TECHNISCHE UNIVERSITÄT
KAISERSLAUTERN

Software Entwicklung 1

Annette Bieniusa / Arnd Poetzsch-Heffter

AG Softech
FB Informatik
TU Kaiserslautern

Ströme zur Ein- und Ausgabe

Ströme zur Ein- und Ausgabe I

Ein- und Ausgabe von Daten wird heutzutage meist durch Ströme modelliert.

Begriffsklärung: Datenstrom

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).

Ströme zur Ein- und Ausgabe II

Bei Operationen auf einem Strom kann es zu Verzögerungen kommen:

- beim Lesen, weil augenblicklich 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.

Einführung in Ströme I

Wir betrachten zunächst Ströme zum Lesen:

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

Einführung in Ströme II

Diese Schnittstelle abstrahiert von der Quelle aus der gelesen wird. Mögliche Quellen:

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

Wir betrachten hier die Fälle 1, 2 und 6.

Lesen aus einer Datenstruktur

Wir betrachten das schrittweise Lesen der Zeichen eines Strings. Die Quelle des Stroms wird dem Konstruktor übergeben.

```
public class StringLeser implements CharEingabeStrom {
    private char[] zeichen;
    private int    index = 0;

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

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

Zusammenbauen von Strömen I

Wir betrachten 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!

Zusammenbauen von Strömen II

```
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 {
            return Character.toUpperCase( (char)z );
        }
    }
}
```

Zusammenbauen von Strömen III

```
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 {
            ...
```

Zusammenbauen von Strömen IV

```
            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;
            }
} } }
```

Folgendes Programm zeigt Zusammenbau und Anwendung von Strömen:

Zusammenbauen von Strömen V

```
public class StreamTestMain {
  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();
  }
}
```

Javas Stromklassen I

Stromklassen werden nach den Datentypen, die sie verarbeiten, und ihren Datenquellen bzw. -senken klassifiziert.

Stromklassen sind wichtige programmiertechnische Hilfsmittel.

Die Reader-/Writer-Klassen aus dem Paket `java.io` verarbeiten `char`-Ströme; die Input-/Output-Stromklassen verarbeiten `byte`-Ströme.

Javas Stromklassen II

Die Reader-Klassen unterstützen:

- das Lesen einzelner Zeichen: `int read();`
- das Lesen mehrerer Zeichen aus der Quelle und Ablage in ein char-Array: `int read(char[]);`
- das Überspringen einer Anzahl von Zeichen der Eingabe: `long skip(long);`
- die Abfrage, ob der Strom für das Lesen des nächsten Zeichens bereit ist;
- das Schließen des Eingabestroms: `void close();`
- Methoden zum Markieren und Zurücksetzen des Stroms.

Javas Stromklassen III

Die Writer-Klassen unterstützen:

- das Schreiben einzelner Zeichen:

```
void write( int );
```

- das Schreiben mehrerer Zeichen eines char-Arrays:

```
void write(char[])
```

u. ä.;

- das Schreiben mehrerer Zeichen eines String:

```
void write(String)
```

u. ä.;

- die Ausgabe ggf. im Strom gepufferter Zeichen:

```
void flush() ;
```

Javas Stromklassen IV

- das Schließen des Ausgabestroms:

```
void close()
```

Die genannten Methoden lösen möglicherweise eine `IOException` aus.

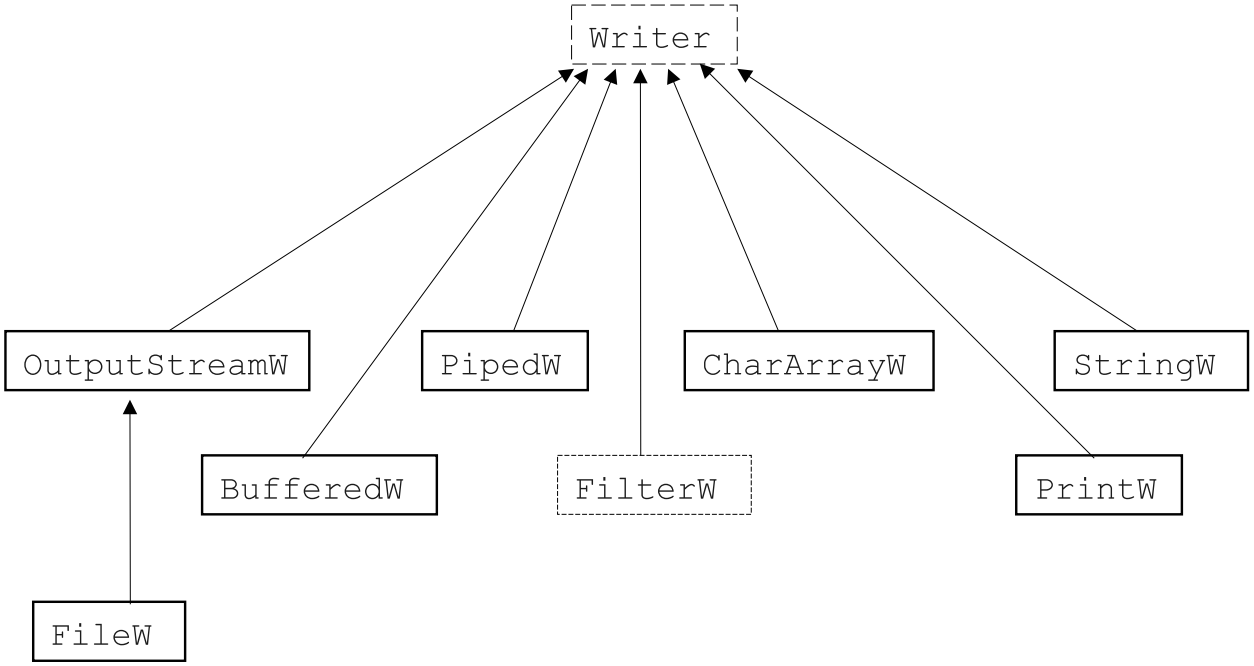
Die von `InputStream` bzw. `OutputStream` abgeleiteten Klassen leisten Entsprechendes für Daten vom Typ `byte`.

Reader-/Writer-Klassen I

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

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

Reader-/Writer-Klassen II



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

`PrintWriter` unterstützen die formatierte Ausgabe von Daten durch die Methoden `print` und `println`, die alle Standarddatentypen als Parameter nehmen.

Bemerkung

Die Konstruktoren ermöglichen das Zusammenhängen von Strömen; hier am Beispiel eines Konstruktors der Klasse `PrintWriter`:

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

Beispiel: Reader-/Writer-Klassen I

```
public class DateiZugriff {
    public static String lesen( String datei )
            throws FileNotFoundException, IOException {
        BufferedReader in =
                new BufferedReader(new FileReader(datei));
        String line, inputstr = "";
        line = in.readLine();
        while( line != null ){
            inputstr = inputstr.concat( line+"\n");
            line = in.readLine();
        }
        in.close();
        return inputstr;
    }
```

Beispiel: Reader-/Writer-Klassen II

```
    public static void schreiben(String datei,String s)
                                        throws IOException {
        PrintWriter out =
            new PrintWriter( new FileWriter(datei) );
        out.print(s);
        out.close();
    }
}
```

Beispiel: Reader-/Writer-Klassen III

```
public class DateiZugriffTest {
  public static void main(String[] args){
    String s;
    try {
      s = DateiZugriff.lesen( argf[0] );
    } catch( FileNotFoundException e ){
      System.out.println("Can't open "+ args[0]);
      return;
     } catch( IOException e ){
      System.out.println("IOException: "+ args[0]);
      return;
     }
```

Beispiel: Reader-/Writer-Klassen IV

```
        try {
            DateiZugriff.schreiben("ausgabeDatei",s);
        } catch( IOException e ){
            System.out.println("Can't open "+ args[0] );
        }
    }
}
```

Schließen von Strömen I

Ströme müssen geschlossen werden, wenn sie nicht mehr verwendet werden. Um dies sicherzustellen, 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.

Schließen von Strömen II

Java 7 bietet eine alternative Syntax für die Deklaration und Verwendung von Strömen an:

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

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

Ein- und Ausgabe von Objekten

Ein- und Ausgabe von Objekten I

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 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.

Ein- und Ausgabe von Objekten II

Andererseits ist Ein- und Ausgabe von Objekten wichtig, um

- Objekte zwischen Prozessen auszutauschen;
- Objekte für nachfolgende Programmläufe zu speichern, d.h. **persistent** zu machen.

Beispiel: Ausgabe von Objekten I

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

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

TodoList tdl = new TodoList();
tdl.add("Lernen", new Date(9,3,2016));
tdl.add("Einkaufen", new Date(1,2,2016));
```

Beispiel: Ausgabe von Objekten II

Was bedeutet es, das von `ll` referenzierte Objekt auszugeben(?):

- nur das `TodoList`-Objekt ausgeben;
- das `TodoList`-Objekt und die zugehörigen `Task`-Objekte ausgeben;
- das `TodoList`-Objekt, die zugehörigen `Task`-Objekte sowie die `String`-Objekte und das `Date`-Objekt ausgeben.

