Ein- und Ausgabe von Daten

Software Entwicklung 1

Annette Bieniusa, Mathias Weber, Peter Zeller

In diesem Kapitel werden wir uns näher mit Konzepten zur Ein-/Ausgabe von Daten beschäftigen. In der Vorlesung haben wir uns bisher auf die Verarbeitung von Daten beschränkt, die auf der Konsole eingegeben wurden; Ausgaben wurden auch wieder auf die Konsole getätigt. Datenströme sind eine wichtige Abstraktion bei der Datenverarbeitung. Wir werden anhand verschiedener Beispiele sehen, wie wir Datenströme verarbeiten können, aus welchen Quellen Datenströme versorgt werden können und wie Datenströme in Senken ausgegeben werden können.
Bisweilen müssen wir auch Objekte persistent abspeichern, beispielsweise um sie zwischen Programmaufrufen verfügbar zu machen oder zwischen Prozessen auszutauschen. Dazu gibt es verschiedene Möglichkeiten; wir werden hier das JSON diskutieren, ein standardisiertes Format, das weltweit in unzähligen Programmen zur Anwendung kommt.

Lernziele dieses Kapitels:

- Das Konzept eines Datenstroms an Beispielen zu erläutern.
- `Reader`-/`Writer`-Klassen in Java zu verwenden.
- Dateien und Ordern zu erzeugen, zu lesen und zu schreiben.
- Objekte in JSON zu serialisieren und deserialisieren.

1 Ein- und Ausgabe

Bei der Eingabe von Daten gibt es im Wesentlichen zwei Vorgehensweisen:

1. Die komplette Eingabe wird auf einmal bis zum Ende gelesen und dann beispielsweise als `String` oder `byte`-Array zur Verfügung gestellt werden.

2. Die Eingabe wird Stückweise gelesen und zur Verfügung gestellt.

Auch die Ausgabe kann entweder auf einmal oder stückweise geschrieben werden.
Das folgende Beispiel zeigt ein Programm, welches einen zwei Strings `searchString` und `replaceString` und zwei Dateinamen als Programmparameter nimmt. Es wird dann die erste Datei gelesen, alle Vorkommen von `searchString` werden durch `replaceString` ersetzt und das Ergebnis wird in die zweite Datei geschrieben. Die hier verwendeten Features aus der Java Standardbibliothek betrachten wir in Abschnitt 3 genauer.

```
public static void main(String[] args) throws IOException {
    String searchString = args[0];
    String replaceString = args[1];
    Path inFile = Paths.get(args[2]);
    Path outFile = Paths.get(args[3]);
    // Daten aus Eingabe-Datei lesen
    byte[] bytes = Files.readAllBytes(inFile);
    // Bytes in String umwandeln (mit UTF-8 Kodierung)
    String str = new String(bytes, StandardCharsets.UTF_8);
    // searchString durch replaceString ersetzen
    String newString = str.replace(searchString, replaceString);
    // String in Bytes umwandeln (mit UTF-8 Kodierung)
    byte[] newBytes = newString.getBytes(StandardCharsets.UTF_8);
    // Ergebnis in Ausgabe-Datei schreiben
    Files.write(outFile, newBytes);
}
```

Die Ein- und Ausgabe folgt in diesem Beispiel jeweils in einem Schritt. Es wird zuerst die ganze Eingabedatei in ein Byte-Array geladen, dann die eigentliche Berechnung ausgeführt und schließlich ein neues Byte-Array in die Ausgabe-Datei geschrieben.
Ein Nachteil bei diesem Vorgehen ist, dass das Programm mindestens so viel Arbeitsspeicher benötigt, wie die Eingabedatei groß ist. Aus diesem Grund verwendet man in Java oft Datenströme um Ein- und Ausgabe von Daten zu modellieren. Damit lassen sich Programme schreiben, welche unabhängiger von der Eingabe-Größe sind.

2 Streams zur Ein- und Ausgabe

Ein ***Strom*** (engl. *Stream*) ist eine potentiell unendliche Folge von Daten. Er wird von einer oder mehrerer Quellen mit Daten versorgt und erlaubt es, diese Daten der Reihe nach aus dem Strom herauszulesen. Das Ende eines Stromes kann durch ein spezielles Datum markiert werden, z.B. in Java bei `char`-Strömen `-1`.
Bei Operationen auf einem Strom kann es zu Verzögerungen kommen:

- beim Lesen, weil im Moment kein Zeichen vorhanden, der Strom aber noch nicht zu Ende ist;
- beim Schreiben, weil evtl. kein Platz im Strom vorhanden ist.

Die Verzögerungen führen zu einer Blockierung der ausgeführten Methode.

2.1 Beispiel: Ströme von Characters

Wir betrachten zunächst Ströme zum Lesen von `Char`s. Diese Ströme sollen jeweils das `CharEingabeStrom`-Interface implementieren:

```
interface CharEingabeStrom {
    int read() throws IOException;
}
```

Die `read()`-Methode liefert einzelne Character aus einem Eingabestrom. Um das Ende der Eingabe zu markieren, wird der (int-) Wert `-1` geliefert. Da `char`-Werte immer zwischen 0 und 65535 liegen, lässt sich der Rückgabewert `-1` von allen `char`-Werten unterscheiden, wir müssen aber den größeren Typ `int` statt `char` verwenden. Bei Probleme bei der Eingabe soll eine `IOException` ausgelöst werden.
Das Interface abstrahiert von der Quelle, aus der gelesen wird. Als mögliche Quellen sind denkbar:

1. Datenstrukturen wie Array, Liste, String
2. Dateien
3. Netzwerk
4. Standardeingabe, z.B. interaktive Eingabe vom Anwender
5. andere Programme
6. andere Ströme

Wir betrachten hier Ströme über Datenstrukturen und andere Ströme. Das Lesen und Schreiben in Dateien wird in Abschnitt 3 diskutiert.

