Funktionale und objektorientierte Programmierkonzepte
Übungsblatt 12

Prof. Karsten Weihe

Übungsblattbetreuer:	Lukas Klenner
Wintersemester 22/23	v1.0
Themen:	File-IO
Relevante Foliensätze:	08
Abgabe der Hausübung:	01.01.2022 bis 23:50 Uhr

Hausübung 12 — Gesamt: 38 Punkte

File-IO anhand von JSON Dateien

Beachten Sie die Seite *Verbindliche Anforderungen für alle Abgaben* im Moodle-Kurs.

Verstöße gegen verbindliche Anforderungen führen zu Punktabzügen und können die korrekte Bewertung Ihrer Abgabe beeinflussen. Sofern vorhanden, müssen die in der Vorlage mit `TODO` markierten `crash`-Aufrufe entfernt werden. Andernfalls wird die jeweilige Aufgabe nicht bewertet.

Die für diese Hausübung relevanten Verzeichnisse sind `src/main/java/h12` und ggf. `src/test/java/h12`.

Hinweise (Für die gesamte Hausübung**):**

- Sie können, wenn nicht anders in der Aufgabe angegeben, davon ausgehen, dass alle Attribute und Parameter nicht `null` sind.
- Um Zeilenumbrüche zu erzeugen, benutzten Sie `'\n'`.
- Alle für diese Hausübung benötigten Klassen und Methoden sind bereits vorgegeben. Verändern Sie deren Signatur nicht. Entfernen Sie nach dem Implementieren der Methoden die Aufrufe der Methode `crash();`
- Wenn Sie die GUI starten, benutzten Sie die Gradle Task application/run, damit beim Auswählen der Datei der richtige Ordner standardmäßig ausgewählt wird.

Einleitung

In dieser Hausübung werden Sie sich mit dem Einlesen und Erstellen von JSON Dateien beschäftigen. JSON [1] (JavaScript Object Notation) ist ein Dateiformat, mit welchem sich einfach Daten speichern und übertragen lassen. JSON Dateien basieren syntaktisch auf JavaScript, werden aber von allen gängigen Programmiersprachen unterstützt. Sie werden in verschiedensten Bereichen benutzt, wie z.B. zur Speicherung von Einstellungen, dem Datenaustausch bei vielen REST-APIs (z.B. Twitter), in verschiedensten Web Anwendungen, in Datenbanken, und vielen weiteren Bereichen in denen Daten gespeichert oder ausgetauscht werden.

Eine JSON Datei besteht dabei aus genau einem der folgenden Element:

1. **Objekt**: Ein Objekt wird von geschweiften Klammern umschlossen und besteht aus einer beliebigen Anzahl von Objekt Einträgen, welche mit einem Komma getrennt werden.

2. **Objekt Eintrag**: Ein Objekt Eintrag besteht aus einem Bezeichner, welcher als String dargestellt wird, sowie einem zugeordneten Wert, welcher als ein beliebiges JSON Element dargestellt wird. Diese beiden werden mit einem `':'` getrennt.

3. **String**: Eine Zeichenkette, beginnend und endend mit Anführungszeichen. Um es einfach zu halten, werden wir hier nicht Escape-Sequenzen, welche mit `'\'` beginnen, berücksichtigen.

4. **Array**: Ein Array wird von eckigen Klammern umschlossen und besteht aus einer beliebigen Anzahl von geordneten JSON Elementen, welche mit einem Komma getrennt werden.

5. **Zahl**: Eine Zahl, bestehend aus den Ziffern`'0' - '9'`, welche optional mit einem `'+'` oder `'-'` beginnen kann und mit einem `'.'` unterbrochen sein kann. Um es einfach zu halten, werden wir hier keine Exponenten berücksichtigen.

6. **Boolean Wert**: Die Konstanten `true` und `false`.

7. **Null Wert**: Die Konstante `null`.

> **Hinweis:**
>
> Wenn Sie möchten, können Sie gerne die Vereinfachungen, die wir für Zahlen und Strings vorgenommen haben auslassen und diese Aspekte ebenfalls implementieren.

[1] siehe: `https://de.wikipedia.org/wiki/JavaScript_Object_Notation`

Beispiel

Eine korrekte JSON Datei sieht beispielsweise wie folgt aus:

JSON Example

```json
{
  "Modulname": "Funktionale und objektorientierte Programmierkonzepte",
  "Dozent": "Karsten Weihe",
  "CP": 10.0,
  "WS": true,
  "Students": [
    {
      "Vorname": "Max",
      "Nachname": "Mustermann"
    },
    {
      "Vorname": "Erika",
      "Nachname": "Musterfrau"
    }
  ],
  "Noten": null
}
```

Die Vorlage

Die einzelnen JSON Elemente werden mit den Interface `JSONElement` und dessen Subinterfaces im Package `h12.json` dargestellt. Jedes JSON Element verfügt über eine Methode `write`, mit welcher die in ihm gespeicherten Informationen in eine JSON Datei geschrieben werden kann. Diese werden Sie in der Aufgabe H2 implementieren. Eine Implementation für die Interfaces finden Sie im package `h12.json.node`.

