UNIKASSEL
VERSITÄT

SOFTWARE ENGINEERING
UNIVERSITÄT KASSEL

Persistenz und Serialisierung

Jens Kosiol

Wintersemester 23/24

(teilweise basierend auf Folien von Prof. Dr. Gabriele Taentzer)

Inhalt

- Grundlegende Dateneingabe und Datenausgabe in Java
- Persistenz von strukturierten Daten (Objekten)
  - Serialisierung
  - JSON als Speicherformat
  - Jackson als JSON Processor

Persistenz

Unter Persistenz versteht man die Fähigkeit, Daten, die zur Laufzeit eines Programms entstehen, dauerhaft zu speichern, um sie zu einem späteren Zeitpunkt wieder benutzen zu können.

Spezieller Fokus in der objektorientierten Programmierung: Objekte sollen so gespeichert werden, dass sie zu einem späteren Zeitpunkt (und unter Umständen in einer anderen Laufzeitumgebung) wiederhergestellt werden können.

Grundsätzliches zur Ein- und Ausgabe in Dateien

- Häufig wird ein separates Paket für die Ein- und Ausgabe erstellt (Schichtenarchitektur).
- Für Ein- und Ausgabe sollte je eine Serviceklasse (häufig als Singletonklassen) erstellt werden.
- Die interne Datenstruktur bestimmt den Aufbau der Ausgabeklasse: in der Serviceklasse eine separate Methode für jede Datenklasse.
- Das Speicherformat bestimmt den Aufbau der Eingabeklasse: in der Serviceklasse eine separate Methode für jede Einheit im Datenformat.
- Exceptions müssen gefangen und behandelt werden.
- Java stellt Bibliotheken für grundlegende Funktionen zur Verfügung (z.B. java.io).

Ausgabe in eine Datei

- Verwenden z.B. der Bibliothek java.io
- Aufbau:
  - Datei zum Schreiben öffnen
  - Writer erzeugen und in die benötigte Datei lenken
    - IOExceptions abfangen
    - Writer zum Schluss schließen

Einsetzbare Klassen:

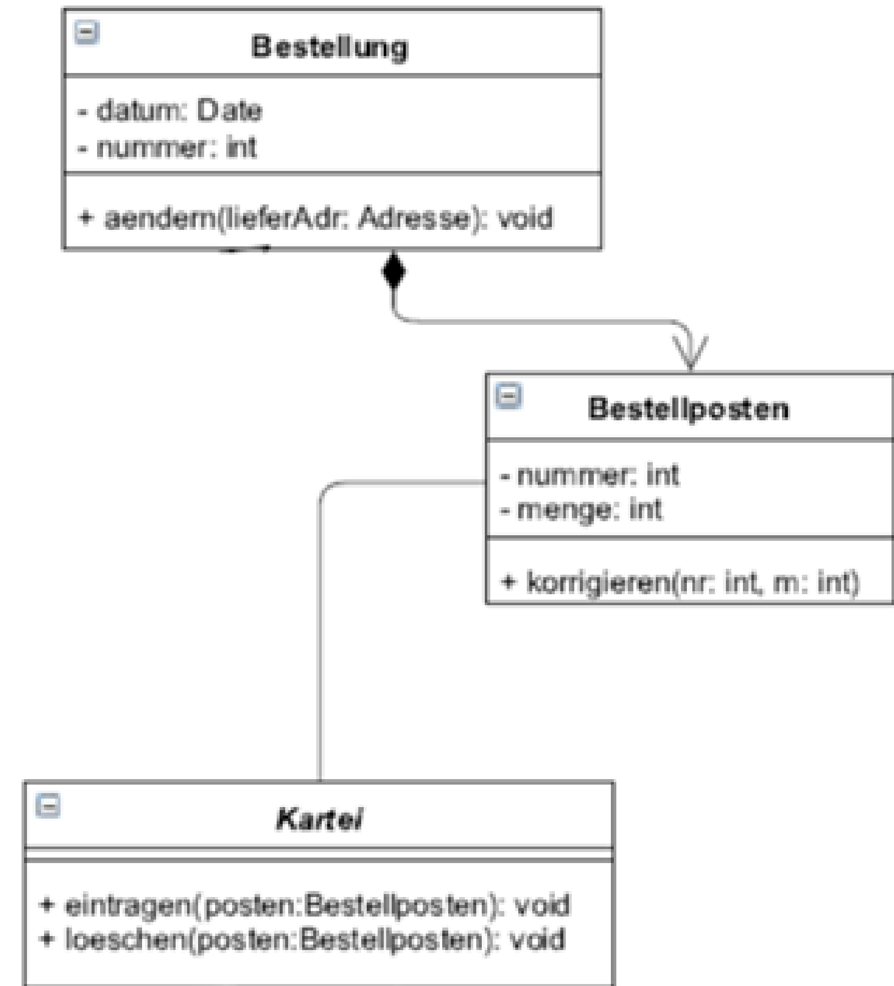
- java.io.FileWriter zum Schreiben von Text
- java.io.BufferedWriter zum Wrappen von FileWritern (aus Effizienzgründen); Methoden:
  - write() zum Schreiben eines Buchstabens oder Strings
  - newLine() zum Schreiben eines Zeilenseparators
  - flush() zum Leeren des Buffers
  - close() zum Schließen des Stroms (ruft flush() auf)

