UNIVERSITÄT
DUISBURG
ESSEN

Distributed Objects & XML

Kapitel 3: Datenrepräsentation

PALUNO
The Ruhr Institute for Software Technology

Teil 1 – Heterogenität

Teil 2 – XML

Teil 3 – JSON

Teil 4 – Google Protocol Buffers

Heterogenität

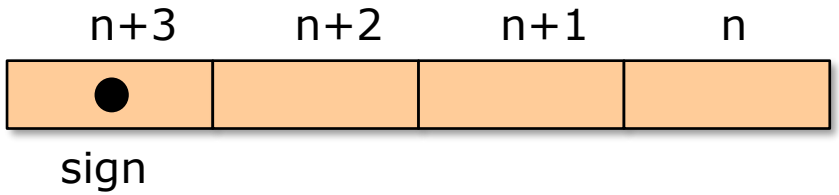
- Verteilte (OO-)Applikationen sind ganz natürlich durch Heterogenität und Flexibilität gekennzeichnet
  - Kein Rechnerknoten ist wie der andere
    - Software
    - Hardware
  - Neue Rechnerknoten und Anwendungen kommen hinzu
- Verschiedene Programmiersprachen
  - Gemeinsames Objektmodell
  - Gemeinsame IDL
  - Anbindung an Programmiersprachen
- Heterogene Middleware
  - Interoperation
  - Interworking
- Heterogene Datenrepräsentationen

Heterogenität – Technische Unterschiede (I)

- Darstellung primitiver Datentypen
- Datenrepräsentation auf der bit-Ebene sehr unterschiedlich
  - Reihenfolge der Bytes im Hauptspeicher und in Maschinenregistern
  - big und little endians

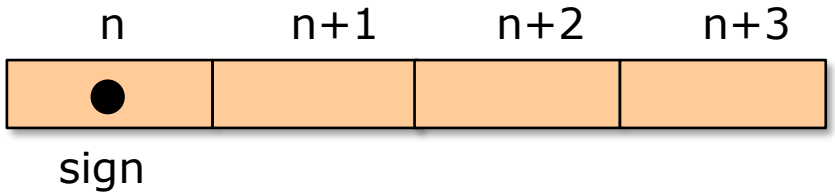
big-endians
(z.B.: Atmel AVR32)
Höchste Wertigkeit zuerst

memory



little-endians
(z.B.: x86-basierte Systeme)
Niedrigste Wertigkeit zuerst

memory



Heterogenität – Technische Unterschiede (II)

- Unterschiedliche Zeichendarstellungen

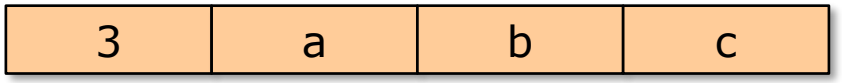
	P	r	e	u	ß	e	n		M	ü	n	s	t	e	r
EBCDIC	D7	99	85	A4	A2	A2	85	95	40	D4	A4	85	95	A2	A3	85	99
ASCII	50	72	65	75	73	73	65	6E	20	4D	75	65	6E	73	74	65	72
ISO-8859-1	50	72	65	75	DF	65	6E	20	4D	FC	6E	73	74	65	72		
UCS	0050	0072	0065	0075	00DF	0065	006E	0020	004D	00FC	006E	0073	0074	0065	0072		

Heterogenität – Technische Unterschiede (II)

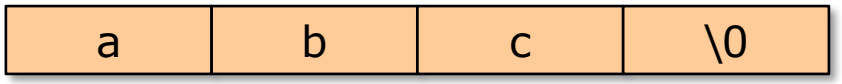
- Unterschiede in den Programmiersprachen

z.B.: der String
„abc“

Pascal memory

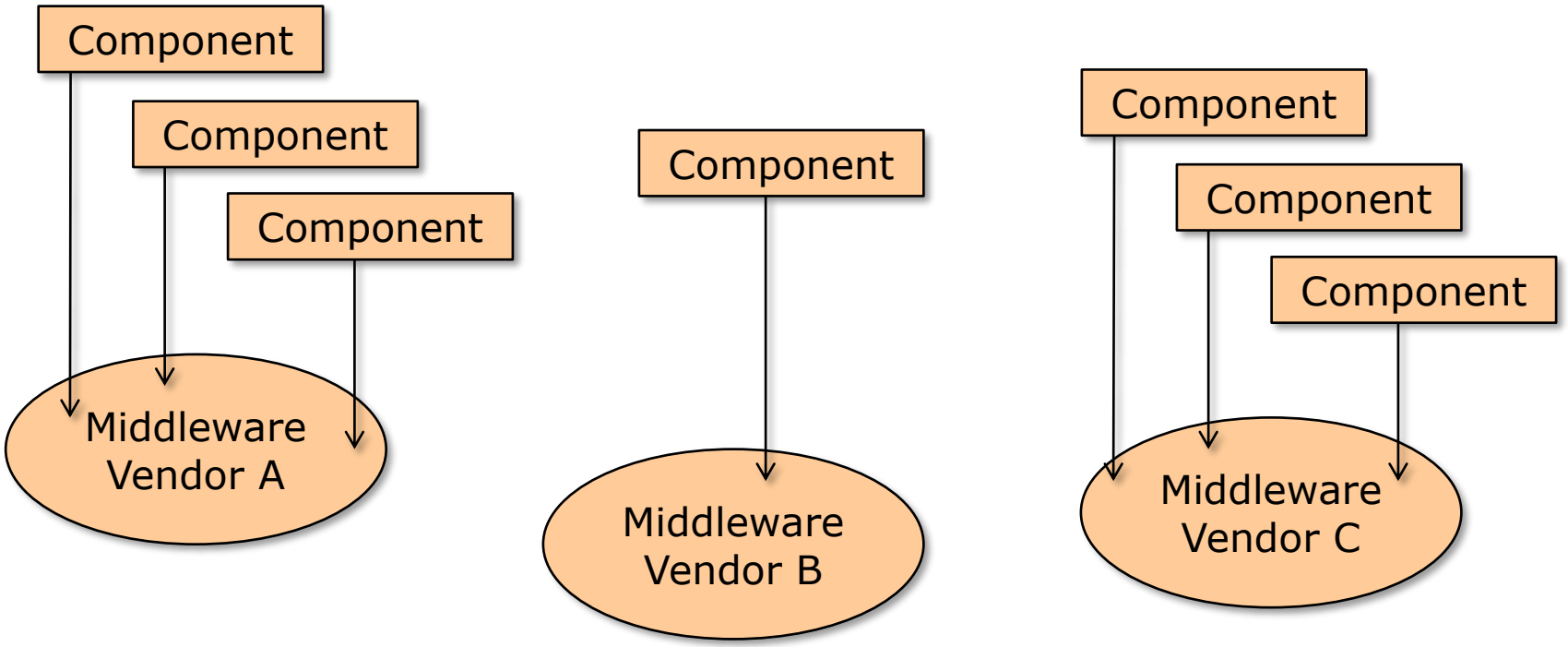


C++ memory



Heterogenität – Unterschiede in der Middleware (I)

- Selbst ohne die gerade betrachteten technischen Unterschiede, kann es Differenzen geben
  - Unterschiedliche Typen von Middleware
  - Unterschiedliche Implementierungen desselben Typs Middleware



Heterogenität – Unterschiede in der Middleware (II)

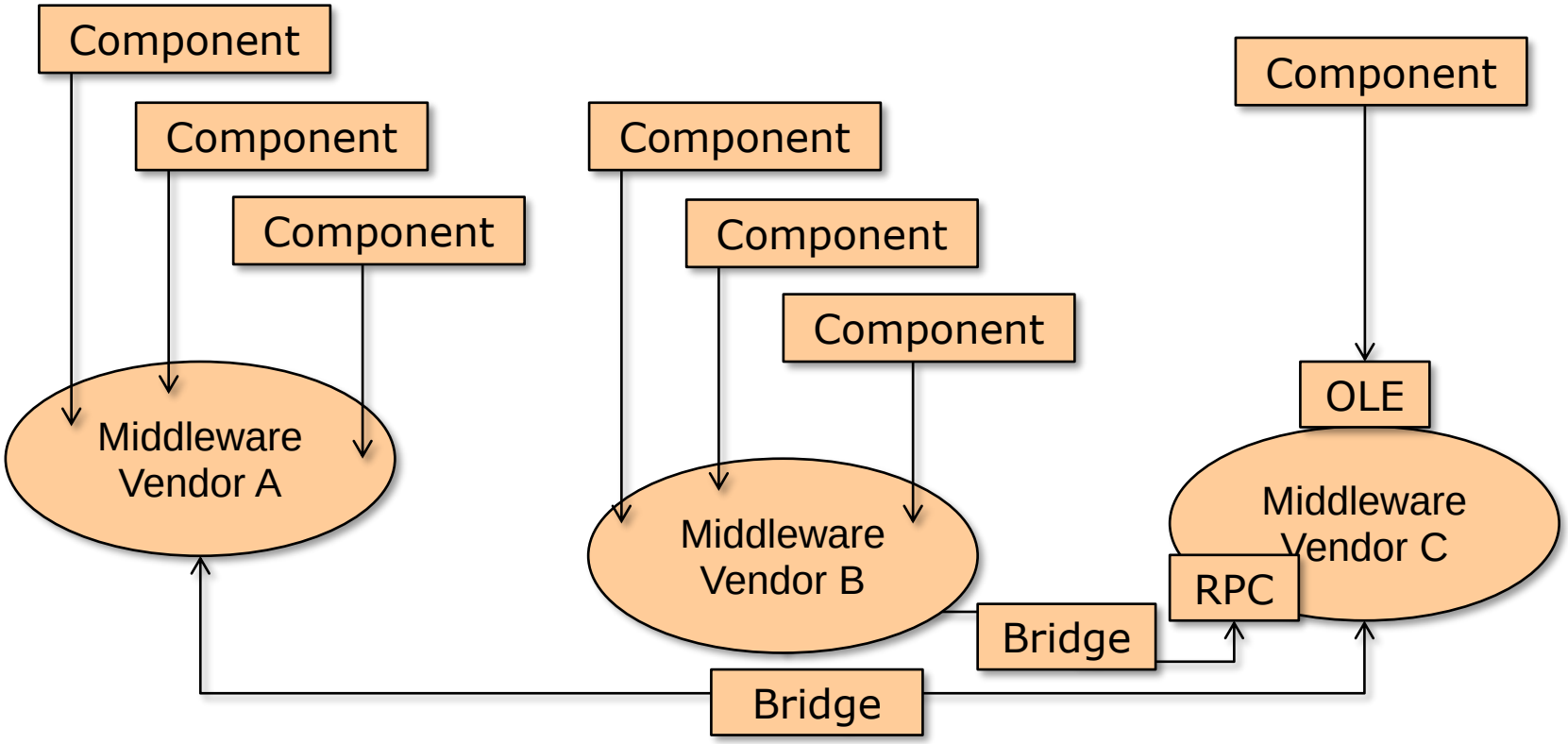
- Unterschiedliche Middleware-Implementierungen
  - Verfügbare Programmiersprachen-Anbindungen
  - Verfügbare Services & Funktionen
  - Unterstützte Hardware / Betriebssysteme / ...
  - Unterstützte Kommunikationsprotokolle
  - Unpräzise Spezifikation
    - Z. B. keine Vorgabe der Zeichenkodierung