Ströme aus Datenstrukturen Als erstes Beispiel behandeln wir zunächst das schrittweise Lesen der Zeichen eines Strings. Die Quelle des Stroms (hier: der String) wird dem Konstruktor übergeben.

```
public class StringLeser implements CharEingabeStrom {
  private String zeichen;
  private int    index;

  public StringLeser(String s) {
    zeichen = s;
    index = 0;
  }

  public int read() {
    if (index == zeichen.length()) {
      return -1;
    } else {
      index++;
      return zeichen.charAt(index - 1);
    }
  }
}
```

Kombinieren von Strömen Stromklassen können auch miteinander kombiniert werden, um Ströme zusammenzufassen oder zu modifizieren. Wir betrachten im Folgenden zwei Stromklassen, die aus anderen Strömen lesen und die Ströme modifizieren. Die Konstruktoren nehmen dabei einen beliebigen `CharEingabeStrom` als Quelle:

→ Subtyping at its best!

Das erste Beispiel ist ein `Char`-Eingabestrom, der alle Buchstaben eines Eingabestromes in Großbuchstaben umwandelt.

```
public class GrossBuchstabenFilter implements CharEingabeStrom {
   private CharEingabeStrom eingabeStrom;

   public GrossBuchstabenFilter(CharEingabeStrom cs) {
      eingabeStrom = cs;
   }

   public int read() throws IOException {
      int z = eingabeStrom.read();
      if (z == -1) {
         return -1;
      } else {
         // sicherer Cast, da z hier im else-Fall ein char-Wert ist
         return Character.toUpperCase((char) z);
      }
   }
}
```

Als zweites Beispiel betrachten wir den `UmlautSzFilter`, der alle Umlaute und das ß durch einen Doppelvokal (ü → ue, etc.) bzw. ss ersetzt. Dabei wird ein Puffer von einem Zeichen verwendet, da der Strom bei jedem `read()`-Aufruf nur ein Zeichen liefern kann und bei der Umwandlung der Umlaute bzw. von ß der zweite Buchstabe für den nächsten Aufruf von `read()` zwischengespeichert werden muss.

```
public class UmlautSzFilter implements CharEingabeStrom {
   private CharEingabeStrom eingabeStrom;
   private int puffer = -1;

   public UmlautSzFilter(CharEingabeStrom cs) {
      eingabeStrom = cs;
   }

   public int read() throws IOException {
      if (puffer != -1) {
         int z = puffer;
         puffer = -1;
         return z;
      } else {
         int z = eingabeStrom.read();
         if (z == -1) return -1;
         switch ((char) z) {
           case 'Ä':  puffer = 'e'; return 'A';
           case 'Ö':  puffer = 'e'; return 'O';
           case 'Ü':  puffer = 'e'; return 'U';
           case 'ä':  puffer = 'e'; return 'a';
           case 'ö':  puffer = 'e'; return 'o';
           case 'ü':  puffer = 'e'; return 'u';
           case 'ß':  puffer = 's'; return 's';
           default :                return z;
         }
      }
```

```
   }
}
```

Der hier gezeigte Code verwendet Umlaute, was zu Problemen führen kann, wenn eine falsche Kodierung verwendet wird. Alternativ lassen sich Umlaute und andere Zeichen auch durch durch die entsprechenden Unicode-Codepoints im Code verwendet werden. Zum Beispiel steht der Character `'\u00C4'` für das Zeichen `'Ä'` und die Auswahlanweisung oben kann alternativ wie folgt geschrieben werden:

```
switch( (char)z ) {
  case '\u00C4':  puffer = 'e'; return 'A';
  case '\u00D6':  puffer = 'e'; return 'O';
  case '\u00DC':  puffer = 'e'; return 'U';
  case '\u00E4':  puffer = 'e'; return 'a';
  case '\u00F6':  puffer = 'e'; return 'o';
  case '\u00FC':  puffer = 'e'; return 'u';
  case '\u00DF':  puffer = 's'; return 's';
  default:        return z;
}
```

Folgendes Programm zeigt den Zusammenbau und die Anwendung der verschiedenen Ströme:

```
public class StreamTest {
  public static void main(String[] args) throws IOException {
    String s = "Äneas opfert den Göttern edle Öle,\n"
             + "auf daß überall das Übel sich ändert.";

    CharEingabeStrom cs = new StringLeser(s);
    cs = new UmlautSzFilter(cs);
    cs = new GrossBuchstabenFilter(cs);
    int z = cs.read();
    while (z != -1) {
       System.out.print((char) z);
       z = cs.read();
    }
    System.out.println();
  }
}
```

Frage 1: Was ist die Ausgabe des Programms?

2.2 Stromklassen in Java

Stromklassen sind wichtige programmiertechnische Hilfsmittel, die in Java durch Bibliotheken in `java.io` unterstützt werden. Stromklassen werden nach den Datentypen, die sie verarbeiten, und ihren Datenquellen bzw. -senken klassifiziert.

- Die Reader-/Writer-Klassen verarbeiten `char`-Ströme.
- Die Input-/Output-Stromklassen verarbeiten `byte`-Ströme.

