Sven Eric Panitz

Lehrbriefe Programmierung in Java

XML

XML

Sven Eric Panitz

28. April 2023

Inhaltsverzeichnis

1 XML

XML ist eine Sprache, die es erlaubt Dokumente mit einer logischen Struktur zu beschreiben. Die Grundidee dahinter ist, die logische Struktur eines Dokuments von seiner Visualisierung zu trennen. Ein Dokument mit einer bestimmten logischen Struktur kann für verschiedene Medien unterschiedlich visualisiert werden, z.B. als HTML-Dokument für die Darstellung in einem Webbrowser, als pdf- oder postscript-Datei für den Druck des Dokuments und das für unterschiedliche Druckformate. Eventuell sollen nicht alle Teile eines Dokuments visualisiert werden. XML ist zunächst eine Sprache, die logisch strukturierte Dokumente zu schreiben, erlaubt.

Dokumente bestehen hierbei aus den eigentlichen Dokumenttext und zusätzlich aus Markierungen dieses Textes. Die Markierungen sind in spitzen Klammern eingeschlossen.

Beispiel 1.2 Der eigentliche Text des Dokuments sei:

ex1.xml

```
The Beatles White Album
```

Die einzelnen Bestandteile dieses Textes können markiert werden:

ex1.xml

```
<cd>
  <artist>The Beatles</artist>
  <title>White Album</title>
</cd>
```

Die XML-Sprache wird durch ein Industriekonsortium definiert, dem (`www.w3c.org`). Dieses ist ein Zusammenschluß vieler Firmen, die ein gemeinsames Interesse eines allgemeinen Standards für eine Markierungssprache haben. Die eigentlichen Standards des W3C heißen nicht Standard, sondern Empfehlung *(recommendation)* , weil es sich bei dem W3C nicht um eine staatliche oder überstaatliche Standartisiertungsbehörde handelt. Die aktuelle Empfehlung für XML liegt seit August 2006 als Empfehlung in der Version 1.1 vor [PSMY+06].

XML enstand Ende der 90er Jahre und ist abgeleitet von einer umfangreicheren Dokumentenbeschreibungssprache: SGML. Der SGML-Standard ist wesentlich komplizierter und krankt daran,

dass es extrem schwer ist, Software für die Verarbeitung von SGML-Dokumenten zu entwickeln. Daher fasste SGML nur Fuß in Bereichen, wo gut strukturierte, leicht wartbare Dokumente von fundamentaler Bedeutung waren, so dass die Investition in teure Werkzeuge zur Erzeugung und Pflege von SGML-Dokumenten sich rentierte. Dies waren z.B. Dokumentationen im Luftfahrtbereich.[1]

Die Idee bei der Entwicklung von XML war: eine Sprache mit den Vorteilen von SGML zu Entwickeln, die klein, übersichtlich und leicht zu handhaben ist.

Wie wir im Laufe des Kapitels sehen werden, ist die logische Struktur eines XML-Dokuments auch wieder eine hierarchische Baumstruktur, so dass wir alle unsere Überlegungen zu Bäumen direkt auf XML-Dokumente anwenden können.

1.1 XML Kopfdeklaration

Die grundlegendste Empfehlung des W3C legt fest, wann ein Dokument ein gültiges XML-Dokument ist, die Syntax eines XML-Dokuments. Die nächsten Abschnitte stellen die wichtigsten Bestandteile eines XML-Dokuments vor.

Jedes Dokument beginnt mit einer Anfangszeile, in dem das Dokument angibt, dass es ein XML-Dokument nach einer bestimmten Version der XML Empfehlung ist:

ex1.xml

```
<?xml version="1.1"?>
```

Dieses ist die erste Zeile eines XML-Dokuments. Vor dieser Zeile darf kein Leerzeichen stehen. Die derzeitig aktuellste und einzige Version der XML-Empfehlung ist die Version 1.1. vom 16. August 2006. Nach Aussage eines Mitglieds des W3C ist es sehr unwahrscheinlich, dass es jemals eine Version 2.0 von XML geben wird. Zuviele weitere Techniken und Empfehlungen basieren auf XML, so dass die Definition von dem, was ein XML-Dokument ist kaum mehr in größeren Rahmen zu ändern ist.

1.2 Elementknoten

Der Hauptbestandteil eines XML-Dokuments sind die Elemente. Dieses sind mit der Spitzenklammernotation um Teile des Dokuments gemachte Markierungen. Ein Element hat einen *Tagnamen* , der ein beliebiges Wort ohne Leerzeichen sein kann. Für einen Tagnamen *name* beginnt ein Element mit `<name>` und endet mit `</name>`. Zwischen dieser Start- und Endemarkierung eines Elements kann Text oder auch weitere Elemente stehen.

Es wird für XML-Dokument verlangt, dass es genau ein einziges oberstes Element hat.

Beispiel 1.3 Somit ist ein einfaches XML-Dokument ein solches Dokument, in dem der gesammte Text mit einem einzigen Element markiert ist:

[1] Man sagt, ein Pilot brauche den Copiloten, damit dieser die Handbücher für das Flugzeug trägt.

ex1.xml

```
<?xml version="1.0"?>
<myText>Dieses ist der Text des Dokuments. Er ist
mit genau einem Element markiert.
</myText>
```

Im einführenden Beispiel haben wir schon ein XML-Dokument gesehen, das mehrere Elemente hat. Dort umschließt das Element `<cd>` zwei weitere Elemente, die Elemente `<artist>` und `<title>`. Die Teile, die ein Element umschließt, werden der Inhalt des Elements genannt.

Ein Element kann auch keinen, sprich den leeren Inhalt haben. Dann folgt der öffnenden Markierung direkt die schließende Markierung.

Beispiel 1.4 Folgendes Dokument enthält ein Element ohne Inhalt:

ex1.xml

```
<?xml version="1.0"?>
<skript>
  <page>erste Seite</page>
  <page></page>
  <page>dritte Seite</page>
</skript>
```

1.3 Leere Elemente

Für ein Element mit Tagnamen *name* , das keinen Inhalt hat, gibt es die abkürzenden Schreibweise: `<name/>`

Beispiel 1.5 Das vorherige Dokument lässt sich somit auch wie folgt schreiben:

ex1.xml

```
<?xml version="1.0"?>
<skript>
  <page>erste Seite</page>
  <page/>
  <page>dritte Seite</page>
</skript>
```

1.4 Gemischter Inhalt

Die bisherigen Beispiele haben nur Elemente gehabt, deren Inhalt entweder Elemente oder Text waren, aber nicht beides. Es ist aber auch möglich Elemente mit Text und Elementen als Inhalt zu schreiben. Man spricht dann vom gemischten Inhalt *(mixed content)* .

Beispiel 1.6 Ein Dokument, in dem das oberste Element einen gemischten Inhalt hat:

ex1.xml

```
<?xml version="1.0"?>
<myText>Der <landsmann>Italiener</landsmann>
<eigename>Ferdinand Carulli</eigename> war  als Gitarrist
ebenso wie der <landsmann>Spanier</landsmann>
<eigename>Fernando Sor</eigename> in <ort>Paris</ort>
ansäßig.</myText>
```

Tatsächlich ist gemischter Inhalt nicht bei jedermann beliebt. Es gibt zwei große Anwendungshintergründe für XML. Zum einen die Dokumentenbeschreibung. In diesem Kontext ist ein gemischter Inhalt ganz natürlich, wie man am obigen Beispiel erkennen kann. Die zweite Fraktion von Anwendern, die XML nutzt, kommt eher aus dem Bereich allgemeiner Datenverarbeitung. Hier stehen eher Daten, wie in Datenbanken im Vordergrund und nicht Textdokumente. Es wird da zum Beispiel XML genutzt um an einen Webservice Daten zu übermitteln, die dort verarbeitet werden sollen. In diesem Kontext stört gemischter Inhalt mitunter oder hat zumindest wenig Sinn.

Die Problematik wird sofort ersichtlich, wenn man sich überlegt, dass Leerzeichen auch Text sind. Unter dieser Prämisse haben mit Sicherheit alle XML-Dokumente gemischten Inhalt, weil zur optischen besseren Lesbarkeit auch in einem Datendokument, das eigentlich keinen gemischten Inhalt hat, zwischen den einzelnen Elementknoten Leerzeichen in Form von Einrückungen und neuen Zeilen stehen.

1.5 Hierarchische Struktur

Die wohl wichtigste Beschränkung für XML-Dokumente ist, dass sie eine hierarchische Struktur darstellen müssen. Zwei Elemente dürfen sich nicht überlappen. Ein Element darf erst wieder geschlossen werden, wenn alle nach ihm geöffneten Elemente wieder geschlossen wurden.