Heterogenität – Unterschiede in der Middleware (III)

- Unterschiedliche Middleware-Typen
- Das gleiche Problem wie bei den Implementierungen
- Zusätzlich:
  - Unterschiedliche Objektmodelle
  - Evtl. nicht einmal ein gemeinsamer Kern

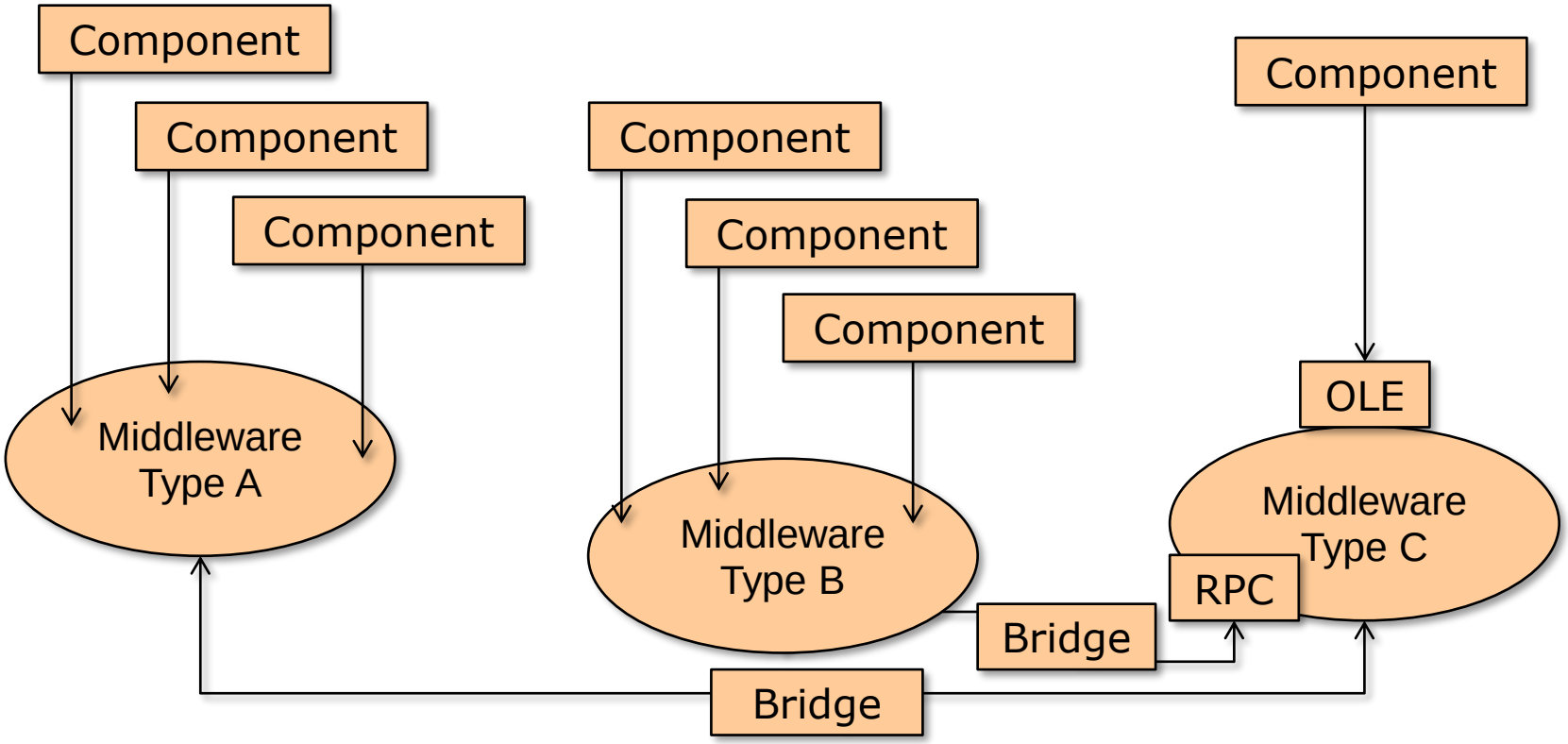
Heterogenität – Integration von Middleware (I)

- Interoperation
  - Unterschiedliche Implementierungen des gleichen Standards arbeiten zusammen
  - Definition von Interaktionsprotokollen



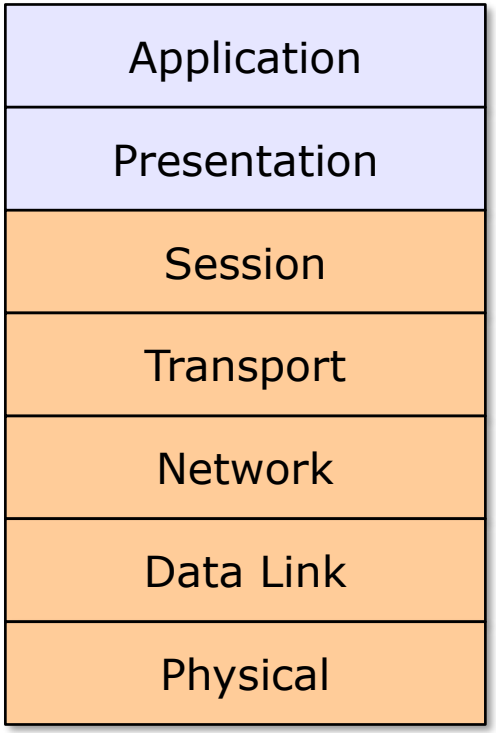
Heterogenität – Integration von Middleware (II)

- Interworking
  - Unterschiedliche Middleware-Ansätze arbeiten zusammen
  - Abbildung der Objektmodelle
  - Definition von Interaktionsprotokollen



Heterogenität – Charakteristiken der Lösung des Problems

- Datenrepräsentationen müssen umgewandelt werden
- Umwandlungen sollten transparent für den Anwendungsentwickler sein
- Umwandlungen können implementiert sein in:
  - Presentation Layer
  - Application Layer
    - Jede Anwendung einzeln
    - Plattform



Heterogenität – Ansätze zur verteilten Datenrepräsentation

- Presentation Layer
  - Sun: XDR
  - OMG: CDR
  - ...
- Anwendungsebene
  - ASN.1
  - XML
  - ...
- Plattform
  - Virtuelle Maschinen, z.B. Java

Teil 1 – Heterogenität

Teil 2 – XML

Teil 3 – JSON

Teil 4 – Google Protocol Buffers

XML – Motivation

- Datenauszeichnungssprache (Extensible Markup Language)
- Strukturbeschreibung von Daten als Basis für Flexibilität
  - Verteilte Systeme müssen auf Änderungen nach dem eigentlichen Design reagieren
  - Strukturbeschreibung nicht statisch, sondern direkt beim Inhalt
- Anforderungen an Systeme
  - Verarbeitung von Dokumenten soll system- und herstellerunabhängig sein
  - Darstellung und Verbreitung von Information soll über unterschiedliche Medien möglich sein

XML – Herkunft

- XML ist Untermenge der Standard Generalized Markup Language (SGML) mit ein paar Einschränkungen
  - Meta-Sprache
  - 1986 als ISO-Standard 8879 verabschiedet
  - Weniger Elemente, einfacher und besser automatisch zu verarbeiten
- Unterschiede zwischen XML und SGML: http://www.w3.org/TR/NOTE-sgml-xml-971215.html

XML – Aufbau (I)

