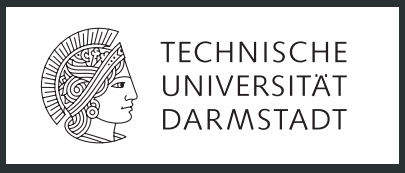
Funktionale und objektorientierte Programmierkonzepte
Übungsblatt 12



Prof. Karsten Weihe

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

Hausübung 12 — Gesamt: 34 Punkte

File-IO anhand von JSON Dateien

Beachten Sie die Seite *Verbindliche Anforderungen für alle Abgaben* in unserem 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 in der Vorlage relevanten Verzeichnisse sind `src/main/java/h12` und `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 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.

Eine JSON Datei besteht aus einem 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. Ein Objekt Eintrag besteht aus einem String, welcher einem beliebigen JSON Element zugeordnet wird. Diese werden mit einem `':'` getrennt.

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

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

4. **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.

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

6. **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.

Eine JSON Datei sieht beispielsweise wie folgt aus:

JSON Example

```json
{
  "ModuleName": "Funktionale und objektorientierte Programmierkonzepte",
  "Dozent": "Karsten Weihe",
  "CP": 10.0,
  "WS": true,
  "Students": [
    {
      "firstName": "Max",
      "lastName": "Mustermann"
    },
    {
      "firstName": "Erika",
      "lastName": "MusterFrau"
    }
  ],
  "grades": null
}
```

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

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 write Methode, mit welcher die in ihm gespeicherten Informationen in eine JSON Datei geschrieben werden kann. Diese werden Sie in 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[1].

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}`	`Map<JSONString, JSONElement> getObjectEntries` `JSONElement getEntry(String)`

Tabelle 1: Übersicht der JSON Element Interfaces

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.node finden Sie eine Implementierung dieses `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. Diese 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.

[1] 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.

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` verweist auf die übergebene Ressource.
Die Methode `createWriter` funktioniert analog, nur benutzt Sie die Klassen `BufferedWriter` und `FileWriter`.

H1.2: LookaheadReader 1 Punkt

Implementieren Sie in der 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 nächstes 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 erweiter, aber keine entfernen.

H2: Herausschreiben in JSON Dateien 6 Punkte

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

Vervollständigen Sie zunächst in der Klasse `JSONNode` die Hilfsmethode `writeIndentation(BufferWriter, int)`, welche in den übergebenen Writer die im zweiten Parameter übergebene Anzahl an Einrückungen schreibt. Eine Einrückung entspricht dabei 2 Leerzeichen.

Implementieren Sie nun die Methode `write(Bufferwriter, 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` Methode schreibt die String Repräsentation dieser Zahl in den übergebenen `Writer`.

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

- **`JSONConstantNode`** Diese Klasse repräsentiert die zulässigen Konstanten `true`, `false`, `null`. Diese Konstanten werden in dem Enum `JSONConstants` im Package h12.json gespeichert. Die Methode `getSpelling()` 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` schreibt die in dem Attribut gespeicherte Konstante Schreibweise der Konstante in den Writer.

- **`JSONArrayNode`** Diese Klasse repräsentiert ein Array, dessen Elemente in dem Objektattribut `list` vom Typ `List<JSONElement>` gespeichert sind. Die `write` 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 Zeilen soll mit der Methode `writeIndentation` die korrekte Einrückung in den Writer geschrieben werden. Schreiben Sie die einzelnen JSONElemente dabei mithilfe deren `write` Methode in den `Writer`. Dabei soll die Einrückung der JSON Elemente eins höher sein, als die momentane Einrückung, welche im zweiten Parameter übergeben wird.

- **`JSONObjectNode`** Diese Klasse repräsentiert ein Objekt, dessen Objekt Einträge in den Einträgen des Objektattributes `objectEntries` vom Typ `Map<JSONString, JSONElement>` gespeichert sind. Die `write` Methode funktioniert äquivalent zur Klasse `JSONArrayNode` mit dem Unterschied, dass das Objekt vin geschweiften Klammern umgeben wird, sowie, dass vor den einzelnen Elemente in der selben Zeile der zugehörige String steht. Zwischen dem String und dem JSON Element steht ein Doppelpunkt gefolgt von einem Leerzeichen.