Beispiel 1.7 Das folgende ist kein gültiges XML-Dokument. Das Element `<bf>` wird geschlossen bevor das später geöffnete Element `<em>` geschlossen worde.

ex1.xml

```
<?xml version="1.0"?>
<illegalDocument>
  <bf>fette Schrift <em> kursiv und fett</bf>
  nur noch kursiv</em>.
</illegalDocument>
```

Das Dokument wäre wie folgt als gültiges XML zu schreiben:

ex1.xml

```
<?xml version="1.0"?>
<validDocument>
  <bf>fette Schrift <em> kursiv und fett</em></bf>
  <em> nur noch kursiv</em>.
</validDocument>
```

Dieses Dokument hat eine hierarchische Struktur.

Die hierarchische Struktur eines XML-Dokuments lässt sich in der Regel gut erkennen, wenn man das Dokument in einem Web-Browser öffnet. Fast alle Web-Browser stellen dann die logische Baumstruktur so dar, dass die Kindknoten eines Elementknoten aufgeklappt und zugeklappt werden können.

1.6 Character Entities

Sobald in einem XML-Dokument eine der spitzen Klammern < oder > auftaucht, wird dieses als Teil eines Elementtags interpretiert. Sollen diese Zeichen hingegen als Text und nicht als Teil der Markierung benutzt werden, sind also Bestandteil des Dokumenttextes, so muss man einen Fluchtmechanismus für diese Zeichen benutzen. Diese Fluchtmechanismen nennt man *character entities* . Eine Character Entity beginnt in XML mit dem Zeichen `&` und endet mit einem Semikolon `;`. Dazwischen steht der Name des Buchstabens. XML kennt die folgenden Character Entities:

Entity	Zeichen	Beschreibung
`<`	<	(less than)
`>`	>	(greater than)
`&`	&	(ampersant)
`&quot;`	"	(quotation mark)
`&apos;`	'	(apostroph)

Somit lassen sich in XML auch Dokumente schreiben, die diese Zeichen als Text beinhalten.

Beispiel 1.8 Folgendes Dokument benutzt Character Entities um mathematische Formeln zu schreiben:

ex1.xml

```
<?xml version="1.0"?>
<gleichungen>
  <gleichung>x+1&gt;x</gleichung>
  <gleichung>x*x&lt;x*x*x für x&gt;1</gleichung>
</gleichungen>
```

1.7 CDATA Section

Manchmal gibt es große Textabschnitte in denen Zeichen vorkommen, die eigentlich durch character entities zu umschreiben wären, weil sie in XML eine reservierte Bedeutung haben. XML bietet die Möglichkeit solche kompletten Abschnitte als eine sogenannte *CData Section* zu schreiben. Eine *CData section* beginnt mit der Zeichenfolge `<![CDATA[` und endet mit der Zeichenfolge: `]]>`. Dazwischen können beliebige Zeichenstehen, die eins zu eins als Text des Dokumentes interpretiert werden.

Beispiel 1.9 Die im vorherigen Beispiel mit Character Entities beschriebenen Formeln lassen sich innerhalb einer CDATA-Section wie folgt schreiben.

ex1.xml

```
<?xml version="1.0"?>
<formeln><![CDATA[
  x+1>x
  x*x<x*x*x für x > 1
]]></formeln>
```

Strukturell sind die CDATA-Sektionen auch nur Textknoten im Dokument. Sie unterscheiden sich von anderen Textknoten nur in der Darstellung in der serialisierten Textform.

1.8 Kommentare

XML stellt auch eine Möglichkeit zur Verfügung, bestimmte Texte als Kommentar einem Dokument zuzufügen. Diese Kommentare werden mit `<!--` begonnen und mit `-->` beendet. Kommentartexte sind nicht Bestandteil des eigentlichen Dokumenttextes.

Beispiel 1.10 Im folgenden Dokument ist ein Kommentar eingefügt:

ex1.xml

```
<?xml version="1.0"?>
<drehbuch>
<titel>Ben Hur</titel>
<akt>
  <szene>Ben Hur am Vorabend des Wagenrennens.
    <!--Diese Szene muß noch ausgearbeitet werden.-->
  </szene>
</akt>
</drehbuch>
```

Kommentarknoten werden nicht als eigentliche Knoten der Baumstruktur des Dokuments betrachtet.

1.9 Processing Instructions

In einem XML-Dokument können Anweisung stehen, die angeben, was mit einem Dokument von einem externen Programm zu tun ist. Solche Anweisungen können z.B. angeben, mit welchen Mitteln das Dokument visualisiert werden soll. Wir werden hierzu im nächsten Kapitel ein Beispiel sehen. Syntaktisch beginnt eine *processing instruction* mit <? und endet mit ?>. Dazwischen stehen wie in der Attributschreibweise Werte für den Typ der Anweisung und eine Referenz auf eine externe Quelle.

Beispiel 1.11 Ausschnitt aus dem XML-Dokument diesen Skripts, in dem auf ein Stylesheet verwiesen wird, dass das Skript in eine HTML-Darstellung umwandelt:

ex1.xml

```
<?xml version="1.0"?>
<?xml-stylesheet
   type="text/xsl"
   href="../transformskript.xsl"?>

<skript>
<titelseite>
<titel>Grundlagen der Datenverarbeitung<white/>II</titel>
<semester>WS 02/03</semester>
</titelseite>
</skript>
```

Processing Instruction-knoten werden nicht als eigentliche Knoten der Baumstruktur des Dokuments betrachtet.

1.10 Attribute

Die Elemente eines XML-Dokuments können als zusätzliche Information auch noch Attribute haben. Attribute haben einen Namen und einen Wert. Syntaktisch ist ein Attribut dargestellt durch den Attributnamen gefolgt von einem Gleichheitszeichen gefolgt von dem in Anführungszeichen eingeschlossenen Attributwert. Attribute stehen im Starttag eines Elements.

Attribute werden nicht als Bestandteils des eigentlichen Textes eines Dokuments betrachtet.

Beispiel 1.12 Dokument mit einem Attribut für ein Element.

ex1.xml

```
<?xml version="1.0"?>
<text>Mehr Information zu XML findet man auf den Seiten
des <link address="www.w3c.org">W3C</link>.</text>
```

Mit Hilfe der Attribute ist es so möglich eine Abbildung an jeden Elementknoten zu definieren. Wie wir im letzten Semester gelernt haben, ist eine Abbildung eine Liste von Paaren. Die Paare sind dabei als Schlüssel/Wert-paare zu verstehen. Ein XML-Attribut stellt ein solches Schlüssel/Wert-Paar dar. Vor dem Gleichheitszeichen steht der Schlüssel und eingeschlossen in Anführungsstrichen findet sich der Wert, auf dem der Schlüssel abgebildet wird.

1.11 Textcodierung

Im Zusammenhang mit IO-Operationen wurde im letzten Semester bereits eingeführt, dass Textdokumente in unterschiedlichen Encodings physikalisch auf einem Datenträger gespeichert werden können.

Auch XML ist ein Dokumentenformat, das nicht auf eine Kultur mit einer bestimmten Schrift beschränkt ist, sondern in der Lage ist, alle im Unicode erfassten Zeichen darzustellen, seien es Zeichen der lateinischen, kyrillischen, arabischen, chinesischen oder sonst einer Schrift bis hin zur keltischen Keilschrift. Jedes Zeichen eines XML-Dokuments kann potentiell eines dieser mehreren zigtausend Zeichen einer der vielen Schriften sein. In der Regel benutzt ein XML-Dokument insbesondere im amerikanischen und europäischen Bereich nur wenige kaum 100 unterschiedliche Zeichen. Auch ein arabisches Dokument wird mit weniger als 100 verschiedenen Zeichen auskommen.

Wenn ein Dokument im Computer auf der Festplatte gespeichert wird, so werden auf der Festplatte keine Zeichen einer Schrift, sondern Zahlen abgespeichert. Diese Zahlen sind traditionell Zahlen die 8 Bit im Speicher belegen, ein sogenannter Byte (auch Oktett). Ein Byte ist in der Lage 256 unterschiedliche Zahlen darzustellen. Damit würde ein Byte ausreichen, alle Buchstaben eines normalen westlichen Dokuments in lateinischer Schrift (oder eines arabischen Dokuments darzustellen). Für ein Chinesisches Dokument reicht es nicht aus, die Zeichen durch ein Byte allein auszudrücken, denn es gibt mehr als 10000 verschiedene chinesische Zeichen. Es ist notwendig, zwei Byte im Speicher zu benutzen, um die vielen chinesischen Zeichen als Zahlen darzustellen.

Die *Codierung* eines Dokuments gibt nun an, wie die Zahlen, die der Computer auf der Festplatte gespeichert hat, als Zeichen interpretiert werden sollen. Eine Codierung für arabische Texte wird den Zahlen von 0 bis 255 bestimmte arabische Buchstaben zuordnen, eine Codierung für deutsche Dokumente wird den Zahlen 0 bis 255 lateinische Buchstaben inklusive deutscher Umlaute und dem ß zuordnen. Für ein chinesisches Dokument wird eine *Codierung* benötigt, die den 65536 mit 2 Byte darstellbaren Zahlen jeweils chinesische Zeichen zuordnet.