Wiederholung: Bestellsystembeispiel

```java
public class Bestellposten {
   private int nummer;
   private int menge;
   public void korrigieren(int nr, int m) {
   //...
   }
}
```

```java
public class Bestellung {
  private Date datum;
  private int nummer;
  public List<Bestellposten> bestellposten;
  //...
  b = new Bestellposten(nummer,menge);
  //…
}
```

```java
public abstract class Kartei {
  //...
  public void eintragen(Bestellposten posten){}
  public void loeschen(Bestellposten posten){}
}
```



Beispiel: Ausgabe in eine Datei


```java
import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;

import data.Bestellposten;
import data.Kartei;

public class KarteiAusgabe {
    private static KarteiAusgabe instanz = null;

    private KarteiAusgabe() {}

    public static KarteiAusgabe gibInstanz(){
    if (instanz == null){
        instanz = new KarteiAusgabe();
        }
        return instanz;
    }
```

```java
private void schreibeBestellposten(Bestellposten b,BufferedWriter out){
    try{
        out.write( str: b.gibNummer() + " " + b.gibProdukt().gibName() + " " + b.gibMenge());
        out.newLine();
    }
    catch (IOException e){
        System.out.println(e.getMessage());
    }
}
```

```java
public void schreibeKartei(Kartei kartei,String dateiname){
    try{
        BufferedWriter out = new BufferedWriter(new FileWriter(dateiname));
        for(Bestellposten p:kartei.gibListe()){
            schreibeBestellposten(p,out);
        }
        out.close();
    }
    catch (IOException e){
        System.out.println(e.getMessage());
    }
}
```

Einlesen aus einer Datei

- Verwenden z.B. der Bibliotheken java.io und java.util
- Aufbau:
  - Datei zum Lesen öffnen
  - Reader erzeugen und mit der Datei verknüpfen
    - IOExceptions abfangen
    - Strom zum Schluss schließen
  - StreamTokenizer oder Scanner erlauben strukturiertes Einlesen von Daten

Einsetzbare Klassen:

- java.io.FileReader zum Lesen von Text
- java.io.BufferedReader zum Wrappen von FileReadern aus Effizienzgründen
  - Stellt analoge Methoden wie der BufferedWriter zur Verfügung: read(), readLine(), close()
- java.util.Scanner zum strukturierten Einlesen; Methoden:
  - hasNext() – hat noch Eingabe
  - nextInt() – nächster Integer
  - next() – nächster String
  - nextLine() – nächste Zeile

Beispiel: Einlesen aus einer Datei

```java
import java.io.FileReader;
import java.io.IOException;
import java.util.Scanner;

import data.Bestellposten;
import data.Produkt;
import data.Kartei;

public class KarteiEingabe {
    private static KarteiEingabe instanz = null;

    private KarteiEingabe() {}

    public static KarteiEingabe gibInstanz(){
    if (instanz == null){
        instanz = new KarteiEingabe();
        }
        return instanz;
    }
```

```java
public void liesExterneKarteiEin(Kartei kartei,String dateiname){

   try{
        FileReader fr = new FileReader(dateiname);
        Scanner scanner = new Scanner(fr).useDelimiter("(;|\\s)");
        while(scanner.hasNext()){
            Bestellposten posten = liesBestellpostenEin(scanner);
            kartei.eintragen(posten);
        }
        scanner.close();
    }
    catch (IOException e){
        System.out.println(e.getMessage());
    }
}
```

Stabilität der Lösung?