Im Interface `JSONElement` sind bereits Getter Methoden für jede Art von JSON Elemente definiert. Diese werfen standardmäßig eine `UnsupportedOperationException` und werden in den zugehörigen Subklassen korrekt implementiert. Dadurch müssen Sie die JSON Elemente nicht downcasten, um auf deren Informationen zuzugreifen, wenn diese einem vorgeschriebenes Format folgen sollen, wie z.B. in der Aufgabe H5. In der folgenden Tabelle finden Sie eine Übersicht über die vorhanden Interfaces, welche JSON Elemente darstellen[2].

Interface	Beispiel	unterstützte Methoden
`JSONString`	`"Hello World"`	`String getString()`
`JSONConstant`	`true`	`JSONConstants getConstant()`

`JSONNumber`	+123.456	`Number getNumber()`
		`Integer getInteger()`
		`Double getDouble()`

`JSONArray`	`["a", "b"]`	`JSONElement[] getArray()`
`JSONObject`	`{"a": true, "b": -10}`	`Set<JSONObjectEntry> getObjectEntries` `JSONElement getValueOf(String)`
`JSONObject.JSONObjectEntry`	`"a": true`	`String getIdentifier()` `JSONElement getValue()`

Tabelle 1: Übersicht der JSON Element Interfaces

In jedem der Interfaces ist zusätzlich eine statische Methode `of(...)` implementiert, mit welcher Sie einfach ein Objekt des Interfaces in Form einer `JSONElementNode` erzeugen können.

In dem Package `h12.json.parser` finden Sie die Klasse `JSONParser`, welche für das Einlesen von JSON Dateien verantwortlich ist. Diese kriegt im Konstruktor eine `JSONParserFactory` übergeben, mit welcher ein `JSONElementParser` erzeugt wird, in welchem die eigentlich Logik für das Parsen implementiert ist. Im Package `h12.json.parser.implementation.node` finden Sie eine Implementierung eines `JSONElementparsers`, welche Sie in der Aufgabe H3 vervollständigen werden.

Im Package `h12.gui` finden Sie eine bereits fertig implementierte GUI, mit welcher man Zeichnungen anfertigen kann. Die einzelnen darstellbaren Formen finden Sie im Package `h12.gui.shapes` als Subklassen der Klasse `MyShape`. In der Aufgabe H5 werden Sie diese GUI um die Möglichkeit erweitern die Zeichnung in JSON Dateien zu speichern und wieder zu aus diesen zu laden. In der Methode `main` der Klasse `Main` im Package `h12` finden Sie den nötigen Code um die GUI zu starten.

[2]Die Formatierung der Beispiele für `JSONArray` und `JSONObject` entsprechen nicht der in Aufgabe H2 benutzten Formatierung

H1: Vorbereitungen 2 Punkte

Bevor Sie sich aber mit JSON Dateien beschäftigen, müssen Sie noch zunächst noch zwei Hilfsstrukturen implementieren, welche später nützlich im Umgang mit JSON Dateien sein werden. Dazu gehört eine Klasse, die für das Erstellen von `Writern` und `Readern` zuständig ist, sowie einen `LookaheadReader` mit einer `peek()` Methode, der benötigt wird um beim Parsen entscheiden zu können, welcher Ableitungspfad als nächstes genommen werden muss.

H1.1: IOFactory 1 Punkt

Implementieren Sie in der Klasse `FileSystemIOFactory` im Package `h12.ioFactory` die Methoden `createWriter(String)` und `createReader(String)`.

Die Methode `createReader` gibt einen neuen `BufferedReader` zurück, welcher auf einem `FileReader` basiert. Dieser `FileReader` soll mit dem im Parameter übergebenen Dateinamen erstellt werden.
Die Methode `createWriter` funktioniert analog, nur benutzt Sie die Klassen `BufferedWriter` und `FileWriter`.

H1.2: LookaheadReader 1 Punkt

Vervollständigen Sie die Klasse `LookaheadReader` im Package `h12.json`, welche auf dem Objektattribute `reader` basiert.

Die Methode read() funktioniert äquivalent zu der Methode `read()` der Klasse `Reader`.

Der Unterschied zu einem normalen Reader liegt in der Methode `peek()`. Diese gibt genauso wie die Methode `read()` das nächste Zeichen zurück, welches eingelesen werden würde, geht aber nicht intern zum nächsten einzulesenden Zeichen über. D.h. die Rückgabe zwei aufeinanderfolgender Aufrufe der Methode `read()` verändert sich nicht, wenn zwischen diesen die Methode `peek()` aufgerufen wurde. Sie dürfen dafür den Konstruktor der Klasse um weitere Instruktionen erweitern, aber keine entfernen.