Man sieht, dass es *Codierungen* geben muss, die für ein Zeichen ein Byte im Speicher belegen, und solche, die zwei Byte im Speicher belegen. Es gibt darüber hinaus auch eine Reihe Mischformen, manche Zeichen werden durch ein Byte andere durch 2 oder sogar durch 3 Byte dargestellt.

Im Kopf eines XML-Dokuments kann angegeben werden, in welcher Codierung das Dokument abgespeichert ist.

Beispiel 1.13 Dieses Skript ist in einer Codierung gespeichert, die für westeuropäische Dokumente gut geeignet ist, da es für die verschiedenen Sonderzeichen der westeuropäischen Schriften einen Zahlenwert im 8-Bit-Bereich zugeordnet hat. Die Codierung mit dem Namen: `iso-8859-1`. Diese wird im Kopf des Dokuments angegeben:

ex1.xml

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<skript><kapitel>blablabla</kapitel></skript>
```

Wird keine Codierung im Kopf eines Dokuments angegeben, so wird als Standardcodierung die sogenannte `utf-8` Codierung benutzt. In ihr belegen lateinische Zeichen einen Byte und Zeichen anderer Schriften (oder auch das Euro Symbol) zwei bis drei Bytes.

Eine Codierung, in der alle Zeichen mindestens mit zwei Bytes dargestellt werden ist: `utf-16`, die Standardabbildung von Zeichen, wie sie im *Unicode* definiert ist.

1.12 DTD

In der XML-Empfehlung integriert ist die Definition der document type description language, kurz DTD.

Die DTD ermöglicht es zu formulieren, welche Tags in einem Dokument vorkommen sollen. DTD ist keine eigens für XML erfundene Sprache, sondern aus SGML geerbt.

DTD-Dokumente sind keine XML-Dokumente, sondern haben eine eigene Syntax. Allerdings kann ein XML Dokument mit einer DTD beginnen. DTD-Abschnitte zu Beginn des Dokuments sind ein legaler und normaler Bestandteil eines XML Dokuments.

Wir stellen im einzelnen diese Syntax vor:

- `<!DOCTYPE` *root-element* `[` *doctype-declaration* `...` `]>`

  Legt den Namen des top-level Elements fest und enthält die gesammte Definition des erlaubten Inhalts.

- `<!ELEMENT` *element-name* *content-model* `>`

  assoziiert einen *content model* mit allen Elementen, die diesen Namen haben.

  content models können wie folgt gebildet werden.:

  - `EMPTY`: Das Element hat keinen Inhalt.
  - `ANY`: Das Element hat einen beliebigen Inhalt
  - `#PCDATA`: Zeichenkette, also der eigentliche Text.
  - durch einen regulären Ausdruck, der aus den folgenden Komponenten gebildet werden kann.
    * Auswahl von Alternativen (oder): `(...|...|...)`
    * Sequenz von Ausdrücken: `(...,...,...)`

    - Option: `...?`
    - Ein bis mehrfach Wiederholung: `...*`
    - Null bis mehrfache Wiederholung: `...+`
- `<!ATTLIST` *`element-name attr-name attr-type attr-default`* `...>`

  Liste, die definiert, was für Attribute ein Element hat:

  Attribute werden definiert durch:

  - `CDATA`: beliebiger Text ist erlaubt
  - `(`*`value`*` |...)`: Aufzählung erlaubter Werte

  Attribut *default* sind:

  - `#REQUIRED`: Das Attribut muß immer vorhanden sein.
  - `#IMPLIED`: Das Attribut ist optional
  - `"value"`: Standardwert, wenn das Attribut fehlt.
  - `#FIXED "value"`: Attribut kennt nur diesen Wert.

Beispiel 1.14 Ein Beispiel für eine DTD, die ein Format für eine Rezeptsammlung definiert.

ex1.xml

```
<!DOCTYPE collection SYSTEM "collection.dtd" [
<!ELEMENT collection (description,recipe*)>

<!ELEMENT description ANY>

<!ELEMENT recipe (title,ingredient*,preparation
                 ,comment?,nutrition)>

<!ELEMENT title (#PCDATA)>

<!ELEMENT ingredient (ingredient*,preparation)?>
<!ATTLIST ingredient name CDATA #REQUIRED
                     amount CDATA #IMPLIED
                     unit CDATA #IMPLIED>

<!ELEMENT preparation (step)*>

<!ELEMENT step (#PCDATA)>

<!ELEMENT comment (#PCDATA)>

<!ELEMENT nutrition EMPTY>
<!ATTLIST nutrition fat CDATA #REQUIRED
                    calories CDATA #REQUIRED
                    alcohol CDATA #IMPLIED>]>
```

Es ist auch möglich DTDs als externe Dateien zu haben und zu Beginn eines XML-Dokuments aus diese zu referenzieren:

ex1.xml

