attribut `static ProductLineManagement plm`, auf das Sie zugreifen können.

V8.1 Setup-Methode

Vor jedem Test muss die (sehr komplexe) Produktionskette initialisiert werden. Dies erfolgt durch den Aufruf des Konstruktors der Klasse `ProductLineManagement` mit dem Namen der Firma als `String`. Den Firmennamen dürfen Sie beliebig wählen. Geben Sie eine mit JUnit-Annotationen versehene öffentlich sichtbare Methode an, die diese Initialisierung vor jedem Test durchführt.

Lösungsvorschlag:

```
1 @BeforeEach
2 public void init() {
3   plm = new ProductLineManagement("FoP");
4 }
```

V8.2 Normalfall

Schreiben Sie einen Test für eine normale Produktion. Hierbei soll ein einziger Artikel `normalProduct` durch die Methode `Product produce (String)` der Klasse `ProductLineManagement` (siehe oben) produziert werden. Stellen Sie sicher, dass der gelieferte Artikel nicht `null` ist und eine Mindestqualität - abfragbar via `getQuality()` - von 89 besitzt. Als Titel des Artikels können Sie einen beliebigen `String` angeben.

Lösungsvorschlag:

```
1 @Test
2 public void testProduce() {
3   Product normalProduct = plm.produce("Hausuebungen");
4   assertNotNull(normalProduct);
5   assertTrue(normalProduct.getQuality() >= 89);
6 }
```

Übungsblatt 07: Lösungsvorschläge von Nhan Huynh und Darya Nikitina

V8.3 Behandlung von Exceptions

Schreiben Sie nun einen weiteren Test, der auch wie im vorherigen Aufgabenteil ein neues Produkt erstellt. Nur diesmal reichen Sie dieses Produkt mittels `boolean submit(Product, int)` aus der Klasse `ProductLineManagement` für die Qualitätskontrolle ein. Wurde die gewünschte Qualität erreicht, liefert die Methode `true`, andernfalls wirft die Methode eine `InsufficientQualityException`.

Testen Sie das Verhalten und das Auftreten der Exception mit einem Produkt, indem Sie dieses einmal auf die (garantierte) Mindestqualität von 89 und einmal auf die unerreichbare Qualität von 101 testen.

Lösungsvorschlag:

```
1  @Test
2  public void testSubmit() {
3    try {
4      Product normalProduct = plm.produce("Hausuebungen");
5      // Throws an exception
6      assertTrue(plm.submit(normalProduct, 89));
7      assertThrows(InsufficientQualityException.class, () -> plm.submit(normalProduct, 101));
8
9    } catch (InsufficientQualityException e) {
10     fail("InsufficientQualityException thrown");
11   }
12 }
```

Übungsblatt 07: Lösungsvorschläge von Nhan Huynh und Darya Nikitina

V9 Testen: Racket und Java

Sie haben nun sowohl das Testen in Java mittels JUnit, als auch das Testen in Racket mittels Checks kennengelernt. In dieser Aufgabe sollen Sie zuerst eine Problemstellung in beiden Sprachen lösen und anschließend Ihre Implementierungen testen.

Gegeben ist eine Zahlenliste. In Racket ist diese als Liste von `numbers` gegeben, in Java als Array von Typ `int[]`. Außerdem sind zwei Parameter `lower` und `upper` gegeben. Ziel ist es, alle Werte aus der Zahlenliste zu sortieren, welche nicht zwischen diesen beiden Grenzwerten `lower` und `upper` liegen (jeweils exklusive).

Ergänzen Sie die beiden untenstehenden Codeausschnitte und Verträge, um diese Problemstellung zu lösen.

```
;; Type: (list of number) number number -> (list of  number)
;; Returns:
(define (numbersBetween alon lower upper)
  ..... )
```

```
/**
 * @param a
 * @param lower
 * @param upper
 * @return
 */
public int[] betweenNumbers(int[] a, int lower, int upper) {
  .....
}
```

Sollte der Parameter `lower` dabei größer als der Parameter `upper` sein, so soll in Java eine `LowerBiggerThanUpperException` geworfen werden. Ergänzen Sie dies in Ihrer Implementierung.

In Racket haben Sie die Möglichkeit einen Fehler auszulösen. Dies geschieht über den Befehl (`error msg`), wobei `msg` ein `String` mit der gewünschten Fehlermeldung ist. Konventionsmäßig einigen wir uns darauf, dass wir bei `msg` zuerst den Funktionsnamen nennen, gefolgt von einem Doppelpunkt und einer Beschreibung des Fehlers. Lösen Sie äquivalent zur `LowerBiggerThanUpperException` auch in der Racketfunktion einen Fehler für diesen Fall aus.