In dem JavaDoc der Klasse finden Sie ein Beispiel, wie die Methoden `peek()` und `read()` funktionieren.

H2: Herausschreiben in JSON Dateien 6 Punkte

Damit können Sie nun beginnen sich mit dem Erstellen von JSON Dateien zu beschäftigen. Dafür finden Sie im Package `h12.json.implementation.node` die Klassen `JSONStringNode`, `JSONConstantNode`, `JSONIntegerNode`, `JSONArrayNode`, `JSONObjectNode` und `JSONObjectEntryNode`, welche das entsprechende JSON Element darstellen. Diese Klassen besitzen alle ein Objektattribut, in welchem die in diesem Element gespeicherten Informationen enthalten sind.

Vervollständigen Sie zunächst in der Klasse `JSONNode` die Hilfsmethode `writeIndentation(BufferedWriter, int)`, welche in den übergebenen Writer die im zweiten Parameter übergebene Anzahl an Einrückungen schreibt. Eine Einrückung entspricht dabei zwei Leerzeichen.

Implementieren Sie nun die Methode `write(Bufferedwriter, int)` in den oben genannten Klassen, welche das repräsentierte JSON Element in den übergebenen Writer schreibt. Der erste Parameter ist der `Writer`, in welchen die Daten geschrieben werden sollen. Der zweite Parameter beschreibt die Einrückungen, des zur schreibenden Elementes.

Im folgenden finden Sie eine genauere Beschreibung für die einzelnen Klassen:

- **`JSONNumberNode`** Diese Klasse repräsentiert eine ganze Zahl, welche in dem Objektattribut `number` gespeichert ist. Die `write(BufferedWriter, int)` Methode schreibt die String Repräsentation dieser Zahl in den übergebenen `Writer`.

- **`JSONStringNode`** Diese Klasse repräsentiert einen String, welcher in dem Objektattribut `string` gespeichert ist. Die `write(BufferedWriter, int)` Methode schreibt diesen String, umgeben von Anführungszeichen, in den übergebenen `Writer`.

- **`JSONConstantNode`** Diese Klasse repräsentiert eine der zulässigen Konstanten `true`, `false`, `null`. Diese Konstanten werden in dem Enum `JSONConstants` im Package `h12.json` gespeichert. Die Methode `getSpelling()` des Enums gibt dabei die benutzte Schreibweise in JSON Dateien zurück. In der Klasse `JSONConstantNode` wird die repräsentierte Konstante in dem Objektattribut `constant` gespeichert. Die Methode `write(BufferedWriter, int)` schreibt die, durch die Methode `getSpelling()` vorgegebene, Schreibweise der Konstante in dem Attribut in den Writer.

- **`JSONArrayNode`** Diese Klasse repräsentiert ein Array, dessen Elemente in dem Objektattribut `list` vom Typ `List<JSONElement>` gespeichert sind. Die `write(BufferedWriter, int)` Methode schreibt die Elemente des Arrays umgeben von eckigen Klammern in den `Writer`. Zwei aufeinanderfolgende JSON Elemente werden dabei von einem Komma getrennt. Jedes Element, sowie die schließende Klammer, sollen in einer neuen Zeile stehen. Zu Beginn jeder neuen Zeile soll mit der Methode `writeIndentation(int)` die korrekte Einrückung in den Writer geschrieben werden. Schreiben Sie die einzelnen JSONElemente in den `Writer`, indem Sie deren `write(BufferedWriter, int)` Methode aufrufen. Dabei soll die Einrückung vor den JSON Elemente eins höher sein, als die momentane Einrückung, welche im zweiten Parameter übergeben wird.

- **`JSONObjectEntry`** Diese Klasse repräsentiert ein Objekt Eintrag, dessen Bezeichner in dem Objektattribut `identifier` und der zugehörige Wert in dem Objektattribut `value` gespeichert ist. Die `write(BufferedWriter, int)` Methode schreibt den Bezeichner des Objekt Eintrages, gefolgt von dem zugehörigen Wert, in den übergebenen `Writer`. Diese beiden Werte werden von einem Doppelpunkt und einem Leerzeichen getrennt. Benutzten Sie die `write(BufferedWriter, int)` Methode des Bezeichners und des zugehörigen Wertes um diese in den `Writer` zu schreiben. Sie finden diese Klasse innerhalb der Klasse `JSONObject`.