```
<!DOCTYPE note SYSTEM "collection.dtd">
```

2 Weitere wichtige XML Spezifikationen

Mit der Spezifikation des XML Formats war noch lange nicht schluss. Darauf bauen unzählige Spezifikationen auf, die im W3C veröffentlicht werden. Daher haben wir in unserem Curriculum auch ein eigenes Wahlfach, dass diese in die Tiefe behandelt. In diesem Abschnitt seien die wichtigsten Spezifikationen, die auf XML aufbauen, genannt.

2.1 Namespaces

Mit Namensräume in XML[HLTB06] können die Tagnamen von Elementen in einen bestimmten Namensraum gelegt werden. Dieses ist wichtig, wenn mit XML-Dokumenten unterschiedlicher Art gearbeitet werden soll, die aber unglücklicher Weise gleiche Tagnamen für unterschiedliche Dinge verwenden. Wenn beide einen anderen Namensraum zugeordnet sind, lassen sie sich am Namensraum unterscheiden. Die Namensräume für ein Unterdokument können in bestimmten Attributen gesetzt werden.

2.2 XML Schema

Die Struktur eines XML-Dokuments lässt sich bereits mit einer DTD beschreiben. Dort kann spezifiziert sein, welche Tagnamen es gibt, und welche Art von Kindknoten ein Element mit einem bestimmten Tagnamen haben kann. XML Schema[MTB$^+$12] geht einen Schritt weiter. Es ist eine Art Typsystem für XML-Dokumente, in dem es auch Zahlentypen gibt. Ein XML-Schema ist wiederum in XML notiert, also ein eigenes XML-Dokument.

2.3 XPath

Zur Navigation innerhalb der Achsen eines XML-Dokuments dient die Sprache:
XPath[DRS17].

Für Bäume stellen Pfade ein wichtiges Konzept dar. Entlang der Pfade navigiert man durch einen Baum. Anhand eines Pfades können Teilbäume selektiert werden. Deshalb hat das W3C eine Empfehlung heraus gegeben, mit deren Hilfe Pfade in XML-Dokumenten beschrieben werden können. Die Empfehlung für *XPath*.

2.3.1 Achsen

Eines der wichtigsten Konzepten von XPath sind die sogenannten Achsen. Sie beschreiben bestimmte Arten sich in einem Baum zu bewegen. XPath kennt 12 Achsen. Eine Achse beschreibt ausgehend von einem Baumknoten eine Liste von Knoten, die diese Achse definiert. Die am häufigste verwendete Achse ist, die Kinderachse. Sie beschreibt die Liste aller direkten Kinder eines Knotens. Entsprechend gibt es die Elternachse, die den Elternknoten beschreibt. Diese Liste hat maximal einen Knoten. Eine sehr simple Achse beschreibt den Knoten selbst. Diese Achse hat immer eine einelementige Liste als Ergebnis. Achsen für Vorfahren und Nachkommen sind der transitive Abschluss der Eltern- bzw. Kinderachse. Zusätzlich gibt es noch Geschwisterachsen, eine für die Geschwister, die vor dem aktuellen Knoten stehen und einmal für die Geschwister, die nach dem aktuellen Knoten stehen. Eine eigene Achse steht für Attribute. Schließlich gibt es noch eine Achse für Namensraumdeklarationen.

self Die einfachste Achse liefert genau den Knoten selbst. Die entsprechende Methode liefert die einelementige Liste des Eingabeknotens.

child Die gebräuchlichste Achse beschreibt die Menge aller Kinderknoten.

descendant Die Achse der Nachkommen eines Knotens beschreibt die Liste aller Kinder, Kindeskinder usw. also aller Knoten, die Unterhalb des Ausgangsknotens liegen.

descendant-or-self In der Nachkommenachse taucht der Knoten selbst nicht auf. Diese Achse fügt den Knoten selbst zusätzlich vorne ans Ergebnis an

parent Die Kinderachse lief in den Baum Richtung Blätter eine Ebene nach unten, die Elternachse läuft diese eine Ebene Richtung Wurzel nach oben. Die Ergebnisliste hat maximal ein Element, eventuell 0 Elemente, wenn wir bereits die Wurzel sind.

ancestor Die Vorfahrenachse ist der transitive Abschluß über die Elternachse, sie bezeichnet also die Eltern und deren Vorfahren. Diese Achse enthält also auf jeden Fall den Wurzelknoten. Sie enthält tatsächlich den Pfad von der Wurzel zum Ausgangsknoten, allerdings ohne den Ausgangsknoten.

ancestor-or-self Ebenso wie für die Achse aller Nachkommen ist auch für die Achse aller Vorfahren eine Version definiert, in der der Ausgangsknoten selbst noch enthalten ist.

following-sibling Auch für die Geschwister eines Knotens sind Achsen definiert. Geschwister sind Knoten, die denselben Elternknoten haben. Die *following-sibling* -Achse definiert alle Geschwister, die in der Kinderliste meines Elternknotens nach mir kommen. Sozusagen alle jüngeren Geschwister.

preceding-sibling Die *preceding-sibling* -Achse definiert alle Geschwister, die in der Kinderliste meines Elternknotens vor mir kommen. Sozusagen alle älteren Geschwister.

following Die Achse *following* bezieht sich auf die *following-sibling* -Achse. Sie bezeichnet alle *following-sibling* -Knoten und deren Nachkommen, sowie dann noch alle weiteren following-siblings der Vorfahren und wieder deren Nachkommen. Textuell bezeichnet diese Achse alle Knoten des XML-Baums, die beginnen, nachdem der Ausgangsknoten beendet ist. Daher auch der Name, alle Knoten die folgen.

Mathematisch lässt sich die Achse wie folgt beschreiben:

following(n)
$= \{f | a \in$ ancestor-or-self$(n), s \in$ following-sibling$(a), f \in$ descendant-or-self$(s)\}$

preceding Entsprechend bezeichnet die Achse *preceding* von allen Vorfahren die *preceding-sibling* -Knoten und deren Nachkommen. Dieses bedeutet, dass genau die Knoten beschrieben werden, die in der textuellen Darstellung beendet wurden, bevor der Ausgangsknoten begonnen hat..

preceding(n)
$= \{f | a \in$ ancestor-or-self$(n), s \in$ preceding-sibling$(a), f \in$ descendant-or-self$(s)\}$

Es werden also von allen Vorfahren und dem Knoten selbst die vorhergehenden Geschwister berechnet und diese und deren Nachkommen genommen.

attribute Die bisherigen Achsen selektierten Knoten aus der Baumhierarchie eines XML-Dokuments. Die letzten beiden Achsen beziehen sich auf die Attribute. Diese sind nicht teil der Kindknoten eines Elements. Daher werden über die bisherigen Achsen niemals Attribute selektiert. Hierfür gibt es die besondere Achse *attribute* , die die Liste aller Attribute eines Knoten selektiert.

2.3.2 Knotentest

Die Kernausdrücke in XPath sind von der Form:

axis::*nodeTest*

axis beschreibt dabei eine der 12 Achsen. *nodeTest* ermöglicht es, aus den durch die Achse beschriebenen Knoten bestimmte Knoten zu selektieren. Syntaktisch gibt es 8 Arten des Knotentests:

- `*`: beschreibt Elementknoten mit beliebigen Namen.
- *pref*`:*`: beschreibt Elementknoten mit dem Prefix *pref* und beliebigen weiteren Namen.
- *pref*:*name*: beschreibt Elementknoten mit dem Prefix *pref* und den weiteren Namen *name*. Der Teil vor den Namen kann dabei auch fehlen.
- `comment()`: beschreibt Kommentarknoten.
- `text()`: beschreibt Textknoten.
- `processing-instruction()`: beschreibt Processing-Instruction-Knoten.
- `processing-instruction(`*target*`)`: beschreibt Processing-Instruction-Knoten mit einem bestimmten Ziel.
- `node()`: beschreibt beliebige Knoten.

Ein XPath-Ausdruck bezieht sich immer auf einen Ausgabgsknoten im Dokument. Die Bedeutung eines XPath-Ausdrucks

axisType`::`*nodeTest*

ist dann:

Selektiere vom Ausgangsknoten alle Knoten mit der durch *axisType* angegebenen Achse. Von dieser Liste selektiere dann alle Knoten, für die der Knotentest erfolgreich ist.

Beispiel 2.15 Der folgende Ausdruck selektiert alle Nachkommen des Ausgangsknoten, die den Elementknoten mit den Tagnamen `code` sind:
`descendant-or-self::code`

Dieses sind in dem XML-Dokument, das dieses Skript beschreibt gerade alle Teile, in denen Java Quelltext steht.

Mehrere XPath-Ausdrücke können laut der XPath Definition nacheinander ausgeführt werden. Hierzu schreibt man zwischen den einzelnen XPath-Audrücken einen Schrägstrich. Der so zusammengesetzte Ausdruck selektiert wieder eine Liste von Knoten ausgehend von einem Ausgangsknoten. Die Bedeutung von einer solchen Verkettung von XPath-Ausdrücken ist:

Selektiere erst mit dem ersten XPath-Teilausdruck ausgehend von einem Startknoten die Liste der Ergebnisknoten. Dann nehme jeden der Knoten in dieser Ergebnisliste als neuen Ausgangsknoten für den zweiten XPath-Ausdruck und vereinige deren Ergebnissse.

Beispiel 2.16 Für das Beispiel des XML-Dokuments dieses Skriptes bezeichnet der Ausdruck

`child::kapitel/child::section/descendant::code/child:text()`

Den Text, der direkt unterhalb eines `code` Elements in steht, wobei das `code` Element Nachfahre eines `section` Elements ist, welches direktes Kind eines `kapitel` Elements ist, welches wiederum ein Kind des Ausgangsknotens ist.

2.3.3 Abkürzende Schreibweise

In obiger Einführung haben wir bereits gesehen, dass XPath den aus dem Dateissystem bekannten Schrägstrich für Pfadangaben benutzt. Betrachten wir XPath-Ausdrücke als Terme, so stellt der Schrägstrich einen Operator dar. Diesen Operator gibt es sowohl einstellig wie auch zweistellig.

Die Grundsyntax in XPath sind eine durch Schrägstrich getrennte Folge von Kernausdrücke mit Achsentyp und Knotentest.

Beispiel 2.17 Der Ausdruck

`child::skript/child::*/descendant::node()/self::code/attribute::class`

beschreibt die Attribute mit Attributnamen `class`, die an einem Elementknoten mit Tagnamen `code` hängen, die Nachkommen eines beliebigen Elementknotens sind, die Kind eines Elementknotens mit Tagnamen `skript` sind, die Kind des aktuellen Knotens, auf dem der Ausdruck angewendet werden soll sind.

Vorwärts gelesen ist dieser Ausdruck eine Selektionsanweisung:
Nehme alle `skript`-Kinder des aktuellen Knotens. Nehme von diesen beliebige Kinder. Nehme von diesen alle `code`-Nachkommen. Und nehme von diesen jeweils alle `class`-Attribute.

Der einstellige Schrägstrichoperator bezieht sich auf die Dokumentwurzel des aktuellen Knotens.

Der doppelte Schrägstrich XPath kennt einen zweiten Pfadoperator `//`. Auch er existiert jeweils einmal einstellig und einmal zweistellig. Der doppelte Schrägstrich ist eine abkürzende Schreibweise, die übersetzt werden kann in Pfade mit einfachen Schrägstrichoperator.

- `//`*expr*: Betrachte beliebige Knoten unterhalt des Dokumentknotens, die durch *expr* charakterisiert werden.
- $e_1//e_2$: Betrachte beliebiege Knoten unterhalb der durch e_1 charkterisierten Knoten.und prüfe diese auf e_2.

Der Doppelschrägstrich ist eine abkürzende Schreibweise für: `/descendant-or-self::node()/`

Der Punkt Für die Selbstachse kann als abkürzende Schreibweise ein einfacher Punkt `.` gewählt werden, wie er aus den Pfadangaben im Dateisystem bekannt ist.

Der Punkt `.` ist die abkürzende Schreibweise für: `self::node()`.

Der doppelte Punkt Für die Elternachse kann als abkürzende Schreibweise ein doppelter Punkt `..` gewählt werden, wie er aus den Pfadangaben im Dateisystem bekannt ist.

Der Punkt `..` ist die abkürzende Schreibweise für: `parent::node()`.

Vereinfachte Kindachse Ein Kernausdruck, der Form `child::`*nodeTest* kann abgekürzt werden durch *nodeTest*. Die Kinderachse ist also der Standardfall.

Vereinfachte Attributachse Auch für die Attributachse gibt es eine abkürzende Schreibweise: ein Kernausdruck, der Form `attribute::`*pre*`:`*name* kann abgekürzt werden durch `@`*pre*`:`*name*.

Beispiel 2.18 Insgesamt lässt sich der obige Ausdruck abkürzen zu: `skript/*//code/@class`

2.4 XSL

Ein Sprache zum Transformieren von XML Dokumenten ist XSL[TWZ+10].

XML-Dokumente enthalten keinerlei Information darüber, wie sie visualisiert werden sollen. Hierzu kann man getrennt von seinem XML-Dokument ein sogenanntes *Stylesheet* schreiben. XSL ist eine Sprache zum Schreiben von Stylesheets für XML-Dokumente. XSL ist in gewisser Weise eine Programmiersprache, deren Programme eine ganz bestimmte Aufgabe haben: XML Dokumente in andere XML-Dokumente zu transformieren. Die häufigste Anwendung von XSL dürfte sein, XML-Dokumente in HTML-Dokumente umzuwandeln.

Um in einem Dokument Teildokumente zu zu selektieren verwendet xsl die XPath-Ausdrücke.

Wir werden die wichtigsten XSLT-Konstrukte mit folgendem kleinem XML Dokument ausprobieren:

cds.xml

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<?xml-stylesheet type="text/xsl" href="cdTable.xslt"?>
<cds>
 <cd>
  <artist>The Beatles</artist>
  <title>White Album</title>
  <label>Apple</label>
 </cd>
 <cd>
  <artist>The Beatles</artist>
  <title>Rubber Soul</title>
  <label>Parlophone</label>
 </cd>
 <cd>
  <artist>Duran Duran</artist>
  <title>Rio</title>
  <label>Tritec</label>
 </cd>
 <cd>
  <artist>Depeche Mode</artist>
  <title>Construction Time Again</title>
  <label>Mute</label>
 </cd>
</cds>
```

