



# Verteilte Verarbeitung

## Kapitel 4 Serialisierung

# Lernziele

- Sie wissen ...
  - Was Serialisierung bedeutet
  - Wie Sie **manuell** Objekte in Java serialisieren
  - Wie Sie die **eingebaute Serialisierung** nutzen
  - Wie Sie Objekte in ****XML und JSON****-Strukturen serialisieren
  - Was das Serializer Pattern ist

# Was ist Serialisierung?

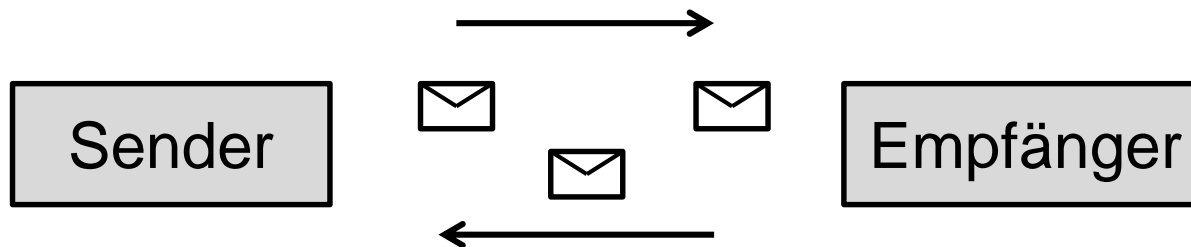
# Übertragung von Objekten im Netzwerk

## Kommunikation *immer* seriell



- Sender und Empfänger vereinbaren Format für Übertragung von Objekten (Format z.B. Java-Serialisierung, XML, ..., IIOP/CDR)
- Objekt wird beim Sender in das Format serialisiert
- Objekt wird beim Empfänger aus dem Format deserialisiert
- Das gilt auch für embedded Software (UART, RS232, ...)

# Wozu müssen Sie sich mit Serialisierung befassen?



- Kommunikation zwischen Sender und Empfänger  
(Achtung: Sowohl Paketorientiert wie Stromorientiert)  
**= Austausch von Nachrichten** (Nachricht als Datei abspeicherbar)  
(unabhängig von der Technologie)
- Preisfrage: Welches Format haben diese Nachrichten?
- Optionen: Java Serialisiert, XML, JSON, Binärformat?
- **-> Eigenschaften von Middleware von Nachrichtenformat abhängig**

# Welche Rolle spielt das Format der Nachrichten?

- Java Serialisierung (z.B. bei Java RMI verwendet)
  - Nur für Kommunikation Java <-> Java geeignet
- Wenn Sender und Empfänger in verschiedenen Programmiersprachen z.B. JavaScript <-> PHP
  - **Sprach- und Plattform-Neutrales Format** zwingend erforderlich: JSON oder XML (oder CDI oder ...)
  - Bei Embedded Systems eher eigenes Binärformat
- Welche Eigenschaften von JSON / XML sind relevant?
  - XML-Dateien / Nachrichten können auf syntaktische und teilweise auf semantische Korrektheit geprüft werden (dazu wird XML-Schema verwendet).
  - XML ist aber aufwendig zu parsen und zu übertragen (große Dateien, aufwendig implementierter Parser)
  - JSON leichtgewichtig (kurze Dateien/Nachrichten, leicht zu parsen) aber: Keine/kaum Syntaxprüfung möglich

# Grundprobleme der Serialisierung

# Grundprobleme der Serialisierung

= Übersetzung von komplexen Datenstrukturen in ein Array aus Bytes.