Testen Sie abschließend die beiden Implementierungen mit jeweils 3 Tests. Ein Test sollte dabei das korrekte Werfen der Fehlermeldung testen.

Übungsblatt 07: Lösungsvorschläge von Nhan Huynh und Darya Nikitina

Lösungsvorschlag:

```java
/**
 * Steps through the array and picks one element from the input data. Compares this element
 * to its adjacent element and moves through all element of the sorted array
 * until the picked element is at its correct position.
 *
 * @param array          the unsorted array which should be sorted
 * @param startInclusive the minimum (inclusive) index of the array
 * @param endInclusive   the maximum (inclusive) index of the array
 */
private void insertionSort(int[] array, int startInclusive, int endInclusive) {
  for (int j = startInclusive + 1; j <= endInclusive; j++) {
    int key = array[j];
    int i = j - 1;

    while (i >= startInclusive && array[i] > key) {
    array[i + 1] = array[i];
      i--;
    }
    array[i + 1] = key;
  }
}
```

```java
/**
 * Returns an array, in which all elements are sorted that are not in the
 * specified interval
 *
 * @param a     the array, in which all elements should be sorted that are not in the
 *              specified interval
 * @param lower the lower value of the interval
 * @param upper the upper value of the interval
 * @return an array, in which all elements are sorted that are not in the
 * specified interval
 * @throws LowerBiggerThanUpperException, if lower is not smaller than upper
 */
public int[] betweenNumbers(int[] a, int lower, int upper)
  throws LowerBiggerThanUpperException {
  if (lower > upper) {
    throw new LowerBiggerThanUpperException();
  }

  int[] sorted = new int[a.length];
  int size = 0;
  int i = 0;

  for (; i < a.length; i++) {
    // Copy in new array
    sorted[i] = a[i];
    // Compute interval
    if (a[i] <= lower || a[i] >= upper) {
      size++;
    }
    // Sort
    else if (size > 1) {
      insertionSort(sorted, i - size, i - 1);
      size = 0;
    }
  }
  // Sort
  if (size > 1) {
    insertionSort(sorted, i - size, i - 1);
  }
  return sorted;
}
```

```
@Test
public void testNoSorting() {
  int[] a = {1, 5, 3, 10, 6, 2, 3, 4, 5, 7, 6, 7, 8, 9, 2};
  int[] expected = {1, 5, 3, 10, 6, 2, 3, 4, 5, 7, 6, 7, 8, 9, 2};
  int[] actual = assertDoesNotThrow(() -> betweenNumbers(a, 0, 20));
  assertArrayEquals(expected, actual);
}

@Test
public void testSorting() {
  int[] a = {1, 5, 3, 10, 6, 3, 4, 2, 5, 7, 6, 7, 8, 9, 2};
  int[] expected = {1, 5, 3, 6, 10, 3, 4, 2, 5, 6, 7, 7, 8, 9, 2};
  int[] actual = assertDoesNotThrow(() -> betweenNumbers(a, 0, 4));
  assertArrayEquals(expected, actual);
}

  @Test
public void testException() {
  int[] a = {1, 5, 3, 10, 6, 2, 3, 4, 5, 7, 6, 7, 8, 9, 2};
  assertThrows(LowerBiggerThanUpperException.class,
      () -> betweenNumbers(a, 4, 3));
 }
```

```
;; Type: (list of number) number number -> (list of number)
;; Returns: a list of numbers, in which all elements should be
;; sorted that are not in the specified interval
(define (numbersBetween alon lower upper)
  (local
    (
     ;; Type: (list of number) (list of number) (list of number)
     ;; -> (list of number)
     ;; Returns: a sorted list in descending order
     (define (numbersBetween-sort alon sorted acc)
       (cond
         ; Empty list
         [(empty? alon)
          ; If acc still contains elements, append it to sorted
          (if (empty? acc)  sorted (append (sort  acc >) sorted))]
         ; Element must be greater than lower and smaller than upper
         [(and (< lower (first alon)) (> upper (first alon)))
          ; If acc is empty, we add the element to sorted, else we add acc
          ; in descending order to sorted
          (if (empty? acc)
              (numbersBetween-sort (rest alon) (cons (first alon) sorted) acc)
              (numbersBetween-sort (rest alon) (append (list (first alon))
                                                       (sort acc >)sorted ) empty))]
         ; Add elements to acc
         [else (numbersBetween-sort (rest alon) sorted
                                    (cons (first alon) acc))])))
    ; Error, if upper is smaller than lower
    (if (< upper lower)
        (error 'numbersBetween: "Lower cannot be greater than upper!")
        ; Reverses the list from descending order to ascending order
        (reverse (numbersBetween-sort alon empty empty)))))

;; Tests
(check-expect (numbersBetween (list  1 5 3 10 6 2 3 4 5 7 6 7 8 9 2) 0 20)
              (list  1 5 3 10 6 2 3 4 5 7 6 7 8 9 2))
(check-expect (numbersBetween (list 1 5 3 10 6 3 4 2 5 7 6 7 8 9 2) 0 4)
              (list 1 5 3 6 10 3 4 2 5 6 7 7 8 9 2 ))
(check-error (numbersBetween (list  1 5 3 10 6 2 3 4 5 7 6 7 8 9 2 ) 4 3))
```

