Streams

Software Entwicklung 1

Annette Bieniusa, Mathias Weber, Peter Zeller

1 Streams zur Ein- und Ausgabe

Ein- und Ausgabe von Daten wird heutzutage meist durch Datenströme modelliert. 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 wird durch ein spezielles Datum markiert (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.

1.1 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` ein Subtyp von `int` ist, ist der gewählte Rückgabetyp der Methode `int`. 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 2 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 char[] zeichen;
    private int    index;

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

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

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:

$\rightarrow$ 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 '\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;
         }
      }
   }
}
```

Die Character `'\u00C4'`, usw. sind die Unicode-Codepoints der Umlaute bzw. ß. Unicode-Character können mit dieser Notation im Quellcode vermerkt werden, z.B. wenn Unicode-Zeichen nicht direkt eingegeben werden können.
Folgendes Programm zeigt den Zusammenbau und die Anwendung der verschiedenen Ströme:

```
public class StreamTest {
  public static void main(String[] args) throws IOException {
    String s = new String("\u00C4neas opfert den "
       + "G\u00D6ttern edle \u00D6le,\nauf da\u00DF "
```

```
        + "\u00FCberall das \u00DCbel sich \u00E4ndert.");

    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?

1.2 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.

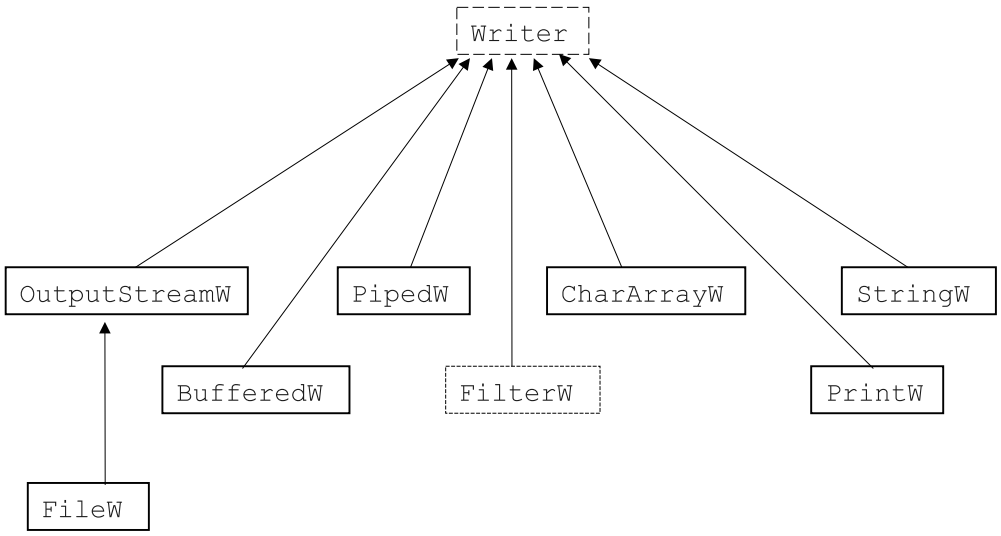
Die Reader-Klassen unterscheiden sich im Wesentlichen durch ihre Quelle:

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 Ströme müssen geschlossen werden, wenn sie nicht mehr verwendet werden. Um dies sicherzustellen, selbst wenn während der Verwendung des Stroms eine Exception auftritt, muss die Anweisung in einem `try-finally` Block verwendet 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.

Beispiel: Liebesbrief-Adapter Im Laufe der Jahre haben Sie eine umfangreiche digitale Liebesbriefsammlung angelegt. Um die Briefe einfach wiederverwenden zu können, verwenden Sie einen Platzhalter `NAME` für den Namen Ihrer aktuellen Liebschaft. Das folgende Programm personalisiert einen Brief, indem es `NAME` durch den tatsächlichen Namen ersetzt; das Ergebnis wird in eine neue Textdatei geschrieben. Da die Briefe zum Teil sehr umfangreich sind, verwendet der Liebesbrief-Adapter gepufferte Reader und Writer, um die Ersetzung zeilenweise vorzunehmen.

```
import java.io.*;

public class LiebesbriefAdapter {
  // Erster Programmparameter: Dateiname
  // Zweiter Programmparameter: Adressat
  public static void main(String[] args){
    String brief = args[0];
```

```
    String person = args[1];

    try (BufferedReader br = new BufferedReader(new FileReader(brief))) {
      try (BufferedWriter bw = new BufferedWriter(new FileWriter(person +
     brief))) {
        String line = br.readLine();
        while(line != null) {
          bw.write(line.replace("NAME", person));
          bw.newLine();

          line = br.readLine();
        }
      }
    } catch (IOException e) {
      System.out.println(e.getMessage());
    }
  }
}
```

2 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.