XSL Dateien XSLT-Skripte sind syntaktisch auch wieder XML-Dokumente. Ein XSLT-Skript hat feste Tagnamen, die eine Bedeutung für den XSLT-Prozessor haben. Diese Tagnamen haben einen festen definierten Namensraum. Das äußerste Element eines XSLT-Skripts hat den Tagnamen `stylesheet`. Damit hat ein XSLT-Skript einen Rahmen der folgenden Form:

trans1.xsl

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<xsl:stylesheet
  version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
</xsl:stylesheet>
```

Wir können mit einer *Processing-Instruction* am Anfang eines XML-Dokumentes definieren, mit welchem XSLT Stylesheet es zu bearbeiten ist. Hierzu wird als Referenz im Attribut `href` die XSLT-Datei angegeben.

trans2.xsl

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<?xml-stylesheet type="text/xsl" href="cdTable.xslt"?>
<cds>
 <cd>........
```

Vorlagen Das wichtigste Element in einem XSLT-Skript ist das Element `xsl:template`. In ihm wird definiert, wie ein bestimmtes Element transformiert werden soll. Es hat schematisch folgende Form:

`<xsl:template match="`*xpath*`">`
zu erzeugender Code
`</xsl:template>`

Das XSLT-Skript aus Abbildung 1 transformiert die XML-Datei mit den CDs in eine HTML-Tabelle, in der die CDs tabellarisch aufgelistet sind.

Öffnen wir nun die XML Datei, die unsere CD-Liste enthält im Webbrowser, so wendet er die Regeln des referenzierten XSLT-Skriptes an und zeigt die so generierte Webseite als Tabelle an.

Läßt man sich vom Browser hingegen den Quelltest der Seite anzeigen, so wird kein HTML-Code angezeigt sondern der XML-Code.

2.5 XQuery

Eine weitere speziell für XML entwickelte Programmiersprache ist XQuery[CDSR14]. Ziel dieser Sprache ist es als Anfragesprache für Dokumentendatenbanken zu dienen. Kern sind eine

cdTable.xsl

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<xsl:stylesheet
  version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<!-- Startregel für das ganze Dokument. -->
<xsl:template match="/">
  <html><head><title>CD Tabelle</title></head>
    <body>
      <xsl:apply-templates/>
    </body>
  </html>
</xsl:template>

<!-- Regel für die CD-Liste. -->
<xsl:template match="cds">
  <table border="1">
    <tr><td><b>Interpret</b></td><td><b>Titel</b></td></tr>
    <xsl:apply-templates/>
  </table>
</xsl:template>

<xsl:template match="cd">
  <tr><xsl:apply-templates/></tr>
</xsl:template>

<xsl:template match="artist">
  <td><xsl:apply-templates/></td>
</xsl:template>

<xsl:template match="title">
  <td><xsl:apply-templates/></td>
</xsl:template>

<!-- Regel für alle übrigen Elemente.
     Mit diesen soll nichts gemacht werden -->
<xsl:template match="*">
</xsl:template>

</xsl:stylesheet>
```

Abbildung 1: Beispiel XSL-Skript.

Art select-Anweisungen, die sogenannten flower-expressions, die mit den Schlüsselwörtern `for`, `let`, `where` und `result` gebildet werden, daher *flower*-expression.[2]

Auch XQuery verwendet zum selektieren von Teildokumenten XPath-Ausdrücke.

Wir werden in Rahmen dieses Lehrbriefs nicht weiter auf XQuery eingehen.

2.6 SVG

Ein XML-Format zur Beschreibung von skalierbaren Vektorgrafiken ist SVG[BWS+18].

Wir geben ein kleines Beispiel in Abbildung 2.

2.7 XHTML

XHTML[McC18] ist eine XML konforme Version von HTML 4. Leider hat sich die Idee, auch HTML streng als hierarchische XML konforme Dokumentstruktur zu behandeln nicht durchgesetzt. Für HTML 5 gibt es keine XHTML-Version.

2.8 Webservices und SOAP

SOAP[Ved07] ist ein Standard zur Repräsentation der Daten bei Webdiensten. SOAP Nachrichten sind als XML Dokumente formuliert.

3 XML-Apis in Java

Betrachten wir nun, wie man in Java mit XML Dokumenten arbeiten kann. Hierzu werden wir eine Reihe unterschiedlicher Standardbibliohteken nutzen:

XML.java

```
package name.panitz.xml;
import java.util.*;
import java.util.concurrent.ConcurrentHashMap;

import java.io.*;
import org.w3c.dom.*;
import javax.xml.parsers.*;
import org.xml.sax.*;
import org.xml.sax.helpers.*;
import javax.xml.stream.*;
import javax.xml.stream.events.*;
import javax.xml.xpath.*;
import javax.xml.transform.*;
```

[2] Hätte man statt des Schlüsselworts `for` sich für das Schlüsselwort `select` entschieden, dann wären es *slower*-expression, was keine gute Werbung gewesen wär.

ubahn.xml

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<svg
    xmlns:dc="http://purl.org/dc/elements/1.1/"
    xmlns:cc="http://creativecommons.org/ns#"
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:svg="http://www.w3.org/2000/svg"
    xmlns="http://www.w3.org/2000/svg"
    version="1.1"
    viewBox="0 0 100 110" >
  <g>
    <path style="stroke:#0000f9;stroke-width:3" d="M 5,100 80,100" />
    <g transform="translate(5,100)">
      <circle r="3"/>
      <text
        style="font-size:7px;line-height:105%;font-family:Sans;"
        transform="translate(3,-5),rotate(315 0 0)"
        text-anchor="begin">Preungesheim</text></g>
    <g  transform="translate(43,100)">
      <circle r="3"/>
      <text
        style="font-size:7px;line-height:105%;font-family:Sans;"
        transform="translate(3,-5),rotate(315 0 0)"
        text-anchor="begin">Sigmund-Freud-Straße</text></g>
    <g transform="translate(81,100)">
      <circle r="3"/>
      <text
        style="font-size:7px;line-height:105%;font-family:Sans;"
        transform="translate(3,-5),rotate(315 0 0)"
        text-anchor="begin"
      >Ronneburgstraße</text>
    </g>
  </g>
</svg>
```

Abbildung 2: Beispiel für eine SVG Graphik.

```
import javax.xml.transform.stream.*;
```

Die Funktionalität wird innerhalb einer Schnittstelle `XML` entwickelt, die gleichzeitig auch für unsere erste eigene XML-API Implementierung als Knotentyp dient.

XML.java

```
public interface XML{
```

3.1 Eigenes XML-API

Bevor wir uns Stadardbibliohteken zur Verarbeitung von XML-Dokumenten widmen, soll ein eigenes kleines XML-Api in Java entwickelt werden.

Dieses soll die Baumstruktur eines XML-Dokuments darstellen. Gemeinsame Schnittstelle ist die Schnittstelle `XML`.

XML.java

```
  XML setParent(XML parent);
  XML parentNode();
```

Wir kennen zwei Knotentypen in der Baumstruktur: Elementknoten und Textknoten. Textknoten bestehen nur aus einen String und folgende kleine Record-Klasse kann diese darstellen.

XML.java

```
  public static record Text(String text,XML parentNode) implements XML{
    public XML setParent(XML parent){return new Text(text,parent);}

    @Override public String toString(){
      return "Text["+text+"]";
    }
  }
```

Elementknoten haben einen Tagnamen, eine Liste von Kindknoten und eine Abbildung in Sinne von Schlüssel-Wert-Paaren für die Attribute.