Ausgabe von Objektgeflechten

Um Objekte in ihrem Zusammenwirken mit anderen Objekten wieder einlesen zu können, müssen sie gemeinsam mit allen erreichbaren Objekten ausgegeben werden.

Wegen möglicher Zyklen ist die Implementierung der Ausgabe und des Einlesens von Geflechten nicht einfach.

Bemerkung

- Gibt man ein Objekt mit den erreichbaren Objekten aus und liest es wieder ein, entsteht eine Kopie.
- Referenziert man von mehreren Variablen Teile des gleichen Geflechts, kommt es beim Einlesen ggf. zu mehreren Kopien eines Objekts des ursprünglichen Geflechts.

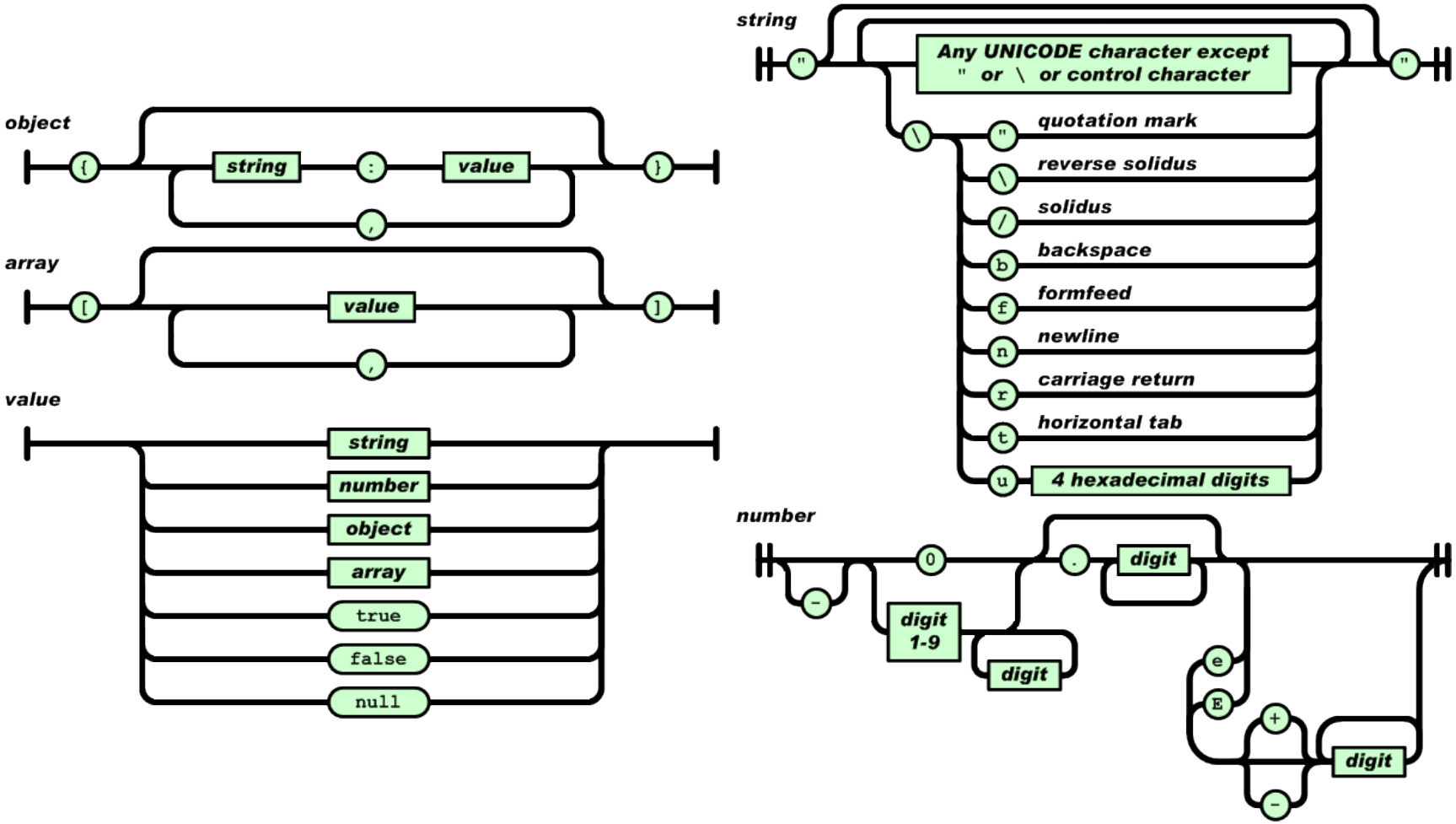
JSON

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


object
{
string
:
value
}
,
array
[
value
]
,
value
string
number
object
array
true
false
null
string
"
Any UNICODE character except
" or \ or control character
"
\
"
quotation mark
\
reverse solidus
/
solidus
b
backspace
f
formfeed
n
newline
r
carriage return
t
horizontal tab
u
4 hexadecimal digits
number
-
0
digit
1-9
digit
.
digit
e
E
+
-
digit

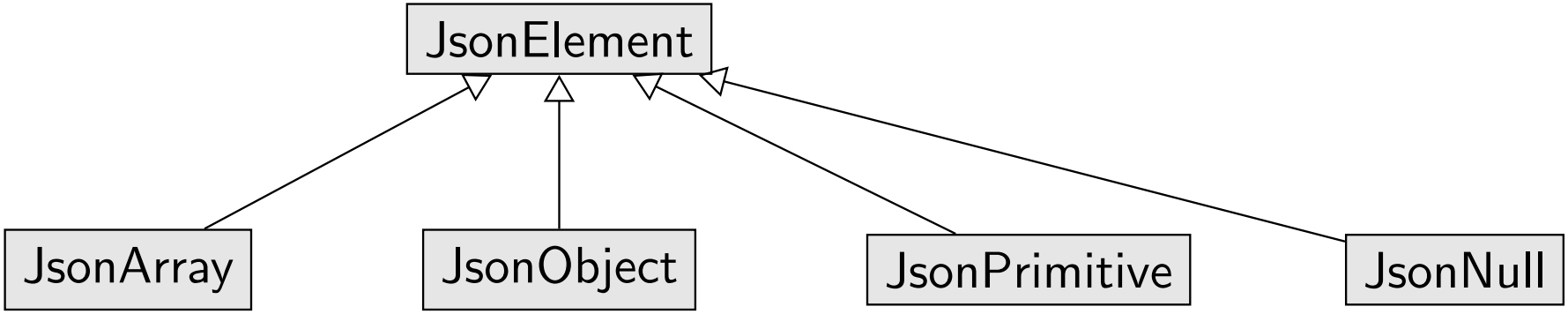
Gson

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

- direkte Beschreibung der zu erzeugenden JSON Ausgabe und die
- Umwandlung von Java-Objekten in JSON
- entsprechende Methoden zum parsen von JSON

Direkte Beschreibung von JSON I

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.

Direkte Beschreibung von JSON II

```
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("Insy");
university.add("courses",
               courses);

gson.toJson(university);
```

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

Direkte Beschreibung von JSON III

```
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(), "Insy");
```

Direkte Beschreibung von JSON IV

- Code stimmt mit ausgegebenem JSON überein
- Struktur direkt beeinflussbar
- Volle Kontrolle, somit auch bei
  - genauen Vorgaben des Ausgabeformats
  - Zyklen im Objektgeflecht

Umwandlung von Java-Objekten in JSON I

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

- Objekte werden als JSON-Objekt,
- Attribute von Objekten werden als Eigenschaften (properties) des JSON Objects,
- Listen und Arrays werden als JSON arrays und
- Maps werden ebenfalls als JSON object ausgegeben.

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

Umwandlung von Java-Objekten in JSON II

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

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

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

Umwandlung von Java-Objekten in JSON III

```
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);
```

Bemerkungen I

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

```
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();
```

Bemerkungen II

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)
```