> **Verbindliche Anforderung:**
>
> Zeilenumbrüche werden mit `'\n'` dargestellt.

H3: Einlesen von JSON Dateien — 10 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 einen rekursiv absteigenden Parser[1] implementieren.

H3.1: Hilfsmethoden — 3 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.

- **`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(int)`**: Die Methode `accept` 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.

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

- **`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.

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

- **`readUntil(Predicate)`** Die Methode `readUntil` 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 `LookaheadReader` eingelesen werden, noch in der Rückgabe enthalten sein.

H3.2: Erkennen der JSON Elemente — 1 Punkt

Implementieren Sie nun in der selben Klasse, wie die Hilfsmethoden, die Methode `parse(LookaheadReader)`. 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	`JSONIntegerNodeParser`
Restliche Zeichen	Konstante	`JSONConstantNodeParser`

Tabelle 2: Übersicht der Parser Klassen

Falls kein Nächstes Zeichen vorhanden ist, weil das Ende des Readers bereits erreicht ist, geben Sie `null` zurück.

H3.3: Die eigentlichen parser — 6 Punkte

Implementieren Sie zuletzt noch die `parse` Methoden der oben aufgezählten Parser Klassen, 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` zurück. Jeder der Parser Klassen besitzen ein Objektattribut 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` um das nächste Zeichen einzulesen und zu validieren.

- Wenn als nächstes ein JSON Element erwartet wird, benutzten Sie die `parse` Methode des `parser` Objektattributes um dieses JSON Element vollständig einzulesen.

- Wenn als nächste eine Zeichenkette unbekannter Länge erwartet wird, wie z.B. bei Konstanten, verwenden Sie die Methode `readUntil`.

- 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 Botschaft der empfangenen Exception geworfen werden.

- Falls eine `JSONObjectNodeParser` auf einen String trifft, welcher mehrfach in dem selben Objekt verwendet wird, wird nur das letzte Vorkommen dieses Strings berücksichtigt.

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.

H4: JSON Handler — 3 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 — 1 Punkt

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 — 2 Punkte

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

Die Methode `write` 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` Methode des `root` Objektattributes 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. Wenn `root` `null` ist, tut die Methode nichts, außer potenziell eine Exception zu werfen, wenn die `IOFactory` kein Schreiben unterstützt.

Die Methode `parse` 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 eine mithilfe des Objektattributes `parserFactory` ein neues `JSONParser` Objekt erstellt, welches den soeben erstellten `LookaheadReader` benutzt. Zum Schluss wird mit der Methode `parser()` des `JSONParsers` der Inhalt der Datei geparst und in dem `root` Objektattribut gespeichert. 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` Objekte müssen innerhalb eines try-with-ressource Blockes erstellt werden.

H5: Speichern und Einlesen von Zeichnungen 13 Punkte

Als 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 3 Punkte

Implementieren Sie als in den Subklassen `MyCircle`, `MyRectangle`, `MyPolygon`, und `CustomLine` von `MyShape` im Package h12.gui.shapes die Methode `toJSON()`. Diese Methoden konvertieren die dargestellte Form in eine `JSONObjectNode`. Die jeweiligen Einträge, die in diesem Objekt vorhanden sein sollen, finden Sie in den JavaDocs zu den Methoden. 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`. Verwenden Sie die bereits implementierte Methode `toJSON(Color)` der Klasse `ColorHelper` um Farben in eine `JSONArrayNode` zu konvertieren.