- Darstellung/Codierung der **elementaren Datentypen** der jeweiligen Programmiersprache
  - String-Darstellung (ASCII / EBCDIC / UTF 8, UTF 32 [Unicode])
  - Zahlen: Integer, Float, Double (Little Endian / Big Endian / ...)
  - Enum?, Boolean?, Date?,
- Darstellung **zusammengesetzter Datentypen**
  - = C-Struct, Java-Klassen wie Kunde / Konto / Vertrag / ...
- Darstellung der **Zeiger / Referenzen**
- Darstellung **objektorientierter Konzepte**
  - **Vererbung**
  - **Listen, Sets, Maps**

Damit haben Sie bei  
Middleware  
häufig Probleme



# Grundprobleme der Serialisierung Software-Lebenszyklus

- Verschiedenen Versionen derselben **Programmiersprache**
  - Wenn interne Repräsentation elementarer Datentypen geändert
  - Z.B. Client läuft auf Java 11, der Server aber noch auf Java 8
  - Folge: Neutrales Format erforderlich?
- Verschiedene Versionen der **Software** (Migrationsproblem)
  - = Alter Client und neuer Server?
  - Z.B. Kundenklasse hat beim Client ein Attribut mehr?
  - Sie müssen also auch an die Migration ihres Nachrichtenformats denken!

# Standardlösungen für diese Probleme

## Web und Smartphone

- Serialisierung selber bauen z.B. nach **Serializer Pattern**
  - Eigener Code übersetzt Datenstrukturen in Byte-Array
  - Typisches Format z.B. **CSV** (= Comma Separated Values)
- Interne Serialisierung der Programmiersprache (Java) nutzen
  - Praktisch kein Aufwand (siehe unten), aber hohes Risiko
- Serialisierung in Standard-Formate, derzeit aktuell
  - CDR = Corba Data Record
  - **XML** = eXtended Markup Language
  - **JSON** = JavaScript Object Notation

