Funktionale und objektorientierte Programmierkonzepte

Florian Kadner und Lukas Röhrig
(Gesamtleitung: Prof. Karsten Weihe)

Wintersemester 18/19
v1.2

Übungsblatt 7

Themen: Assertions, JUnit und Fehlerbehandlung

Relevante Folien: Fehlerbehandlung

Abgabe der Hausübung: 14.12.2018 bis 23:55 Uhr

Einbinden und Verwendung von JUnit

Auf diesem Übungsblatt werden wir erstmals mit JUnit arbeiten. In dieser Veranstaltung verwenden wir die aktuellste Version, also JUnit 5. JUnit 5 wird bereits mit Eclipse ausgeliefert und muss daher nicht erneut heruntergeladen werden. Allerdings muss die Bibliothek dem Projekt noch hinzugefügt werden. Dazu führt man die folgenden Schritte aus:

1. Öffnen des Eclipse-Projekts, an dem gearbeitet und für das die Bibliothek genutzt werden soll.
2. Öffnen der Projekt-Eigenschaften durch Rechtsklick auf dem Projektnamen.
3. Auswahl des Eintrags *Java Build Path*.
4. Im rechten Teil des Fensters zum Karteireiter *Libraries* wechseln und auf *Add Library...* klicken.
5. In dem nun erscheinenden Dialog wählt man *JUnit*, klickt auf *Next* und wählt dann JUnit 5.
6. Nun kann man auf *Finish* klicken.

V Vorbereitende Übungen

V1 Theoriefragen ☆ ☆ ☆

V1.1 Grundlegendes

1. Wie hängen die Begriffe `throws` und `throw` 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.

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 gleichgültig.

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

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

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

V2 Try/Catch-Block ★ ☆ ☆

Was ist das Problem mit dem folgenden Codeausschnitt?

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

Modifizieren Sie den Code mittels `try`/`catch`-Blockes um das Problem zu beheben.

V3 Exceptions ★ ☆ ☆

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

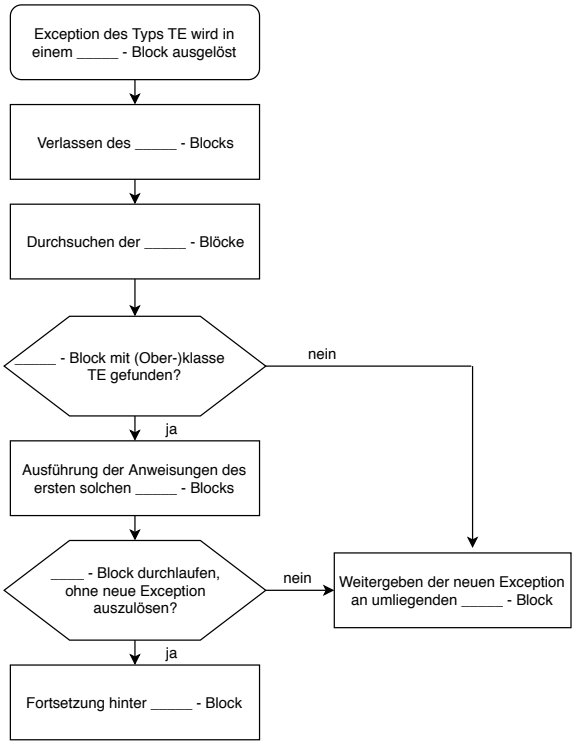
(1) Welche Ausgabe wird dieses Programm beim Aufruf der Methode `test()` liefern?
(2) Welche Ausgabe erfolgt bei einer Änderung von Zeile 4 in String `test = "yes";` ?

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

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`



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!

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

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?

V6 Erster Test mittels BeforeEach ★ ★ ☆

In dieser Aufgabe wollen wir einen Blick auf die `BeforeEach`-Annotation von JUnit5 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. Geben Sie eine entsprechend mit JUnit-Annotationen versehene Methode namens `setup` an, die in einer Testklasse steht und die für jeden Testfall eine neue Instanz von `GeoLib` in dem bereits deklarierten Attribut `geoLib` speichert.

V6.2 Testfall

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

V7 Fehlertypen ★★☆

Zur Erinnerung: Die Syntax ist die Struktur und die Semantik die Bedeutung des Programms. Dies lässt uns zwei verschiedene Arten von Fehlern definieren:

1. **Syntaxfehler** findet und moniert der Compiler. Typische Beispiele sind falsch geschriebene oder unbekannte Worte oder eine falsche Anordnung von Worten und Operanden.
2. Bei **Semantikfehlern** unterscheiden wir nochmals in...
   - ... solche, die der Compiler nicht findet, sondern die sich zur Laufzeit durch Programmabbruch auswirken; und
   - ... solche, die der Compiler findet, wie zum Beispiel nicht erreichbarer Programmcode oder auch Typfehler.