- **`JSONObjectNode`** Diese Klasse repräsentiert ein Objekt, dessen Objekt Einträge in den Einträgen des Objektattributes `objectEntries` vom Typ `Set<JSONObjectEntry>` gespeichert sind. Die Formatierung für die `write(BufferedWriter, int)` Methode funktioniert äquivalent zur Klasse `JSONArrayNode`. Der Unterschied ist, dass das Objekt von geschweiften Klammern umgeben wird. Schreiben Sie die ObjektEntry Objekte ebenfalls mithilfe deren `write(BufferedWriter, int)` Methode in den übergebenen Writer.

H3: Einlesen von JSON Dateien — 12 Punkte

Als nächstes beschäftigen Sie sich damit die JSON Dateien, die Sie in H2 geschrieben haben, auch wieder einzulesen. Dafür werden Sie in dieser Aufgabe Teile eines rekursiv absteigenden Parsers[3] implementieren.

H3.1: Hilfsmethoden — 4 Punkte

Vervollständigen Sie zunächst die folgenden Hilfsmethoden in der Klasse `JSONElementNodeParser` im Package `h12.json.parser.node`, welche die Interaktion der Parser Methoden mit dem, im entsprechende Objektattribut gespeicherten, `LookaheadReader` vereinfachen. Erinnern Sie sich dafür an die Methode `peek()` aus der H1.2.

[3] siehe: `https://en.wikipedia.org/wiki/Recursive_descent_parser`

- **`skipIndentation()`**: Die Methode `skipIndentation()` liest so lange Zeichen von dem Objektattribut `reader` ein, bis das nächste Zeichen kein Leerzeichen ist. Verwenden Sie zum erkennen von Leerzeichen die Methode `isWhiteSpace(int)` der Klasse `Character`.

- **`acceptIt()`**: Die Methode `acceptIt()` ruft zunächst die Methode `skipIndentation()` auf und ließt danach mittels der Methode `read()` das nächste Zeichen des `LookaheadReaders` ein. Die Rückgabe der Methode ist dieses eingelesene Zeichen.

- **`accept(char)`**: Die Methode `accept(char)` funktioniert äquivalent zur Methode `acceptIt()`, überprüft aber zusätzlich noch, ob das eingelesene Zeichen gleich dem übergebenen Parameter ist und wirft eine `UnexpectedCharacterException`, falls dies nicht der Fall ist. Wenn das Ende des `Readers` bereits erreicht wurde, wird stattdessen eine `BadFileEndingException` geworfen.

- **`peek()`**: Die Methode `peek()` ruft zunächst die Methode `skipIndentation()` und liefert dann das Ergebnis der `peek()` Methode des `LookaheadReaders` zurück.

- **`checkEndOfFile()`**: Die Methode `checkEndOfFile()` ruft zunächst die Methode `skipIndentation` und überprüft danach, ob das Ende des `LookaheadReader` erreicht wurde. Falls dies nicht der Fall ist, wird eine `BadFileEndingException` geworfen.

- **`readUntil(Predicate)`** Die Methode `readUntil(Predicate)` liest solange Zeichen von dem `LookaheadReader` ein, bis das übergebene Prädikat für das nächste Zeichen, welches der `LookaheadReader` zurückliefern würde, `true` zurückgibt. Zum Schluss gibt die Methode alle eingelesenen Zeichen in einem String zurück. Das Zeichen, für welches das Prädikat `true` zurückgibt, soll dabei weder mit der Methode `read()` des `LookaheadReaders` eingelesen werden, noch in der Rückgabe enthalten sein. Die Methode überspringt dabei Leerzeichen nicht, sondern gibt diese ebenfalls zurück. Falls das Ende des `LookaheadReaders` erreicht wurde, bevor das Prädikat für ein Zeichen `true` zurückgegeben hat, wird eine `BadFileEndingException` geworfen.

H3.2: Erkennen der JSON Elemente — 1 Punkt

Implementieren Sie nun in der selben Klasse, wie die Hilfsmethoden, die Methode `parse()`. Diese Methode ist dafür zuständig zu erkennen, welche Art von JSON Element als nächstes eingelesen wird und danach die korrekte Methode aufzurufen. Dafür ruft `parse()` anhand der folgenden Tabelle die `parse()` Methode der zuständigen Klasse über die bereits vorhandenen Objektattribute auf. Die Methode liest dabei kein Zeichen mithilfe der Methode `read()` des LookaheadReaders ein.

Nächstes Zeichen	JSON Element	Zuständige Klasse
`'{'`	Objekt	`JSONObjectNodeParser`
`'['`	Array	`JSONArrayNodeParser`
`'"'`	String	`JSONStringNodeParser`
`'+', '-', '.', '0' - '9'`	Zahl	`JSONNumberNodeParser`
Restliche Zeichen	Konstante	`JSONConstantNodeParser`

Tabelle 2: Übersicht der Parser Klassen

Verwenden Sie zum Erkennen des nächsten Zeichen die in H3.1 implementierte Methode `peek()`. Falls kein nächstes Zeichen vorhanden ist, weil das Ende des `Readers` bereits erreicht ist, geben Sie `null` zurück.