Übungsblatt 07: Lösungsvorschläge von Nhan Huynh und Darya Nikitina

V10 Exceptionklassen

V10.1

Schreiben Sie eine `public`-Klasse `MyException`, die von `Exception` erbt. Der Konstruktor dieser Klasse hat einen Parameter `str` vom Typ `String` und einen Parametern vom Typ `int`. Ein Objekt von `MyException` hat ein `private`-Attribut `message` vom Typ `String`. Der Konstruktor weist `message` die Konkatenation aus beiden Parametern zu. Die `public`-Methode `getMessage` von `Exception` soll so überschrieben werden, dass `message` zurückgeliefert wird.

Lösungsvorschlag:

```
1  public class MyException extends Exception {
2
3    private String message;
4
5    public MyException(String str, int n) {
6      message = str + n;
7    }
8
9    @Override
10   public String getMessage() {
11     return message;
12   }
13 }
```

V10.2

Schreiben Sie eine `public`-Klasse `X` mit einer `public`-Klassenmethode `km`, die einen `int`-Parameter `n` hat, `int` zurückliefert und potentiell `MyException` wirft. Und zwar wirft `km` eine `MyException` mit `"n cannot be negative"` und `n` als Parameterwerten, wenn negativ ist. Andernfalls liefert `km` das Quadrat von `n` zurück.

Lösungsvorschlag:

```
1  public class X {
2
3    public static int km(int n) throws MyException {
4      if (n < 0) {
5        throw new MyException("n can not be negative", n);
6      }
7      return n * n;
8    }
9  }
```

V10.3

Schreiben Sie eine `public`-Klasse `Y` mit einer `public`-Objektmethode `m`, die einen `int`-Parameter `n` hat und `int` zurückliefert. Diese Methode ruft `km` von `X` mit `n` auf, ohne ein Objekt von `X` dafür einzurichten, und liefert das Ergebnis

von `km` zurück. Sollte `km` eine `Exception` werfen, dann soll die Botschaft der `Exception` auf dem Bildschirm ausgegeben werden.

Lösungsvorschlag:

```
1  public class Y extends X {
2
3    public int m(int n) {
4      try {
5        return km(n);
6      } catch (MyException e) {
7        System.out.println(e.getMessage());
8      }
9      return 0;
10   }
11 }
```

V11 Welcher Belag darf es sein?

Schreiben Sie eine Klasse `NoBreadException` welche von `Exception` erbt, im Konstruktor einen Parameter `String topping` erhält und damit den Konstruktor der Basisklasse mit der Konkatenation `"There is no bread, only" + topping` aufruft.

Schreiben Sie dann ein Functional Interface namens `Lunch`. Dieses enthält die funktionale Methode `String getTopping(String s)`, welche eine `NoBreadException` wirft.

Initialisieren Sie nun das Functional Interface `Lunch` durch einen Lambda-Ausdruck. Geprüft werden soll, ob der `String` ein korrektes Sandwich ist. Dabei besteht ein korrektes Sandwich aus zweimal dem Substring `"bread` und einem Topping dazwischen. Korrekte Sandwiches sind also beispielsweise `"breadtunabread"` oder `"breadabcdefgbread"`. Vordem ersten `"bread"` und nach dem zweiten `"bread"` darf kein Substring mehr stehen. Zurückgegeben werden soll immer das Topping, also der Substring zwischen den beiden `"bread"`'s. Das Topping muss dabei nicht „sinnvoll" sein, sondern irgendein beliebiger `String`. Ist kein Brot vorhanden, so soll eine `NoBreadException` mit dem alleinigen Topping geworfen werden.

Sie dürfen davon ausgehen, dass niemals nur eine Brotscheibe verwendet wird, sondern entweder zwei oder keine.