Wie einfach ist diese Implementierung?

- Welcher Aufwand ist für die Implementierung der (fehlenden) Methode `liesBestellpostenEin` nötig?

Wie robust / korrekt ist diese Implementierung?

- Was passiert, wenn ein Produktname Leerzeichen oder Semikolons enthält?
- Was passiert, wenn die Datei die Daten in unterschiedlicher Reihenfolge enthält?

Wie einfach ist es, die Implementierung robuster zu machen?

- Ggf. Produktnamen in Anführungszeichen (macht vor allem das Einlesen etwas komplizierter)
- Speichern und Berücksichtigen von Attributnamen

Serialisierung

Serialization is the process of converting the state of an object into a form that can be persisted or transported. The complement of serialization is deserialization, which converts a stream into an object. Together, these processes allow data to be stored and transferred.

[Quelle: https://learn.microsoft.com/en-us/dotnet/standard/serialization/]

Serialisierung bezeichnet auch häufig spezifisch die Umwandlung von Daten/Objekten, in einen Byte Stream (der persistiert werden kann).

Java-Schnittstelle Serializable

- Eine Klasse, die die Schnittstelle `Serializable` implementiert, kann zu einem Byte Stream konvertiert werden (genauso ihre Unterklassen).
- Die Schnittstelle hat keine Felder oder Methoden, sondern dient nur zur Anzeige der Serialisierbarkeit.
- Beim Serialisieren eines Objekts werden alle seine Felder gespeichert, die nicht als `transient` oder `static` markiert sind.
- Stößt der Serialisierungsvorgang (transitiv) auf ein Objekt, das nicht serialisierbar ist, wird eine `NotSerializableException` geworfen.
- Bei der Deserialisierung wird der Byte Stream wieder in ein Objekt umgewandelt.
- Serialisierung und Deserialisierung können angepasst werden.

Implementierung Serialisierung

- Jede serialisierbare Klasse muss eine `serialVersionUID` vom Typ `long` (außerdem `static` und `final`) definieren (oder diese wird zur Laufzeit automatisch berechnet), die verwendet wird, um beim Deserialisieren zu vergleichen, ob die IDs übereinstimmen. Wenn nicht, wird eine `InvalidClassException` geworfen.
- Schreiben zum Beispiel in einen `FileOutputStream`, Lesen aus einem `FileInputStream`.
- `ObjectOutputStream` bietet Methoden zum Serialisieren von Objekten, `ObjectInputStream` Methoden zum Deserialisieren:
  - `write()`, `writeObject()`, `readObject()`, `close()`, `flush()`, …
- Serialisierbare Klassen können Methoden `readObject` und `writeObject` implementieren, um die Serialisierung anzupassen.
  - `defaultWriteObject` und `defaultReadObject` können verwenden werden, um dabei auf die Standardimplementierung zuzugreifen.

Eigenschaften Serialisierung

- Speichern im Byteformat ist oft kompakt und effizient.
- Da zusätzlich zu den Werten auch die Attributnamen gespeichert werden, toleriert das Deserialisieren per `readObject` kleinere Modifikationen der Originalklasse (Hinzufügen von Feldern, Umsortieren von Feldern, Änderungen an Methoden).
- Da im Byteformat gespeichert wird, ist der Ansatz gut mit zusätzlicher Kompression oder Verschlüsselung von Daten kombinierbar.
- Die entstehende Datei ist im Allgemeinen nicht von Menschen lesbar.
- Objekte, deren Ausgangsklasse umfangreicher modifiziert wurde, können nicht mehr deserialisiert werden.

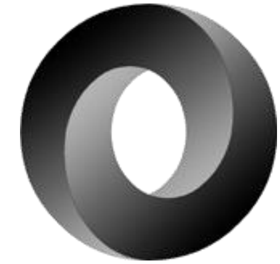
Vorsicht!!!