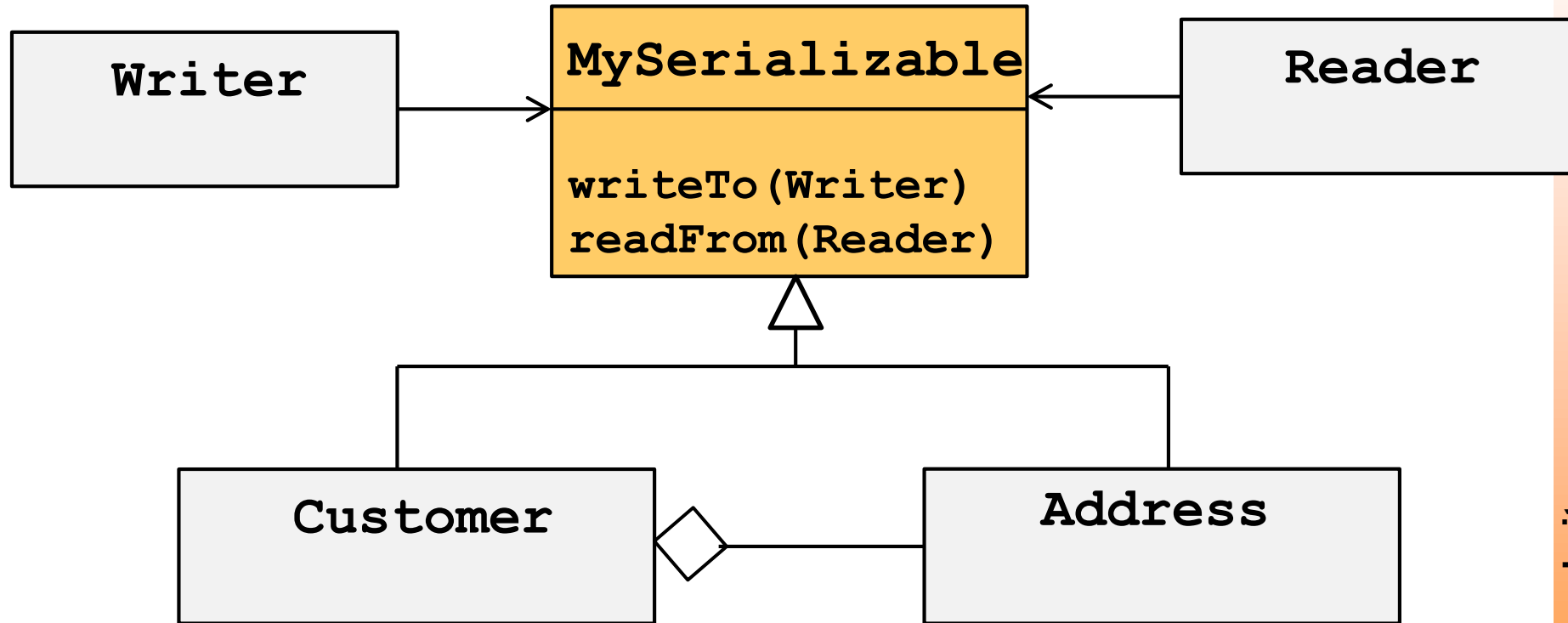
# Standardlösungen für diese Probleme

## Embedded Software

- Datenübertragung ggf. in einem eigenen binären Format
  - Jedes Byte hat Bedeutung
  - Position des Bytes in der Nachricht festgelegt
- Sonderprobleme
  - Grammatik zum Parsen der Nachrichten im Code vergraben
  - Also keine explizite Grammatik
  - Problem der Evolution von Systemen, z.B. Erweiterung des Nachrichtenformats mit zusätzlichen Inhalten
- Lösungsmöglichkeiten z.B. Google Protobuf

# Das Serializer Pattern

# Serializer Pattern



Vgl. Serializer Pattern (Riehle et al.), PLOP 1996

# Idee Serializer-Pattern (selbst gemacht)

## Kapselung der Codierung in Reader/Writer

```
public interface Reader {  
    // Elementare Datentypen  
    public int readInt();  
    public String readString();  
    ...  
    // Zusammengesetzte Datentypen  
    public Serializable readObject();  
}
```

```
public interface Writer { // Analog wie oben  
    public void writeInt(int i);  
    public void writeString(String s);  
    public void writeObject(Serializable object);  
}
```

# Beispiel für die Nutzung der Interfaces

```
public class Kunde implements MySerializable {  
    private String nummer;  
    private String name;  
    private Adresse firmenAdresse;  
    private Kunde geworbenerKunde;  
  
    public void readFrom(Reader r) {  
        this.name = r.readString();  
        this.nummer = r.readString();  
        this.firmenAdresse = (Adresse) r.readObject();  
        this.geworbenerKunde = (Kunde) r.readObject(); }  
  
    public void writeTo(Writer w) {  
        w.writeString(name);  
        w.writeString(nummer);  
        w.writeObject(firmenAdresse);  
        w.writeObject(geworbenerKunde); }  
}
```

# Problem der Referenzen

- Allgemeines Problem mit Objektgeflechten: Wie werden Referenzen zu anderen Objekten (de)serialisiert? [Sonderproblem: **Zyklen**]
- Serialisierung: Anstelle des referenzierten Objekts nur seine Objektidentität (z.B. **fachlicher Schlüssel**) verwenden.  
(vgl. „Aggregate“ aus Domain Driven Design nach Evans)
- Phase 1: in Deserialisierung – Objekte lesen
  - Alle gelesenen Objekte werden auch in einer Map gespeichert
  - Schlüssel der Map: Objektidentität
- Phase 2: Referenzen setzen
  - Um Referenz aufzulösen: get(OID) in der Map