H3.3: Die eigentlichen Parser — 7 Punkte

Implementieren Sie zuletzt noch die `parse()` Methoden der oben aufgezählten Parser Klassen, sowie der Klasse `JSONObjectEntryNodeParser`, in denen die eigentliche Logik enthalten ist. Diese lesen jedes Zeichen ein, welches zu dem JSON Element gehört, das im `LookaheadReader` als nächstes folgt. Die Rückgabe ist der Inhalt dieses JSON Elementes als Objekt der zugehörigen Subklasse von `JSONElementNode`. Jeder der Parser Klassen besitzt ein Objektattribut `parser` vom Typ `JSONElementNodeParser`, mit welchem Sie auf die zuvor implementierten Hilfsmethoden zugreifen können. Dadurch müssen Sie keine Leerzeichen berücksichtigen. Die `parse()` Methoden funktionieren dabei wie folgt:

- Wenn das nächste erwartete Zeichen bereits feststeht, wie z.B. bei einer öffnenden Klammer, benutzen Sie die Methode `accept(char)` um das nächste Zeichen einzulesen und zu validieren, indem Sie der Methode das erwartete Zeichen übergeben.
- Wenn als nächstes ein beliebiges JSON Element erwartet wird, wie z.B. innerhalb eines `JSONArray`, benutzten Sie die `parse()` Methode des `parser` Objektattributes um dieses JSON Element vollständig einzulesen. Falls diese Methode `null` zurückgibt, wird eine `BadFileEndingException` geworfen.
- Wenn als nächste eine Zeichenkette unbekannter Länge erwartet wird, wie z.B. bei Konstanten, verwenden Sie die Methode `readUntil(Predicate)`.
- Ein `JSONConstantNodeParser` liest alle folgenden Buchstaben ein und wirft eine `InvalidConstantException`, wenn die eingelesene Zeichenkette nicht der Schreibweise einer validen Konstanten entspricht, wie im Enum `JSONConstants` definiert. Verwenden Sie zum Erkennen von Buchstaben die Methode `isLetter(int)` der Klasse `Character`
- Ein `JSONNumberNodeParser` liest so lange Zeichen ein, bis dass nächste Zeichen kein `'+'`, `'-'`, `'.'` oder eine Ziffer ist. Danach wird, abhängig davon, ob in der eingelesenen Sequenz ein Punkt ist, versucht diese mit der Methode `parseInt(String)` der Klasse `Integer`, bzw mit der Methode `parseDouble(String)` der Klasse `Double` in eine Zahl umzuwandeln. Falls dabei eine `NumberFormatException` geworfen wird, soll diese gefangen werden und stattdessen eine `InvalidNumberException` mit der eingelesenen Zeichenkette als Botschaft geworfen werden. Verwenden Sie zum Erkennen von Ziffern die Methode `isDigit(int)` der Klasse `Character`
- Ein `JSONObjectEntryNodeParser` benutzt den im `JSONElementNodeParser` gespeicherten `JSONStringParser` um den Bezeichner des Objekt Eintrages zu parsen. Auf diesen können Sie über die entsprechende Getter Methode zugreifen.
- Ein `JSONObjectNodeParser` benutzt den `JSONObjectEntryNodeParser`, der im `JSONElementNodeParser` hinterlegt ist, um die einzelnen Objekt Einträge zu parsen. Auf diesen können Sie über die entsprechende Getter Methode zugreifen.
- Falls bei einem `JSONObjectNodeParser` oder einem `JSONArrayNodeParser` auf ein trennendes Komma eine schließende Klammer (`'}'` bzw, `']'`) folgt, wird eine `TrailingCommaException` geworfen.
- Falls ein `JSONObjectNodeParser` auf ein `ObjectEntry` trifft, welches mehrfach in dem selben Objekt verwendet wird, wird nur das letzte Vorkommen dieses Strings berücksichtigt.[4]

In der Klasse `JSONStringNodeParser` finden Sie bereits eine beispielhafte Implementierung eines solchen Parsers.

Hinweis:

In den Parser Klassen finden Sie im JavaDoc jeweils ein Beispiel, wie ein eingelesene Teilsequenz einer JSON Datei und die zugehörige Ausgabe des Parser aussieht.

[4] Dieses Verhalten wird bereits durch die Methode `add(E)` des Interfaces `Set<E>` erzeugt

H4: JSON Handler — 6 Punkte

Damit haben Sie nun die Möglichkeit JSON Dateien zu erstellen und wieder einzulesen, aber noch keine Möglichkeit diese Funktionalität von außen einfach zu verwenden. Dafür werden Sie in dieser Aufgabe zwei Klassen implementieren, die dies ermöglichen.