Das Deserialisieren von Daten ist ein Sicherheitsrisiko! Serialisierung und Deserialisierung per Serializable umgeht die Java-Zugriffskontrollen!

Typische Sicherheitsmaßnahmen:

- Sicherheitskritische Daten nicht serialisierbar machen (`transient`).
- Quellen und Netzwerkverbindungen auf Vertrauenswürdigkeit prüfen.
- `readObject` anpassen und eingelesen Werte validieren
- Kurze Einführung: https://www.oracle.com/java/technologies/javase/seccodeguide.html

JSON

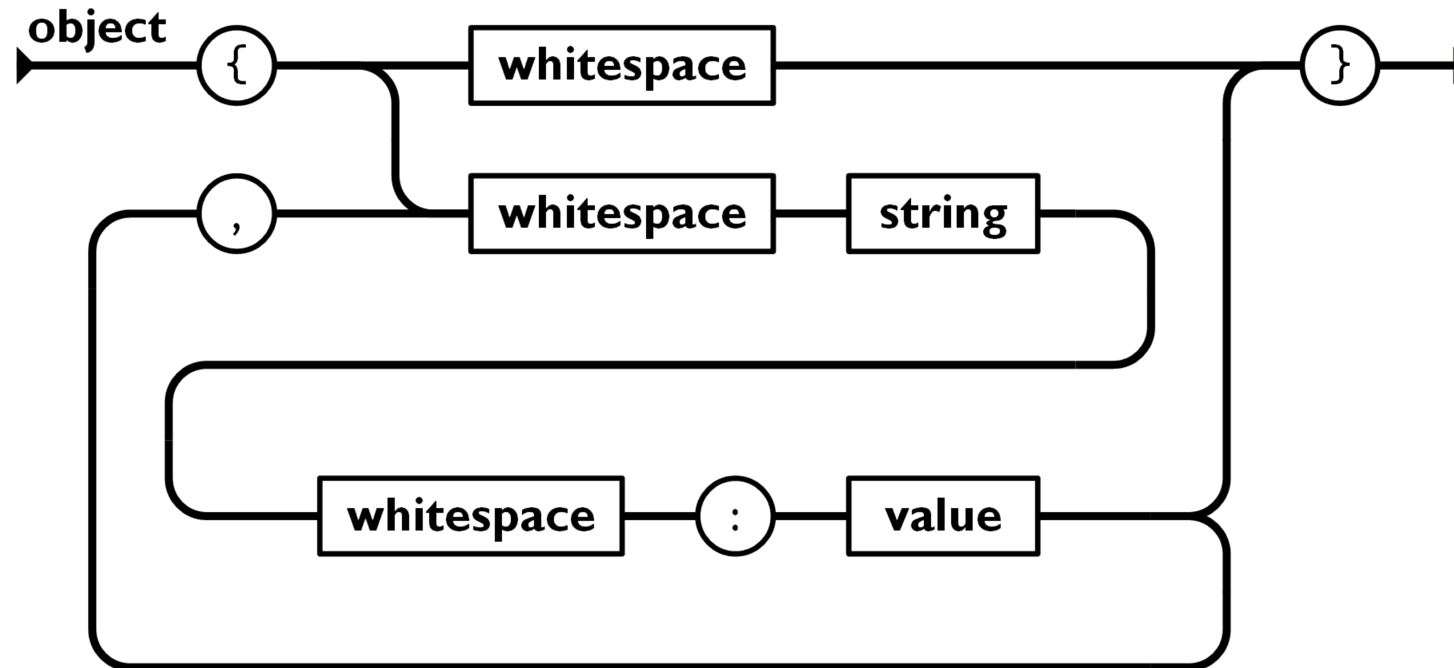


- Kurz für: JavaScript Object Notation
- Leichtgewichtiges Format zum Datenaustausch
- Programmiersprachenunabhängig
- Einfache Syntax; kann von Menschen gelesen und geschrieben werden
- Bibliotheken für alle größeren Programmiersprachen (Java, C Varianten, Python, Rust, Haskell, …)
- Website: https://www.json.org/json-en.html

JSON Syntax

- JSON bietet eine Syntax, aber bewusst keine Semantik. Wie ein JSON-File in einer Programmiersprache/einem Projekt Bedeutung haben soll, kann jeweils festgelegt werden.
- Die Syntax von JSON basiert auf Name-Wert-Paaren und Listen von Werten.
  - Ein object ist eine ungeordnete Menge von Name-Wert-Paaren, eingeschlossen in geschweifte Klammern. Name und Wert sind durch einen Doppelpunkt getrennt, Paare durch Kommata.
  - Ein array ist eine kommaseparierte Liste von Werten, eingeschlossen in eckige Klammern.