H5.2: Formen aus JSON Dateien einlesen 4 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, welche 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, ersetzten. Sonst lesen Sie, abhängig von dem Wert der Konstante, die erwarteten Werte (vgl. H5.1) ein und geben Sie ein Passendes Objekt der Subklassen von `MyShape` mit den entsprechenden Eigenschaften zurück. Benutzten Sie den Konstruktor der Subklasse um das zurückgegebene Objekt zu erzeugen. Verwenden Sie die Methode `fromJSON(JSONElement)` der Klasse `ColorHelper` um ein `JSONElement` in eine Farbe zu konvertieren. Beachten Sie dabei, dass Sie für die `ShapeType` Konstanten `StraightLine` und `MyTriangle` keine Funktionalität implementieren müssen, da diese als Polygone gespeichert werden. Falls doch eine solche Konstante eingelesen wird, werfen Sie eine `JSONParseException` mit der selben Botschaft, wie im Fall `null`. Wenn bei diesem Prozess eine `UnsupportedOperationException` geworfen wird, fangen Sie diese und werfen Sie stattdessen eine `JSONParseException` mit der Botschaft `"Invalid MyShape format!"`.

> **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`), wird eine `UnsupportedOperationException` geworfen.

H5.3: Validieren von ausgewählten Dateien — 1 Punkt

Als nächste implementieren Sie in der Klasse `FileOperationHandler` im Package h12.gui.components die Methode `checkFileName`. Die Methode überprüft, ob die vom Benutzter 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`, falls keiner dieser Fälle eintritt.

H5.4: Speichern von Zeichnungen in JSON Dateien — 2 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 im Eintrag `"background"` die Hintergrundfarbe der Zeichnung enthält und im Eintrag `"shapes"` ein `JSONArray`, welches alle Formen auf der Zeichnung enthält, und gibt dieses JSON Objekt zurück. 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` die Methode `toJSON()` der Klasse `MyShape`.

Danach können Sie nun ebenfalls in der Klasse `SaveCanvasHandler` die Methode `save()` implementieren, welche dafür zuständig ist den Benutzter 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 Benutzter eine Datei auswählen zu lassen. Übergeben Sie dieser Methode das Arbeitsverzeichnis der Anwendung, welches Sie mit `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, setzten Sie mit der Methode `setIOFactory` der Klasse `JSON` die benutzte `IOFactory` auf eine `FileSystemIOFactory` und speichern Sie die Zeichnung mithilfe der `write` Methode eines `JSON` Objektes, welches mit der Rückgabe der `canvasToJSONObject` Methode erstellt wurde. Rufen Sie zum Schluss die Methode `showSuccessDialog(String)` auf, welche den Namen der ausgewählten Datei übergeben kriegt. Falls beim Schreiben eine `JSONWriteException` geworfen werden fangen Sie diese und rufen Sie stattdessen die Methode `showErrorDialog(String)` auf und übergeben Sie ihr die Botschaft der gefangenen `JSONWriteException`.

H5.5: Laden von Zeichnungen aus JSON Dateien — 3 Punkte

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

Implementieren Sie dazu zunächst die Methode `canvasFromJSONObject()` in der Klasse `LoadCanvasHandler` im Package h12.gui.components. Diese liest den Inhalt der Einträge `"background"` und `"shapes"` ein und

speichert Sie in den Objektattributen `backgroundColor` und `shapes`. Für die genaue Formatierung der Einträge siehe H5.4. Auf das root Element der eingelesenen Datei können Sie über die Methode `getRoot()` des Objektattributes `json` zugreifen. Falls das zurückgegebene root Element `null` ist, werfen Sie eine `JSONParseException` mit der Botschaft `"The given File is empty!"`. Benutzten Sie zum konvertieren der eingelesenen JSON Arrays die Methoden `ColorHelper.fromJSON(Color)` und `MyShape.fromJSON()`. Falls dabei eine `UnsupportedOperationException` geworfen wird, fangen Sie diese und werfen stattdessen eine `JSONParseException`, welche die selbe Botschaft, wie die gefangene Exception enthält.

Zum Schluss müssen Sie noch in der selben Klasse die Methode `load()` implementieren, welche dafür zuständig ist den Benutzter eine Datei auswählen zu lassen und dann den Inhalt dieser Datei einzulesen um dann 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 `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, setzten Sie mit der Methode `setIOFactory` der Klasse `JSON` die benutzte `IOFactory` auf eine `FileSystemIOFactory` und lesen Sie dann den Inhalt der Datei mit der Methode `parse` des Objektattributes `json` ein. Rufen Sie danach die zuvor implementierte Methode `canvasFromObject` 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.