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

```
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)` vollständig gelesen werden, so dass der Dateiinhalt als Liste von Strings 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.

```
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)
```

Beispiel: Liebesbrief-Adapter Unser LiebesbriefAdapter braucht als Eingabe in der aktuellen Version den Dateinamen der Eingabedatei. Damit müssen die Liebesbriefe im gleichen Verzeichnis abgelegt werden wie das Java-Programm. Sollen die Briefe in Unterordnern organisiert werden, so müssen wir den Pfad zur Ausgabedatei auf andere Weise erzeugen. Dazu können wir die `Path`-API verwenden.

```
import java.nio.file.*;
import java.io.*;

public class LiebesbriefAdapterXP {
  // Erster Programmparameter: Dateipfad
  // Zweiter Programmparameter: Adressat
  public static void main(String[] args){
    String brief = args[0];
    String person = args[1];

    Path pin = Paths.get(brief);
    Path parent = pin.getParent();
    String filename = pin.getFileName().toString();
```

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

```
    Path pout = parent.resolve(person + filename);

    try (BufferedReader br = Files.newBufferedReader(pin)) {
      try (BufferedWriter bw = Files.newBufferedWriter(pout)) {
        String line = br.readLine();
        while(line != null) {
          bw.write(line.replace("NAME", person));
          bw.newLine();

          line = br.readLine();
        }
      }
    } catch (IOException e) {
      System.out.println(e.getMessage());
    }
  }
}
```

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`).

3 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:

- Der Zustand reicht bisweilen zur Repräsentation eines Objektes nicht aus.
- 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();
tdl.add("Lernen", new Date(9,3,2016));
tdl.add("Einkaufen", new Date(1,2,2016));
```

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?

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.

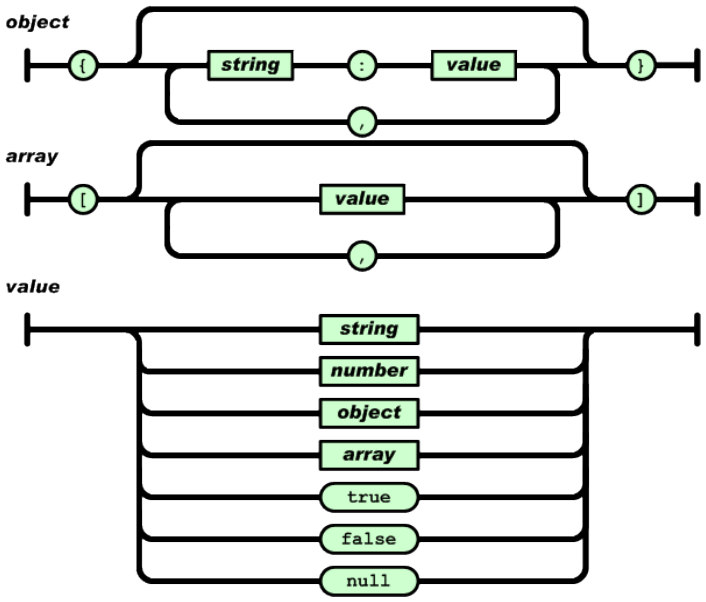
In Java gibt es einen native Möglichkeit der Serialisierung (JOS, Java Object Serialization). Dabei wird die Serialisierbarkeit der Objekte einer Klasse K dadurch ausgedrückt, dass K das Interface `Serializable` implementiert. Dies erlaubt es, Objekte direkt in einem Format in Byte-Streams zu schreiben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