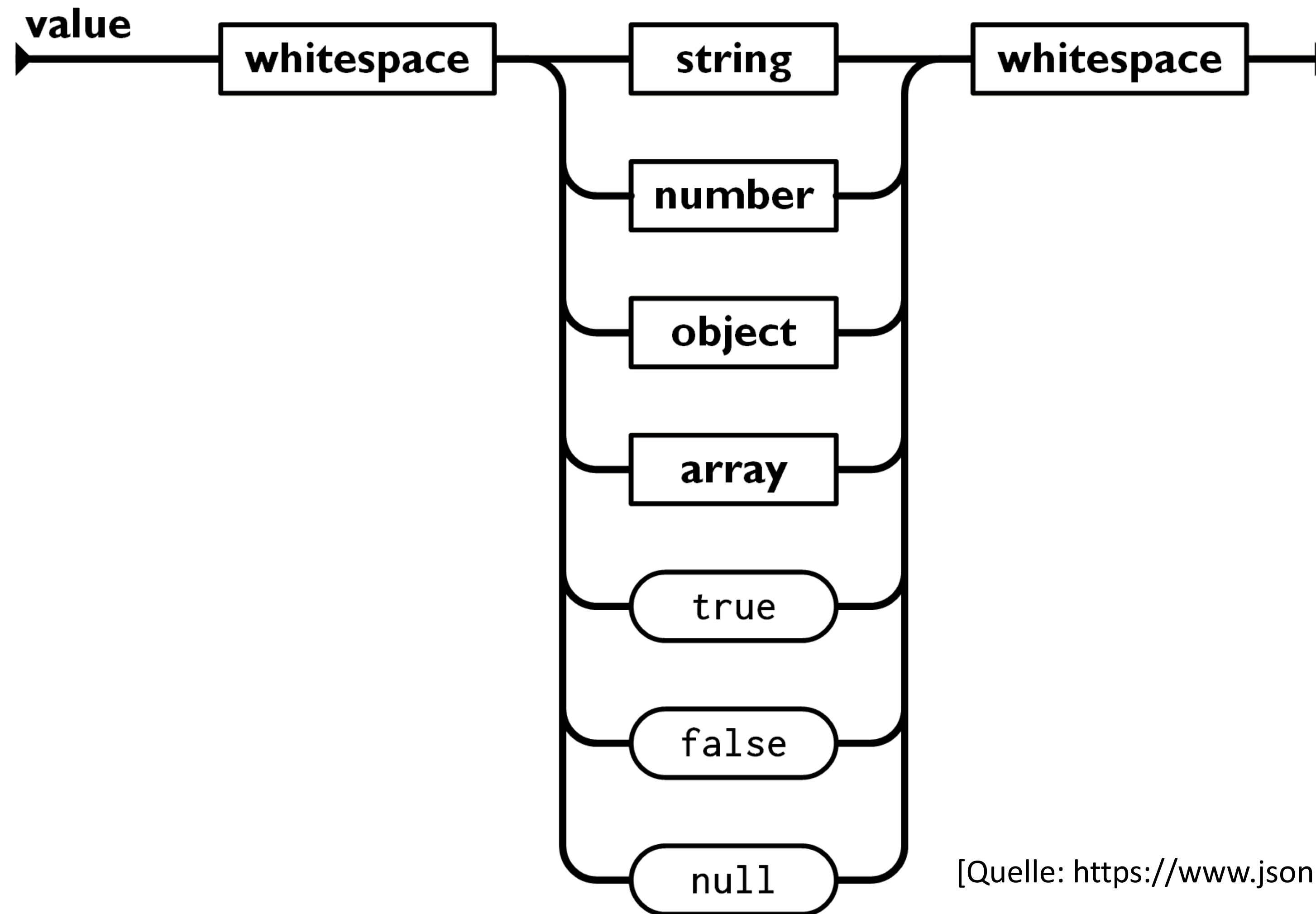
Syntax JSON Object



[Quelle: https://www.json.org/json-en.html]