Reader-Klassen Die Reader-Klassen unterstützen:

Lesen einzelner Zeichen	`int read()`
Lesen mehrerer Zeichen aus der Quelle und Ablage in ein char-Array	`int read(char[] c)`
Überspringen einer Anzahl von Zeichen der Eingabe	`long skip(long l)`
Abfrage, ob der Strom für das Lesen des nächsten Zeichens bereit ist	`boolean ready()`
Schließen des Eingabestroms	`void close()`

sowie Methoden zum Markieren und Zurücksetzen des Stroms.
Die genannten Methoden lösen möglicherweise eine `IOException` aus.

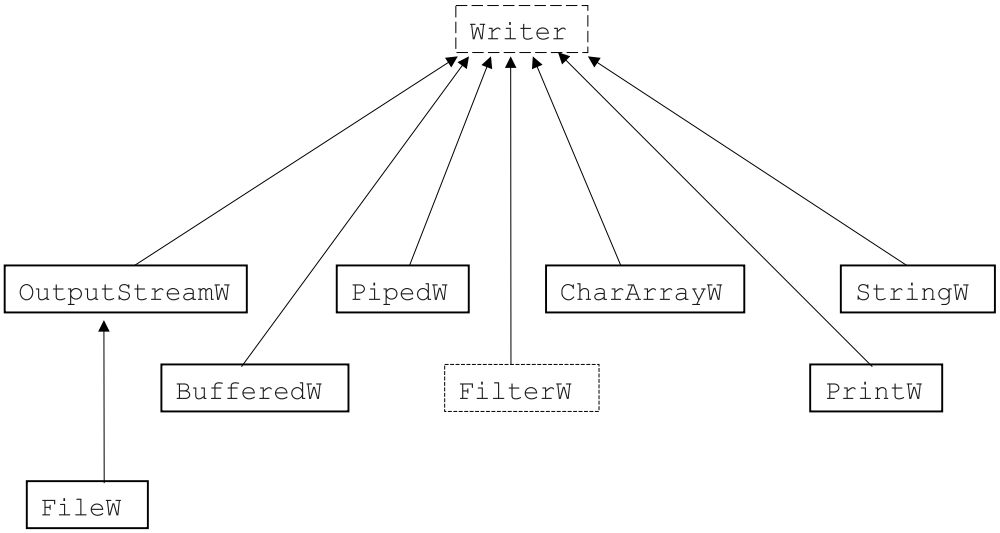
Die Reader-Klassen unterscheiden sich im Wesentlichen durch ihre Quelle:

Reader-Klasse	Quelle	Bemerkung
InputStreamReader	InputStream	
FileReader	byte-Strom aus Datei	
BufferedReader	Reader	zeilenweise puffernd
LineNumberReader	Reader	zeilenweise puffernd
PipedReader	PipedWriter	
FilterReader	Reader	
PushBackReader	Reader	Methode `unread`
CharArrayReader	char[]	
StringReader	String	

Write-Klassen Die Writer-Klassen unterstützen:

Schreiben einzelner Zeichen	`void write(int i)`
Schreiben mehrerer Zeichen eines char-Arrays	`void write(char[] c)` u. ä.
Schreiben mehrerer Zeichen eines String	`void write(String s)` u. ä.
Ausgabe ggf. im Strom gepufferter Zeichen	`void flush()`
Schließen des Ausgabestroms	`void close()`

Writer arbeiten analog zu Reader-Klassen, nur in umgekehrter Richtung.



Beispiel `PrintWriter` unterstützen die formatierte Ausgabe von Daten durch die Methoden `printf` bzw. `format`, sowie `print` und `println`, die alle Standarddatentypen als Parameter nehmen.
Wie wir bereits im vorherigen Abschnitt erläutert haben, ermöglichen die Konstruktoren das Zusammenhängen von Strömen; hier am Beispiel eines Konstruktors der Klasse `PrintWriter`:

```
public PrintWriter(OutputStream o, boolean autoFlush) {
   this(new BufferedWriter(
          new OutputStreamWriter(o)), autoFlush);
}
```

Dieser `PrintWriter`-Konstruktor erzeugt einen `PrintWriter` für einen existierenden `OutputStream`. Dazu erzeugt er einen entsprechenden `OutputStreamWriter`, der die zu schreibenden Character in Bytes konvertiert. Der Parameter `autoFlush` sorgt dafür (falls `true`), dass die Methoden `println`, `printf`, `format` den Ausgabepuffer automatisch leeren.

Schließen von Strömen Um geöffnete Dateien oder andere Ressourcen wieder freizugeben, müssen Ströme mit ihrer `close`-Methode geschlossen werden, wenn sie nicht mehr verwendet werden. Das Schließen eines kombinierten Stroms schließt auch automatisch den ursprünglichen Strom.
Um das Schließen sicherzustellen, selbst wenn während der Verwendung des Stroms eine Exception auftritt, muss die `close`-Methode in einem `try-finally` Block aufgerufen werden.

```
BufferedReader reader = null;
try {
    reader = new BufferedReader(...);
```

```
    // Strom verwenden
} finally {
    if (reader != null) {
        reader.close();
    }
}
```

Der `finally`-Block wird immer nach dem `try`-Block ausgeführt, sowohl wenn eine Exception ausgelöst wurde als auch im anderen Fall.
Java 7 bietet eine alternative Syntax für die Deklaration und Verwendung von Strömen an:

```
try (BufferedReader reader = new BufferedReader(...)) {
    // Strom verwenden
}
```