- Grundsätzlich ganz einfach:
  - Öffnende und schließende Tags
    - Erkennbar an < und >
    - Paarweise, immer richtig schachteln
    - Bessere Lesbarkeit durch Einrückungen
  - Jeder Tag mit Inhalt und Attributen
    - Spezielle „leere“ Tags ohne Inhalt möglich
    - Attribute in Anführungszeichen

XML – Aufbau (Beispiel)

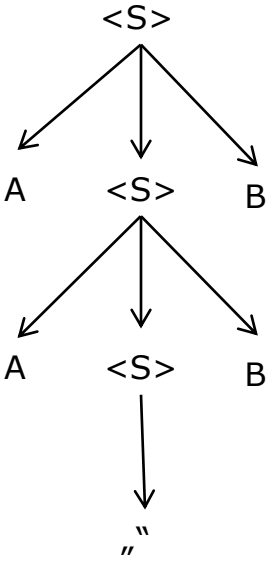
```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<?xml-stylesheet href="personalkartei.xsl" type="text/xsl" ?>
<!DOCTYPE personalkartei SYSTEM "personalkartei.dtd">
<personalkartei>
   <person>
      <name>
         <vorname>Alfons</vorname>
         <nachname>Viertel vor Zwölfte</nachname>
      </name>
      <strasse>Muster Str. 11</strasse>
      <plz>47111</plz>
      <ort>Lummerland</ort>
      <telefon art="privat">01234/56789</telefon>
      <telefon art="buero">054321/9999999</telefon>
   </person>
</personalkartei>
```

XML – Aufbau (II)

- Strukturierung
  - Inhalt mit Nutzdaten und Strukturbeschreibung
  - Strukturdefinition (= Grammatik von Dokumenten)
    - Document Type Definition (DTD)
    - XML Schema
- Darstellung
  - Document Object Model (DOM)
  - Standardisiertes Datenzugriffsmodell
  - Plattform- und sprachunabhängige Schnittstelle für dynamischen Zugriff auf Inhalt, Struktur und Layout eines XML-Dokumentes

XML – Struktur
Exkurs: Kontextfreie Grammatiken

- Formale Definition einer Strukturbeschreibung
- G = (T, N, P, S)
  - T = Terminalsymbole
  - N = Nonterminalsymbole
  - P = Produktionen
  - S = Startsymbol
- Ableitung als zentrale Möglichkeit der Anwendung
  - Für ein Wort einer Sprache gibt es eine Folge von Produktionsanwendungen, die mit dem Wort endet
  - P = { <S> -> A <S> B, <S> -> „“}
  - <S> -> A <S> B -> AA <S> BB -> AABB
  - Zu jeder Ableitung kann ein Baum gefunden werden



XML – Struktur – DTD (I)

- DTD: Document Type Definition
  - https://www.w3schools.com/xml/xml_dtd_intro.asp
- Grammatik für XML (Strukturdefinition)
- Grundelemente:
  - Elementtypen
  - Attributlisten
  - Entitäten
  - Notationen
- XML-Dokument entspricht genau der Baumstruktur, die sich aus der Anwendung der Produktionsregeln der Elementtypen ergibt.

XML – Struktur – DTD (II)

- Elementtypen
  - Terminal- und Nichtterminalsymbole
  - Inhalt: EMPTY, ANY, andere Elementtypen
  - Operatoren:
    - Reihenfolgen (,)
    - Alternativen (|)
    - Gruppierung („(…)“)
    - Kardinalität (*, +, ?, keines davon)
  - Terminale: #PCDATA (Parsed Character Data)
  - Beispiel:

```
<!ELEMENT person (name,email*)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT email (#PCDATA)>
```

XML – Struktur – DTD (III)

- Attributlisten
  - Wichtigste Attributtypen
    - CDATA
    - ID
    - IDREF / IDREFS
    - Alternative Werte
  - Attributvorgaben:
    - #REQUIRED (Notwendig)
    - #IMPLIED (Optional)
    - "…" (Standardwert)
    - #FIXED "…" (Fester Wert)
  - Beispiel

```
<!ATTLIST person id ID #REQUIRED>
<!ATTLIST person note CDATA #IMPLIED>
<!ATTLIST person contr (true|false) "false">
```

XML – Struktur – DTD (IV)

- Entities
  - Definierte Kürzel, wie Textbausteine
  - Beispiele aus HTML:
    - ü (ü)
    -   (Geschütztes Leerzeichen)
  - PUBLIC (öffentlich bekannt) und SYSTEM (externe Referenz)
  - Beispiele:

```
<!ELEMENT uebungsblatt(#PCDATA)>
<!ENTITY praefix "DOX-" >

<!– Beispiel für eine externe Referenz -->
<!ENTITY datenquelle SYSTEM "news.txt" >
```

  - Nutzung:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE uebungsblatt SYSTEM "uebungsblatt.dtd">
<uebungsblatt>&praefix;Übung 1</uebungsblatt>
```

XML – Struktur – DTD (V)

- Notationen
  - Hinweise zur Datenverarbeitung
  - PUBLIC und SYSTEM
  - Hinweise nicht standardisiert
  - Beispiel:

```
<!NOTATION gif PUBLIC "-//APP/Photoshop/4.0
photoshop.exe">
```

XML – Struktur – Beispiel DTD (I)

```
<!ELEMENT personnel (person+)>
<!ELEMENT person (name,email*,url*,link?)>
<!ATTLIST person id ID #REQUIRED
                 note CDATA #IMPLIED
                 contr (true|false) "false"
                 salary CDATA #IMPLIED>
<!ELEMENT name ((family,given)|(given,family))>
<!ELEMENT family (#PCDATA)>
<!ELEMENT given (#PCDATA)>
<!ELEMENT email (#PCDATA)>
<!ELEMENT url EMPTY>
<!ATTLIST url href CDATA "http://">
<!ELEMENT link EMPTY>
<!ATTLIST link manager IDREF #IMPLIED
               subordinates IDREFS #IMPLIED>
<!NOTATION gif PUBLIC "-//APP/Photoshop/4.0 photoshop.exe">
```

XML – Struktur – Beispiel DTD (II)

- Dazu passende XML-Datei:

```
<personnel>
     <person id="1" note="Notiz" contr="true">
               <name>
                         <family>Musterfrau</family>
                         <given>Franziska</given>
               </name>
               <email>franziska.musterfrau@example.org</email>
               <url href="http://www.example.org" />
               <url href="http://www.company.org" />
               <link subordinates="2" />
     </person>
     <person id="2" note="Notiz">
               <name>
                         <family>Krautsalat</family>
                         <given>Herbert</given>
               </name>
               <email>herbert.krautsalat@company.org</email>
               <url href="http://www.company.org" />
               <link manager="1" />
     </person>
</personel>
```

XML – Struktur – XML Schema (I)

- XSD: XML Schema Definition (XSD)
  - https://www.w3schools.com/xml/schema_intro.asp
- Alternative Grammatik für XML (Strukturdefinition)
- Selbst XML
  - Nutzt XML Namespaces
- Mehr Möglichkeiten als DTD
  - Typsystem mit verschiedenen Basistypen
  - Min/Max für Auswahl und Sequenzen
  - Erweiterbarkeit
  - Einschränkungen für Inhalte für Basistypen
- Grundelemente:
  - Atomare Datentypen
  - Einfache und komplexe Elemente
  - Attribute
  - Restrictions

XML – Struktur – XML Schema (II)

- Einfache Elemente
  - Haben Name und einfachen Typ
    - string, decimal, integer, boolean, date, time
  - Dürfen nur Inhalt vom angegebenen Typ haben (insbesondere keine anderen Elemente als Kinder)
  - Können Standardwert oder festen Wert haben
  - Können Attribute bekommen
- Beispiele:

```
<xs:element name="name" type="xs:string"/>
<xs:element name="birthday" type="xs:date"/>
```

XML – Struktur – XML Schema (III)

- Komplexe Elemente
  - Möglichkeit 1: Leere Elemente
    - Haben keinen Inhalt und keine Kindelemente
    - Können Attribute haben
- Beispiel:

```
<xs:element name="student">
  <xs:complexType>
    <xs:attribute name="matrikelnummer" type="xs:integer"/>
  </xs:complexType>
</xs:element>
```

- Ergebnis:

```
<student matrikelnummer="12345" />
```

XML – Struktur – XML Schema (IV)

- Komplexe Elemente
  - Möglichkeit 2: Elemente mit Kindelementen
    - Definieren eine Auswahl möglicher Kindelemente über sog. Indicators
      - all, choice, sequence
      - minOccurs, maxOccurs
    - Können Text (character data) zwischen die Kindelemente mischen
    - Können Attribute haben
- Beispiel:

```
<xs:element name="person">
  <xs:complexType>
    <xs:all>
      <xs:element name="first" type="xs:string"/>
      <xs:element name="given" type="xs:string"/>
    </xs:all>
  </xs:complexType>
</xs:element>
```

XML – Struktur – XML Schema (V)

- Attribute
  - Haben Name und einfachen Typ
    - string, decimal, integer, boolean, date, time
  - Können Standardwert oder festen Wert haben
  - Können als erforderlich markiert werden
- Beispiele:

```
<xs:attribute name="name" type="xs:string" use="required"/>
<xs:attribute name="birthday" type="xs:date"/>
```

XML – Struktur – XML Schema (VI)

- Restrictions
  - Erlauben Einschränkungen für den Inhalt von einfachen Elementen und Attributen
    - Aufzählung möglicher Werte (enumeration)
    - Grenzwerte für Zahlen (maxExclusive, maxInclusive, minExclusive, minInclusive)
    - Präzision von Zahlen (fractionDigits, totalDigits)
    - Länge des Inhalts (length, maxLength, minLength)
    - Reguläre Ausdrücke (pattern)
    - Umgang mit Nicht-Character-Zeichen (whiteSpace)

XML – Struktur – XML Schema (VII)

- Beispiele:

```
<xs:element name="role">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:enumeration value="Student"/>
      <xs:enumeration value="Lecturer"/>
      <xs:enumeration value="Admin"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>

<xs:element name="month">
  <xs:simpleType>
    <xs:restriction base="xs:integer">
      <xs:minInclusive value="1"/>
      <xs:maxInclusive value="12"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

XML – Struktur – XML Schema (VIII)

- Erweiterbarkeit mit Vererbung
  - Geht nur für komplexe Elemente
  - Ähnlich zu Restrictions
- Beispiel:

```
<xs:element name="student" type="studentinfo"/>

<xs:complexType name="personinfo">
  <xs:element name="name" type="xs:string"/>
</xs:complexType>

<xs:complexType name="studentinfo">
  <xs:complexContent>
    <xs:extension base="personinfo">
      <xs:element name="studentid" type="xs:string"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

XML – Struktur – XML Schema (IX)

- Erweiterbarkeit für beliebige Elemente
  - Erlaubt das Hinzufügen von Elementen oder Attributen, die nicht im Schema definiert sind.
- Beispiel:

```
<xs:element name="task">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="title" type="xs:string"/>
      <xs:element name="description" type="xs:string"/>
      <xs:any minOccurs="0"/>
    </xs:sequence>
    <xs:anyAttribute/>
  </xs:complexType>
</xs:element>
```

XML – Struktur – Beispiel XSD (I)

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:bsp="http://de.wikipedia.org/wiki/XML_Schema#Beispiel"
        targetNamespace="http://de.wikipedia.org/wiki/XML_Schema#Beispiel"
        elementFormDefault="qualified">

  <element name="head">
      <complexType>
        <sequence>
          <element name="title" type="string"/>
        </sequence>
      </complexType>
    </element>

  <element name="doc">
    <complexType>
      <sequence>
        <element ref="bsp:head"/>
        <element name="body" type="string"/>
      </sequence>
    </complexType>
  </element>

</schema>
```

Entlehnt von: http://de.wikipedia.org/w/index.php?title=XML_Schema&oldid=103206632

XML – Struktur – Beispiel XSD (II)

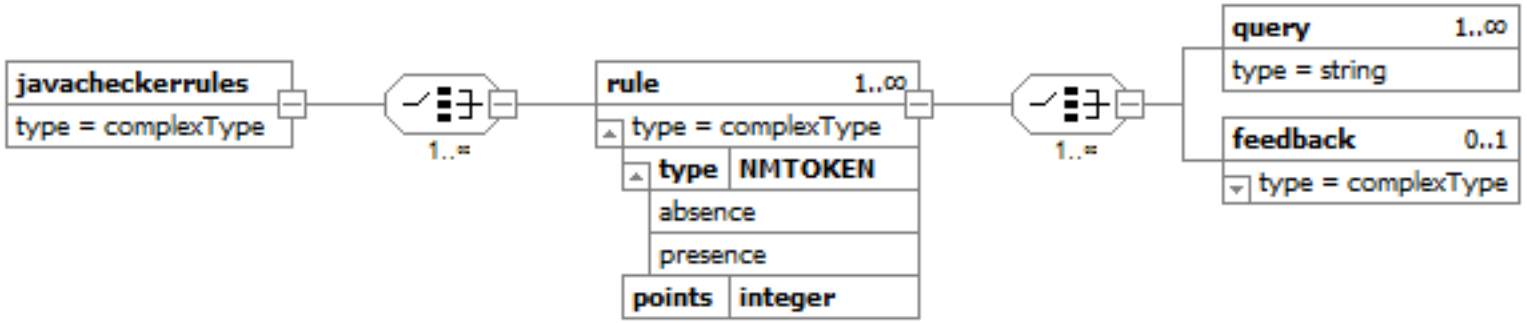
- XML, dass der Strukturdefinition entspricht:

```
<?xml version="1.0" encoding="UTF-8"?>
<doc xmlns="http://de.wikipedia.org/wiki/XML_Schema#Beispiel">
  <head>
    <title>
      Dies ist der Titel
    </title>
  </head>
  <body>
    Dies ist der Text.
  </body>
</doc>
```

Quelle: http://de.wikipedia.org/w/index.php?title=XML_Schema&oldid=103206632

XML – Struktur – Grafische Darstellung

- XSD ist selbst XML, also ein Baum
- XSD kann daher als Baumstruktur grafisch dargestellt werden
  - Kompaktere Darstellung
  - Intuitive Lesbarkeit
- Freie Tools zur Erstellung von Diagrammen verfügbar
- Beispiel:



XML – Inhalt – Parsen (I)

- Parser ermöglichen Zugriff auf Inhalte eines XML-Dokuments
- Drei Konzepte zur Auswahl:
  - DOM-Parser
  - SAX-Parser
  - StAX-Parser
- Viele konkrete Werkzeuge/Implementierungen
  - Große Anzahl für fast alle Programmiersprachen
  - Z.B. Java API for XML Processing (JAXP)
- Praktisches Tutorial: http://tutorials.jenkov.com/java-xml/index.html

XML – Inhalt – Parsen (II)

- DOM-Parser
  - DOM = Document Object Model
  - Verarbeitet stets ganzes Dokument
  - Transformation in hierarchische Baumstruktur (DOM)
- Objekt-orientierter Zugriff auf die Inhalte der XML-Datei
  - Client-Code ruft Methoden der Parser-API auf und verarbeitet deren Datentypen
  - Beispiele (Pseudocode):
    - `element.getAttribute("...")`
    - `element.getChild("...")`
- Vorteil:
  - Intuitive Nutzung
- Nachteil: Schlechte Performanz, gerade wenn nur ein paar Elemente betrachtet werden sollen
  - Lange Laufzeit für das Laden des gesamten XML-Dokuments
  - Hoher Speicherbedarf

XML – Inhalt – Parsen (III)

- SAX-Parser
  - SAX = Simple API for XML
  - Schrittweiser Zugriff durch Dokument
  - Dokument als Datenstrom (push)
- Zugriff auf einzelne Inhalte durch Events
  - Client-Code muss Handler-Interface implementieren, um auf die Events zu reagieren
  - Beispiele (Pseudocode):
    - `startDocument()`
    - `startElement(namespace, name, attributes)`
    - `endDocument()`

XML – Inhalt – Parsen (IV)

- Vorteile:
  - Code kann einige Events ignorieren
  - Nicht das ganze Dokument permanent im Speicher
  - Verarbeitung kann ggf. frühzeitig abgebrochen werden
- Nachteile:
  - Interface muss implementiert werden
  - Keine Kontrolle über die eintreffenden Events

XML – Inhalt – Parsen (V)

- StAX-Parser
  - StAX = Streaming API for XML
  - Cursor in Dokument-Datei
  - Dokumententeile als Datenstrom (pull)
- Kombiniert das Vorgehen von DOM und SAX
  - Client-Code entscheidet, wann er das nächste Event liest und kann es dann verarbeiten
  - Beispiel (Pseudocode):
    - ```
      while(events.hasNext()) {
            Event e = events.next()
            if (e == START_ELEMENT) { … }
            if (e == END_ELEMENT) { … }
            …
      }
      ```
      ```

XML – Inhalt – Parsen (VI)

- Vorteile:
  - Volle Kontrolle
  - Keine Interface-Implementierung nötig
  - StAX unterstützt auch das Schreiben in die XML-Datei
- Nachteil:
  - Wenig intuitiv
  - Mehr Verantwortung bei der Entwicklung

XML – Inhalt – Validierung (I)

- Parser können XML-Dokumente nur verarbeiten, wenn sie syntaktisch korrekt sind.
- Validierende Parser können zudem Konformität zu DTD bzw. XML Schema prüfen.
- Damit drei mögliche Ergebnisse:
  - Syntaktisch inkorrekt (invalid)
  - Wohlgeformt (well-formed)
  - Gültig (valid)

XML – Inhalt – Validierung (II)

- XML-Dokumente sind wohlgeformt, also syntaktisch korrekt, wenn sie folgende Eigenschaften erfüllen:
  - Zu Beginn steht eine XML-Deklaration
  - Es gibt genau ein Wurzelelement
  - Alle öffnenden Tags wurden wieder geschlossen
  - Alle Elemente sind korrekt verschachtelt
  - Alle Elemente beachten Groß- und Kleinschreibung
  - Alle Attributwerte stehen in Anführungszeichen
  - Sonderzeichen (einschließlich Markup-Zeichen) sind durch Entities ausgedrückt
- XML-Dokumente sind gültig, wenn sie außerdem die Strukturvorgaben (DTD oder XSD) beachten.