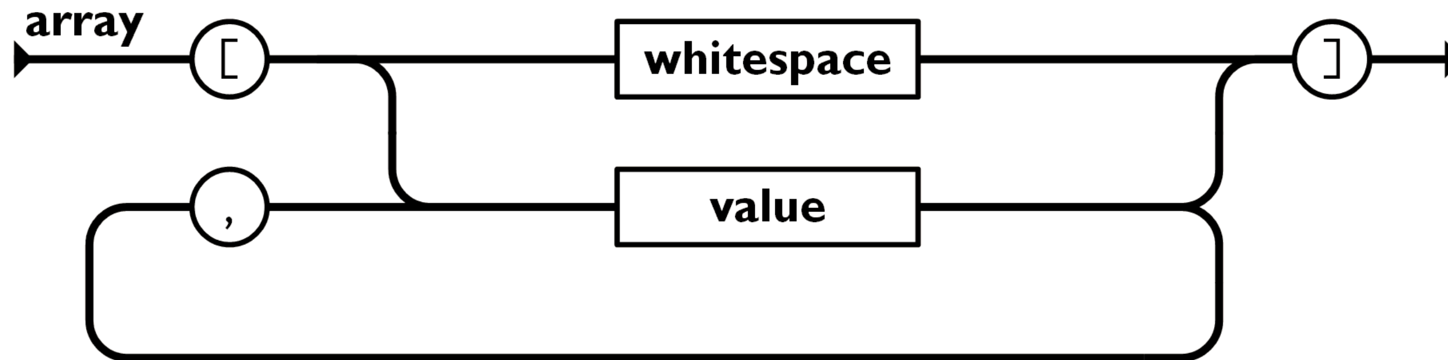
SOFTWARE ENGINEERING
UNIVERSITÄT KASSEL

Syntax JSON Value



[Quelle: https://www.json.org/json-en.html]

Syntax JSON Array



[Quelle: https://www.json.org/json-en.html]

Jackson als JSON Prozessor

- Jackson ist eine umfangreiche Java-Bibliothek zum Speichern und Laden verschiedenster Datenformate (JSON, CSV, XML, YAML, …).
  - Parsieren von Daten in unterschiedlichsten Datenformaten und Laden als Objekte.
  - Speichern von Objekten in verschiedensten Datenformaten.
  - Bewirbt sich als schnell, zuverlässig, leichtgewichtig; ist umfangreich konfigurierbar.
- Weblinks:
  - Allgemeine Homepage: https://github.com/FasterXML/jackson
  - Einstieg in die Javadocs: https://github.com/FasterXML/jackson-docs/wiki/Finding-Javadoc
  - Ein Tutorial: https://www.baeldung.com/jackson
- Neueste Version: 2.16 (15. November 2023)

Jackson ObjectMapper

```
public class PartyService {
   private final ObjectMapper objectMapper = new
ObjectMapper();

   public PartyService() {
      this.objectMapper
         .enable(SerializationFeature.INDENT_OUTPUT);
   }
}
```

Ausschnitt aus `PartyService.java`

`ObjectMapper` ist eine zentrale Klasse zum Speichern und Laden von JSON Strings und Dateien. Typische Verwendung:

- Ein finaler `ObjectMapper` wird in einer Serviceklasse angelegt.
- Über die `enable`-Methode lassen sich viele Einstellungen für Serialisierung und Deserialisierung treffen (Darstellung von Daten, Sortierung von Maps, Bedingungen für das Scheitern von Ausleseprozessen, …).

writeValue und readValue

```
public class PartyService {
    private static final String DATA_FOLDER_NAME = "data";
    private static final Path DATA_FOLDER = Paths.get(DATA_FOLDER_NAME);
    private final ObjectMapper objectMapper = new ObjectMapper();

    public void saveCurrentParty() {
        …
        String fileName = …;
        this.objectMapper.writeValue(Paths.get(fileName).toFile(),
            this.currentParty);
        …
    }

    public ArrayList<Party> load() {
        …
        Party party = objectMapper.readValue(Paths.get(fileName).toFile,
            Party.class);
    }
}
```

Vereinfachter Auszug aus `PartyService.java`

Der Jackson ObjectMapper definiert die Methoden `writeValue` und `readValue` für das Speichern und Laden von Objekten.