H4.1: JSON Parser — 2 Punkte

Implementieren Sie die Methode `parse()` der Klasse `JSONParser` im Package `h12.json.parser`. Diese benutzt die `parse` Methode des Objektattribut `elementParser` um den Inhalt des dort hinterlegten Reader zu parsen. Bevor das geparste JSON Element zurückgegeben wird, wird noch die Methode `checkEndOfFile()` aufgerufen, um zu überprüfen, ob das Ende des Readers korrekt erreicht wurde. Falls dabei eine `IOException` geworfen wird, fangen Sie diese und werfen stattdessen eine `JSONParseException` mit der selben Botschaft, wie die gefangene Exception.

H4.2: JSON — 4 Punkte

Implementieren Sie als nächstes die beiden Methoden `parse(String)` und `write(String)` in der Klasse `JSON` im Package `h12.json`.

Die Methode `write(String, JSONElement)` überprüft zunächst, ob das Objektattribut `IOFactory` Schreiben unterstützt und wirft eine `JSONWriteException` mit der Botschaft `"The current ioFactory does not support writing!"`, falls dies nicht der Fall ist. Ansonsten erstellt die Methode mithilfe der `ioFactory` und dem übergebenen Parameter einen neuen `BufferedWriter` und ruft mit diesem die `write(BufferedWriter, int)` Methode des zweiten Parameters auf. Die initiale Einrückung ist 0. Falls beim Schreiben eine `IOException` geworfen wird, fangen Sie diese und werfen stattdessen eine `JSONWriteException` mit der selben Botschaft, wie die gefangene Exception.

Die Methode `parse(String)` überprüft zunächst, ob das Objektattribut `IOFactory` Lesen unterstützt und wirft eine `JSONWriteException` mit der Botschaft `"The current ioFactory does not support reading!"`, falls dies nicht der Fall ist. Ansonsten erstellt die Methode mithilfe dem Objektattribut `ioFactory` einen `BufferedReader` und erzeugt einen `LookaheadReader`, welcher auf diesem `BufferedReader` basiert. Danach wird mithilfe des Objektattributes `parserFactory` ein neues `JSONParser` Objekt erstellt, welches den soeben erstellten `LookaheadReader` benutzt. Rufen Sie zum Schluss die Methode `parse()` des `JSONParsers` auf, um den Inhalt der Datei zu parsen und geben Sie die Rückgabe der Methode zurück. Falls beim Lesen eine `IOException` geworfen wird, fangen Sie diese und werfen stattdessen eine `JSONParseException` mit der selben Botschaft, wie die gefangene Exception.

> **Verbindliche Anforderung:**
>
> Alle `Reader` und `Writer` müssen innerhalb eines try-with-ressource Blockes erstellt werden.

H5: Speichern und Einlesen von Zeichnungen — 12 Punkte

In dieser letzten Aufgabe werden Sie sich mit einer praktischen Anwendung für JSON Dateien beschäftigen. Im Package `h12.gui` finden Sie eine bereits implementierte GUI für ein Zeichenprogramm, mit welchem sich die verschiedenen Formen im Package `h12.gui.shapes` zeichnen lassen. Sie werden nun diese um die Funktionalität, Zeichnungen in JSON Dateien zu speichern und wieder zu Laden, erweitern.

H5.1: Formen als JSON Dateien darstellen — 1 Punkt

Implementieren Sie dafür als erstes in der Subklasse `MyPolygon` von `MyShape` im Package `h12.gui.shapes` die Methode `toJSON()`. Diese Methode konvertiert das dargestellte Polygon in eine `JSONObjectNode`. Die jeweiligen Einträge, die in dieser `JSONObjectNode` vorhanden sein sollen, finden Sie in dem JavaDoc der Methode. Auf die zugehörigen Werte können Sie über die gleichnamigen Objektattribute zugreifen. Der Eintrag `"name"` entspricht der Rückgabe der Methode `getSpelling()`, aufgerufen auf der Klassenkonstanten `TYPE` vom statischen Typ `ShapeType`. Verwenden Sie die bereits implementierte Methode `toJSON(Color)` der Klasse `ColorHelper` um ein Objekt vom Typ `java.awt.Color` in eine `JSONArrayNode` zu konvertieren. Die einzelnen `JSONElemente` können Sie über die `of(...)` Methoden der Interfaces, oder über die Konstruktoren der Subklassen von `JSONElementNode`, erstellen.