# CSV-Format als Brücke zu Excel

## Beispiel für Serializer Pattern

# Serialisierungsformat: CSV

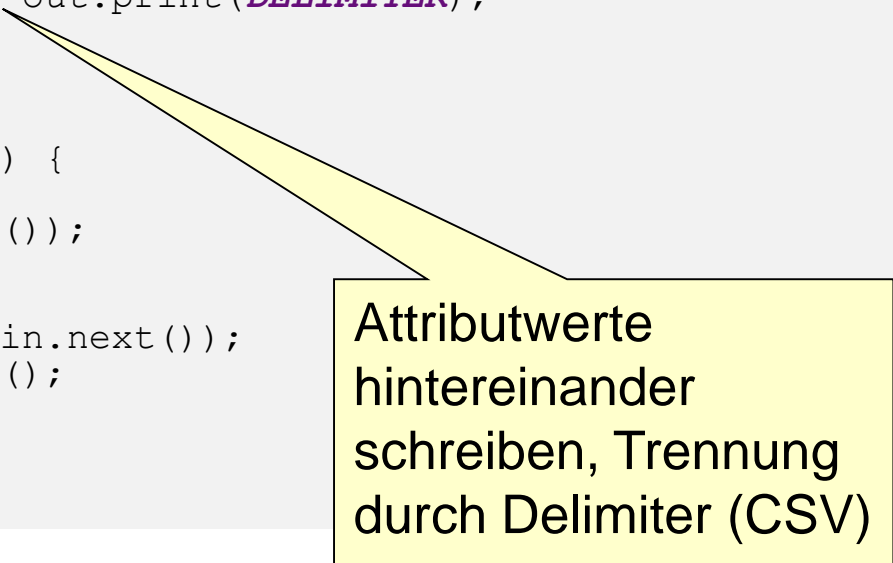
## *CSV = Comma-Separated Values*

- Leicht zu parsen / zu schreiben
- Kann mit MS-Excel oder MS-Project geöffnet werden
- Standard z.B.: RFC 4180 - Common Format and MIME Type for Comma-Separated Value
- Beispiel:  
**Gerd;Beneken;Professor;B 1.21; ...**  
**Markus;Breunig;Professor; B 1.21; ...**

# Manuelle Serialisierung ... (etwa als CSV)

```
public interface CSVSerializer {  
    void readFrom(Scanner in);  
    void writeTo(PrintStream out);  
}
```

```
public class Customer implements CSVSerializer {  
    public static final String DELIMITER = ";";  
    @Override  
    public void writeTo(PrintStream out) {  
        out.print(id);          out.print(DELIMITER);  
        out.print(firstname);  out.print(DELIMITER);  
        out.print(lastname);   out.print(DELIMITER);  
        out.print(birthday);   out.print(DELIMITER);  
        postalAddress.writeTo(out); out.print(DELIMITER);  
    }  
  
    @Override  
    public void readFrom(Scanner in) {  
        in.useDelimiter(DELIMITER);  
        id = Long.parseLong(in.next());  
        firstname = in.next();  
        lastname = in.next();  
        birthday = LocalDate.parse(in.next());  
        postalAddress = new Address();  
        postalAddress.readFrom(in);  
    }  
}
```



Attributwerte  
hintereinander  
schreiben, Trennung  
durch Delimiter (CSV)

# Manuelle Serialisierung ...

```
Customer harry = new Customer(1L, "Harry", "Hirsch",  
    LocalDate.of(1950, 3, 18),  
    new Address("Hochschulstr. 1", "83024", "Rosenheim"));  
Customer gerd = new Customer(2L, "Gerd", "Beneken",  
    LocalDate.of(1971, 5, 10),  
    new Address("Schütteweg 4", "26384", "Wilhelmshaven"));  
  
try (PrintStream printStream  
    = new PrintStream("customers.csv")) {  
  
    harry.writeTo(printStream);  
    printStream.println();  
    gerd.writeTo(printStream);  
    printStream.println();  
  
} catch (FileNotFoundException e) {  
    e.printStackTrace();  
}
```

# Diskussion: Manuelle Serialisierung und CSV

- Einfache Brücke zu Excel und anderen Datenquellen / Senken
- Relativ leicht implementierbar
- Beliebiges Format möglich: String, Byte-Array, Cobol-Copy-Strecke
- Aber: Daten müssen als EINE Tabelle darstellbar sein
- Aber: Hohe Wahrscheinlichkeit von Fehlern, damit hoher Testaufwand

# Problem der Referenzen

- Allgemeines Problem mit Objektgeflechten: Wie werden Referenzen zu anderen Objekten (de)serialisiert? [Sonderproblem: **Zyklen**]
- Serialisierung: Anstelle des referenzierten Objekts nur seine Objektidentität (z.B. **fachlicher Schlüssel oder Seriennummer**) verwenden.
- Phase 1: in Deserialisierung – Objekte lesen
  - Alle gelesenen Objekte werden auch in einer Map gespeichert
  - Schlüssel der Map: Objektidentität
- Phase 2: Referenzen setzen
  - Um Referenz aufzulösen: get(OID) in der Map