XML.java

```
public static record
Element(String name,List<XML> childNodes,Map<String,String> attributes,XML parentNode)
        implements XML{
  public XML setParent(XML parent){
    var newChildNodes = new ArrayList<XML>();
    var r =  new Element(name,newChildNodes,attributes,parent);
    for (var cn:childNodes) newChildNodes.add(cn.setParent(r));
    return r;
  }
  @Override public String toString(){
    return "Element["+name+","+childNodes+","+attributes+"]";
  }
}
```

Beispiel 3.19 Wir haben im letzten Lehrbrief bereits umfangreiche Erfahrungen mit Bäumen gemacht und können für unsere XML-Struktur schnell einfache Algorithmen schreiben.

Als ersten kleines Beispiel laufen wir einmal durch das ganze Dokument und zählen die Knoten.

XML.java

```
default int count(){
  if (this instanceof Text t) return 1;
  else if (this instanceof Element e)
    return 1+e.childNodes().parallelStream()
       .map(x->x.count()).reduce(0,(r,x)->r+x);
  throw new RuntimeException("unmatched pattern: "+this);
}
```

Beispiel 3.20 Um unsere Dokumente wieder als String zu serialisieren, sei eine Methode definiert, die die serialisierte Form des Baums in ein Writer-Objekt schreibt.

XML.java

```
default void write(Writer out) throws IOException{
  if (this instanceof Text t) out.write(t.text);
  else if (this instanceof Element e){
    out.write("<");
    out.write(e.name());
    for (var atr:e.attributes().entrySet()){
      out.write(" ");
      out.write(atr.getKey());
```

```
        out.write("=\"");
        out.write(atr.getValue());
        out.write("\"");
      };
      out.write(">");
      for (var c:e.childNodes())c.write(out);
      out.write("</");
      out.write(e.name());
      out.write(">");
    }
  }
```

mit Hilfe eines `StringWriter` können wir nun ein XML-Baum als String darstellen:

XML.java

```
default String show() {
  try{
    var out = new StringWriter();
    write(out);
    return out.toString();
  }catch(IOException e){ return "";}
}
```

3.2 DOM

DOM (`document object model`) ist ein vom W3C veröffentlichtes API zur Bearbeitung von XML und HTML Dokumenten[BCH+04]. Es repräsentiert die Baumstruktur der Dokumente. Die Geschichte von DOM liegt in den frühen Tagen des Internets verwurzelt, als man anfing mit Hilfe von Javascript im Browser clientseitig Html-Dokumente dynamisch zu erzeugen und zu verändern. Dahei resultierte der Bedarf, nach einem einheitlichen API zur Bearbeitung der Dokumente. Mit der Definition von XML wurde DOM auch für XML-Dokumente definiert.

Das DOM-Api ist programmiersprachlich unabhängig definiert. Somit verwendet es keine auch noch so schönen speziellen Konstrukte von Programmiersprachen. Für Java heißt das, dass es keine Java-Standardlisten, keine Iterable-Objekte, keine forEach-Schleife und schon gar keine Strom-Objekte für die Bearbeitung von XML-Dokumenten im Java DOM-Api gibt.

Das DOM-API ist in der Standard-Javaentwicklungsumgebung implementiert. Hierzu findet sich im API das Paket `org.w3c.dom`. Es fällt auf, dass dieses Paket nur Schnittstellen und keine Klassen enthält. Der Grund hierfür ist, dass es sich bei DOM um ein implementierungsunabhängiges API handelt. So soll der Programmierer auch gar keine Klassen kennenlernen, die das API implementieren.

Die zentrale Schnittstelle in *dom* ist `Node`. Sie hat als Unterschnittstellen alle Knotentypen, die es in XML gibt. Folgende Graphik gibt über diese Knotentypen einen Überblick.

```
interface org.w3c.dom.Node
 |
 |--interface org.w3c.dom.Attr
 |--interface org.w3c.dom.CharacterData
 |   |
 |   |--interface org.w3c.dom.Comment
 |   |--interface org.w3c.dom.Text
 |       |
 |       |--interface org.w3c.dom.CDATASection
 |
 |--interface org.w3c.dom.Document
 |--interface org.w3c.dom.DocumentFragment
 |--interface org.w3c.dom.DocumentType
 |--interface org.w3c.dom.Element
 |--interface org.w3c.dom.Entity
 |--interface org.w3c.dom.EntityReference
 |--interface org.w3c.dom.Notation
 |--interface org.w3c.dom.ProcessingInstruction
```

Wie man sieht, findet sich die Struktur, die wir in unserem eigenen Versuch, ein XML-Baum-API zu entwickeln, entworfen haben, wieder. Die allgemeine Schnittstelle `Node` und deren beiden Unterschnittstellen `Element` und `Text`.

Eine der entscheidenen Methoden der Schnittstelle `Node` selektiert die Liste der Kinder eines Knotens:

```
NodeList getChildNodes()
```

Knoten, die keine Kinder haben können (Textknoten, Attribute etc.) geben bei dieser Methode die leere Liste zurück. Attribute zählen auch wie in unserer Modellierung nicht zu den Kindern eines Knotens. Um an die Attribute zu gelangen, gibt es eine eigene Methode:

```
NamedNodeMap getAttributes()
```

Wie man sieht, benutzt Javas *dom* Umsetzung keine von Javas Listenklassen zur Umsetzung einer Knotenliste, sondern nur genau die in *DOM* spezifizierte Schnittstelle `NodeList`. Eine `NodeList` hat genau zwei Methoden:

```
int getLength()
Node item(int index)
```

Mit diesen beiden Methoden lässt sich über die Elemente einer Liste über den Indexzugriff iterieren. Die Schnittstelle `NodeList` implementiert nicht die Schnittstelle `Iterable`. Dieses ist insofern schade, da somit nicht die neue *for-each*-Schleife aus Java für die Knotenliste des *DOM* benutzt werden kann.

Die Schnittstelle `Node` enthält ebenso die anderen Methoden, die wir uns vorher in der eigenen Umsetzung überlegt haben:

```
String getNodeName();
String getNodeValue();
short getNodeType();
```

Die Methode `getNodeType()` gibt keine Konstante einer Aufzählung zurück, sondern eine Zahl des Typs `short`, die den Knotentyp kodiert. In der Schnittstelle `Node` sind Konstanten für die einzelnen Knotentypen definiert. Die anderen beiden Methoden verhalten sich so, wie wir es in unserer Minimalumsetzung auch implementiert haben.

Die Schnittstelle `Node` des DOM-APIs ist wesentlich umfangreicher als die kleine Schnittstelle `Node`, die wir uns zum Einstieg überlegt haben. Insbesondere gibt es eine Methode, mit der von einem Knoten auch wieder hoch zu seinem Elternknoten navigiert werden kann.

```
Node getParentNode()
```

Sofern ein Knoten einen Elternknoten besitzt, kann dieser mit dieser Methode erfragt werden. Ansonsten ist das Ergebnis `null`.

3.2.1 Parsen eines XML-Dokuments

Wir benötigen einen Parser, der uns die Baumstruktur eines XML-Dokuments erzeugt. In der Javabibliothek ist ein solcher Parser integriert, allerdings nur über seine Schnittstellenbeschreibung. Im Paket `javax.xml.parsers` gibt es nur Schnittstellen. Um einen konkreten Parser zu erlangen, bedient man sich einer Fabrikmethode: In der Schnittstelle `DocumentBuilderFactory` gibt es eine statische Methode `newInstance` und über das `DocumentBuilderFactory`-Objekt, lässt sich mit der Methode `newDocumentBuilder` ein Parser erzeugen.

Wir können so eine statischen Methode zum Parsen eines XML-Dokuments schreiben:

XML.java

```
public static Document parseXml(String xmlFileName){
  try{
   return
    DocumentBuilderFactory
     .newInstance()
     .newDocumentBuilder()
     .parse(new File(xmlFileName));
  }catch(Exception e){return null;}
}
```

Wir können jetzt z.B. den Quelltext dieses Lehrbriefs parsen.

Shell

```
jshell> import static name.panitz.xml.XML.*

jshell> parseXml("master.xml")
$2 ==> [#document: null]

jshell> parseXml("master.xml").getChildNodes().item(0)
$3 ==> [exercise: null]
```

Das Ergebnis ist ein Knoten vom Typ `Document`. Dieser repräsentiert in DOM das ganze Dokument. Das eigentliche Wurzeldokument ist ein Kind des Knotens `Document`.

Wie man sieht ist die Methode `toString` in der implementierenden Klasse der Schnittstelle `Document` und `Element`, die unser Parser benutzt, nicht sehr aufschlußreich. Es wird nicht das Dokument in seiner inhaltlichen Form dargestellt, sondern nur der Knotennamen und Knotenwert, der hier `null` ist.