(1) Beheben Sie im folgenden Codeausschnitt alle lexikalischen und syntaktischen Fehler:

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

(2) Was passiert generell beim Aufruf der Methode `reverseArray`? Warum kann der Code auch ohne vorhandene syntaktische Fehler nicht kompiliert werden? Was müsste man beheben um den Code kompilierbar 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 fehlerhafte Stellen im Code, um das gewünschte Ergebnis zu erreichen.

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% des Optimums).

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.

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.

V8.3 Behandlung von Exceptions

Schreiben Sie nun einen weiteren Test der auch wie im vorherhigen 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.

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 Aufgaben 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.

Racket:

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

Java:

```
/**
  *
  * @param arr
  * @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. In Racket können Sie dies mit `(check-error fct msg)` bewerkstelligen. Dabei steht `(fct)` für den Funktionsaufruf und `msg` für die Fehlermeldung.

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 Parameter `n` 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.

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 can not be negative"` und `n` als Parameterwerten, wenn `n` negativ ist. Andernfalls liefert `km` das Quadrat von `n` zurück.

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.

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 Sandwiche sind also beispielsweise `"breadtunabread"` oder `"breadabcdefgbread"`. Vor dem 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.

V12 Arrays, Exceptions und Vererbung ★ ★ ★

V12.1 Klasse X

Gegeben sei die folgende Klasse:

```
public class X {
  public int a[];
  public boolean writable[];
}
```

Wir benutzen das Array `a[]` um `int`-Werte zu speichern. Im Array `writable[]`, wird festgehalten ob ein `int`-Wert im Array `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 initalisiert 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.

V12.2 Klasse Y

Schreiben Sie nun eine Klasse `Y`, die von der Klasse `X` aus Aufgabe V12.1 erbt. 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 `writeable[]` 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.

H Siebte Hausübung — Gesamt 13 Punkte
Terme und Arithmetik

In der Mathematik ist ein Term ein sinnvoller Ausdruck, der Zahlen, Variablen, Symbole für mathematische Verknüpfungen und Klammern enthalten kann. In dieser siebten Hausübung wollen wir uns mit solchen Termen und grundlegender Arithmetik beschäftigen.

H1 Eigene Exception implementieren — 1 Punkt

Schreiben Sie zunächst eine neue Klasse `InvalidTermException` im Package `Exceptions`, die von der Klasse `Exception` abgeleitet ist. Die neue `Exception` besitzt zwei `private` `String`-Attribute `term` und `message`, sowie die `public`-Methoden `String getTerm()` und `String getMessage()`, die die jeweiligen Attribute zurückgeben. Der Konstruktor der Exception bekommt als ersten Parameter den Term übergeben, der die Exception ausgelöst hat und als zweiten Parameter eine Botschaft übergeben und weist die übergebenen Werte den entsprechenden Objektattributen zu.

H2 Terme modellieren — 3 Punkte

Wir wollen nun zuerst eine Klasse implementieren, die es uns ermöglicht Terme abzuspeichern und im weiteren Verlauf der Aufgabenstellung auszuwerten. Schreiben Sie nun eine Klasse `Term` im Package `Main` mit zwei `private` `String`-Attributen `term` und `result`. Der Konstruktor der Klasse bekommt einen angeblichen Term als `String` übergeben und entfernt zunächst alle Whitespaces im übergebenen `String`. Anschließend wird geprüft, ob nun dieser `String` einem sinnvollen Ausdruck entspricht, also ob die Zahlen, Klammern und Symbole für mathematische Verknüpfungen in genau diesem `String` sinnvoll angeordnet sind und ob somit dieser `String` wirklich einen Term darstellt. Der Einfachheit halber, wollen wir in unserer Implementation auf Variablen in Termen verzichten. Terme können somit aus folgenden Elementen bestehen:

(1) **Zahlen**:
Entweder als ganze Zahl oder als Gleitkommazahl mit einem Punkt als Dezimaltrennzeichen. Negative Zahlen tragen ein Minus (−) als Vorzeichen.
Beispiele für Zahlen: `-3` oder `5.239`.

(2) **Symbole für mathematische Verknüpfungen**:
Wir beschränken uns auf die Grundrechenarten, also auf die vier mathematischen Operationen Addition, Subtraktion, Multiplikation und Division. Dargestellt durch deren Symbole: Plus (+), Minus (−), Mal (∗) und Geteilt (/).

(3) **Klammern**:
Zur Gruppierung von Termen benutzen wir nur runde Klammern.

Hinweis: Eine Auslassung des Malzeichens, wie beispielsweise bei dem Term $3(2+1)$, ist nicht zulässig! Der korrekte Term lautet dann $3 * (2+1)$.

Folgend einige Beispiele für Strings, die zulässige Terme darstellen:

(1) `"120"`

(2) `"3*2+4-2"`

(3) `"(5+5.25)*2-10/(2+1)"`

(4) `"0+3.2*(-4+-5)*((2+2)*(1+(6-2.25125)))"`

Folgend einige Beispiele für Strings, die **keine** zulässigen Terme darstellen:

(1) `"3++2"` → Der Additionsoperator darf nicht mehrmals aufeinander folgen.

(2) `"(3+2*(2-4)"` → Eine schließende Klammer fehlt.

(3) `"4(4+8)"` → Ein Operator fehlt.

Sollte der übergebene String kein Term sein, so ist eine `InvalidTermException` mit einer Botschaft zu werfen, die beschreibt, wieso der übergebene String eben gerade kein Term ist, also gegen welche Anforderung verstoßen wurde. Passen Sie den Methodenkopf des Konstruktors der Klasse `Term` dahingehend an!

H3 Binäre Operatoren 1 Punkt

Wir wollen nun eine Klasse implementieren, die es uns ermöglicht die vier Grundrechenarten mit jeweils zwei Operanden vom Typ `String` auszuführen, um damit im weiteren Verlauf der Aufgabe unsere Terme auswerten zu können. Dazu schreiben wir eine Klasse `DoubleStringMath` im Package `Math`, die das `Interface StringMath` implementiert. Alle vier von Ihnen zu implementierenden Methoden, die die Grundrechenarten darstellen, sollen zunächst die beiden übergebenen Operanden zum Typ `double` konvertieren und dann damit die entsprechende Berechnung durchführen. Die Ergebnisse der Berechnungen sollen wieder als `String` zurückgegeben werden. Nutzen Sie dafür die Klassenmethode `String Utils.doubleToString(double d)`, diese konvertiert die übergebene Zahl vom Typ `double` zum Typ `String`.

H4 Terme auswerten 4 Punkte

Erweitern Sie nun die Klasse `Term` um eine Methode `public String getResult()`. Die Methode überprüft zunächst, ob das Objektattribut `result` noch `null` ist. Ist dies der Fall, so wird der Term solange ausgewertet, bis er nur noch aus einer Zahl besteht. Diese Zahl, also das Ergebnis der Auswertung, wird dann `result` zugewiesen und von der Methode zurückgegeben. Ist das Objektattribut `result` ungleich `null`, so wird es direkt zurückgegeben. Um einen Term auszuwerten, wenden Sie die Rechenregeln in folgender Reihenfolge an:

(1) Klammern

(2) Punktrechnung (Multiplikation und Division)

(3) Strichrechnung (Addition und Subtraktion)

(4) Von links nach rechts

Führen Sie die für die Auswertung benötigten Berechnungen zunächst über ein Objekt der von ihnen geschriebenen Klasse `DoubleStringMath` durch. Dafür weisen Sie dem Klassenattribut `public static StringMath math` der Klasse `Utils`, zu einem geigneten Zeitpunkt, ein neues Objekt der Klasse `DoubleStringMath` zu und greifen auf dieses Klassenatribut innerhalb der von Ihnen zu implementierenden Methode `public getResult()` der Klasse `Term` zu, um die Berechnungen durchzuführen. Dies ermöglicht uns einen einfachen Austausch der Klasse, die für die arithmetischen Operationen zuständig ist. Wollen wir die zuständige Klasse austauschen, so weisen wir dem Klassenattribut `math` der Klasse `Utils`, einfach ein Objekt einer anderen Klasse, die das `Interface StringMath` implementiert, zu.

Unverbindliche Hinweise:

Sie können sich folgende Methoden anlegen, um einen Term auszuwerten:

1. `boolean isAtom()`: Gibt `true` genau dann zurück, wenn der Term nur aus einer einzigen Zahl besteht wie beispielsweise unter (1) in den Beispielen von Aufgabe H2 gezeigt.
2. `int findAddOrSubOperation()`: Gibt die Position des linken Operanden einer Addition oder Subtraktion innerhalb eines Strings wieder.
3. `int findMulOrDivOperation()`: Gibt die Position des linken Operanden einer Multiplikation oder Division innerhalb eines Strings wieder.
4. `int findInnerMostExpression()`: Gibt die Position des meist innerliegenden Ausdrucks innerhalb des Terms zurück, also genau den Ausdruck, der nach Prioritäten als erstes berechnet werden muss.
5. `String evaluateSimpleExpression(String simple)`: Bekommt einen einfachen Term als String übergeben, also einen Term ohne Klammern, und liefert dessen Ergebnis als String zurück. Ein Beispiel für solch einen einfachen Term finden Sie in den Beispielen von H2 unter (2).

Folgend ein Beispiel für die Auswertung eines Terms, Schritt für Schritt:

1. `"54 + (545 * 234 + (4545 - 54))* 6 + 34"`
2. `"54 + (545 * 234 + 4491)* 6 + 34"`
3. `"54 + 132021 * 6 + 34"`
4. `"792214"`

H5 TermIO 2 Punkte

Jetzt wollen wir eine Klasse schreiben, die es uns ermöglicht Terme aus Dateien zu laden und die anschließend erfolgende Auswertung der Terme in einer Datei zu speichern. Legen Sie zunächst eine neue Klasse `TermIO` im Package `Main` an. Die Klasse hat nur ein `public`-Array `terms` vom Typ `Term`, in diesem Array wollen wir die Terme speichern, die wir aus einer beliebigen Datei laden wollen.

H5.1 Terme aus Datei einlesen 1 Punkt

Schreiben Sie eine Methode `public boolean readTermsFromFile(String filePath)`. Die Methode bekommt einen Pfad zu einer angeblichen Datei, die Terme enthalten soll, übergeben und benutzt die Klassenmethode