Dieses `try with resource`-Konstrukt sorgt dafür, dass der Strom automatisch am Ende des `try`-Blocks geschlossen wird.
Syntax:

Anweisung →
    try ResourceSpecification {
      DeklAnweisListe
    } catchKlauselListe finallyBlock

ResourceSpecification →
    (ResourceList)
  | ϵ

ResourceList →
    Resource ; ResourceList
  | Resource

Resource →
    Typ ≪ *Bezeichner* ≫ = Ausdruck

CatchKlauselListe →
    CatchKlausel CatchKlauselListe
  | ϵ

CatchKlausel →
    catch (≪ *Typ* ≫ ≪ *Bezeichner* ≫) {
        DeklAnweisListe
    }

finallyBlock →
    finally {
        DeklAnweisListe
    }
  | ϵ

3 Zum Umgang mit Dateien und Dateisystemen

Seit Java 1.7 gibt es ein modernes, umfangreiches Paket, `java.nio.files` zum Umgang mit Dateien und Dateisystem. Dateien werden in einem Dateisystem durch Pfade adressiert. Pfade sind hierarchisch angeordnet und bestehen, ausgehend von einem Wurzelelement (Root), aus einer Sequenz von Dateiordnern und Dateinamenselementen (z.B. Name, Kürzel).
Unter Windows wird dabei als Trennzeichen ein Backslash verwendet (\):

```
C:\Users\karlheinz\privat\brief.txt
```

Unter unixartigen System ist das Trennzeichen ein Slash (/):

```
/home/karlheinz/privat/brief.txt
```

Im Java-Programm werden Pfade durch Objekte vom Typ `Path` repräsentiert[1]. Sie können u.a. durch die statische Methode `get(String)` der Klasse `Paths` konstruiert werden:

```
Path p = Paths.get("/home/karlheinz/privat/brief.txt");
```

Die Methoden der Klasse `Path` dienen dazu, Pfade zu untersuchen und zu erzeugen. Die Methode `getParent()` gibt den Pfad des Verzeichnisses zurück, in dem sich die aktuelle Datei/das aktuelle Verzeichnis befinden.

```
assertEquals("/home",
   Paths.get("/home/karlheinz").getParent().toString())
```

Die Methoden `resolve(String path)` und `resolve(Path path)` erzeugen einen neuen Pfad indem der übergebene Pfad relativ zum aktuellen Pfad aufgelöst wird.

```
assertEquals("/home/karlheinz",
   Paths.get("/home").resolve("karlheinz").toString())
```

Die Methode `getFileName()` liefert den Namen der Datei zurück, die durch den aktuellen Pfad adressiert wird.

```
assertEquals("brief.txt",
   Paths.get("/home/karlheinz/brief.txt").getFileName().toString())
```

Auf diese Weise können Pfade verarbeitet werden, ohne die platformabhängigen Trennzeichen und Pfadformate zu berücksichtigen.
Die Klasse `Files`[2] bietet statische Methoden, die auf Dateien, Dateiordnern und anderen Arten von Dateien operieren.
Kleinere Dateien können mittels `readAllLines(Path path)` oder `readAllBytes(Path path)` vollständig gelesen werden, so dass der Dateiinhalt als Liste von Strings oder als Byte-Array vorliegt:

```
List<String> vorsaetze = Files.readAllLines(p);
```

Um die Inhalte einer Datei neu zu schreiben, gibt es eine Methode `write` zum Schreiben. Dabei wird die Datei auch erzeugt, falls noch nicht existent.

[1] `https://docs.oracle.com/javase/8/docs/api/java/nio/file/Path.html`
[2] `https://docs.oracle.com/javase/8/docs/api/java/nio/file/Files.html`

```
List<String> neueVorsaetze = Arrays.asList("Weltfrieden schaffen",
   "keine Listen mit Vorsaetzen erstellen");
Files.write(p, neueVorsaetze, StandardCharsets.UTF_8);
```

Bei **größeren Dateien** sollten zum Lesen und Schreiben die Eingabe bzw. Ausgabe gepuffert werden. Dadurch muss der Inhalt der Datei nicht vollständig im Programmspeicher vorgehalten werden, sondern kann sukzessive gelesen bzw. geschrieben werden. Aus der Klasse `Files`:

```
static BufferedReader newBufferedReader(Path path)
static BufferedWriter newBufferedWriter(Path path)
```

Neben den hier kurz vorgestellten Methoden gibt es noch zahlreiche weitere Varianten Optionen in der Klasse `Files`, um Dateiinhalte und Datei-Attribute zu lesen und manipulieren, Verzeichnisse zu durchlaufen, Dateien zu erzeugen, zu löschen, zu verschieben, etc. Näheres dazu finden Sie in der Dokumentation (`http://docs.oracle.com/javase/tutorial/essential/io/index.html`).

