Michael Inden

Der Java-Profi: Persistenzlösungen und REST-Services

Datenaustauschformate, Datenbankentwicklung und verteilte Anwendungen

dpunkt.verlag

Dipl.-Inform. Michael Inden ist Oracle-zertifizierter Java-Entwickler für JDK 6. Nach seinem Studium in Oldenburg war er lange Zeit als Softwareentwickler und -architekt bei verschiedenen internationalen Firmen tätig.

Dabei hat er über 15 Jahre Erfahrung beim Entwurf objektorientierter Softwaresysteme gesammelt, an diversen Fortbildungen und an mehreren Java-One-Konferenzen in San Francisco teilgenommen. Sein besonderes Interesse gilt dem Design qualitativ hochwertiger Applikationen mit ergonomischen, grafischen Oberflächen sowie dem Coaching von Kollegen.

Michael Inden

Der Java-Profi: Persistenzlösungen und REST-Services

Datenaustauschformate, Datenbankentwicklung und verteilte Anwendungen

dpunkt.verlag

Michael Inden
michael_inden@hotmail.com

Lektorat: Dr. Michael Barabas
Fachgutachter: Torsten Horn
Copy-Editing: Ursula Zimpfer, Herrenberg
Satz: Michael Inden
Herstellung: Susanne Bröckelmann
Umschlaggestaltung: Helmut Kraus, www.exclam.de
Druck und Bindung: M.P. Media-Print Informationstechnologie GmbH, 33100 Paderborn

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.d-nb.de abrufbar.

ISBN:
Print 978-3-86490-374-8
PDF 978-3-86491-960-2
ePub 978-3-86491-961-9
mobi 978-3-86491-962-6

1. Auflage 2016

Wieblinger Weg 17
69123 Heidelberg

5 4 3 2 1 0

Inhaltsverzeichnis

Vorwort

Zunächst einmal bedanke ich mich bei Ihnen, dass Sie sich für dieses Buch entschieden haben. Hierin finden Sie Informationen zu den Datenaustauschformaten XML und JSON sowie zum Zugriff auf Datenbanken mit JDBC und JPA als auch auf MongoDB. Darüber hinaus werden RESTful Webservices mit JAX-RS und Jersey behandelt. Diese für Unternehmensanwendungen wichtigen Themen möchte ich Ihnen anhand von praxisnahen Beispielen näherbringen. Dabei kommen die vielfältigen Neuerungen aus JDK 8 zum Einsatz, um die Beispiele prägnanter zu machen. Für einen fundierten Einstieg in Java 8 möchte ich Sie auf meine Bücher »Java 8 – Die Neuerungen« [9] oder alternativ »Der Weg zum Java-Profi« [8] verweisen. Beide können ergänzend, aber auch unabhängig von diesem Buch gelesen werden.

Motivation

Wenn Sie bereits komplexe Java-Applikationen für den Desktop-Bereich schreiben und sich vertraut mit der Sprache Java fühlen, dann sind Sie schon recht gut für das Berufsleben gerüstet. Allerdings kommen Sie dort früher oder später mit Datenbanken, dem Informationsaustausch basierend auf XML oder JSON und vermutlich auch verteilten Applikationen in Berührung. Darunter versteht man Programme, die auf mehreren JVMs (und gewöhnlich somit auf mehreren Rechnern) ausgeführt werden. Um zusammenzuarbeiten, müssen diese miteinander kommunizieren, wodurch ganz neue Herausforderungen, aber auch Möglichkeiten entstehen.

Vielleicht haben Sie sich bisher auf den Desktop-Bereich konzentriert und wollen nun per JDBC oder JPA mit einer Datenbank kommunizieren. Dann erhalten Sie in diesem Buch eine fundierte Einführung in die Persistenz mit Java, SQL, JDBC und JPA. Oftmals benötigen Sie aber weiteres Know-how, da die Programmanwender zunehmend anspruchsvoller werden: Neben einer gut bedienbaren Benutzeroberfläche kommt für viele Applikationen der Wunsch auf, deren Funktionalität – zumindest teilweise – auch im Netzwerk bereitzustellen. Dazu existieren vielfältige Technologien. In diesem Buch wollen wir uns auf die populären RESTful Webservices konzentrieren und mit der Programmierung einer sogenannten Client-Server-Applikation beschäftigen.

Wie Sie sehen, sind Unternehmensanwendungen ein spannendes, aber auch weitreichendes Feld, was deutlich mehr Anforderungen als reine Java-SE-Anwendungen an den Entwickler stellt. Dieses Buch gibt Ihnen einen fundierten Einstieg. Wie schon

bei meinem Buch »Der Weg zum Java-Profi« war es auch diesmal mein Ziel, ein Buch zu schreiben, wie ich es mir selbst immer als Hilfe gewünscht habe, um mich auf die Herausforderungen und Aufgaben im Berufsleben vorzubereiten.

Wer sollte dieses Buch lesen?

Dieses Buch ist kein Buch für Programmierneulinge, sondern richtet sich an all diejenigen Leser, die solides Java-Know-how besitzen und ihren Horizont auf die interessante Welt der Unternehmensanwendungen erweitern wollen. Dazu werden die dafür benötigten Themen Datenaustauschformate (XML, JSON) sowie Datenbankentwicklung (RDBMS, SQL, JDBC, JPA und auch NoSQL-DBs mit MongoDB) sowie die Kommunikation in verteilten Applikationen mit REST-Webservices (JAX-RS) vorgestellt.

Dieses Buch richtet sich im Speziellen an zwei Zielgruppen:

1. Zum einen sind dies engagierte Hobbyprogrammierer, Informatikstudenten und Berufseinsteiger, die Java als Sprache beherrschen und nun neugierig auf die zuvor genannten Themen sind.
2. Zum anderen ist das Buch für erfahrene Softwareentwickler und -architekten gedacht, die ihr Wissen ergänzen oder auffrischen wollen.

Was soll mithilfe dieses Buchs gelernt werden?

Dieses Buch zeigt und erklärt einige wesentliche Themen, die bei der Realisierung von Unternehmensapplikationen von Bedeutung sind. Sollte ein Thema bei Ihnen besonderes Interesse wecken und Sie weitere Informationen wünschen, so finden sich in den meisten Kapiteln Hinweise auf weiterführende Literatur.

Zwar ist Literaturstudium hilfreich, aber nur durch Übung und Einsatz in der Praxis können wir unsere Fähigkeiten signifikant verbessern. Deshalb ermuntere ich Sie, die gezeigten Beispiele (zumindest teilweise) durchzuarbeiten. Manchmal werde ich bei der Lösung eines Problems bewusst zunächst einen Irrweg einschlagen, um anhand der anschließend vorgestellten Korrektur die Vorteile deutlicher herauszustellen. Mit dieser Darstellungsweise hoffe ich, Ihnen mögliche Fallstricke und Lösungen aufzeigen zu können.

Des Weiteren lege ich Wert darauf, auch den kleinen, scheinbar nicht ganz so wichtigen Dingen ausreichend Beachtung zu schenken. Zum Beispiel ist es von großem Nutzen, wenn Klassen, Methoden, Attribute usw. einen sinnvollen Namen tragen.

Auch auf der Ebene des Designs lässt sich einiges falsch machen. Die Komplexität in der zu modellierenden Fachlichkeit dient häufig als Ausrede für konfuse und verwirrende Lösungen. Beim professionellen Entwickeln sollte man aber viel Wert auf klares Design legen. Grundsätzlich sollte alles möglichst einfach und vor allem gut verständlich gehalten werden, sodass eine ausgereifte und wartbare Lösung entsteht.