# Serialisierung in Java (Interface Serializable)

# Automatische Serialisierung in Java

- Objekte werden automatisch (de-) **serialisiert**
- = Hin- und Zurückverwandlung in einen **Bytestrom**.
- Stream-Pärchen, das Objekte lesen und schreiben kann:

**ObjectInputStream: Object readObject()**

**ObjectOutputStream: void writeObject(Object)**

- Objekte müssen das **Serializable-Interface** implementieren!
- Falls eine zu serialisierende Klasse das Serializable-Interface nicht unterstützt: **NotSerializableException**



# Interface Serializable

`Serializable` ist nur ein Marker-Interface

```
public interface Serializable { };
```

Also:

```
public class MyClass(...) implements Serializable  
{  
    ...  
}
```

# Lesen und Schreiben von Objekten

- Schreiben mit `writeObject(...)`.
  - Alle Felder und Objekte der Klasse rekursiv
  - **transient** oder `static` Attribute werden nicht geschrieben

```
FileOutputStream f= new FileOutputStream(...);  
ObjectOutputStream s= new ObjectOutputStream(f);  
s.writeObject(new Date());  
s.close();
```

- Lesen über `readObject(...)`

```
FileInputStream g= new FileInputStream(...);  
ObjectInputStream p= new ObjectInputStream(g);  
Date d = (Date) p.readObject();  
p.close();
```

# Beispiel Customer und Adresse

```
public class Customer implements Serializable {  
    private Long id;  
    private String firstname;  
    private String lastname;  
    private transient String password;  
    private LocalDate birthday;  
  
    private Address postalAddress;  
    // ...  
}
```

```
public class Address implements Serializable {  
    private String street;  
    private String postalCode;  
    private String city;  
    // ...  
}
```

# Beispiel für serialisierten Kunden (vgl. Code zur Übung)

```

-í  sr    de.fhr.inf.vv.exp2.javaser.Kunde      LL  L
firmenAdresset $Lde/fhr/inf/vv/exp2/javaser/Adresse;L
geworbenerKundet "Lde/fhr/inf/vv/exp2/javaser/Kunde;L
namet  Ljava/lang/String;L nummerq ~  xpsr
"de.fhr.inf.vv.exp2.javaser.Adresse      LL  L  ortq ~
L  plzq ~  L  strasseq ~  xpt Clausthal-Zellerfeldt
35678t  Schö;tteweg 3sq ~  q ~  pt
Hugo Habichtt  4711t
Willi Winzigt  0815

```

## serialVersionUID

- Compiler erzeugt für jede Serializable-Klasse beim Compilerlauf eine ID (Hashcode [SHA] der Klasse)
  - → Serialisierte Objekte lassen sich nur mit der **identischen** **.class**-Datei deserialisieren
  - Abhilfe: Eigene **serialVersionUID** steuert die **Kompatibilität** von serialisierten Daten

```
public class Kunde implements Serializable
{
    private static final long serialVersionUID = 0x01;

    // Methoden ...
}
```

# Diskussion Java Serialisierung

- ***Nur für Kommunikation Java <-> Java*** geeignet in einer anderen Sprache müssten Sie einen de-serialisierer nachbauen
- Einfach zu verwenden (nur ein Interface)
- Kommt mit ***beliebig komplexen Objektgeflechten*** klar (= Vererbung, Listen, Maps kein Problem, „Kreise“)
- Probleme
  - Bei Weiterentwicklung der Software (-> daher die serialVersionUID)
  - Kommunikation mit Systemen auf anderen Plattformen