H5.2: Formen aus JSON Dateien einlesen — 3 Punkte

Implementieren Sie nun in der Klasse `MyShape` im Package `h12.gui.shapes` die Methode `fromJSON(JSONElement)`, welche das Gegenstück zur Methode `toJSON()` darstellt und ein Objekt vom Typ `MyShape` zurückgibt, welches die in dem übergebenen `JSONObject` enthaltene Eigenschaften besitzt. Fragen Sie dafür zunächst den im Eintrag `"name"` enthaltenen String ab und wandeln Sie diesen dann mit der Methode `fromString(String)` des Enums `ShapeType` in die zugehörige `ShapeType` Konstante um. Falls die Rückgabe dieser Methode `null` ist, werfen Sie eine `JSONParseException` mit der Botschaft `"Invalid shape type: <shapeType>!"`, wobei Sie den Substring `<shapeType>` mit dem String, welcher aus dem JSON Objekt eingelesen wurde, ersetzen. Ansonsten rufen Sie, abhängig von dem Wert der Konstanten, die entsprechende Methode des Objektattributes `jsonToShapeConverter` auf und geben Sie die Rückgabe dieser Methode zurück. Beachten Sie, dass es für die Formen `MyTriangle` und `StraightLine` keine entsprechende Methode gibt, da diese als Polygone dargestellt werden. Wenn dennoch die zu eine dieser beiden Klassen gehörige Konstante eingelesen wird, werfen Sie eine `JSONParseException` mit der selben Botschaft, wie im Fall, dass `null` zurückgegeben wird. Falls dabei eine `UnsupportedOperationException` oder `NoSuchElementException` geworfen wird, fangen sie diese und werfen stattdessen eine `JSONParseException` mit der selben Botschaft, wie die gefangene Exception.

Vervollständigen Sie zusätzlich noch in der Klasse `JSONToShapeConverter` die Methode `polygonFromJSON(JSONElement)`. Diese liest, genauso wie die anderen in der Klasse bereits implementierten Methoden, die Einträge des übergebenen `JSONElements` ein und gibt ein Objekt des Types `MyPolygon`, welches die eingelesenen Information enthält, zurück. Wenn bei diesem Prozess eine `UnsupportedOperationException` oder `NoSuchElementException` geworfen wird, fangen Sie diese und werfen Sie stattdessen eine `JSONParseException` mit der Botschaft `"Invalid MyShape format!"`. Die im `JSONElement` erwarteten Einträge finden Sie im JavaDoc der Methode `toJSON` der Klasse `MyPolygon`.

> **Hinweis:**
>
> Sie können die Informationen, die in den einzelnen JSON Elementen gespeichert sind mit dem Getter Methoden aus dem Interface `JSONElement` abfragen. Da dort bereits alle Methoden definiert sind, müssen Sie die einzelnen Elemente nicht downcasten. Um Zahlenwerte abzufragen, verwenden Sie die Methode `getInteger()`. Falls das JSON Element nicht die abgefragte Information enthält (z.B. wenn `getInteger` auf einem `JSONArray` aufgerufen wird), wird eine `UnsupportedOperationException` geworfen. Wenn die Methode `getEntry(String)` auf einem `JSONObject` aufgerufen wird, welches den übergebenen String nicht als Bezeichner enthält, wird eine `NoSuchElementException` geworfen.

H5.3: Validieren von ausgewählten Dateien — 1 Punkt

Als nächstes implementieren Sie in der Klasse `FileOperationHandler` im Package `h12.gui.components` die Methode `checkFileName(String)`. Die Methode überprüft, ob die vom Benutzer ausgewählte Datei eine valide JSON Datei ist. Wenn der übergebene Dateiname `null` ist, rufen Sie die Methode `showErrorDialog(String)`

mit der Fehlermeldung `"No file selected!"` auf. Wenn der Dateiname nicht mit `".json"` endet, rufen Sie die Methode `showErrorDialog(String)` mit `"Invalid file type!"` auf. Geben Sie in beiden Fällen `false` zurück und geben Sie `true` zurück, falls keiner dieser Fälle eintritt.

H5.4: Speichern von Zeichnungen in JSON Dateien — 3 Punkte

Nun können Sie das Speichern einer erstellten Zeichnung in einer JSON Datei implementieren.

Implementieren Sie dazu zunächst die Methode `canvasToJSONObject()` in der Klasse `SaveCanvasHandler` im Package `h12.gui.components`. Diese erstellt ein `JSONObject`, welches die beiden Einträge `"background"` und `"shapes"` hat. im Eintrag `"background"` ist die Hintergrundfarbe der Zeichnung enthalten und im Eintrag `"shapes"` ist ein `JSONNodeArray` enthalten, welches aus allen Formen, die in der Zeichnung enthalten sind, besteht. Auf die Hintergrundfarbe und die Formen können Sie über die entsprechenden Objektattribute zugreifen. Benutzten Sie zum Konvertieren der Hintergrundfarbe in ein JSON Element die Methoden `ColorHelper.toJSON(Color)` und zum Konvertieren der einzelnen `MyShapes` Objekte die Methode `toJSON()` der Klasse `MyShape`.