Sourcecode und ausführbare Programme

Da der Fokus des Buchs auf dem praktischen Nutzen und der Vorbereitung auf das Berufsleben bzw. dessen besserer Meisterung liegt, werden praxisnahe Beispiele vorgestellt. Um den Rahmen des Buchs nicht zu sprengen, stellen die Listings häufig nur Ausschnitte aus lauffähigen Programmen dar – zum besseren Verständnis sind wichtige Passagen dort mitunter fett hervorgehoben. Die in den Listings abgebildeten Sourcecode-Fragmente stehen als kompilierbare und lauffähige Programme (Gradle-Tasks) auf der Webseite zu diesem Buch `www.dpunkt.de/java-persistenz` zum Download bereit. Der Programmname bzw. der Name des ausführbaren Gradle-Tasks wird in Kapitälchenschrift, etwa FIRSTSAXEXAMPLE, angegeben.

Neben dem Sourcecode befindet sich auf der Webseite ein Eclipse-Projekt, über das sich alle Programme ausführen lassen. Idealerweise nutzen Sie dazu Eclipse 4.5 oder neuer, weil diese Version der IDE bereits Java 8 unterstützt und die Beispiele dieses Buchs immer wieder auch Funktionalitäten aus JDK 8 nutzen.

Neben dem Eclipse-Projekt wird eine Datei `build.gradle` mitgeliefert, die den Ablauf des Builds für Gradle beschreibt. Dieses Build-Tool besitzt viele Vorzüge wie die kompakte und gut lesbare Notation und vereinfacht die Verwaltung von Abhängigkeiten enorm. Gradle wird im Anhang A einführend beschrieben. Als Grundlage für spätere Ergänzungen dient folgende Datei `build.gradle`, die JUnit als Abhängigkeit definiert und trotz der Kürze schon ein vollständiges Release als `jar`-Datei namens `java-profi-db-rest.jar` erzeugt:

```
apply plugin: 'java'
apply plugin: 'eclipse'

sourceCompatibility=1.8

// create special jar containing the starter app
jar
{
    baseName = "java-profi-db-rest"

    manifest
    {
        attributes ( "Main-Class" : "de.inden.starter.ApplicationStarter" )
    }
}

repositories
{
    mavenCentral()
}

dependencies
{
    testCompile 'junit:junit:4.11'

    // Weitere Abhängigkeiten hier eintragen
}
```

Aufbau dieses Buchs

Nachdem Sie nun einen groben Überblick über den Inhalt dieses Buchs haben, möchte ich die Themen der einzelnen Kapitel kurz vorstellen.

Kapitel 1 – Einstieg in XML und JSON Weil proprietäre Formate beim Datenaustausch oftmals Probleme bereiten, spielt in heutigen Applikationen die standardisierte Darstellung von Daten eine immer größere Rolle. Kapitel 1 stellt die Datenaustauschformate XML und JSON vor, die die Interoperabilität zwischen verschiedenen Programmen erleichtern und sogar einen Austausch erlauben, wenn diese Programme in unterschiedlichen Programmiersprachen erstellt wurden.

Kapitel 2 – Einführung in Persistenz und relationale Datenbanken Dieses Kapitel stellt wichtige Grundlagen zu Datenbanken und zu SQL vor. Insbesondere wird auch auf Möglichkeiten der Transformation von Objekten in entsprechende Repräsentationen in Datenbanken eingegangen. Dabei wird vor allem auch der sogenannte Impedance Mismatch, die Schwierigkeiten bei der Abbildung von Objekten auf Tabellen einer Datenbank, thematisiert.

Kapitel 3 – Persistenz mit JDBC Wie man mit Java-Bordmitteln auf Datenbanken zugreifen kann, ist Thema von Kapitel 3. Zunächst betrachten wir JDBC als Basistechnologie und erstellen verschiedene Beispielapplikationen bis hin zum Mapping von Objekten in die Datenbank, dem sogenannten ORM (Object-Relational Mapping).

Kapitel 4 – Persistenz mit JPA Das JPA (Java Persistence API) stellt eine Alternative zu JDBC dar und erleichtert die Realisierung von Persistenzlösungen mit Java, insbesondere das ORM. In Kapitel 4 werden zunächst Grundlagen besprochen und dann gezeigt, wie sich selbst komplexere Objektgraphen mithilfe von JPA persistieren lassen.

Kapitel 5 – NoSQL-Datenbanken am Beispiel von MongoDB Neben relationalen Datenbanken gewinnen NoSQL-Datenbanken immer mehr an Bedeutung. Ein Vertreter ist MongoDB, das in Kapitel 5 behandelt wird. Neben einer Einführung in die Theorie und ersten Experimenten mit der Mongo Console schauen wir uns die Verarbeitung mit Java, im Speziellen unter Zuhilfenahme von Spring Data MongoDB, an.

Kapitel 6 – REST-Services mit JAX-RS und Jersey Kapitel 6 stellt RESTful Webservices vor, die seit geraumer Zeit im praktischen Alltag immer wichtiger werden: Viele Systeme binden die Funktionalität anderer Systeme basierend auf REST ein. Nach einer Einführung in die Thematik zeige ich einige Varianten, wie sich die Funktionalität eigener Applikationen als RESTful Webservice bereitstellen lassen.

Kapitel 7 – Entwurf einer Beispielapplikation Den Abschluss des Hauptteils dieses Buchs bildet ein Kapitel, in dem eine Beispielapplikation entwickelt wird, die eine Vielzahl der zuvor im Buch vorgestellten Technologien einsetzt. Wir folgen einer iterativ inkrementellen Vorgehensweise und sehen dabei, wie man Erweiterungen schrittweise geschickt in bestehende Applikationen integrieren kann.

Anhang A – Einführung Gradle Anhang A liefert eine Einführung in das Build-Tool Gradle, mit dem die Beispiele dieses Buchs übersetzt wurden. Mit dem vermittelten Wissen können Sie dann auch kleinere eigene Projekte mit einem Build-System ausstatten.

Anhang B – Einführung Client-Server und HTTP Im Anhang B erhalten Sie einen Einstieg in die Client-Server-Kommunikation und HTTP, weil beides für verteilte Applikationen und REST-Services von zentraler Bedeutung ist.

Anhang C – Grundlagenwissen HTML HTML und XML sind wichtige Standards, deren Kenntnis einem Java-Entwickler immer mal wieder nützlich sein kann. In diesem Anhang werden verschiedene Grundlagen zu HTML so weit vorgestellt, wie diese für das Verständnis einiger Beispiele aus diesem Buch notwendig sind.

Anhang D – Grundlagenwissen JavaScript Das in Kapitel 7 entwickelte Abschlussbeispiel verwendet mitunter etwas JavaScript, um ein interaktives Web-GUI mit Zugriffen auf REST-Services zu erstellen. Die dazu benötigten Grundlagen zur Sprache JavaScript werden in diesem Anhang behandelt.

Konventionen

Verwendete Zeichensätze

In diesem Buch gelten folgende Konventionen bezüglich der Schriftart: Neben der vorliegenden Schriftart werden wichtige Textpassagen *kursiv* oder ***kursiv und fett*** markiert. Englische Fachbegriffe werden eingedeutscht großgeschrieben, etwa Event Handling. Zusammensetzungen aus englischen und deutschen (oder eingedeutschten) Begriffen werden mit Bindestrich verbunden, z. B. Plugin-Manager. Namen von Programmen und Entwurfsmustern werden in KAPITÄLCHEN geschrieben. Listings mit Sourcecode sind in der Schrift `courier` gesetzt, um zu verdeutlichen, dass dies einen Ausschnitt aus einem Java-Programm darstellt. Auch im normalen Text wird für Klassen, Methoden, Konstanten und Parameter diese Schriftart genutzt.