- Erster Parameter von `writeValue` ist der Ort für den Output (als `File`, `OutputStream`, `Writer`, …), zweiter Parameter ist das Objekt, das gespeichert werden soll. Rückgabewert ist `void`.
- Erster Parameter von `readValue` ist der Ort, von dem ausgelesen werden soll (als `String`, `File`, `InputStream`, `Reader`, …), zweiter Parameter typischerweise ein Klassenobjekt eines passenden Typs `T`. Rückgabewert ist ein Objekt vom Typ `T`.
- `readValue` kann auch Listen von Objekten oder Maps konstruieren.

Typische Speicher- und Ladevorgänge mit Jackson

- `writeValue` speichert ein Objekt als JSON-File:
  - Jedes Feld der Klasse, das `public` ist oder über einen öffentlichen Getter verfügt, wird gespeichert.
  - Syntax: `"<Feldname>" : <Wert>`
  - Collections werden als JSON-Arrays gespeichert, Referenzen auf andere Objekte als JSON-Objects.
  - Statische Felder werden ignoriert (sie gehören zur Klasse, nicht zum Objekt!).

- `readValue` liest ein JSON-File und erstellt ein Objekt:
  - Jedes Feld der Klasse, das `public` ist oder über einen öffentlichen Setter verfügt, wird aus dem JSON-File ausgelesen und der entsprechende Wert gesetzt.
  - Verfügt die Klasse über ein öffentliches Feld, für das das JSON-File keinen Eintrag hat, wird das entsprechende Feld ignoriert.
  - Hat die JSON-Datei Einträge, die keinen Feldern der Klasse entsprechen, wird eine `UnrecognizedPropertyException` geworfen.

Konfiguration der Serialisierung

Der Serialisierungs- und Deserialisierungsprozess kann in Jackson konfiguriert werden:

- Klassen, Felder und Methoden können annotiert werden.
- `SerializationFeature` und `DeserializationFeature` eines `ObjectMappers` können gesetzt werden.

Beispiele Jackson Annotationen

- `@JsonIgnore`: Ein Feld, das damit annotiert ist, wird beim Serialisieren ignoriert.
- `@JsonIgnoreProperties`: Klassenannotation
  - Kann ein Array von zu ignorierenden Feldern mitgegeben werden.
  - Per `ignoreUnknown=true` kann eingestellt werden, dass unbekannte Einträge in einem JSON-File ignoriert werden und trotzdem ein Objekt konstruiert wird.
- `@JsonGetter`: Kennzeichnet eine Methode (deren Name von der typischen Namensgebung abweicht) als Getter.
- `@JsonPropertyOrder` ist eine Klassenannotation mit der die Reihenfolge, in welcher Felder serialisiert werden, bestimmt werden kann.
- `@JsonManagedReference`, `@JsonBackReference` und `@JsonIdentityInfo` um mit bidirektionalen Referenzen umzugehen.
  - Das mit `@JsonManagedReference` annotierte Feld wird serialisiert, das mit `@JsonBackReference` annotierte ignoriert.
  - Alternativ kann `@JsonIdentityInfo` verwendet werden, um Objekten eindeutige Ids zuzuweisen, die dann verwendet werden, um zyklische Verweise zu unterbrechen.

Ausblick: JSON Schema

```
{
   "$schema": "https://json-schema.org/draft/2020-12/schema",
   "$id": "https://de.uniks.se.PMWS2324_PartyApp/party.schema.json",
   "title": "Party",
   "description": "Existing party",
   "type": "object",
   "properties": {
      "id": {
        "description": "Unique identifier of party",
        "type": "string"
      },
      "name": {
        "description": "Name of party",
        "type": "string"
      },
      …
    },
  "required": [ "id", "name", … ]
}
```

- JSON Schema erlaubt die Definition von Schemata
- JSON-Files können gegen ein Schema validiert werden.
- Schemata können automatisiert aus Klassen abgeleitet werden.
- Website: https://json-schema.org/