Beispiel: Suchen und Ersetzen Wir wollen nun Streams verwenden, um das Beispiel aus der Einleitung (Abschnitt 1) zu implementieren und so das angesprochene Speicherproblem vermeiden.

```
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;

import static java.nio.charset.StandardCharsets.UTF_8;
import static java.nio.file.StandardCopyOption.ATOMIC_MOVE;
import static java.nio.file.StandardCopyOption.REPLACE_EXISTING;

public class SearchReplace2 {
    public static void main(String[] args) throws IOException {
        String searchString = args[0];
        String replaceString = args[1];
        Path inFile = Paths.get(args[2]).toAbsolutePath();
        Path outFile = Paths.get(args[3]).toAbsolutePath();

        Path tempF;
        if (inFile.equals(outFile)) {
            // Falls Eingabe und Ausgabe gleich sind, zuerst in temporäre Datei schreiben
            tempF = Files.createTempFile(outFile.getParent(), "tmp", "");
        } else {
            tempF = outFile;
        }

        // Dateien mit try-witch-resource Statement öffnen und Ersetzung ausführen
        try (BufferedReader in = Files.newBufferedReader(inFile, UTF_8);
             BufferedWriter out = Files.newBufferedWriter(outFile, UTF_8)) {
            searchReplace(searchString, replaceString, in, out);
        }
```

```
        if (!tempF.equals(outFile)) {
            Files.move(tempF, outFile, ATOMIC_MOVE, REPLACE_EXISTING);
        }
    }

    private static void searchReplace(String searchString, String replaceString,
                                      BufferedReader reader, BufferedWriter writer)
            throws IOException {
        while (true) {
            String line = reader.readLine();
            if (line == null) {
                return;
            }
            String newLine = line.replace(searchString, replaceString);
            writer.write(newLine);
            writer.newLine();
        }
    }
}
```

Die Methode `searchReplace` liest die Eingabe Zeilenweise ein und, führt jeweils die Ersetzung pro Zeile aus und schreibt die neue Zeile direkt in den Ausgabestrom. So hängt der Speicherverbrauch des Programms nicht mehr von der Gesamtgröße der Datei ab, sondern nur noch von der maximalen Zeilenlänge in der Datei.
In der `main`-Methode werden mit `Files.newBufferedReader` und `newBufferedWriter` die Eingabe- und Ausgabeströme geöffnet. Dies passiert in einem try-with-resources Block, so dass diese nach dem Ersetzen auch wieder geschlossen werden.
Außerdem müssen wir hier noch den Spezialfall behandeln, dass Eingabe- und Ausgabedatei identisch sind. In diesem Fall erstellen wir mit `Files.createTempFile` eine temporäre Datei und verschieben diese am Ende der `main`-Methode mittels `Files.move`.

4 Ein- und Ausgabe von Objekten

Wir haben gesehen, wie wir `byte`- und `char`-Werte in Ströme schreiben und von Strömen lesen können. Nun wollen wir uns ansehen, wie wir Objekte in ein entsprechendes Format umwandeln können.
Serialisieren bedeutet einen Wert eins Typs in Bytes umzuwandeln.
Deserialisieren bezeichnet den umgekehrten Prozess.
Die Serialisieren und Deserialisieren von Objekten ist komplex:

- Nicht alle Attribute eines Objekts müssen gespeichert werden (zum Beispiel Zwischenspeicher, Indexstrukturen, etc).
- Manchmal ist es nicht ausreichend alle Attribute eines Objekts zu speichern (der Zustand könnte von externen Ressourcen oder dem aktuellen Systemzustand abhängig sein).
- Objektreferenzen besitzen nur innerhalb des aktuellen Prozesses eine Gültigkeit.

- Bei Objekten ist häufig ihre Rolle im Objektgeflecht von entscheidender Bedeutung.

Andererseits ist Ein- und Ausgabe von Objekten wichtig, um Objekte zwischen Prozessen auszutauschen oder Objekte für nachfolgende Programmläufe zu speichern, d.h. ***persistent*** zu machen.

Beispiel: Ausgabe von Objekten Wir modellieren eine Liste von Aufgaben, die jeweils aus einer Beschreibung und dem Datum, bis zu welchem die Aufgabe zu erledigen ist, bestehen.

```
public class TodoList {
    private List<Task> tasks;
    public TodoList() {
        this.tasks = new LinkedList<Task>();
    }
    public void addTask(String desc, Date d) {...}
}

public class Task {
    private String description;
    private Date date;
    ... // Getter und Setter
}
```

Die Aufgabenliste kann folgendermaßen verwendet werden:

```
TodoList tdl = new TodoList();
Date heute = new Date(16, 1, 2018);
tdl.add("Lernen", heute);
tdl.add("Einkaufen", heute);
tdl.add("Lernen", new Date(17, 1, 2018));
```

Um die Aufgabenliste auch in anderen Programmen (Kalender, Webservice, etc.) oder bei Neustart des Programms verwenden zu können, muss sie in einem geeigneten Format "ausgegeben" werden. Was bedeutet es aber, das von `tdl` referenzierte Objekt "auszugeben"?

- Genügt es nur das `TodoList`-Objekt ausgeben?
- Muss das `TodoList`-Objekt und die zugehörigen `Task`-Objekte ausgegeben werden?
- Oder ist es notwendig, das `TodoList`-Objekt, die zugehörigen `Task`-Objekte sowie die `String`-Objekte und das `Date`-Objekt auszugeben?
- Wie sieht es mit den `Entry`-Objekten der `LinkedList` aus?
- Und muss das Objekt `heute` zwei mal ausgegeben werden, weil es zweimal verwendet wird?

Um Objekte in ihrem Zusammenwirken mit anderen Objekten wieder einlesen zu können, müssen sie i.d.R. gemeinsam mit allen erreichbaren Objekten ausgegeben werden.
Dabei ist folgendes zu beachten:

- Gibt man ein Objekt mit den erreichbaren Objekten aus und liest es wieder ein, entsteht eine Kopie.
- Wegen möglicher Zyklen ist die Implementierung der Ausgabe und des Einlesens von Objekt-Geflechten nicht einfach.
- Referenziert man von mehreren Variablen Teile des gleichen Geflechts, kommt es beim Einlesen ggf. zu mehreren Kopien eines Objekts des ursprünglichen Geflechts.