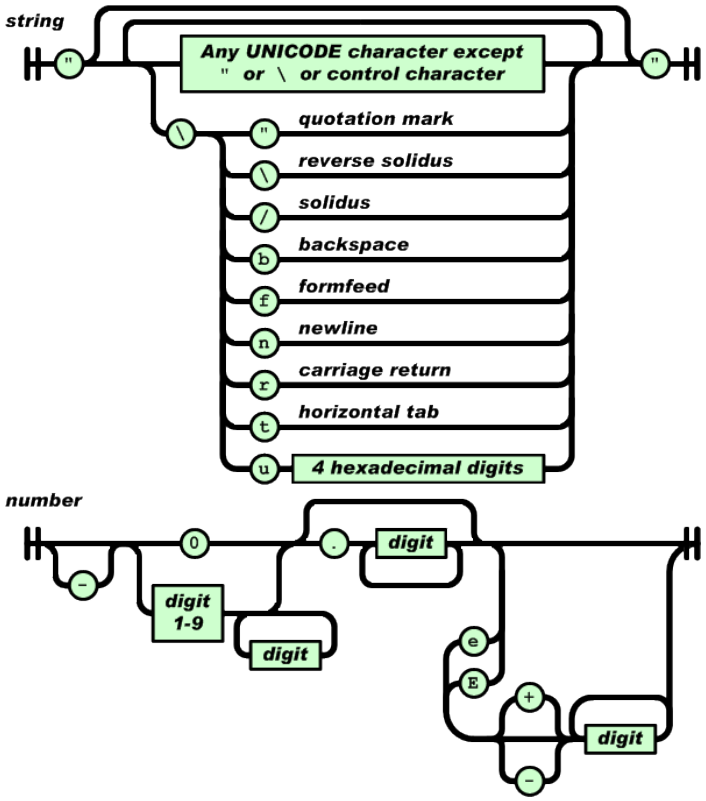
3.1 JSON

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) (⇒ JSON Object)
- Eine geordnete Liste von Werten (⇒ JSON Array)

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





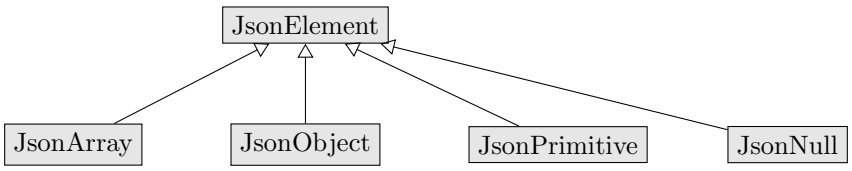
Als Beispiel betrachten wir die JSON Struktur von Universitätsobjekten, die für eine Universität den Namen sowie eine Liste von Kursen enthält:

```
{"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:

```
JsonParser parser = new JsonParser();

JsonObject parsedUni = parser.parse(jsonString).getAsJsonObject();
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:

```
Gson gson = new Gson();
University unikl = new University("TU KL");
unikl.addCourse("SE 1");
unikl.addCourse("SE 2");
unikl.addCourse("SE 3");
unikl.addCourse("Insy");
String json = gson.toJson(unikl);

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

Benutzerdefinierte Übersetzungen für Java-Klassen implementiert man als Subtypen von `JsonSerializer<T>` (von `T` nach JSON) und `JsonDeserializer<T>` (von JSON nach `T`).

```
public class UniSerializer implements JsonSerializer<University> {
    public JsonElement serialize(University uni, Type type,
                                 JsonSerializationContext context) {
        ...
    }
}
```

Instanzen dieser Klassen können in Gson wie folgt eingebunden werden:

```
GsonBuilder builder = new GsonBuilder();
builder.registerTypeAdapter(University.class,
                            new UniSerializer());
builder.registerTypeAdapter(University.class,
                            new UniDeserializer());
Gson gson = builder.create();
```

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

```
// schreibe das University Objekt unikl in Datei unikl.json
gson.toJson(unikl, new JsonWriter(
                     new FileWriter("unikl.json")))

// lies das University Objekt aus der Datei unikl.json
gson.fromJson(new FileReader("unikl.json"),
              University.class)
```

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-2.5.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-2.5.jar:. Main.java
  ```

- Unter Windows:

  ```
  javac -cp gson-2.5.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-2.5.jar:junitrunner.jar:. Test.java
  ```

- Unter Windows:

  ```
  javac -cp gson-2.5.jar;junitrunner.jar;. Test.java
  ```

Beim Ausführen muss der Classpath erneut mit angegeben werden:

- Unter Linux/Mac:

  ```
  java -cp gson-2.5.jar:. Main
  ```

- Unter Windows:

  ```
  java -cp gson-2.5.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-2.5.jar:. softech.junitrunner.Runner Test
  ```

- Unter Windows:

  ```
  java -cp junitrunner.jar;gson-2.5.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 o) {
    if (this == obj) {
       return true;
    }
    if (obj == null) {
       return false;
    }
    if (!(obj instanceof University )) {
       return false; // andere Klasse
    }
    University other = (University) o;
    return this.name.equals(other.name)
      && this.courses.equals(other.courses);
  }
}
```