XML – Inhalt – Arbeiten mit Inhalten

- Transformation (XSL Transformation = XSLT)
  - XSLT übersetzt XML-Quelldateien von einer XML-Strukturdefinition in eine andere
  - Technisch im Wesentlichen Graphtransformation
  - Werkzeug: z. B. Xalan
- Selektoren (XPath)
  - Ermöglichen strukturierte Abfragen auf Datenbeständen, die als XML vorliegen
  - Konzeptuell ähnlich zu SQL für Datenbanken wegen Bezug zu Schema
- Gestaltungsaspekte (Formatting Objects)
  - XML-FO ist eine Anwendung von XML für die Darstellung von XML-Daten in Druckerzeugnissen
  - XHTML ist eine Anwendung von XML für die Darstellung von XML-Daten auf Bildschirmen

Verwendung von XML – Bewertung

- Verwendung von XML kann die Flexibilität erhöhen
- Verschiedene Varianten möglich
  - Kommunikation
  - Verarbeitung (Client- / Server-)
- Vor- und Nachteile
  - Performance
    - Verschiedene Parser sind unterschiedlich schnell
    - Hoher Speicherbedarf bei DOM-Parsern auf großen Dokumenten
  - Skalierbarkeit
  - Flexibilität

Verwendung von XML – Direkte Verwendung zur Kommunikation

- Datenaustausch „von Hand“
  - Strukturierte Daten werden beispielsweise per TCP/IP ausgetauscht
  - Protokollfragen müssen separat behandelt werden (insbesondere Fehlerbehandlung)
  - Vorgehen
    - Client stellt Anfrage zusammen
    - Versenden an den Server
    - Auspacken der Anfrage und Bearbeitung
  - Im Prinzip Standardverfahren des Marshalling/Unmarshalling
    - aber hier auf der Ebene der Applikation!
  - Vorteile
    - Berücksichtigung spezieller Eigenschaften möglich
  - Nachteile
    - Spezielle Lösungen
    - Fehlerbehandlung
    - …

Verwendung von XML – Voraussetzungen

- Definition der gemeinsamen Sprache
  - Definition der Grammatik der Sprache: DTD oder XSD
  - Ggfs. Kodierung der Terminalsymbole beachten
    - Character Encoding
    - Zahlenformate
    - …
- Es ist zu beachten, dass Erweiterungen später eingebaut werden können
- Änderung oder Entfernen von Elementen ist problematisch

Verwendung von XML – Anfragen auf Client-Seite

- Durchlauf der zu verpackenden (internen) Struktur
- Zu beachten ist der Aufbau dieser Struktur
  - Regelmäßigkeit erhöht Strukturiertheit und Systematik der Lösung
  - Wünschenswert:
    - „Schöne" Datenstrukturen wie Baum, Graph, usw.
    - Möglichst keine speziellen Verzweigungen, Verkettungen
- Achtung: alle Gemeinheiten von Strukturdurchläufen sind zu berücksichtigen
  - Baum-Strukturen
  - Z.B. modifizierter Inorder-Durchlauf
    - { Ausgabe der anführenden Tags;
      Verarbeitung der Kind-Knoten;
      Ausgabe der abführenden Tags; }

Verwendung von XML – Anfragen auf Server-Seite

- Zunächst muss entschieden werden, wie die Daten verarbeitet werden
  - Passenden Parser (SAX, StAX, DOM) wählen
- Baumstruktur von XML liefert die Basis für einen systematischen Durchlauf
- Der Durchlauf kann fehlende oder zusätzliche Elemente „entdecken“
  - Insbesondere IDREFS auf Elemente, die noch nicht geparst sind
  - Systematische Behandlung dieser Ausnahmen notwendig
- Standardmäßig werden dann die Elemente des Baums in die Zielstrukturen im Server übertragen

XML – Bewertung

- Motivation für Verwendung von XML
  - Datenaustausch außerhalb von Middleware
    - Interoperation und Interworking
  - Höhere Flexibilität
  - Persistenz
- Ausdrucksfähigkeit hoch durch
  - Strukturbeschreibung und Inhalt + Strukturdefinition
- Verwendung
  - Hohe Flexibilität
  - Weniger Funktionalität, da kein Framework sondern Textauszeichnungssprache
  - Performance problematisch

Teil 1 – Heterogenität

Teil 2 – XML

Teil 3 – JSON

Teil 4 – Google Protocol Buffers

JSON – Motivation

- JSON: „JavaScript Object Notation“
- Kompaktere Darstellung von Daten als XML
- Schlanke Sprachdefinition
  - Einfach zu verstehen
  - Einfach zu verarbeiten
- Standardisiertes (IETF RFC 4627) Datenaustauschformat
  - System- und Hersteller-Unabhängig
  - Darstellung und Verbreitung über verschiedene Medien möglich
  - http://tools.ietf.org/html/rfc4627
- Abgeleitet von den Objekt-Literalen von JavaScript
  - Von jedem JavaScript-Interpreter zu parsen
- JSON ist Untermenge der YAML ain‘t Markup Language (YAML)

JSON – Aufbau

- Strukturierung
  - Inhalt mit Nutzdaten und Strukturbeschreibung
  - Strukturdefinition (= Grammatik von Dokumenten)
    - JSON-Schema
  - Objekte als Key-Value-Paare
  - Verschachtelung über { und } für Objekte und [und] für Arrays
- Darstellung nicht betrachtet
  - Es werden Datenstrukturen modelliert, kein ausgezeichnetes Dokument
  - Mischung von Elementen mit Character Data daher grundsätzlich nicht möglich.

JSON – Inhalt – Beispiel

```
{
        "name":{
                "vorname":"Alfons",
                "nachname":"Viertel vor Zwölfte"
        },
        "straße":"Muster Str. 11",
        "plz":"47111",
        "ort":"Lummerland",
        "telefon":[
                {       "art":"privat",
                        "nummer":"01234/56789"},
                {       "art":"buero",
                        "nummer":"054321/9999999" }
        ]
}
```

JSON – Inhalt – Parsen

- Parser ermöglichen Zugriff auf Inhalte eines JSON-Dokuments
- Zwei eingebaute Möglichkeiten in JavaScript
  - `eval(mein_JSON_String)`
    - zu unsicher
      - könnte auch ausführbarer Code sein!
    - Zuweisungen und Operations-Aufrufe nicht gefiltert
  - `JSON.parse()`
    - Sicherer: evaluiert nur JSON-Objekte, nicht JavaScript
- Es gibt auch Parser-Libraries

JSON – Inhalt – Parsen

- DOM-Parser
  - Ganzes Dokument
  - Transformation in hierarchische Baumstruktur (DOM)
- SAX-Parser
  - Schrittweiser Zugriff durch Dokument
  - Dokument als Datenstrom (push)
  - Zugriff auf einzelne Inhalte durch Events
- StAX-Parser
  - Cursor in Dokument-Datei
  - Dokumententeile als Datenstrom (pull)
- Werkzeuge
  - Große Anzahl für fast alle Programmiersprachen
  - Z. B. org.JSON für Java, jsoncpp für C++, json-framework für Objective-C, … (siehe z.B. http://www.json.org)

JSON – Inhalt – Validierung

- Validierung
  - einfacher Parser
  - validierender Parser (Zugriff auf JSON-Schema)
- Ergebnisse
  - gültig (valid)
  - wohlgeformt (well-formed), grundlegende Anforderungen an JSON-Syntax
  - ungültig

JSON – Struktur – JSON-Schema

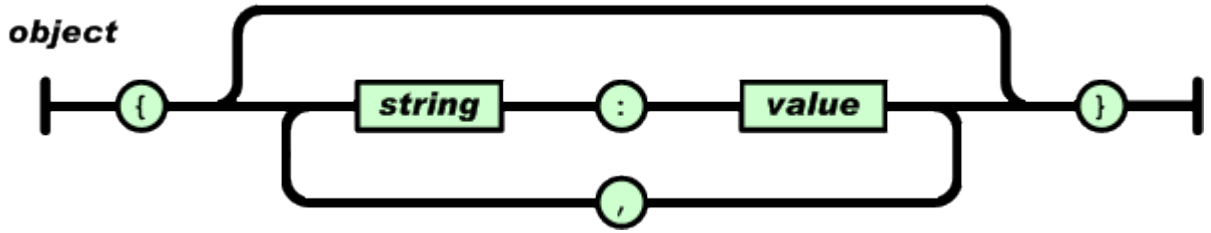
- Weniger komplex als für XML
- Selbst wieder JSON
- Inhalte:
  - Typ eines Elements
    - object, array, number, string
  - Inhalte eines Elements
    - Ggf. manche davon „required“
  - Optionale Beschreibung für Elemente

JSON – Struktur – Beispiel JSON-Schema

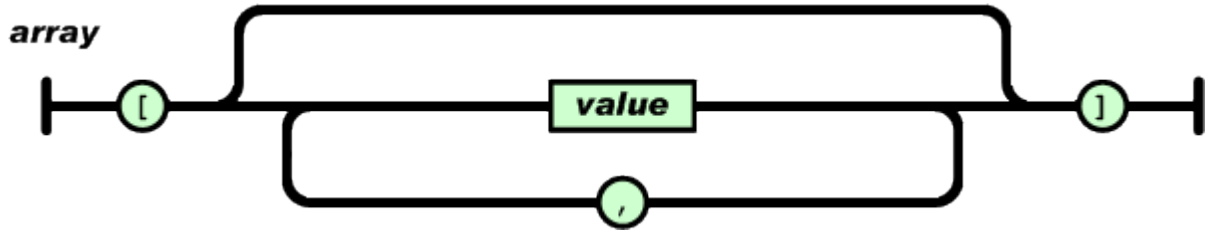
```
{
        "type" : "array"
        "title" : "personalkartei",
        "items" : {
                "type" : "object",
                "properties" : {
                        "name" : {
                                "type" : "object",
                                "properties" : {
                                        "vorname" : {
                                                "type" : "string",
                                                "description" : "first name"},
                                        "nachname" : {
                                                "type" : "string",
                                                "description" : "last name"}
                                }
                        },
                        "straße" : {
                                "type" : "string"},
                        [...]
                },
                "required" : ["name", "straße"]
        }
}
```