Verwendete Klassen aus dem JDK

Werden Klassen des JDKs erstmalig im Text erwähnt, so wird deren voll qualifizierter Name, d. h. inklusive der Package-Struktur, angegeben: Die Klasse `String` würde dann einmal als `java.lang.String` notiert – alle weiteren Nennungen erfolgen dann ohne Angabe des Package-Namens. Diese Regelung erleichtert initial die Orientierung und ein Auffinden im JDK und zudem wird der nachfolgende Text nicht zu sehr aufgebläht. Die voll qualifizierte Angabe hilft insbesondere, da in den Listings eher selten `import`-Anweisungen abgebildet werden.

Im Text beschriebene Methodenaufrufe enthalten in der Regel die Typen der Übergabeparameter, etwa `substring(int, int)`. Sind die Parameter in einem Kontext nicht entscheidend, wird mitunter auf deren Angabe aus Gründen der besseren Lesbarkeit verzichtet – das gilt ganz besonders für Methoden mit generischen Parametern.

Klassen- und Tabellennamen

Zur besseren Unterscheidbarkeit von Objekt- und Datenbankwelt werde ich für dieses Buch als Konvention deutsche Tabellen- und Spaltennamen verwenden. Zudem werden Tabellen im Plural benannt, z. B. `Personen`. Beides hilft, die in Englisch gehaltene Objektwelt leichter von der in Deutsch repräsentierten Datenbankwelt abzugrenzen.

Verwendete Abkürzungen

Im Buch verwende ich die in der nachfolgenden Tabelle aufgelisteten Abkürzungen. Weitere Abkürzungen werden im laufenden Text in Klammern nach ihrer ersten Definition aufgeführt und anschließend bei Bedarf genutzt.

Abkürzung	Bedeutung
API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
DB	Datenbank
(G)UI	(Graphical) User Interface
IDE	Integrated Development Environment
JDBC	Java Database Connectivity
JDK	Java Development Kit
JLS	Java Language Specification
JPA	Java Persistence API
JRE	Java Runtime Environment
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
XML	eXtensible Markup Language

Tipps und Hinweise aus der Praxis

Dieses Buch ist mit diversen Praxistipps gespickt. In diesen werden interessante Hintergrundinformationen präsentiert oder es wird auf Fallstricke hingewiesen.

Tipp: Praxistipp

In derart formatierten Kästen finden sich im späteren Verlauf des Buchs immer wieder einige wissenswerte Tipps und ergänzende Hinweise zum eigentlichen Text.

Danksagung

Ein Fachbuch zu schreiben ist eine schöne, aber arbeitsreiche und langwierige Aufgabe. Alleine kann man eine solche Aufgabe kaum bewältigen. Daher möchte ich mich an dieser Stelle bei allen bedanken, die direkt oder indirekt zum Gelingen des Buchs beigetragen haben. Insbesondere konnte ich bei der Erstellung des Manuskripts auf ein starkes Team an Korrekturlesern zurückgreifen. Es ist hilfreich, von den unterschiedlichen Sichtweisen und Erfahrungen profitieren zu dürfen.

Zunächst einmal möchte ich mich bei Michael Kulla, der als Trainer für Java SE und Java EE bekannt ist, für sein Review vieler Kapitel und die fundierten Anmerkungen bedanken. Auch Tobias Trelle als MongoDB-Experte hat sein Know-how in das Kapitel zu NoSQL-Datenbanken eingebracht. Vielen Dank!

Merten Driemeyer, Dr. Clemens Gugenberger und Prof. Dr. Carsten Kern haben mit verschiedenen hilfreichen Anmerkungen zu einer Verbesserung beigetragen. Zudem hat Ralph Willenborg mal wieder ganz genau gelesen und so diverse Tippfehler gefunden. Vielen Dank dafür! Ein ganz besonderer Dank geht an Andreas Schöneck für die schnellen Rückmeldungen auch zu später Stunde mit wertvollen Hinweisen und Anregungen.

Schließlich möchte ich verschiedenen Kollegen meines Arbeitgebers Zühlke Engineering AG danken: Jeton Memeti, Joachim Prinzbach, Marius Reusch, Dr. Christoph Schmitz, Dr. Hendrik Schöneberg und Dr. Michael Springmann. Sie trugen durch ihre Kommentare zur Klarheit und Präzisierung bei.

Ebenso geht ein Dankeschön an das Team des dpunkt.verlags (Dr. Michael Barabas, Martin Wohlrab, Miriam Metsch und Birgit Bäuerlein) für die tolle Zusammenarbeit. Außerdem möchte ich mich bei Torsten Horn für die fundierte fachliche Durchsicht sowie bei Ursula Zimpfer für ihre Adleraugen beim Copy-Editing bedanken.

Abschließend geht ein lieber Dank an meine Frau Lilija für ihr Verständnis und die Unterstützung. Glücklicherweise musste sie beim Entstehen dieses Erweiterungsbandes einen weit weniger gestressten Autor ertragen, als dies früher bei der Erstellung meines Buchs »Der Weg zum Java-Profi« der Fall war.

Anregungen und Kritik

Trotz großer Sorgfalt und mehrfachen Korrekturlesens lassen sich missverständliche Formulierungen oder sogar Fehler leider nicht vollständig ausschließen. Falls Ihnen etwas Derartiges auffällt, so zögern Sie bitte nicht, mir dies mitzuteilen. Gerne nehme ich auch sonstige Anregungen oder Verbesserungsvorschläge entgegen. Kontaktieren Sie mich bitte per Mail unter:

michael_inden@hotmail.com

Zürich und Aachen, im April 2016
Michael Inden

1 Einstieg in XML und JSON

Oftmals müssen Programme gewisse Daten speichern oder untereinander austauschen. Dazu gibt es verschiedene Möglichkeiten. Während früher oft proprietäre Formate verwendet wurden, hat sich dies mittlerweile geändert. Vielfach setzt man nun auf Standards wie ***XML*** (***eXtensible Markup Language***) und neuerdings auch ***JSON*** (***JavaScript Object Notation***). Um dem Trend nach Standardisierung Rechnung zu tragen, beschäftigt sich dieses Kapitel mit der Verarbeitung von Dokumenten. Wir schauen auf XML und JSON und welche Vorzüge man durch deren Einsatz erzielt.

Zunächst einmal ist erwähnenswert, dass XML eine strukturierte, standardisierte und doch flexible Art der Darstellung und Verarbeitung von Informationen erlaubt. Hilfreich ist dabei vor allem, dass die Daten und die strukturierenden Elemente (nahezu) frei wählbar sind. Dabei werden spezielle Zeichenfolgen, sogenanntes Markup, verwendet. Man spricht bei XML deshalb auch von einer sogenannten ***Auszeichnungssprache*** (Markup Language). Ähnlich wie Sie es vielleicht von HTML kennen, existieren auch in XML spezielle Zeichenfolgen, die eine steuernde oder besondere Bedeutung tragen. Das erlaubt es, Daten nahezu selbstbeschreibend darzustellen.

XML spielt auch beim Datenaustausch zwischen Programmkomponenten oder Systemen eine wichtige Rolle. Das liegt insbesondere an der einfach zu verarbeitenden Darstellung als textbasiertes Format, das sich zudem problemlos über das Netzwerk übertragen lässt. Für die Auswertung gibt es vom W3C (World Wide Web Consortium) verschiedene Varianten, die eine Vielzahl an Frameworks hervorgebracht haben, sodass eine Verarbeitung von XML in gebräuchlichen Programmiersprachen sowie der Austausch zwischen unterschiedlichen Systemen und Programmen einfach möglich wird.