# JSON und Serialisierung

# Serialisierungsformate: JSON

## ***JSON = JavaScript Object Notation***

- Standard ***rfc8259*** The application/json Media Type for JavaScript Object Notation (JSON)
- Teilweise selbstbeschreibend (nur Objekte mit Attributen)
  - = Name / Wert-Paare
  - Werte können: Strings, Listen oder zusammengesetzte Datenstrukturen sein
  - UTF 8
- Sehr weit verbreitet
  - Insbesondere im Zusammenhang mit REST (später mehr)
  - Wesentlich leichtgewichtiger als XML
  - Inzwischen auch mit Schemadefinition (<http://json-schema.org/>)
  - Und mit Pointern (<https://tools.ietf.org/html/rfc6901>)



# JSON-String für Kunden

```
{  
  "id":2,  
  "firstname":"Gerd",  
  "lastname":"Beneken",  
  "birthday":"1971-05-10",  
  "postalAddress":{  
    "street": "Schütteweg 4",  
    "postalCode": "26384",  
    "city": "Wilhelmshaven"},  
  "hobbies":[  
    "Schach",  
    "Halma",  
    "Fussball"],  
  "isPremium":false  
}
```



Seriali-  
sieren

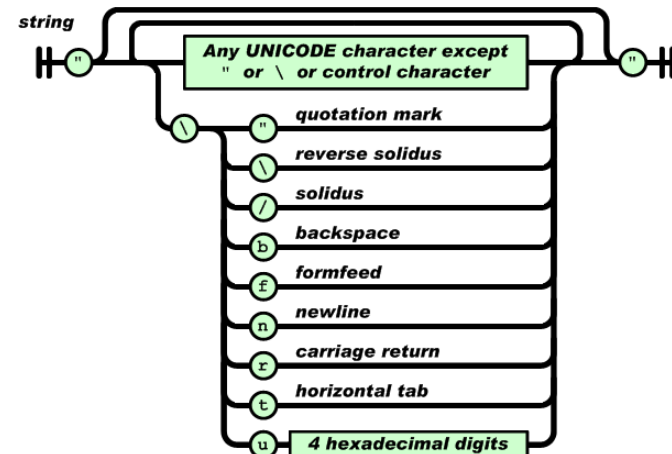
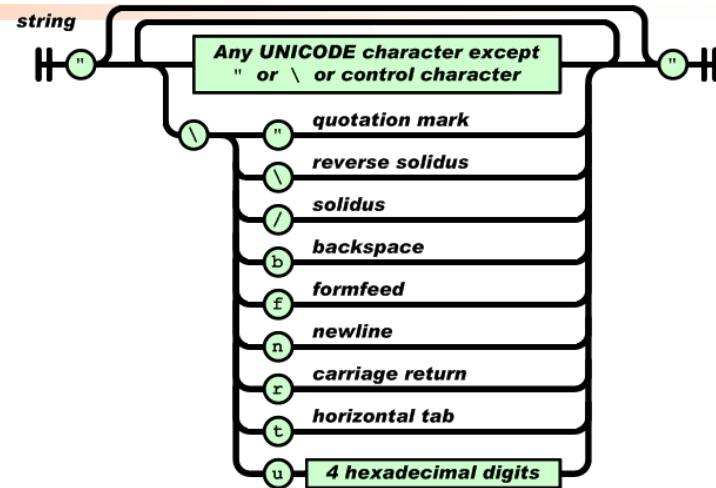
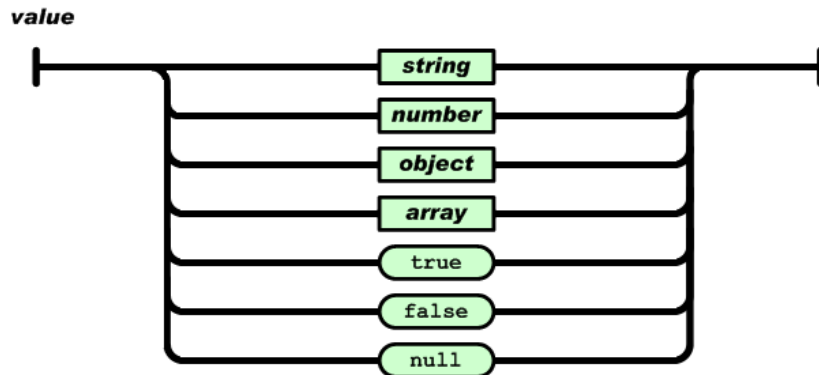
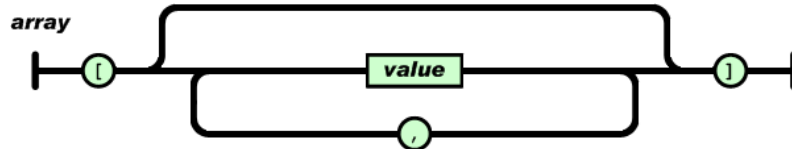
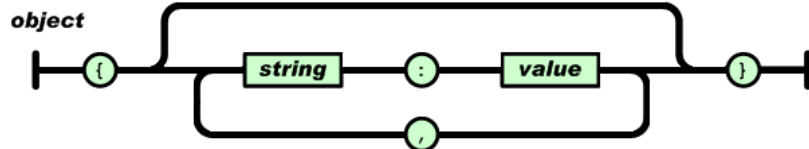