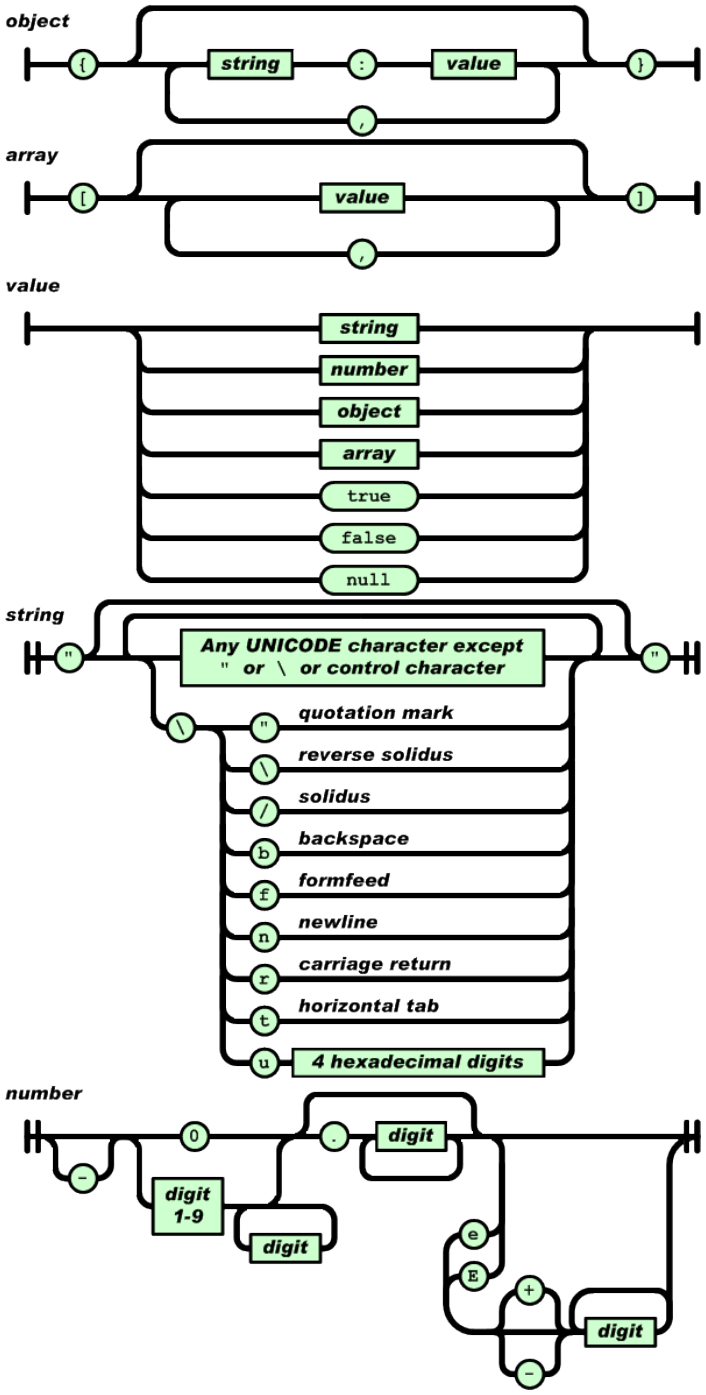
In Java gibt es eine native Möglichkeit der Serialisierung (JOS, Java Object Serialization). Dabei wird die Serialisierbarkeit der Objekte einer Klasse K dadurch ausgedrückt, dass K das Interface `Serializable` implementiert. Dies erlaubt es, Objekte direkt in einem Format in Byte-Streams zu schreiben und auszulesen, das von Java direkt unterstützt wird. Eine umfangreiche Einführung finden Sie hier: `http://openbook.rheinwerk-verlag.de/javainsel9/javainsel_17_010.htm`

4.1 JSON

JSON (JavaScript Object Notation) ist ein leichtgewichtiges Datenaustauschformat. Es ist einfach für Menschen zu lesen und gleichzeitig einfach für Maschinen zu parsen und generieren.
JSON baut auf zwei Arten von Strukturen auf:

- Eine Sammlung von Name-Wert-Paaren (ähnlich einer Map) ($\Rightarrow$ JSON Object)
- Eine geordnete Liste von Werten ($\Rightarrow$ JSON Array)

JSON Syntax Die folgenden Syntaxdiagramme zeigen die Struktur von JSON Objects und JSON Arrays (Quelle: `http://www.json.org`):



Als Beispiel betrachten wir die JSON Struktur von Universitätsobjekten, die für eine

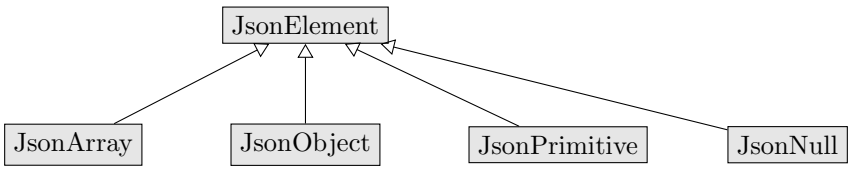
Universität den Namen sowie eine Liste von Kursen enthält:

```
{"name":"TU KL","courses":["SE 1","SE 2","SE 3","Insy"]}
```

Gson Gson ist ein Java-Bibliothek zur Erzeugung von JSON Repräsentationen von Objekten. Die Bibliothek enthält Unterstützung für

- die direkte Beschreibung der zu erzeugenden JSON Ausgabe,
- die Umwandlung von Java-Objekten in JSON, sowie
- entsprechende Methoden zum Parsen von JSON.