JSON – Struktur – Grammatik

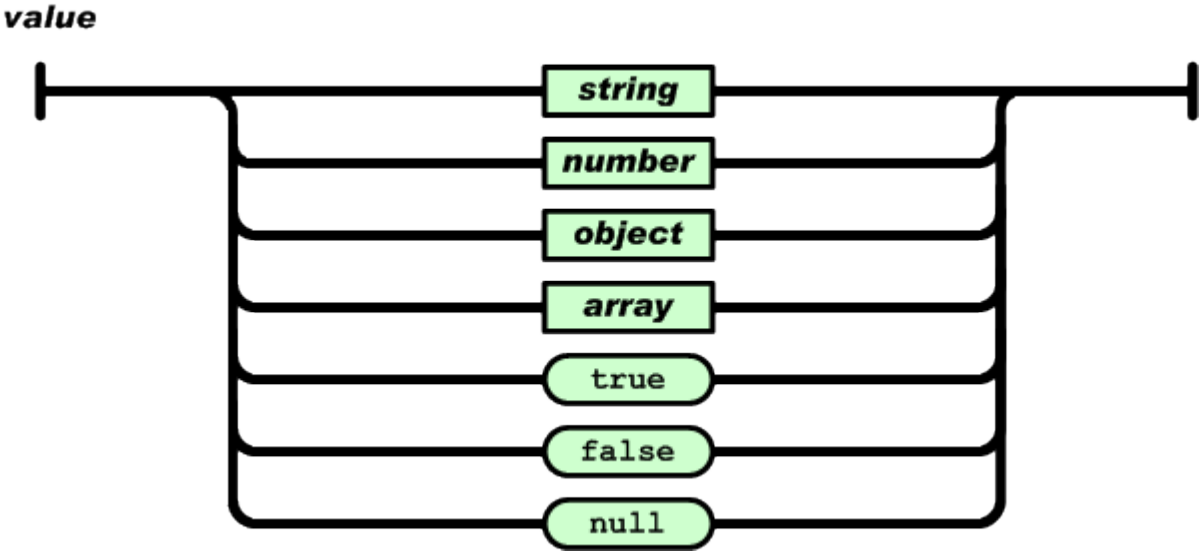
- Object



- Array

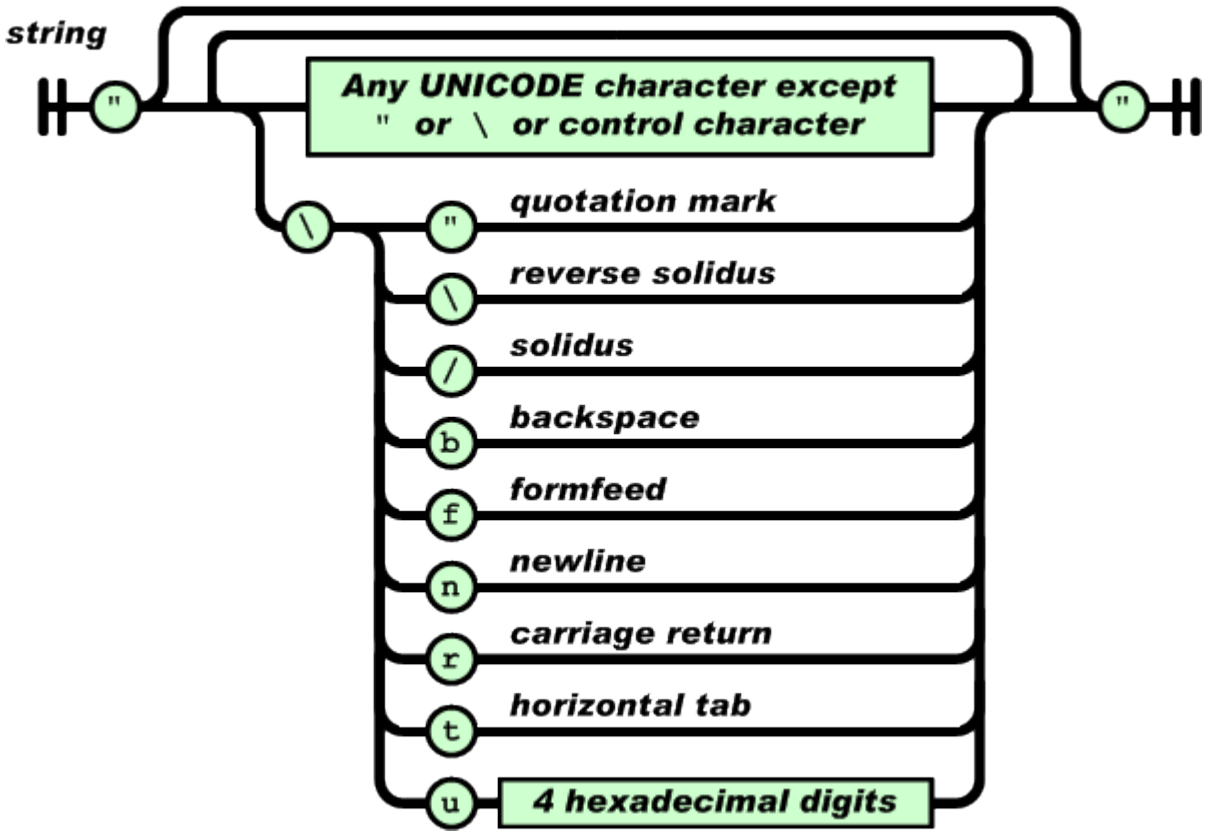


- Value



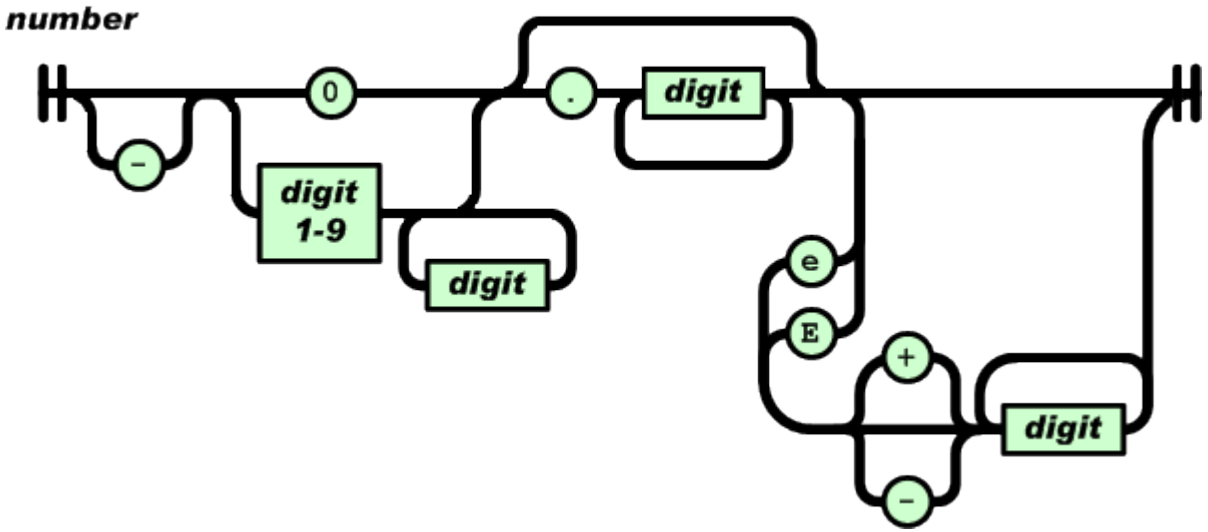
JSON – Struktur – Grammatik

- String



JSON – Struktur – Grammatik

- Number



JSON – Bewertung (I)

- Flexibilität durch Standardisierung und Struktur
- Verwendung für
  - den Austausch von Datenobjekten
  - entfernte Operationsaufrufe (JSON-RPC)
- Vor- und Nachteile
  - Performance
  - Skalierbarkeit
  - Flexibilität

JSON – Bewertung (II)

- Motivation für Verwendung von JSON
  - Datenaustausch außerhalb von Middleware
  - Höhere Flexibilität wegen Plattformunabhängigkeit
  - Persistenz
- Ausdrucksfähigkeit hoch durch
  - Strukturbeschreibung und Inhalt + Strukturdefinition
  - Aber: JSON-Schema kaum genutzt