Nach dieser Motivation möchte ich Ihnen kurz darlegen, was dieses Kapitel beinhaltet. Zu Beginn gebe ich eine kurze Einführung in XML, um für diejenigen Leser, für die dieses Thema neu ist, eine gemeinsame Basis mit bereits erfahreneren Kollegen zu schaffen. Dabei lernen wir einige Grundbausteine eines XML-Dokuments kennen. Anschließend gehe ich kurz auf Möglichkeiten zur semantischen Beschreibung mithilfe von DTDs (Document Type Definition) sowie XSDs (XML Schema Definition) ein. Mit diesem Grundwissen wenden wir uns der Verarbeitung von XML mit verschiedenen Java-APIs zu. Wir betrachten SAX (Simple API for XML) und DOM (Document Object Model) sowie StAX (Streaming API for XML). Zum Auswerten von XML-basierten Daten werfen wir abschließend einen Blick auf XPath und XSLT (eXtensible Stylesheet Language Transformations). Außerdem betrachten wir noch JAXB (Java Architecture for XML Binding) und abschließend JSON.

1.1 Basiswissen XML

Sowohl HTML als auch XML sind textuelle Repräsentationen von Daten. Während in HTML eine vordefinierte Menge an Auszeichnungselementen, sogenannten ***Tags***, etwa `body`, `head`, `ol` und `table` existiert, sind im Gegensatz dazu die Tags in XML nicht fest vorgegeben, sondern frei wählbar. Dies ermöglicht eine flexible Beschreibung und Strukturierung der Daten.

Weil sich Erklärungen anhand eines konkreten Beispiels in der Regel leichter nachvollziehen lassen, werfen wir einen Blick auf ein einfaches XML-Dokument, das eine Menge von Personen wie folgt modelliert:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!-- Dies ist ein Kommentar: Die erste Zeile ist der Prolog -->

<!-- Hier folgen nun die eigentlichen Daten -->
<Personen>
    <Person vorname="Michael" name="Inden" alter="44"></Person>
    <Person vorname="Hans" name="Meyer" alter="32"></Person>

    <!-- Kurzschreibweise für leere Elemente / leer. d. h. keine Subelemente -->
    <Person vorname="Peter" name="Muster" alter="55" />
</Personen>
```

An diesem Beispiel erkennt man verschiedene Dinge: XML-Dokumente speichern ihre Daten in sogenannten ***Elementen***, die ineinander geschachtelt sein können. Das alle anderen umschließende Element nennt sich ***Wurzelelement***. Im Beispiel gibt es drei Subelemente `Person` unter dem Wurzelelement `Personen`. Es wäre aber auch problemlos möglich, weitere Hierarchieebenen – etwa eine Adresse unterhalb eines `Person`-Elements – wie folgt einzufügen:

```
<Person vorname="Hans" name="Meyer" alter="32">
    <!-- Verschachtelte Subelemente -->
    <Adresse>
        <Stadt PLZ="24106">Kiel</Stadt>
    </Adresse>
</Person>
```

Ergänzend zu den textuellen Angaben von Werten innerhalb von Elementen sowie durch deren Verschachtelung können Informationen in sogenannten ***Attributen*** hinterlegt werden, wie dies für die Attribute `vorname`, `name` und `alter` des Elements `Person` gezeigt ist. Zudem können XML-Dokumente auch ***Kommentare*** enthalten, die mit der Zeichenfolge `<!--` eingeleitet und mit `-->` abgeschlossen werden.

Schaut man auf das gezeigte XML-Dokument, so erkennt man, dass jedes Element eine Start- und Endmarkierung, ***Start-*** und ***End-Tag*** genannt, besitzt. Ebenso wie bei HTML werden diese Tags in spitzen Klammern notiert. Gewöhnlich treten öffnendes und schließendes Tag paarweise auf, also in der Notation `<Person>` ... `</Person>`. Die dritte Angabe einer Person zeigt eine Kurzschreibweise, bei der das Element durch die Notation `/>` abgeschlossen werden kann, sofern in dem Element keine Subelemente vorhanden sind.

Wohlgeformte und valide XML-Dokumente

Ergänzend zu diesen ersten syntaktischen, wenig formalen Regeln sollten XML-Dokumente zwei Anforderungen erfüllen: Sie sollten ***wohlgeformt*** und ***valide*** sein. Wohlgeformt bedeutet, dass alle vom XML-Standard geforderten, später ausführlicher genannten Regeln zur Syntax eingehalten werden – etwa dass Elemente ein korrespondierendes Start- und End-Tag besitzen. Neben diesen syntaktischen Anforderungen können auch semantische Anforderungen an ein XML-Dokument aufgestellt werden, unter anderem, welche Tags gültige Elemente darstellen und wie diese miteinander kombiniert werden dürfen. Erfüllt ein XML-Dokument diese Regeln ebenfalls, so ist es nicht nur wohlgeformt, sondern im Hinblick auf diese Anforderungen auch valide.

Abgrenzung von XML zu anderen Formaten

Neben XML sind weitere Formen der Datenspeicherung bzw. -repräsentation denkbar, etwa die in Java integrierte Serialisierung oder eine Repräsentation als Comma Separated Values (CSV) bzw. das immer populärer werdende JSON. Zudem gibt es alternativ auch immer noch proprietäre Formate.

Welche Argumente sprechen für den Einsatz von XML, welche dagegen? Jedes Format besitzt seine spezifischen Stärken und Schwächen. Die Verständlichkeit und die Interoperabilität leiden bei proprietären Formaten – insbesondere weil diese nicht immer voll umfänglich dokumentiert sind. Gerade bei der Kommunikation zwischen verschiedenen Rechnern mit gegebenenfalls unterschiedlichen Betriebssystemen spielen Dinge wie die Byte-Reihenfolge (Big vs. Little Endian) sowie Zeichensatzcodierungen und verschiedene Zeilenumbruchzeichenfolgen (`\r\n` oder nur `\n`) eine Rolle. Diese Feinheiten erschweren den Datenaustausch und machen ihn recht fehleranfällig. Ähnliche Probleme bergen auch binäre Formate, mit denen sich beispielsweise die Inspektion und Modifikation von Daten schwieriger gestalten. Diese Negativpunkte gelten eingeschränkt auch für die in Java integrierte Serialisierung. Wenn eine menschenlesbare Darstellung gewünscht ist, kann man CSV nutzen, das sich vor allem für Listen von Daten eignet, sich gut in Excel importieren und dort verarbeiten lässt. Eine Schwäche von CSV ist aber, dass damit hierarchische Strukturen nur umständlich abbildbar sind. Darüber hinaus lassen sich Metainformationen – wie z. B. die Informationen über die Bedeutung der einzelnen Daten – mithilfe von CSV nur mühsam transportieren. Als Abhilfe sieht man manchmal, dass die erste Zeile als eine Art Kommentar ausgelegt ist und dort die Spalten statt den Nutzinhalt die Bedeutung der Daten bzw. Attributnamen enthalten, etwa wie folgt:

```
Vorname,  Name,          Wohnort
Michael,  Inden,         Zürich
Clemens,  Gugenberger,   Aachen
Carsten,  Kern,          Düren
...
```

Zwar ist diese Art der Dokumentation für uns als Menschen hilfreich, jedoch kann dies die Verarbeitung mit Programmen erschweren, da hier wieder spezielle Prüfungen erfolgen müssen und Sonderfälle zu behandeln sind.

Nach diesem kurzen Ausflug zu CSV kommen wir wieder zu XML. Es ist sicherlich keine eierlegende Wollmilchsau, jedoch besitzt es gegenüber anderen Formaten unter anderem folgende positive Eigenschaften:

- **Einfache Handhabung** – Aufgrund der Darstellung als simples Textdokument ist die maschinelle Verarbeitung von XML relativ einfach. Zudem können – sofern nötig – mithilfe eines Texteditors Änderungen vorgenommen werden.
- **Robustheit** – Informationen lassen sich in XML leicht extrahieren und korrekt zuordnen, während bei CSV Angaben versehentlich aufgrund der falschen Position verwechselt werden können. In XML spielt oftmals die Reihenfolge der Angabe von Elementen keine wesentliche Rolle, da sich aufgrund des Elementnamens die Daten problemlos zuordnen lassen.
- **Einfache Repräsentation** – In XML lassen sich Daten gut strukturieren und im Gegensatz zu HTML findet keine Vermischung mit Darstellungsinformationen statt. Bei geeignetem Layout ergibt sich eine gut erkennbare Dokumentenstruktur.
- **Einfache Verarbeitung** – XML kann recht einfach in eigenen Programmen verarbeitet werden, wenn man die gängigen Frameworks nutzt.
- **Einfache Transformierbarkeit** – Mithilfe der später vorgestellten Transformationen (XSLT) lassen sich die Daten von XML in andere Darstellungsformen, etwa HTML oder CSV, überführen.
- **Umfangreiche Such- und Abfragemöglichkeiten** – Die textuelle Darstellung von XML erlaubt es, Daten auf einfache Weise zu suchen. Es lassen sich sogar komplexere Suchen und Abfragen, die Hierarchieebenen und Filterung nutzen, mithilfe von XPath ausführen.

Neben diesen ganzen positiven Aspekten sollte man allerdings nicht verschweigen, dass XML-Dokumente durch die Vielzahl an Tags mitunter schwerfällig zu lesen sind und schnell recht umfangreich und unübersichtlich werden. Schlimmer noch: Durch die Wiederholung der Tag-Namen in Start- und End-Tag findet man eigentlich fast immer mehr Tags als Nutzinformationen. Als Folge wird es mitunter schwierig, relevante Informationen aus der »Textwüste« herauszulesen. Deswegen wird XML auch als »***overly verbose***« bezeichnet. Gerade wenn viele Datensätze im XML repräsentiert werden, steht die Größe des XML-Dokuments in keinem guten Verhältnis zu den tatsächlich zu transportierenden Nutzdaten. Als Alternative zu XML erfreut sich daher das deutlich kompaktere Datenformat JSON wachsender Beliebtheit. Darauf werde ich in Abschnitt 1.5 eingehen.

Abschließend möchte ich in noch erwähnen, dass XML die Grundlage für verschiedene spezifische Auszeichnungssprachen bildet, z. B. SVG (Scalable Vector Graphics) zur Beschreibung von Vektorgrafiken oder MathML zur Beschreibung mathematischer Formeln.

Tipp: Auswertung von XML-Dokumenten und Parser

Die Extraktion von Daten aus XML-Dokumenten wird zu einer fehlerträchtigen und mühsamen Arbeit, wenn man sie selbst programmiert. Glücklicherweise unterstützen uns hierbei sogenannte ***Parser***. Diese Softwarekomponenten übernehmen den Vorgang der Auswertung von XML-Dokumenten. Dabei werden alle Besonderheiten berücksichtigt, etwa wie das Markup zu interpretieren ist, wo ein Element startet und wo dieses endet, ob und welche Attribute es besitzt u. v. m. Das Java-XML-API bietet zudem noch folgende Vorteile:

- Man muss sich nicht um Zeichensatzcodierungen kümmern.
- Die Behandlung von Whitespaces wird erleichtert.
- Die Daten können bequem abgefragt werden und müssen nicht selbst aus dem Markup extrahiert werden.
- Man kann sich darauf verlassen, dass das Dokument korrekt verarbeitet wird.

Der letzte Punkt klingt so selbstverständlich, ist er aber leider nicht: Mir ist einmal ein firmeneigener, nicht ausgereifter, selbst geschriebener Parser untergekommen, der nicht mit Kommentaren innerhalb von XML klarkam.

1.1.1 Bestandteile und Aufbau eines XML-Dokuments

Wie bereits angedeutet, hat jedes XML-Dokument einen definierten Aufbau und muss gewissen Regeln folgen, beispielsweise müssen Elemente korrekt verschachtelt sein. Schauen wir nun auf einige Vorgaben zum Aufbau eines XML-Dokuments.

Elemente, Tags und Attribute

Die Kombination aus öffnendem und schließendem Tag sowie alles, was dazwischen steht, wird als ***Element*** bezeichnet. Elemente beschreiben Daten und können weitere Subelemente, Attribute und auch textuelle Nutzdaten enthalten. Ein Element definiert eine semantische Einheit und dient zur Strukturierung der Daten.. Nachfolgend schauen wir nochmals auf das schon gezeigte Beispiel der Personen:

```
<Personen>
    <Person vorname="Michael" name="Inden" alter="44"></Person>
    <Person vorname="Hans" name="Meyer" alter="32"></Person>
    <Person vorname="Peter" name="Muster" alter="55" />
</Personen>
```

Die Eigenschaften von Elementen lassen sich entweder durch Subelemente oder in Form von Attributen definieren. In der XML-Gemeinde ist man sich nicht immer einig, wann man Subelemente nutzen sollte und wann Attribute zu bevorzugen sind. Grundsätzlich können allerdings nur solche Eigenschaften als Attribut modelliert werden, die sich sinnvoll als Text darstellen lassen – also Zahlen und Texte, jedoch keine komplexeren Informationen.

Attribute besitzen in XML einen Namen und einen Wert, wobei Attributwerte immer textueller Natur sind und daher in Anführungszeichen notiert werden. Das ist ein Unterschied zu HTML, das die Angabe von Zahlenliteralen erlaubt, etwa `width=100`.

Darüber hinaus ist im Gegensatz zu HTML die Groß-/Kleinschreibung für Tags in XML von Relevanz: Die Angaben `<Person>` und `<person>` werden beim Auswerten eines XML-Dokuments als unterschiedliche Elemente angesehen.

Die Namen für Elemente kann man relativ frei vergeben, jedoch darf der Name nicht mit `xml` oder Zahlen, Bindestrichen usw., sondern nur mit Buchstaben beginnen, ansonsten ist man in den nachfolgenden Zeichen recht frei. Allerdings sollte man bei der Namensgebung auf Verständlichkeit achten.

Bedeutung von Tags und Elementen In XML dienen Elemente bekanntermaßen der Beschreibung der Daten. Sofern die Namen der Elemente und die ihrer Attribute sinnvoll gewählt sind, lässt sich eine verständliche, im Idealfall fast selbsterklärende Beschreibung und Repräsentation von Daten vornehmen.

Die im einführenden Beispiel genutzten Tags hätten auch anders benannt sein können, etwa `Mitglieder` und `Mitglied` für eine Vereinsverwaltung oder `Kundenstamm` und `Kunde` für eine Kundenverwaltung. Das Besondere an XML ist, dass man mit Tags Semantik beschreiben kann und dies dabei hilft, eine sprechende, verständliche und menschenlesbare Repräsentation der Nutzdaten aufzubereiten. Das ist ein großer Vorteil von XML gegenüber vielen anderen Notationsformen.

Prolog und Processing Instructions

Neben den eigentlichen Nutzdaten können in einem XML-Dokument auch Metainformationen in Form von sogenannten ***Processing Instructions*** hinterlegt werden. Diese werden im XML-Dokument durch `<? ?>` markiert. Derart gekennzeichnete Informationen werden bei der Auswertung (dem sogenannten ***Parsing***) des XML-Dokuments speziell behandelt und können spezifische Verarbeitungsschritte auslösen.

Ganz elementar ist jedoch eine Prolog genannte Metainformation, die von ihrer Syntax stark an eine Processing Instruction erinnert. Jedes XML-Dokument sollte mit einem Prolog ähnlich zu Folgendem starten, um festzulegen, dass die nachfolgenden Daten ein XML-Dokument gemäß der Version 1.0 der XML-Spezifikation[1] sind, in dem das angegebene Zeichensatz-Encoding, hier `UTF-8`, genutzt wird:

```
<?xml version="1.0" encoding="UTF-8" ?>
```

Zwar ist ein Prolog optional, wird er aber angegeben, so muss er am Beginn des XML-Dokuments stehen. Weil XML-Dokumente auch Sonderzeichen oder Umlaute enthalten können, die in unterschiedlichen Zeichensatz-Encodings unterschiedlich dargestellt werden, ist es sehr sinnvoll, das verwendete Zeichensatz-Encoding anzugeben und das

[1] Zwar gibt es bereits eine Version 1.1, doch die meisten Parser unterstützen nur Version 1.0.

XML-Dokument auch gemäß des angegebenen Encodings abzuspeichern.[2] Falls nichts angegeben wurde, wird standardmäßig UTF-8 genutzt. Als Codierungen findet man auch `ISO-8859-1` oder `UTF-16`.

Kommentare

Mitunter ist es zum Verständnis der Daten sinnvoll, diese mit Kommentaren zu versehen, die bei der Auswertung des Dokuments ignoriert werden. Im einführenden Beispiel haben wir Kommentare und deren Syntax `<!-- Kommentar -->` bereits kennengelernt – eine Verschachtelung ist jedoch nicht erlaubt.

Spezialzeichen

Bei der Beschreibung der Syntax von XML haben wir gesehen, dass diverse Zeichen eine Bedeutung tragen, etwa <, >, ", ' und &. Teilweise sollen diese Zeichen Bestandteil der Werte von Elementen und Attributen sein, etwa wenn in einer XML-Datei mathematische Ausdrücke abzubilden sind. Etwas naiv könnte man diese wie folgt notieren:

```
<Spezialzeichen>
    <Frage>Bitte antworte: Wie lautet das Ergebnis der Berechnung 44 / 11? &
                wen interessiert das?</Frage>
    <Gleichung formel="a < b & b < c => a < c"/>
</Spezialzeichen>
```

Dadurch käme es beim Parsing zu Fehlern. Weil es mitunter nötig ist, diese Zeichen in XML repräsentieren zu können, gibt es die in Tabelle 1-1 gezeigten vordefinierten Ersatzzeichenfolgen.

Tabelle 1-1 *Vordefinierte Ersetzungszeichenfolgen bei XML*

Zeichen	Bedeutung
<	`<`
>	`>`
"	`&quot;`
'	`&apos;`
&	`&`

[2] Achten Sie beim Bearbeiten penibel darauf, mit welchem Zeichensatz-Encoding Sie Dateien abspeichern sowie einlesen und verarbeiten – insbesondere sollte die Prolog-Angabe konsistent mit der Dateicodierung sein. Ansonsten kann es zu Problemen kommen, im Speziellen bei der Verarbeitung von Sonderzeichen und Umlauten. Vor allem dann, wenn unterschiedliche Programme die XML-Dokumente verarbeiten, führt eine Abweichung zwischen tatsächlich genutztem Zeichensatz-Encoding und dessen Angabe im XML-Dokument zu Fehlern in der Interpretation.

Mit diesem Wissen machen wir uns an die Korrektur und schreiben Folgendes:

```
<Spezialzeichen>
    <Frage>Bitte antworte: Wie lautet das Ergebnis der Berechnung 44 / 11? &amp;
        wen interessiert das?</Frage>

    <!-- a < b & b < c => a < c -->
    <Gleichung formel="a &lt; b &amp; b &lt; c =&gt; a &lt; c"/>

    <Antwort>https://www.google.de/?gws_rd=ssl#q=44%2F11</Antwort>
</Spezialzeichen>
```

Im gezeigten XML habe ich noch eine Antwort hinzugefügt. Gerade im Zusammenhang mit URLs bzw. deren Query-Parametern, die in XML angegeben werden, ist der Einsatz des Zeichens `&` oftmals sinnvoll.

Syntaktische Anforderungen an ein XML-Dokument

Damit ein XML-Dokument als wohlgeformt angesehen wird, müssen gewisse Voraussetzungen erfüllt sein – einige davon habe ich bereits erwähnt:

1. Ein XML-Dokument kann mit einem Prolog beginnen. Davor dürfen keine Zeichen stehen.[3] Im Speziellen auch keine Leerzeilen oder Kommentare!
2. Es darf nur genau ein Wurzelelement geben.
3. Jedes Element besitzt ein Start- und ein End-Tag, wobei das Start-Tag immer vor dem End-Tag notiert werden muss. Als Abkürzung für ein leeres Element (d. h. ohne textuellen Inhalt und ohne Subelemente) gibt es die Kurzform `/>`.
4. Elemente dürfen nur hierarchisch ineinander geschachtelt werden, also wie folgt: `<a> <b> </b> </a>`, aber nicht alternierend: `<a> <b> </a> </b>`.
5. Werte von Attributen sind immer textuell und daher in Anführungszeichen notiert.

Falls eine dieser Bedingungen *nicht* erfüllt ist, so wird gemäß W3C-Spezifikation gefordert, dass bei der Auswertung des XML-Dokuments ein Fehler auftreten muss und die Abarbeitung gestoppt wird. XML-Parser sind da rigoros: Ein kleiner (struktureller) Fehler führt sofort zum Abbruch des Parsings. Allerdings ist das nicht nur zur getreuen Erfüllung des Standards so geregelt; meist ist nach dem ersten Fehler schon keine sinnvolle Interpretation des Inhalts mehr möglich, weil die Verarbeitung semantisch nicht mehr fortgesetzt werden kann.

Das steht stark im Gegensatz zur Auswertung von HTML, bei der der Webbrowser versucht, Fehler auszubügeln und trotz eventuell inkonsistenter Eingabedaten möglichst immer eine Darstellung aufbereiten zu können. Dadurch, dass die erlaubten Tags bekannt sind, kann ein HTML-Parser besser mögliche Korrekturen anwenden. Mitunter ist aber je nach Fehler die Darstellung dann recht dürftig.

[3] Außer einer Byte Order Mark, einer Kennzeichnung der Bytereihenfolge. Können Editoren damit nicht umgehen, so sieht man oftmals »Schmutzzeichen« wie folgende: ï»¿.

Nun kann man sich fragen, wie denn bei benutzerdefinierten Tags eine inhaltliche Prüfung möglich wird. Dazu erlaubt es XML, eine Vorgabe zum Aufbau eines XML-Dokuments zu definieren. Damit beschäftigen wir uns gleich in Abschnitt 1.1.2.

Namensräume

Kommen wir nachfolgend zu Namensräumen. Diese sind immer dann wichtig, wenn man mehrere XML-Dokumente verschiedener Herkunft miteinander kombinieren möchte. Derartige XML-Dokumente könnten gleichnamige Elemente enthalten, die dann nicht mehr eindeutig zuzuordnen wären. Denken Sie beispielsweise an eine Personenliste und eine CD-Sammlung, wo in beiden ein Element `Name` definiert ist.

Mithilfe von Namensräumen verhindert man Konflikte bei ansonsten gleichnamigen Elementen. Namensräume kann man sich ähnlich wie die Strukturierung von Verzeichnissen und Packages vorstellen, die jeweils nur eindeutige Dateinamen bzw. Klassennamen enthalten dürfen.

Zur Definition eines Namensraums nutzt man URIs, die Webadressen repräsentieren können. Für einen Namensraum muss dies keiner realen Webadresse entsprechen. Im nachfolgenden Listing wird der Namensraum `jp` definiert. Dazu wird im ersten Tag das Attribut `xmlns` (xmlns = XML name space, also XML-Namensraum) genutzt.