Für jedes Element in der JSON-Syntax gibt es eine Java-Klasse, die dieses Element repräsentiert.



Mit Objekten dieser Klassen können JSON-Ausgaben beschrieben werden.
Um das Universitätsobjekt aus dem obigen Beispiel zu erzeugen, kann man wie folgt vorgehen:

```
Gson gson = new Gson();
JsonObject university = new JsonObject();
university.addProperty("name", "TU KL");
JsonArray courses = new JsonArray();
courses.add("SE 1");
courses.add("SE 2");
courses.add("SE 3");
courses.add("FGdP");
university.add("courses", courses);

String s = gson.toJson(university); // liefert den gewuenschten String
```

Das Auslesen der Information aus der JSON Repräsentation (Parsen) erfolgt folgendermaßen:

```
Gson gson = new Gson();
JsonObject parsedUni = gson.fromJson(jsonString, JsonObject.class);
assertEquals(parsedUni.get("name").getAsString(), "TU KL");

JsonArray parsedCourses = parsedUni.get("courses").getAsJsonArray();
assertEquals(parsedCourses.get(0).getAsString(), "SE 1");
assertEquals(parsedCourses.get(1).getAsString(), "SE 2");
assertEquals(parsedCourses.get(2).getAsString(), "SE 3");
assertEquals(parsedCourses.get(3).getAsString(), "FGdP");
```

Umwandlung von Java-Objekte in JSON Für viele Java-Objekte ist die Relation zwischen dem Objekt im Speicher und der Ausgabe als JSON klar:

Objekte	⇒	JSON-Objekt
Attribute eines Objekts	⇒	Eigenschaften (properties) des JSON Objects
Listen und Arrays	⇒	JSON Arrays
Maps	⇒	JSON Object

Reichen diese Konventionen aus, kann Gson die Konvertierung in der Regel automatisch durchführen.

```
public class University {
    private String name;
    private List<String> courses;

    public University(String name) {
        this.name = name;
        this.courses = new ArrayList<String>();
    }

    public void addCourse(String course) {
        this.courses.add(course);
    }

    public boolean equals(Object o) {...}
}
```

Frage 2: Geben Sie eine Implementierung der `equals`-Methode für die Klasse `University`!

Für die Klasse `University` kann beispielsweise eine automatische Umwandlung erfolgen:

```
University unikl = new University("TU KL");
unikl.addCourse("SE 1");
unikl.addCourse("SE 2");
unikl.addCourse("SE 3");
unikl.addCourse("Insy");

Gson gson = new Gson();
String json = gson.toJson(unikl);

University parsedUni = gson.fromJson(json, University.class);
assertEquals(parsedUni, unikl);
```

JSON kann auch direkt auf einen `Writer` geschrieben oder von einem `Reader` gelesen werden:

```
// schreibe das University Objekt unikl in Datei unikl.json
Path path = Paths.get("unikl.json");
try (BufferedWriter w = Files.newBufferedWriter(path)) {
  gson.toJson(unikl, w);
}
// lies das University Objekt aus der Datei unikl.json
try (BufferedReader r = Files.newBufferedReader(path)) {
  University uni = gson.fromJson(r, University.class);
}
```

Anpassung der Json Serialisierung/Deserialisierung Manchmal ist die Standard-Serialisierung/Deserialisierung von Gson ungeeignet. Dies ist zum Beispiel der Fall, wenn das Objektgeflecht Kreise enthält oder Subtyping verwendet wird. Ein Beispiel dafür ist die folgende Typhierarchie:

```
abstract class Course {
    protected String name;
    protected int etcs;
    // ...
}

class Seminar extends Course {
    private int numPlaces;
    // ...
}

class Lecture extends Course {
    private boolean writtenExam;
    // ...
}

public class University2 {
    private String name;
    private List<Course> courses;
}
```

Wir können versuchen, diese Typhierarchie wie oben mit Gson zu verwenden:

```
University2 unikl = new University2("TU KL");
unikl.addCourse(new Lecture("SE 1", 10, true));
unikl.addCourse(new Seminar("SE-Seminar", 4, 15));
Gson gson = new Gson();
String json = gson.toJson(unikl);

// Ergibt Json:
{
  "name": "TU KL",
  "courses": [
    {
      "writtenExam": true,
      "name": "SE 1",
      "etcs": 10
    },
    {
      "numPlaces": 15,
      "name": "SE-Seminar",
      "etcs": 4
    }
  ]
}
```

Beim Versuch dieses Json-Dokument wieder in ein `University`-Objekt umzuwandeln erhalten wir jedoch den Fehler “java.lang.RuntimeException: Unable to invoke no-args