Implementieren Sie nun ebenfalls in der Klasse `SaveCanvasHandler` die Methode `save()`, welche dafür zuständig ist den Benutzer eine Datei auswählen zu lassen und dann die momentane Zeichnung in dieser Datei zu speichern. Rufen Sie dafür zunächst die Methode `selectFile(String)` auf, um den Benutzer eine Datei auswählen zu lassen. Übergeben Sie dieser Methode das Arbeitsverzeichnis der Anwendung, welches Sie mit dem Befehl `System.getProperty("user.dir")` erhalten. Die Rückgabe der Methode `selectFile(String)` ist der Name der ausgewählten Datei. Verifizieren Sie danach mit der zuvor geschriebenen Methode `checkFileName(String)`, ob die ausgewählte Datei valide ist, indem Sie die Methode mit dem Namen der ausgewählten Datei aufrufen. Falls dies nicht der Fall ist, tut die Methode nichts weiteres. Falls die eingelesene Datei zulässig ist, setzen Sie mit der Methode `setIOFactory(IOFactory)` des Objektattributes `json` die benutzte `IOFactory` auf eine `FileSystemIOFactory` und speichern Sie die Zeichnung mithilfe der `write(String, JSONElement)` Methode des `json` Objektes. Diese kriegt als ersten Parameter den Namen der ausgewählten Datei und als zweiten Parameter die Rückgabe der Methode `canvasToJSONObject()`. Rufen Sie zum Schluss die Methode `showSuccessDialog(String)` auf, welche den Namen der ausgewählten Datei übergeben kriegt. Falls beim Schreiben eine `JSONWriteException` geworfen wird, fangen Sie diese und rufen Sie stattdessen die Methode `showErrorDialog(String)` auf mit der Botschaft der gefangenen `JSONWriteException` als Wert des Parameters.

H5.5: Laden von Zeichnungen aus JSON Dateien — 4 Punkte

Zum Schluss müssen Sie noch das Laden einer gespeicherten Zeichnung implementieren.

Implementieren Sie dazu zunächst die Methode `canvasFromJSONObject(JSONElement)` in der Klasse `LoadCanvasHandler` im Package `h12.gui.components`. Diese liest den Inhalt der Einträge `"background"` und `"shapes"` des übergebenen `JSONElements` ein und speichert ihn in den Objektattributen `backgroundColor` und `shapes`. Für die genaue Formatierung der Einträge siehe H5.4. Falls das übergebene Element `null` ist, werfen Sie eine `JSONParseException` mit der Botschaft `"The given File is empty!"`. Benutzten Sie zum konvertieren der eingelesenen `JSONArrays`, die in den übergebenen Element gespeichert sind, die Methoden `ColorHelper.fromJSON(Color)` und `MyShape.fromJSON(JSONElement)`. Auf den Inhalt der `JSONArrays` können Sie die Methode `getArray()` aus dem Interface `JSONElement` verwenden. Falls dabei eine `UnsupportedOperationException` oder `NoSuchElementException` geworfen wird, fangen Sie diese und werfen stattdessen eine `JSONParseException` mit der Botschaft `"Invalid MyShape format!"`.

Zum Schluss müssen Sie noch in der selben Klasse die Methode `load()` implementieren, welche dafür zuständig ist den Benutzer eine Datei auswählen zu lassen, den Inhalt dieser Datei einzulesen und die zugehörige Zeichnung anzuzeigen.

Rufen Sie dafür als erstes die Methode `selectFile(String)` der Superklasse auf, welche den Namen der ausgewählten Datei zurückliefert. Übergeben Sie dieser Methode das Arbeitsverzeichnis der Anwendung, welches Sie mit dem Befehl `System.getProperty("user.dir")` erhalten. Verifizieren Sie danach mit der zuvor geschriebenen Methode `checkFileName`, ob die ausgewählte Datei valide ist. Falls dies nicht der Fall ist, tut die Methode nichts weiteres. Falls die eingelesene Datei zulässig ist, setzen Sie mit der Methode `setIOFactory(IOFactory)` des Objektattributes `json` die benutzte `IOFactory` auf eine `FileSystemIOFactory` und lesen Sie dann den Inhalt der Datei mit der Methode `parse(String)` des Objektattributes `json` ein, indem Sie dieser Methode den Namen der ausgewählten Datei übergeben. Rufen Sie danach die zuvor implementierte Methode `canvasFromObject(JSONElement)` mit dem geparsten Inhalt der Datei auf und zum Schluss die Methode `setupNewFrame()`. Falls beim Parsen eine `JSONParseException` geworfen wird, fangen Sie diese und rufen die Methode `showErrorDialog(String)` mit der Botschaft der gefangenen Exception auf.