```
<jp:Person xmlns:jp="http://www.javaprofi.de">
    <jp:Name>Inden</jp:Name>
</jp:Person>
```

Nachfolgend wird das Mixen zweier gleichnamiger Elemente gezeigt:

```
<root>
    <jp:Person xmlns:jp="http://www.javaprofi.de">
        <jp:Name>Inden</jp:Name>
    </jp:Person>
    <CD>
        <Name>Best of 80's</Name>
    </CD>
</root>
```

Ein etwas realistischeres Beispiel liefert SELFHTML.[4] Dort findet man eine Überschneidung der Elemente `nummer` und `name`, die durch die Namensräume `produkt` und `kunde` eindeutig identifiziert werden:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<bestellung xmlns:produkt="http://localhost/XML/produkt"
                    xmlns:kunde="http://localhost/XML/kunde">
    <produkt:nummer>p49393</produkt:nummer>
    <produkt:name>JXY Rasierer VC100</produkt:name>
    <produkt:menge>1</produkt:menge>
    <produkt:preis>69,--</produkt:preis>
    <kunde:nummer>k2029</kunde:nummer>
    <kunde:name>Meier, Fritz</kunde:name>
    <kunde:lieferadresse>Donnerbalkenstr.14, 80111 München</kunde:lieferadresse>
</bestellung>
```

[4] `http://wiki.selfhtml.org/wiki/XML/Regeln/XML-Namensr%C3%A4ume`

1.1.2 Validierung eines XML-Dokuments

Neben den rein syntaktischen Anforderungen an ein XML-Dokument ist es wünschenswert, auch semantische Forderungen aufstellen zu können, die beschreiben, welche Elemente in welcher Kombination und Häufigkeit vorkommen dürfen. Das kann in Form einer DTD (Document Type Definition) oder einer XSD (XML Schema Definition) geschehen. Beide ermöglichen eine Festlegung dessen, was wir für HTML vielfach als selbstverständlich erachten, nämlich, dass es gewisse Schlüsselwörter bzw. Tag-Namen gibt, die wiederum bestimmte Attribute besitzen können und wie Elemente geschachtelt werden dürfen.

Sowohl DTD als auch XSD erlauben es, den strukturellen Aufbau eines XML-Dokuments festzulegen. In der Strukturbeschreibung werden Elemente sowie deren Subelemente und Attribute definiert. Auf DTDs und auf XSDs werden wir einen kurzen Blick werfen, damit Sie in der Praxis zumindest so viel Grundlagenwissen besitzen, dass Sie eine solche Datei grob entschlüsseln können. DTDs werden hier behandelt, obwohl sie veraltet und ein wenig angestaubt sind und beispielsweise die Spezifikation zeitgemäßer zusätzlicher Typeinschränkungen nicht erlauben. Allerdings findet man sie immer noch recht häufig und ihre Kenntnis erleichtert das Verständnis von XSDs.

Validierung mit DTD

Wie zuvor kurz angedeutet, sind im XML-Standard keine Tags zur Datenbeschreibung vordefiniert. Wollen zwei Applikationen Daten untereinander austauschen, so müssen beide das gleiche Verständnis vom strukturellen Aufbau, also der verwendeten Elemente und Attribute sowie deren erlaubten Kombinationen, besitzen. Genau dies wird durch eine DTD festgelegt. Anders gesagt: Eine DTD definiert, welche Elemente im XML-Dokument vorkommen dürfen und wie diese dort angeordnet werden (dürfen).

Schauen wir auf folgende DTD (`Personen.dtd`) für das eingangs des Kapitels präsentierte XML-Dokument, das eine Menge von Personen definiert:

```
<!ELEMENT Personen (Person)* >
<!ELEMENT Person EMPTY >
<!ATTLIST Person vorname CDATA #REQUIRED
                 name    CDATA #REQUIRED
                 alter   CDATA #REQUIRED>
```

Die gezeigte DTD enthält sowohl die Beschreibung für Elemente mit `!ELEMENT` sowie für Attribute mit `!ATTLIST`. Es werden sowohl die für das Beispiel gültigen Elemente (also `Personen` und `Person`) sowie Attribute (`vorname`, `name` und `alter`) als auch deren Kombination festgelegt – in unserem Beispiel gilt, dass alle drei Attribute für das Element `Person` angegeben werden müssen. Durch `(Person)*` wird festgelegt, dass 0 bis beliebig viele `Person`-Elemente innerhalb von `Personen` vorkommen dürfen

Notation von Attributen Die Daten eines `Person`-Elements können durch Attribute festgelegt werden. In der DTD wird durch die Angabe von `!ATTLIST` beschrieben, wie der Aufbau aussieht. Im Beispiel gilt demnach, dass ein `Person`-Element die drei Attribute `vorname`, `name` und `alter` besitzt. Diese sind alle vom Typ `CDATA`[5], was für Character Data steht. Damit sind textuelle Informationen gemeint, die keine Tags (also kein weiteres Markup) enthalten. Darüber hinaus sind die Attribute als notwendig (`#REQUIRED`) definiert.

Integration oder Verweis auf DTD Zur Validierung eines Dokuments kann man die DTD in die XML-Datei aufnehmen. Da es verschiedene Ausprägungen der Daten, aber nur eine beschreibende DTD geben sollte, ***ist es nahezu immer sinnvoller, einen Verweis auf eine Datei aufzunehmen, die die DTD enthält, anstatt die DTD im XML-Dokument explizit aufzuführen***. Den Verweis notiert man wie folgt:

```
<?xml version="1.0" encoding="utf-8" ?>
<!DOCTYPE Personen SYSTEM "Personen.dtd">
<Personen>
    <Person vorname="Michael" name="Inden" alter="44"></Person>
    <Person vorname="Hans" name="Meyer" alter="32"></Person>
    <Person vorname="Peter" name="Muster" alter="55" />
</Personen>
```

Neben der Kürze besitzt dieses Vorgehen den Vorteil, dass eine bessere Trennung von Zuständigkeiten vorgenommen wird und es dadurch auch zu Erleichterungen bei möglichen Änderungen oder Wartungsarbeiten an der DTD kommt: Diese müssen nur einmal zentral ausgeführt werden und nicht in jedem nutzenden XML-Dokument, sodass Inkonsistenzen zwischen den DTDs in den Dateien leichter als bei direkt in XML-Dokumenten definierten DTDs vermieden werden können – zwischen DTD und XML werden diese automatisch verhindert.

Tipp: Öffentliche DTDs

Die gerade gezeigte Form einer sogenannten externen DTD (weil diese in einer eigenen Datei gespeichert ist) nutzt man insbesondere für eigene nicht öffentliche Applikationen. Soll eine DTD auch anderen zugänglich gemacht werden, so kann man eine externe öffentliche DTD definieren. Dazu dient das Schlüsselwort `PUBLIC` sowie die Angabe eines Namens (nachfolgend symbolisch als `<DTD-Name>` notiert) und einer URL (`<DTD-Location>`):

```
<!DOCTYPE root_element PUBLIC "<DTD-Name>" "<DTD-Location>">
```

[5] Das darf nicht mit dem `#PCDATA` für Parsed Character Data für Elemente verwechselt werden. `#PCDATA` ist zwar ähnlich zu `CDATA`, erlaubt aber beliebig viel Text und auch Zeilenumbrüche. Allerdings sind keine inneren Elemente erlaubt. Wollte man den Namen statt als Attribut als Element definieren, so würde man in der gezeigten DTD `<!ELEMENT Person (name)>` notieren. Damit im Namen nur Text auftreten kann, schreibt man: `<!ELEMENT name (#PCDATA)>`.