Übungsblatt 07: Lösungsvorschläge von Nhan Huynh und Darya Nikitina

Lösungsvorschlag:

```
public class NoBreadException extends Exception {

  public NoBreadException(String topping) {
    super("There is no bread, only " + topping);
  }
}

@FunctionalInterface
public interface Lunch {

  String getTopping(String s) throws NoBreadException;
}
```

```
Lunch lunch = s -> {
  if (s.startsWith("bread") && s.endsWith("bread")) {
    return s.substring(5, s.length() - 5);
  }
  throw new NoBreadException(s);
};
```

Übungsblatt 07: Lösungsvorschläge von Nhan Huynh und Darya Nikitina

V12 Arrays, Exceptions und Vererbung ★★★

V12.1 Klasse X

Gegeben sei die folgende Klasse:

```
1 public class X {
2
3   public int[] a;
4   public boolean[] writable;
5 }
```

Wir benutzen das Array, auf das `a` verweist, um `int`-Werte zu speichern. Im Array, auf das `writable` verweist, wird festgehalten ob ein `int`-Wert in `a` überschrieben werden darf, oder nicht. Der `int`-Wert `a[i]` darf überschrieben werden, wenn `writable[i] == true`. Sie können davon ausgehen, dass beide Arrays der Klasse `X` immer die gleiche Länge besitzen, sodass die Indizes der beiden Arrays übereinstimmen.

Implementieren Sie den Konstruktor der Klasse `X`, dieser bekommt einen `int`-Parameter übergeben und initialisiert die Arrays `a` und `writable` mit der gleichen Länge, dabei soll jeder Wert im Array `writable` mit `true` initialisiert werden. Die Länge beider Arrays entsprechen hier dem Wert des übergebenen Parameters. Erweitern Sie nun die Klasse um eine `public`-Methode `save`. Die Methode liefert nichts zurück, bekommt einen `int`-Parameter übergeben und speichert den übergebenen Parameter am kleinsten freien Index im Array `a` ab, an dem ein Wert nach aktueller Definition von `writable` überschrieben werden darf. Zusätzlich setzt sie den entsprechenden Wert im Array `writable` auf `false`. Darf kein Wert überschrieben werden, so soll die Methode eine `ArrayStoreException` mit der Nachricht `"no free space left"` werfen.

Übungsblatt 07: Lösungsvorschläge von Nhan Huynh und Darya Nikitina

Lösungsvorschlag:

```
1  public class X {
2
3    public int a[];
4    public boolean writable[];
5
6    public X(int n) {
7      a = new int[n];
8      writable = new boolean[n];
9      for (int i = 0; i < writable.length; i++) {
10       writable[i] = true;
11     }
12   }
13
14   public void save(int n) throws ArrayStoreException {
15     for (int i = 0; i < writable.length; i++) {
16       // Only if true, do the following operation
17       if (writable[i]) { // equal to writable[i] == true
18         writable[i] = false;
19         a[i] = n;
20         return;
21       }
22     }
23     throw new ArrayStoreException("no free space left");
24   }
25 }
```

V12.2 Klasse Y

Schreiben Sie nun eine Klasse `Y`, die von der Klasse `X` aus Aufgabe rbt. Die Klasse soll die Methode `save` der Oberklasse überschreiben. Die Methode liefert nichts zurück, bekommt einen `int`-Parameter übergeben und soll mit diesem Parameter die Methode `save` der Oberklasse aufrufen. Wird dabei eine `ArrayStoreException` geworfen, so soll diese abgefangen werden. Ist dies der Fall, so sollen die beiden Arrays `a` und `writable` um ihre aktuelle Länge erweitert werden, um dann anschließend den übergebenen Parameter mittels der Methode `save` abspeichern zu können. Die bereits gespeicherten Werten in den beiden Arrays dürfen bei der Erweiterung nicht verloren gehen.

Lösungsvorschlag:

```
1  public class Y extends X {
2
3    public Y(int n) {
4      super(n);
5    }
6
7    @Override
8    public void save(int n) {
9      try {
10       super.save(n);
11     } catch (ArrayStoreException e) {
12       int[] newA = new int[a.length * 2];
13       boolean[] newWritable = new boolean[writable.length * 2];
14       for (int i = 0; i < writable.length; i++) {
15         newA[i] = a[i];
16         newWritable[i] = writable[i];
17       }
18       for (int i = writable.length; i < newWritable.length; i++) {
19         newWritable[i] = true;
20       }
21       a = newA;
22       writable = newWritable;
23       save(n);
24     }
25   }
26 }
```