Beispiel 3.21 Als ersten kleines Beispiel für das DOM-Api laufen wir einmal durch das ganze Dokument und zählen die Knoten.

XML.java

```
static int count(Node n){
  var result = 1;
  var cs = n.getChildNodes();
  for (var i=0;i<cs.getLength();i++){
    result+=count(cs.item(i));
  }
  return result;
}
```

Diese Methode lässt sich auf das Dokument dieses Lehrbriefs anwenden.

Shell

```
jshell> count(parseXml("master.xml"))
$4 ==> 180
```

Beispiel 3.22 In diesem zweiten Beispiel sollen alle Tagnamen eines Dokuments in einer Menge gesammelt werden.

Dazu wird die Menge erzeugt und der eigentlichen Methode übergeben.

XML.java

```
static Set<String> tagNames(Node n){
  var result = new TreeSet<String>();
  tagNames(n,result);
  return result;
}
```

Es muss zunächst abgefragt werden, ob es sich bei dem Knoten um einen Elementknoten hanelt. Nur für diese ist der Knotenname ain Tagname.

XML.java

```
static void tagNames(Node n,Set<String> result){
  if (n.getNodeType()==Node.ELEMENT_NODE) result.add(n.getNodeName());
  var cs = n.getChildNodes();
  for (var i=0;i<cs.getLength();i++){
    tagNames(cs.item(i),result);
  }
}
```

So können wir alle im Quelltext dieses Lehrbriefs verwendeten Tagnamen sammeln.

Shell

```
jshell> tagNames(parseXml("master.xml"))
$5 ==> [both, class, code, div, exercise, extension, java, lang, name, p, shell, solution, solutio
```

Beispiel 3.23 Wir kennen jetzt schon zwei Baum-APIs für XML. Unser eigenes API und das DOM-Api. In diesem Beispiel soll ein DOM-Dokument in unserer Struktur dargestellt werden.

XML.java

```
static XML toMyXML(Node n){
  if (n.getNodeType()==Node.ELEMENT_NODE){
    var ats = new TreeMap<String,String>();
    var as = n.getAttributes();
    for (var i=0;i<as.getLength();i++){
      ats.put(as.item(i).getNodeName(),as.item(i).getNodeValue());
    }
    var children = new ArrayList<XML>();
```

```
      var r = new Element(n.getNodeName(),children,ats,null);
      var cs = n.getChildNodes();
      for (var i=0;i<cs.getLength();i++){
        var cld = toMyXML(cs.item(i));
        if (cld!=null)
          children.add(cld.setParent(r));
      }
      return r;
    }else if (n.getNodeType()==Node.TEXT_NODE
            ||n.getNodeType()==Node.CDATA_SECTION_NODE){
      var t = n.getNodeValue().trim();
      return t.isEmpty()?null:new Text(n.getNodeValue(),null);
    }
    if (n.getNodeType()==Node.DOCUMENT_NODE){
      return toMyXML(n.getChildNodes().item(0));
    }
    throw new RuntimeException("unsuported node type: "+n);
  }
```

So können wir jetzt mit Hilfe dieser Transformation auch ein DOM-Objekt als String Serialisieren.

Shell

```
jshell> System.out.println(toMyXML(parseXml("master.xml")).show())
<exercise>
  <lang>Java</lang>
  <extension>java</extension>
  <name>XML</name>
  <text>
    <div>
      <p>Studieren Sie den zur Aufgabe geh?renden Lehrbrief und l?sen Sie die Aufgaben darin. Testen Sie ihre L
```

3.3 SAX

Oft brauchen wir nie das komplette XML-Dokument als Baum im Speicher. Eine Großzahl der Anwendungen auf XML-Dokumenten geht einmal das Dokument durch, um irgendwelche Informationen darin zu finden, oder ein Ergebnis zu erzeugen. Hierzu reicht es aus, immer nur einen kleinen Teil des Dokuments zu betrachten. Und tatsächlich hätte diese Vorgehensweise, bai allen bisher geschriebenen Programmen gereicht. Wir sind nie im Baum hin und her gegangen. Wir sind nie von einem Knoten zu seinem Elternknoten oder seinen vor ihm liegenden Geschwistern gegangen.

Ausgehend von dieser Beobachtuung hat eine Gruppe von Programmierern ein API zur Bearbeitungen von XML-Dokumenten vorgeschlagen, das nie das gesammte Dokument im Speicher zu

halten braucht. Dieses API heißt *SAX*, für *simple api for xml processing*. *SAX* ist keine Empfehlung des W3C. Es ist außerhalb des W3C entstanden.

Die Idee von *SAX* ist ungefähr die, dass uns jemand das Dokument vorliest, einmal von Anfang bis Ende. Wir können dann auf das gehörte reagieren. Hierzu ist für einen Parse mit einem SAX-Parser stets mit anzugeben, wie auf das Vorgelesene reagiert werden soll. Dieses ist ein Objekt der Klasse `DefaultHandler`. In einem solchen *handler* sind Methoden auszuprogrammieren, in denen spezifiziert ist, was gemacht werden soll, wenn ein Elementstarttag, Elementendtag, Textknoten etc. vorgelsen wird. Man spricht bei einem SAX-Parser von einem ereignisbasierten Parser. Wir reagieren auf bestimmte Ereignisse des Parses, nämlich dem Starten/Enden von Elementen und so weiter.

Auch ein SAX-Parser liegt in Java nur als Schnittstelle vor und kann nur über eine statische Fabrikmethode instanziiert werden.

XML.java

```
public static void parse(File file,DefaultHandler handler)
                              throws Exception{
  SAXParserFactory.newInstance()
                  .newSAXParser()
                  .parse(file,handler);
}
```

XML.java

```
public static void parse(Reader reader,DefaultHandler handler)
                              throws Exception{
  SAXParserFactory.newInstance()
                  .newSAXParser()
                  .parse(new InputSource(reader) ,handler);
}
```

Beispiel 3.24 Als erstes Beispiel wollen wir unser altbekanntes Zählen der Knoten programmieren. Hierzu ist ein eigener `DefaultHandler` zu schreiben, der, sobald beim Vorlesen ihm der Beginn eines Elements gemeldet wird, darauf reagiert, indem er seinen Zähler um eins weiterzählt. Wir überschreiben demnach genau eine Methode aus dem `DefaultHandler`, nämlich die Methode `startElement`.

XML.java

```
public static class Count extends DefaultHandler {
  public int result = 0;

  @Override
  public void startElement(String uri, String localName, String qName,
```

```
      Attributes attributes) throws SAXException {
    result++;
  }
}
```

Shell

```
jshell> var counter = new Count()
counter ==> name.panitz.xml.XML$Count@2e0fa5d3

jshell> parse(new File("master.xml"),counter)

jshell> counter.result
$5 ==> 57
```

Beispiel 3.25 Das zweite kleine SAX-Beispiel sammelt alle Tagnamen in einer Menge auf:

XML.java

```
public static class GetTagNames extends DefaultHandler {
  public Set<String> result = new TreeSet<>();

  @Override
  public void startElement(String uri, String localName, String qName,
      Attributes attributes) throws SAXException {
    result.add(qName);
  }
}
```

Shell

```
jshell> var  tagCollector = new GetTagNames()
tagCollector ==> name.panitz.xml.XML$GetTagNames@5010be6

jshell> parse(new File("master.xml"),tagCollector)

jshell> tagCollector.result
$5 ==> [both, class, code, div, exercise, extension, java, lang, name, p, solution, solutionFQCN, testclasses,
```

Beispiel 3.26 Und schließlich ein etwas komplexeres Beispiel.

XML.java

```
public static class GetProgramCode extends DefaultHandler {
  public StringBuffer result = new StringBuffer();
  boolean isInCodeElement = false;
```

```
  @Override public void startElement
      (String uri, String localName, String qName,Attributes attributes)
      throws SAXException {
    if (qName.equals("code")) {
      isInCodeElement = true;
    }
  }
  @Override public void characters(char[] ch, int start, int length)
      throws SAXException {
    String text = new String(ch, start, length);
    if (isInCodeElement)
      result.append(text);
  }
  @Override
  public void endElement(String uri, String localName, String qName)
      throws SAXException {
    if (qName.equals("code")) {
      isInCodeElement = false;
    }
  }
}
```

Shell

```
jshell> var code = new GetProgramCode()
code ==> name.panitz.xml.XML$GetProgramCode@2e0fa5d3

jshell> parse(new File("master.xml"),code)

jshell> code.result
$5 ==> The Beatles White Album<cd>
  <artist>The Beatles</artist>
  <title>White Album</title>
</cd><?xml version="1.1"?><?xml version="1.0"?>
<myText>Dieses ist der Text des Dokuments. Er ist
mit genau einem Element markiert.
</myText><?xml version="1.0"?>
<skript>...
```

3.4 StAX

Wir haben bisher drei methodisch sehr unterschiedliche Herangehensweisen für die Programmierung von XML Dokumenten kennen gelernt:

- unser eigenes in Java implementiertes XML Api.
- das DOM-API als ein Baum-API.
- das SAX-API als ein Ereignis-basiertes API.