Teil 1 – Heterogenität

Teil 2 – XML

Teil 3 – JSON

Teil 4 – Google Protocol Buffers

Google Protocol Buffers

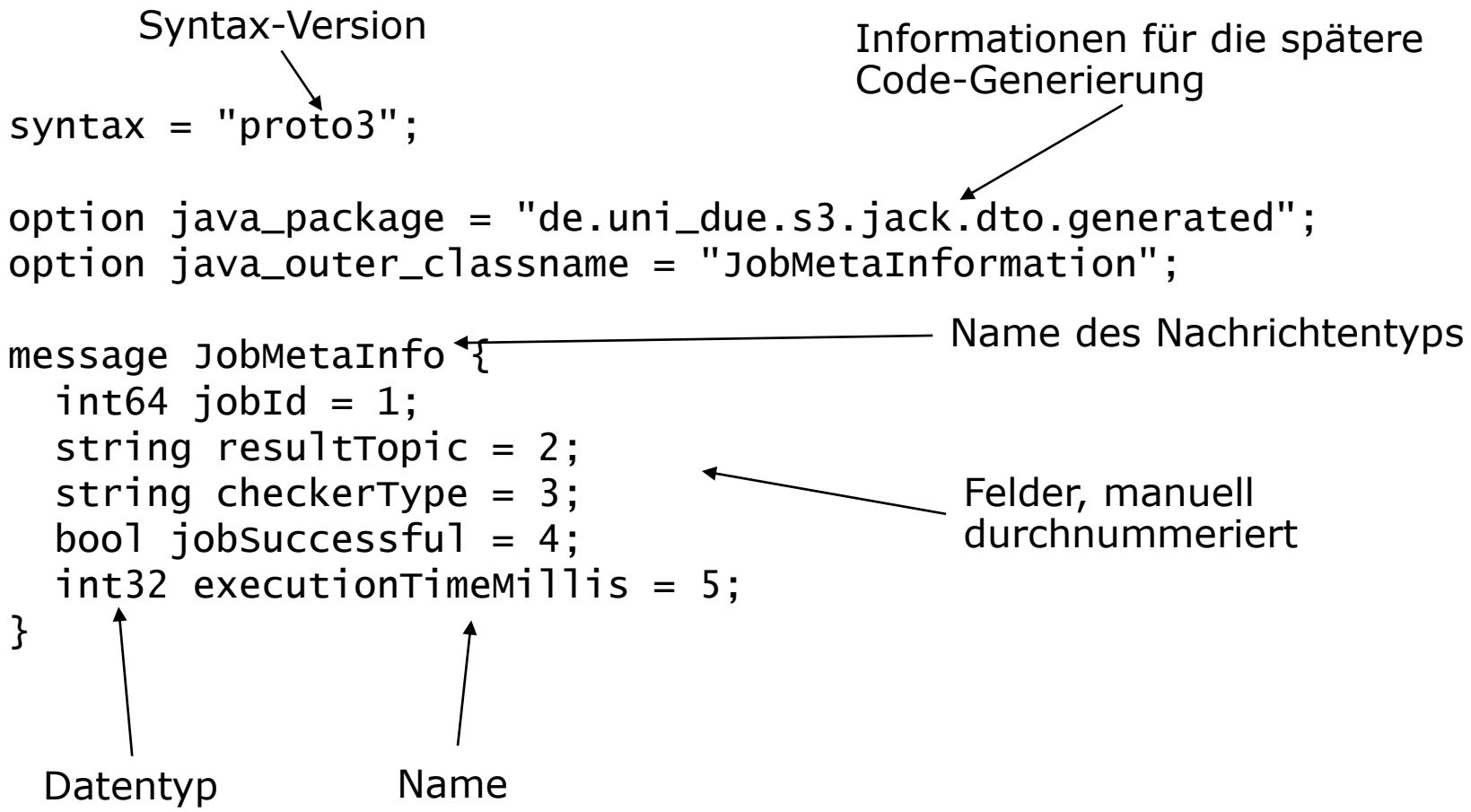
- XML und JSON erzeugen beide „lesbare“ Dokumente.
  - Lesbar für Menschen
  - Von Parsern ohne Zusatzinformationen zu verarbeiten
- Alle nötigen Informationen stecken im Dokument.
  - Viel Overhead in XML
  - Etwas weniger Overhead in JSON
  - Dadurch höhere Netzlast und Verarbeitungsdauer
- Idee von Google Protocol Buffers
  - Schlankere Serialisierung
  - Verarbeitung durch spezifisch generierten Code
  - Nicht mehr universell lesbar, dafür aber kleiner und schneller
- https://developers.google.com/protocol-buffers

Google Protocol Buffers - Idee

- Idee ähnlich zu Middlewares:
  - Dokumentstruktur in unabhängiger Beschreibungssprache definieren
  - Daraus Code analog zu Stubs für verschiedene Programmiersprachen automatisch generieren
  - Restlicher Code interagiert nur mit diesen „Stubs“, eigentliches Dokument bleibt verborgen
- Generierter Code zur Dokumenterzeugung
  - Zum Beispiel über Builder-Pattern
  - Übernimmt das Marshalling
- Generierter Code zur Dokumentverarbeitung
  - Spezifischer, effizienter Parser
  - Übernimmt das Unmarshalling
- Weil Google Protocol Buffers eher für kleine Dokumente gedacht sind, werden diese dort Nachrichten (Messages) genannt.

Google Protocol Buffers – Beispiel (I)

- Beispieldefinition einer Nachricht mit fünf Feldern:



Google Protocol Buffers – Datentypen

- Die Protobuf-Spezifikation kennt eine Menge möglicher (primitiver) Datentypen.
  - Nicht alle sind in jeder Programmiersprache verfügbar
- Beispiele:

Protobuf		**C++**	**Java**	**Python**
double		double	double	float
float		float	float	float
int32	Ineffizient für negative Zahlen, dafür besser sint32 nehmen.	int32	int	int
int64	Ineffizient für negative Zahlen, dafür besser sint64 nehmen.	int64	long	int/long
sint32		int32	int	int
sint64		int64	long	int/long
bool		bool	boolean	bool
string	Muss in UTF-8 oder 7-bit ASCII codiert sein.	string	String	unicode (Python 2) oder str (Python 3)
bytes		string	ByteString	bytes

https://developers.google.com/protocol-buffers/docs/overview#scalar

Google Protocol Buffers – Multiplizitäten

- Im Standardfall sind Felder optional.
  - Wert ist also null- oder genau einmal in der Nachricht enthalten
  - Parser füllt fehlende Felder mit Defaults
    - Leerer String, False, die Zahl 0, …
    - In Java niemals `null`
- Felder können als `repeated` markiert werden.
  - Dann können Werte beliebig oft (auch gar nicht) enthalten sein.
  - Einfügereihenfolge der Werte bleibt erhalten
- Weiteres Feature zur Verbesserung der Kompression:
  - Nur genau eines von vielen, optionalen Feldern setzen lassen:

```
message SampleMessage {
  oneof adressierung {
     string strasseUndHausnummer = 1;
     string postfach = 2;
  }
}
```

Google Protocol Buffers – Beispiel (II)

- Definitionen können einander referenzieren:

```
syntax = "proto3";

option java_package = "de.uni_due.s3.jack.dto.generated";
option java_outer_classname = "BackendResultData";

import "JobMetaInformation.proto";

message BackendResult {
  int32 result = 1;
  JobMetaInfo jobMetaInfo = 2;
  string backendLog = 3;
  repeated Feedback feedback = 4;

  message Feedback {
    string category = 1;
    string title = 2;
    string content = 3;
  }
}
```