constructor for class Course", weil Gson versucht eine Instanz der abstrakten Klasse Course zu erstellen – die Information um welche konkrete Klasse es sich handelt ist verloren gegangen.
Wir können statt die Umwandlung von `Course`-Objekten jedoch anpassen. Benutzerdefinierte Übersetzungen für Java-Klassen implementiert man als Subtypen von `JsonSerializer<T>` (von `T` nach JSON) und `JsonDeserializer<T>` (von JSON nach `T`). Die folgende Klasse `CourseSerialization` implementiert beide Interfaces und speichert den konkreten Typ des `Course`-Objekts in einem Feld "type" ab, so dass diese Information nicht verloren geht.

```
public class CourseSerialization
      implements JsonSerializer<Course>, JsonDeserializer<Course> {
    @Override
    public Course deserialize(JsonElement json, Type typeOfT,
          JsonDeserializationContext context) throws JsonParseException {
        JsonObject obj = json.getAsJsonObject();
        String type = obj.get("type").getAsString();
        switch (type) {
            case "seminar":
                return context.deserialize(json, Seminar.class);
            case "lecture":
                return context.deserialize(json, Lecture.class);
            default:
                throw new JsonParseException("Unknown type: " + type);
        }
    }

    @Override
    public JsonElement serialize(Course course, Type typeOfSrc,
          JsonSerializationContext context) {
        String type;
        if (course instanceof Seminar) {
            type = "seminar";
        } else if (course instanceof Lecture) {
            type = "lecture";
        } else {
            throw new RuntimeException("Unknown type: " + course);
        }
        JsonObject obj = (JsonObject) context.serialize(course);
        obj.add("type", new JsonPrimitive(type));
        return obj;
    }
}
```

Um diese Klasse bei Gson zu registrieren muss ein `GsonBuilder` und dessen Methode `registerTypeAdapter` verwendet werden:

```
GsonBuilder builder = new GsonBuilder();
builder.registerTypeAdapter(Course.class, new CourseSerialization());
Gson gson = builder.create();
String json = gson.toJson(unikl);

// Ergibt Json:
{
  "name": "TU KL",
```

```
  "courses": [
    {
      "writtenExam": true,
      "name": "SE 1",
      "etcs": 10,
      "type": "lecture"
    },
    {
      "numPlaces": 15,
      "name": "SE-Seminar",
      "etcs": 4,
      "type": "seminar"
    }
  ]
}
```

Falls Daten aus externen Quellen geladen werden, ist es wichtig einzuschränken, welche Klassen initialisiert werden können. Ansonsten können Angreifer den Deserialisierungs-Mechanismus eventuell ausnutzen um Code auszuführen, auf den sie keinen Zugriff haben sollten. Siehe auch:
`https://www.owasp.org/index.php/Deserialization_of_untrusted_data`

Praktische Hinweise zu GSON Zur Verwendung der Gson Bibliothek müssen Sie sich die .jar-Datei der GSON-Bibliothek von der Materialien-Seite herunterladen (z.B. `gson-2.8.2.jar`). Beim Ausführen muss die Bibliothek mit der Option -cp in den sogenannten Classpath aufgenommen werden. Der Classpath ist eine durch : (bzw ; unter Windows) getrennte Liste von Ordnern und Dateien.
Zur Übersetzung muss das aktuelle Verzeichnis (geschrieben als einzelner Punkt) und die Gson jar-Datei im Classpath sein:

- Unter Linux/Mac:

  ```
  javac -cp gson-2.8.2.jar:. Main.java
  ```

- Unter Windows:

  ```
  javac -cp gson-2.8.2.jar;. Main.java
  ```

Zum Übersetzen von JUnit Test muss zusätzlich noch die junitrunner.jar in den Pfad aufgenommen werden:

- Unter Linux/Mac:

  ```
  javac -cp gson-2.8.2.jar:junitrunner.jar:. Test.java
  ```

- Unter Windows:

  ```
  javac -cp gson-2.8.2.jar;junitrunner.jar;. Test.java
  ```

Beim Ausführen muss der Classpath erneut mit angegeben werden:

- Unter Linux/Mac:

```
java -cp gson-2.8.2.jar:. Main
```

- Unter Windows:

```
java -cp gson-2.8.2.jar;. Main
```

Beim Ausführen von Tests lässt sich der junitrunner dann nicht mehr mit der Option `-jar` starten, da diese Option nicht kompatibel mit der Option `-cp` ist. Statt dessen müssen Tests wie normale Programme mit Angabe der Klasse `softech.junitrunner.Runner` als Main-Klasse gestartet werden:

- Unter Linux/Mac:

```
java -cp junitrunner.jar:gson-2.8.2.jar:. softech.junitrunner.Runner Test
```

- Unter Windows:

```
java -cp junitrunner.jar;gson-2.8.2.jar;. softech.junitrunner.Runner Test
```

`Test` bezieht sich dabei auf den Namen der Klasse mit den Tests.

Importieren der Gson-Bibliothek Zur Verwendung müssen die Klassen der Gson-Bibliothek am Anfang der Java-Datei importiert werden:

```
import com.google.gson.Gson;
import com.google.gson.JsonArray;
import com.google.gson.JsonElement;
import com.google.gson.JsonObject;
import com.google.gson.JsonParser;
```

Es können auch direkt alle Klassen importiert werden mit:

```
import com.google.gson.*;
```

Details zu Gson finden Sie unter folgenden Links:

- `http://google.github.io/gson/apidocs/`
- `https://github.com/google/gson/blob/master/UserGuide.md`

Hinweise zu den Fragen

Hinweise zu Frage 1:
AENEAS OPFERT DEN GOETTERN EDLE OELE,
AUF DASS UEBERALL DAS UEBEL SICH AENDERT.

Hinweise zu Frage 2:

```
public class University {
    private String name;
    private List<String> courses;

    //...
    public boolean equals(Object obj) {
        if (this == obj) {
            return true;
        }
        if (obj == null) {
            return false;
        }
        if (!(obj instanceof University )) {
            return false; // andere Klasse
        }
        University other = (University) obj;
        return this.name.equals(other.name)
                && this.courses.equals(other.courses);
    }
}
```