```
public static String[] readAllLinesFromFile(String filePath) throws
FileNotFoundException, UnsupportedEncodingException
```

der Klasse `Utils`, um die Datei Zeile für Zeile in ein `String`-Array einzulesen. Sofern eine `FileNotFoundException` geworfen wird, fangen Sie diese ab und geben Sie mittels `System.out.println()` die Nachricht `"File not found: "`, konkateniert mit dem übergebenen Dateipfad auf der Konsole aus und geben Sie `false` zurück. Sollte eine `UnsupportedEncodingException` geworfen werden, so geben Sie die Nachricht `"Unsupported encoding: "` konkateniert mti der Botschaft der Exception auf der Konsole aus und geben Sie ebenfalls `false` zurück. Wird keine Exception beim Lesen der Datei geworfen, so sollten sich nun alle Zeilen der Datei im zurückgegebenen `String`-Array der Methode `readAllLinesFromFile` befinden. Das Array `terms` der Klasse `TermIO` wird nun mit der Länge des `String`-Arrays initialisiert und anschließend wird jeder `String` des Arrays benutzt um ein entsprechendes `Term`-Objekt zu erstellen, das dem Array `terms` zugewiesen wird. Mit dem `String` an Index 0 im `String`-Array erstellen Sie ein neues `Term`-Objekt ebenfalls am Index 0 im Array `terms` usw. Sollte beim Erstellen eines neuen `Term`-Objektes, eine `InvalidTermException` geworfen werden, so fangen Sie diese ab und geben die Nachricht `"Invalid term: "` konkateniert mit dem Term auf der Konsole aus und geben sie `false` zurück. Sollten keine `Exceptions` geworfen werden, so gibt die Methoden am Ende `true` zurück.

H5.2 Terme auswerten und in Datei speichern 1 Punkt

Schreiben Sie die Methode `public boolean writeTermResultsToFile(String filePath)` der Klasse `TermIO`. Diese bekommt einen Pfad zu einer Datei übergeben und nutzt die Methode `public static void writeLinesToFile(String filePath, String[] lines)throws IOException` der Klasse `Utils` um die Ergebnisse der Auswertungen der Terme im Array `terms` in einer Datei abzuspeichern. Das Ergebnis der Auswertung jedes Terms im Array `terms` soll jeweils in einer neuen Zeile in der Datei am übergebenen Dateipfad abgespeichert werden. Fangen Sie eine eventuell geworfene `IOException` ab und geben Sie die Nachricht `"Error writing to file: "`, konkateniert mit der Botschaft der `Exception` auf der Konsole aus und geben Sie `false` zurück. Sollten keine `Exceptions` geworfen werden, so gibt die Methoden am Ende `true` zurück.

H6 Testen mittels JUnit 2 Punkte

Abschließend wollen wir einen Teil unserer implementierten Methoden testen. Legen Sie dazu eine neue Klasse `TermTests` im ebenfalls von Ihnen anzulegenden Package `Tests` an. Implementieren Sie nun folgende Tests in der neu angelegten Klasse:

(1) 3 Tests, die den Konstruktor der Klasse `Term` mit korrekten Termen aufrufen. Hierbei soll der Aufruf des Konstruktors keine `Exception` werfen.

(2) 3 Tests, die den Konstruktor der Klasse `Term` mit Strings aufrufen, die keine Terme

darstellen. Diese Tests sollen also prüfen, ob der Konstruktor in diesem Falle eine `InvalidTermException` wirft.

(3) 3 Tests, die die Methode `getResult()` der Klasse `Term` überprüfen.

Achtung: Selbstverständlich gilt auch hier, genau wie im Racket-Teil der Veranstaltung, dass Ihre Tests auch nicht-triviale Eingaben testen sollen! Von den drei Tests muss mindestens einer davon einen Randfall abdecken!