Validierung mit XSD-Schema

Wie schon erwähnt, wird die eben gezeigte Prüfung eines XML-Dokuments über eine DTD mittlerweile nicht mehr empfohlen. Stattdessen sollte eine XSD genutzt werden. Das hat mehrere Gründe: Einer besteht darin, dass DTDs selbst kein gültiges XML sind. Ein weiterer ist, dass eine DTD als Typangaben nur Zeichenketten, aber keine anderen Datentypen kennt. Dadurch fehlen beim Einsatz einer DTD einige semantische Ausdrucksmöglichkeiten, etwa eine Festlegung eines Werts auf positive Ganzzahlen.

Elementare Typen Mit XSD kann man einfache Datentypen nutzen und wie in Programmiersprachen daraus komplexere Typen definieren. Als Basistypen stehen unter anderem folgende bereit:

- `xs:string` für textuelle Informationen,
- `xs:integer` und `xs:float` für Ganz- und Fließkommazahlen,
- `xs:boolean` für Wahrheitswerte sowie
- `xs:date` für Datumsangaben.

Tipp: Einschränkungen definieren

Im nachfolgenden Beispiel sehen wir die Einschränkung eines Wertebereichs für eine Altersangabe. Dazu stehen einige Attribute mit den selbsterklärenden Namen `minExclusive`, `minInclusive`, `maxExclusive`, `maxInclusive` zur Verfügung:

```
<xs:simpleType name="Alter">
  <xs:restriction base="xs:integer">
    <xs:minInclusive value="1"/>
    <xs:maxInclusive value="150"/>
  </xs:restriction>
</xs:simpleType>
```

Für nicht ganzzahlige Wertebereiche kann mithilfe der Attribute `totalDigits` und `fractionDigits` festgelegt werden, wie viele Ziffern insgesamt vorkommen dürfen und wie viele davon Nachkommastellen sind.

Soll eine Aufzählung gültiger Werte nachgebildet werden, so nutzt man das Element `enumeration`, etwa zur Definition eines Typs `Geschlecht`:

```
<xs:simpleType name="Geschlecht">
  <xs:restriction base="xs:string">
    <xs:enumeration value="männlich"/>
    <xs:enumeration value="weiblich"/>
  </xs:restriction>
</xs:simpleType>
```

Es ist auch möglich, einen regulären Ausdruck als Gültigkeitsmuster anzugeben. Zur Validierung einer internationalen Telefonnummer mit Vorwahl notiert man:
`<xs:pattern value="(+|00)?[0-9]{2}-[0-9]{2,4}-[0-9]{5,8}"/>`

Komplexe Typen Die eben erwähnten elementaren Typen erleichtern die Definition von zusammengesetzten komplexeren Typen, etwa einer Liste von Personen. Zu deren Beschreibung benötigt man noch weitere Modellierungs- bzw. Ausdrucksmittel wie unter anderem die Komposition von Typen aus anderen Typen. Das ist ähnlich zum objektorientierten Programmieren, jedoch arbeitet man bei XML auf textueller Ebene. Im Beispiel wird eine Person durch drei Attribute modelliert:

```
<xs:complexType name="Person">
    <xs:attribute name="vorname" type="xs:string" />
    <xs:attribute name="name" type="xs:string" />
    <xs:attribute name="alter" type="xs:positiveInteger" />
</xs:complexType>
```

Neben einer Zusammensetzung aus Attributen kann man einen neuen Typ auch wie folgt als Menge von Elementen[6] definieren:

```
<xs:element name="Personen">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="Person" minOccurs="0" maxOccurs="unbounded" />
        </xs:sequence>
    </xs:complexType>
</xs:element>
```

Wir sehen hier, dass sich die Menge der Personen im Wurzelelement `Personen` als komplexer Typ (`xs:complexType`) aus einer Folge (`xs:sequence`) von Elementen vom Typ `Person` zusammensetzt. Zudem kann die Anzahl der erlaubten Vorkommen festgelegt werden: Über die Angabe `minOccurs` und `maxOccurs` lässt sich der gültige Wertebereich einschränken, hier von 0 bis unbeschränkt (`unbounded`). Erfolgt keine Mengenangabe in der XSD, so wird standardmäßig genau ein Vorkommen erwartet.

Strukturbeschreibung mit einer XSD

Schauen wir nun auf eine XSD für die `Personen`-XML-Datei aus dem einleitenden Beispiel. Diese definieren wir mithilfe anonymer Typen folgendermaßen:

```
<?xml version="1.0" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:element name="Personen">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="Person" minOccurs="0" maxOccurs="unbounded">
                    <xs:complexType>
                        <xs:attribute name="vorname" type="xs:string" />
                        <xs:attribute name="name" type="xs:string" />
                        <xs:attribute name="alter" type="xs:positiveInteger" />
                    </xs:complexType>
                </xs:element>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
</xs:schema>
```

[6] und in Kombination mit Attributen

Verweis auf XSD Soll zur Validierung eines XML-Dokuments statt einer DTD eine XSD genutzt werden, so ändert sich die Referenzierung im Dokument wie folgt:

```
<?xml version="1.0" ?>
<Personen xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://meinnamespace.meinefirma.de/Personen.xsd"
                xmlns="http://meinnamespace.meinefirma.de">
    <Person vorname="Michael" name="Inden" alter="44" />
    <!-- ... -->
</Personen>
```

Hinweis: IDE zum Validieren nutzen

Das händische Editieren innerhalb umfangreicher XML-Dokumente wird schnell ein wenig fummelig und fehlerträchtig. Um XML-Dateien analysieren und mögliche Fehler einfacher finden zu können, ist die VALIDATE-Funktionalität aus dem Eclipse-Kontextmenü hilfreich. Sofern im XML-Dokument eine DTD oder XSD angegeben und für Eclipse zugreifbar ist, können die Daten validiert werden. In jedem Fall wird geprüft, ob ein XML-Dokument wohlgeformt ist.

1.2 XML-Verarbeitung mit JAXP

Nachdem wir nun wissen, wie wir Daten mit XML beschreiben, wollen wir uns anschauen, wie wir als XML vorliegende Informationen mit Java verarbeiten können. Dafür ist ein sogenannter ***Parser*** zuständig, von dem verschiedene Ausprägungen existieren: Typische Vertreter sind ***SAX***- und ***DOM***-Parser. Dabei steht SAX für ***Simple API for XML*** und DOM für ***Document Object Model***. Beide Varianten sind zwar schon älter, jedoch immer noch recht gebräuchlich. Mit JDK 6 wurde mit ***StAX*** (Streaming API for XML) ein neueres API zur XML-Verarbeitung eingeführt. Diese APIs, die zusammen mit XSLT das ***JAXP*** (***Java API for XML Processing***) bilden, werden nun anhand prägnanter Beispiele vorgestellt. Für Details verweise ich auf die weiterführende Literatur am Kapitelende.

Event-basierte Arbeitsweise (SAX und StAX)

Beim SAX wird ein XML-Dokument stückweise gelesen und bei jedem Auffinden eines relevanten XML-Bestandteils eine spezifische Aktion ausgelöst. Bei diesem Event-basierten Ansatz registriert eine Applikation beim Parser spezielle Callback-Listener (sogenannte Content Handler). Beim Einlesen von Bestandteilen des XML-Dokuments werden Methoden dieser Content Handler aufgerufen. Modifikationen am XML-Dokument können mit SAX nicht vorgenommen werden. Recht ähnlich arbeitet StAX: Hier wird ein Iterator-basierter Ansatz gefahren, bei dem die Applikation die Kontrolle hat und das XML mithilfe eines Parsers analysiert. Jeder Bestandteil eines XML-Dokuments wird vom Parser als Token oder `XMLEvent` repräsentiert. Im Gegensatz zu SAX kann man mit StAX auch XML-Dokumente schreiben.