Import der Definition von Folie 72

Referenz auf das importierte Objekt

Referenz auf einen weiteren, lokal definieren Message-Typen

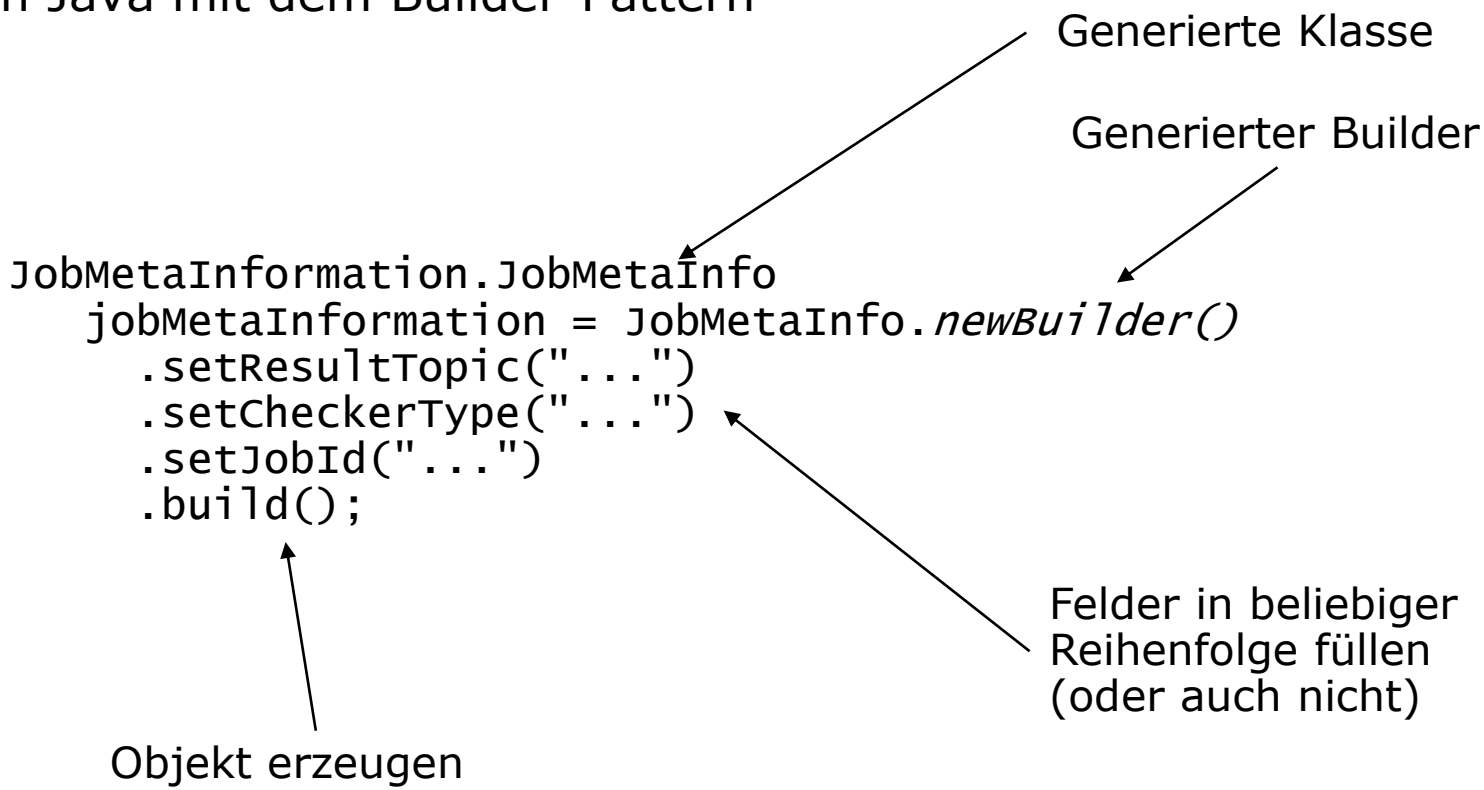
Lokale Definition eines Message-Typen

Google Protocol Buffers – Weitere Features

- Weitere Features (hier nicht im Detail betrachtet):
  - Enumerations
    - Funktionieren wie lokale Message-Definitionen
    - Default-Wert ist das erste Element der Aufzählung
  - Extensions
    - Messages definieren Nummern explizit zur Verwendung in Erweiterungen
    - Erweiterungen verwenden diese Nummern in ihren Felddefinitionen
    - Folge: Dieselbe Feld-Nummer hat je nach Erweiterung eine andere Bedeutung

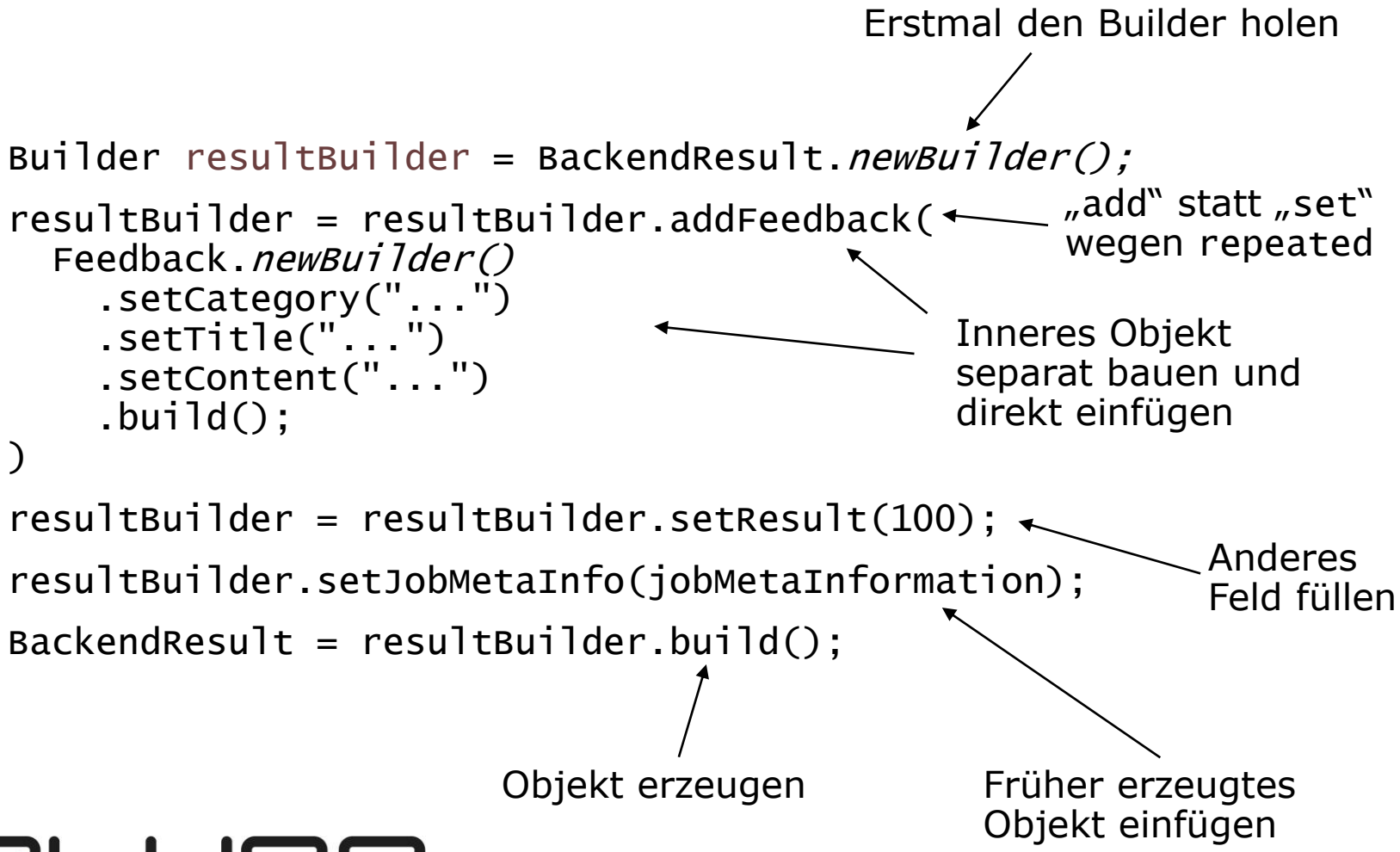
Google Protocol Buffers – Beispiel (III)

- Nachrichten können über generierten Code als Objekt erzeugt und gefüllt werden.
  - In Java mit dem Builder-Pattern



Google Protocol Buffers – Beispiel (IV)

- Befüllen auch über mehrere Schritte verteilt möglich:



Google Protocol Buffers – (De)Serialisierung

- Objekte/Klassen enthalten automatisch Methoden zur Serialisierung und Deserialisierung
  - Serialisierung per Objekt-Methode ergibt ein Byte-Array:
    ```
    byte[] byteString = backendResult.toByteArray();
    ```
  - Deserialisierung per Klassen-Methode macht aus einem Byte-Array wieder ein Objekt:
    ```
    BackendResult backendResult =
    BackendResult.parseFrom(byteString);
    ```
- Ergebnis ist kompakte Darstellung des Nachrichten-Objekts
  - Ohne den jeweiligen Parser nicht lesbar

Google Protocol Buffers – Versionierung

- Protobuf-Spezifikationen können sich weiterentwickeln.
  - Neue Felder einfügen
  - Vorhandene Felder entfernen
    - Nummern dürfen nicht neu vergeben werden
    - Nummern und Namen ehemals vorhandene Felder können über Eintrag mit Schlüsselwort `reserved` geschützt werden:
      `reserved 2, 15, 9 to 11;`
- Kompatibilität durch Nummerierung der Felder
  - Alte Parser ignorieren neue Felder
  - Neue Parser füllen alte Felder mit Defaults
- Weitere Regeln sollten beachtet werden:
  - Nicht alle Typen sind untereinander kompatibel.
  - Nicht alle Typen können einfach auf `repeated` umgestellt werden.
  - Siehe https://developers.google.com/protocol-buffers/docs/overview#updating

Google Protocol Buffers – Unterschiede Syntax Version 2 und 3

- Version 2 hatte Features, die in Version 3 wieder entfernt wurden
- Felder konnten als `required` markiert werden.
  - Mussten dann in einem Objekt genau einmal enthalten sein
  - Konnte man zwar auch wieder entfernen, aber alte Parser behielten die Information.
    - Alte Parser werfen dann bei neuen Nachrichten Fehler.
  - „Some engineers at Google have come to the conclusion that using `required` does more harm than good; they prefer to use only `optional` and `repeated`. However, this view is not universal.“ (https://developers.google.com/protocol-buffers/docs/overview#specifying_field_rules)
- Default-Werte konnten pro Feld gesetzt werden:
  `optional int32 result = 1 [default = 100];`
  - Passende Konstruktoren sind aber nicht in jeder Programmiersprache generierbar
  - Muss in Version 3 manuell durch Wrapper sichergestellt werden

Google Protocol Buffers – Bewertung

- Motivation für die Verwendung von Google Protobuf
  - Effizienteres Datenformat als XML und JSON
  - Nutzung für die reine Maschine-zu-Maschine-Kommunikation
- Vorteile
  - Geringer Größe serialisierter Objekte
  - Effiziente Verarbeitung
  - Saubere, erweiterbare Objektdefinitionen
- Nachteile
  - Serialisierte Objekte nicht menschenlesbar
  - Informationen im generierten Code statt im serialisierten Objekt
    - Repräsentation nicht vollständig/unabhängig
    - Unerwartete Effekte bei inkompatiblen Versionen möglich
  - Compiler (noch?) nicht für alle Programmiersprachen verfügbar

Fazit

- Heterogenität
  - Verteilte (OO-)Anwendungen sind ganz natürlich durch Heterogenität und Flexibilität gekennzeichnet wegen unterschiedlicher Programmiersprachen und heterogenen Middlewares und Datenrepräsentationen
- XML
  - Standardisierte Markup-Sprache für Dokumente
  - Geeignet zum strukturierten Datenaustausch
  - Umfassende, detaillierte Strukturdefinition und –validierung möglich
- JSON
  - Einfache Beschreibung von Datenstrukturen
  - Leichtgewichtiger als XML
- Google Protobuf
  - Effiziente Serialisierung
  - Enge Integration mit spezifischer Codegenerierung