DeSeri-  
alisieren

```
public class Customer {  
  
  private Long id;  
  private String firstname;  
  private String lastname;  
  private LocalDate birthday;  
  
  private Address postalAddress;  
  
  private List<String> hobbies;  
  private boolean isPremium;  
  
}  
  
public class Address {  
  private String street;  
  private String postalCode;  
  private String city;  
  
}
```

# JSON Grammatik

vgl. json.org



# JSON-(De) Serialisierung

## z.B. mit GSON - Bibliothek

### ■ Lesen

```
String jsonString = ...;  
Gson gson = new Gson();  
Customer k = gson.fromJson(jsonString, Customer);
```

### ■ Schreiben

```
Customer k = ...;  
Gson gson = new Gson();  
String json = gson.toJson(k);
```

# JSON Schema Beispiel

vgl. <https://json-schema.org/>

```
{ "title": "Person",  
  "type": "object",  
  "properties": {  
    "firstName": { "type": "string" },  
    "lastName": { "type": "string" },  
    "age": { "description": "Age in years",  
            "type": "integer", "minimum": 0 }  
  },  
  "required": ["firstName", "lastName"]  
}
```

# Diskussion JSON als Nachrichtenformat

- ***Derzeit häufig in Middleware verwendet (Rest, „HTTP-Schnittstelle“, ...)***
  - Neutrales, plattformunabhängiges Format
  - Schema offenbar noch immer in Arbeit
- Java: Einfach zu verwenden, viele Bibliotheken vorhanden (Hier GSON bei google code).
- Sehr leicht zu parsen
- Weitgehend selbstbeschreibendes Format (= Name / Wert Paare)
- Einschränkungen
  - Nur Baumstruktur (keine Kreise)
  - Probleme mit Referenzen, Probleme mit Vererbung

# Binärformate

(Embedded Systems?)

# Lösungsmöglichkeit: Protobuf (Google)

<https://developers.google.com/protocol-buffers/>

- Serialisierer mit expliziter Schnittstellenbeschreibung (Grammatik)
- Sprachen: Java, C++, Python
- Codegenerator erzeugt aus Schnittstellenbeschreibung Java-Code
- Beispiel für Nachrichtenformat

```
message Person {  
    required string name = 1;  
    required int32 id = 2;  
    optional string email = 3;  
}
```

- Programm nutzt dann generierten Code (wie Person)

# Diskussion Binärformat

- Binärformate in Sprachen wie C oder Cobol populär
  - Grund: Byte-Array kann auf interne Datenstruktur/Copy-Strecke gecastet werden
  - Hoffnung: Ersparnisse beim Bau des Parsers (Laufzeit, Hauptspeicher) und Verringerung des Protokoll-Overheads
- Risiko der Fehlinterpretation einzelner Bytes hoch
- Lösung: Frameworks wie Protobuf, welche den Parser und die Nachrichtenerzeugung generieren
- Vorsicht, selbsterfundene Protokolle müssen robust gegen neue Versionen ihrer eigenen Software sein.