Als Abschluss betrachten wir noch eine vierte methodische Herangehensweise. Dabei schließen wir wieder den Kreis zum ersten Kapitel, nämlich zu den Iteratoren. Ein weiteres API zur XML

Verarbeitung in Java ist das strombasierte API StAX. Dahinter verbirgt sich wie bei SAX, die Verarbeitung des Dokuments sequentiell in seiner textuellen Form. Damit steht wieder nicht die reine Baumstruktur direkt im Speicher zur Verfügung. Die sequentielle Verarbeitung wird durch einen Iterator über die verschiedenen XML-Ereignisse bewerkstelligt.

In Java finden sich die entsprechenden Definitionen als Schnittstellen im Paket `javax.xml.stream`.

Die entscheidenen beiden Schnittstellen sind dabei:

- `XMLEventReader`: diese Schnittstelle beschreibt einen Iterator der beim Lesen eines XML-Dokuments über die einzelnen XML-Komponenten des Dokuments iteriert. Leider berücksichtigt diese Schnittstelle nicht die generischen Typen von Java, so dass die Methode `next()` als Rückgabetyp lediglich `Object` hat.
- `javax.xml.stream.events.XMLEvent`: diese Schnittstelle beschreibt allgemein ein XML-Konstrukt, auf dem beim Lesen gestoßen wird. Für die verschiedenen XML-Anteile gibt es Unterschnittstellen dieser Schnittstelle wie z.B. `StartElement`, `EndElement` oder `Characters`. In der Schnittstelle `XMLEvent` sind boolsche Methoden definiert, die testen, ob ein Objekt von einer dieser Unterschnittstellen ist. Es gibt z.B. die Methode `boolean isAttribute();`.

Die Anwendung des StAX-APIs geht sehr analog, zu der Anwendung der anderen beiden XML APIs.

Ein StAX basierter Parser wird durch eine Factory instanziiert:

XML.java

```
static XMLEventReader staxReader(String fileName) throws XMLStreamException, IOException{
  return XMLInputFactory
      .newInstance()
      .createXMLEventReader(new FileReader(fileName));
}
```

Das so erhaltene `XMLEventReader` Objekt kann zum Iterieren durch das Dokument benutzt werden. Leider kennt das StAX-API nicht die Schnittstelle `Iterable`, so dass nicht direkt die for-each Schleife zum Iterieren angewendet werden kann. Wir können uns da mit der Erzeugung eines iterierbaren Objektes durch einen Lambda-Ausdruck behelfen.

Beispiel 3.27 Auch hier sei das erste kleine Beispiel das durchzählen. Allerdings interessiert uns hier nicht die gesamt knotenanzahl sondern die Anzahl der Elementknoten.

XML.java

```
static int count(XMLEventReader reader){
  var result = 0;
  Iterable<Object> it = () -> reader;
  for (Object o : it){
    var ev = (XMLEvent) o;
```

```
      if (ev.isStartElement()) result++;
    }
    return result;
  }
```

Und der entsprechende Testaufruf.

Shell

```
jshell> count(staxReader("master.xml"))
$2 ==> 77
```

3.5 XPath in Java

Im Java-API ist eine XPath-Implementierung enthalten. Die dazugehörigen Klassen befinden sich im Paket `javax.xml.xpath`. Die Implementierung basiert auf DOM, es werden also DOM-Knoten als Eingabe und genommen und auch u.a. eine DOM-Nodelist als Ergebnis berechnet.

Im Folgendem ein Beispiel, wie ein XPath-Ausdruck, der eine Knotenliste als Ergebnis hat, mit dem Java-API für XPath angewendet wird. Wir schreiben eine statische Methode, die einen XPath-Ausdruck als String übergeben bekommt und das Eingabedokument für diesen Ausdruck als einen Reader.

XML.java

```
public static NodeList evalXPath(String xpathExpression, Reader reader)
    throws XPathExpressionException {
```

Zunächst einmal benutzt das XPath-API wieder das Factory-Pattern, so dass ein XPath-Ausdruck nicht direkt durch einem Konstruktoraufruf erzeugt wird, sondern durch eine Methode in einer sogenannten Factory-Klasse, ebenso wie auch der DOM-Parser erzeut wurde.

XML.java

```
  var xpath = XPathFactory.newInstance().newXPath();
```

Die Schnittstelle `XPath` kennt eine Methode `evaluate`, die drei Parameter hat:

- der auszuwertende XPath-Ausdruck als String.
- die Eingabequelle für das XML-Dokuement, auf dem dieser Ausdruck ausgewertet werden soll.
- eine Konstante, die das erwartete Ergebnis anzeigt. Für die meisten XPath-Ausdrücke wird ein Objekt des Typs `Nodelist` als Ergebnis erwartet. Es gibt aber auch XPath-Ausdrücke, die nur Strings, oder Zahlen als Ergebnis haben.

Das Ergebnis der Methode `evaluate` ist nur vom Typ `Object` und muss entsprechend erst noch mit einer Typzusicherung auf den eigentlichen Ergebnistypen überprüft werden.

In unserem Beispiel gehen wir davon aus, dass das Ergebnis eine `Nodelist` ist. Falls dieses nicht der Fall ist, wird eine Ausnahme geworfen.

XML.java

```
 var ns = (NodeList) xpath.evaluate(xpathExpression,
      new InputSource(reader), XPathConstants.NODESET);
  return ns;
}
```

Abschließend ein kleiner Testaufruf dieser Methode, der aus der XML-Datei, die diesem Skript zugrunde liegt, alle code-Elemente selektiert:

3.6 XSL in Java

Ebenso wie für XPath gibt es auch eine Implementierung von XSLT in Java. Die entsprechenden Klassen befinden sich im Paket `javax.xml.transform`. Auch hier wird wieder das Factory-Pattern angewendet. Wir geben im Folgendem eine kleine Beispielanwendung, in der ein XSLT-Skript auf ein XML-Dokument angewendet wird.

Interessant ist, dass es auch möglich ist eine Instanz vom Typ `Transformer` ohne Angabe eines XSL-Skripts als Quelle zu erzeugen. Dieses entspricht dann der Identität, dass heißt, das transformierte Ausgabeausgabedokument ist gleich dem zu transformierenden Eingabedokument. Damit kann dieser Identittäts-Transformer dazu genutzt werden, um einen DOM-Baum wieder als textuelle Datei zu speichern.

XML.java

```
static public String transform(File xslt,File doc){
  return transform(new StreamSource(doc),new StreamSource(doc));
}

static public String transform(Source xslt,Source doc){
  try{
    StringWriter writer = new StringWriter();
    Transformer t =
       TransformerFactory
        .newInstance()
        .newTransformer(xslt);
    t.transform(doc,new StreamResult(writer));
    return writer.getBuffer().toString();
  }catch (TransformerException e){
    return "";
  }
}
```

4 Aufgaben

Aufgabe 1 Zunächst wollen wir ein paar kleine Funktionen auf unserem eigenen kleinem XML-Api schreiben.

a) Schreiben Sie eine Funktion, die die maximale Pfadlänge von der Wurzel bis zu einem Blatt berechnet. Ein Dokument, das nur aus einem Knoten besteht, hat dabei die Länge 1.

XML.java

```
default long height(){
  if (this instanceof Text) return 1;
  /* ToDo */
  throw new RuntimeException("unmatched pattern: "+this);
}
```

Aufgabe 2 In dieser Aufgabe wollen wir für die XPath-Achsen auf unserer eigenen Datenstruktur XML eine Selektion implementieren. Hierzu sei eine Auzählung gegeben, die die unterschiedlichen Achsen enthält.

XML.java

```
public static enum Axes
  {self
  ,child
  ,descendant
  ,descendant_or_self
  ,parent
  ,ancestor
  ,ancestor_or_self
  ,following_sibling
  ,following
  ,preceding_sibling
  ,preceding;
}
```

In dieser Aufgabe ist für jeder der Achsen eine Funktion zu schreiben, die die Elemente auf der entsprechenden Achse in eine Ergebnisliste einfügt.

XML.java

```
public default List<XML> select(Axes ax){
  return switch(ax){
    case self -> self();
    case child -> child();
    case descendant -> descendant();
    case descendant_or_self -> descendantOrSelf();
    case parent -> parent();
    case ancestor -> ancestor();
```

```
      case ancestor_or_self -> ancestorOrSelf();
      case following_sibling -> followingSibling();
      case following -> following();
      case preceding_sibling -> precedingSibling();
      case preceding -> preceding();
    };
  }
```

a) Implementieren die Achse *self* für unsere XML-Objekte.

XML.java

```
public default List<XML> self(){
  return self(new ArrayList<>());
}
```

XML.java

```
public default List<XML> self(List<XML> result){
  //TODO
  return result;
}
```

b) Implementieren die Achse *child* für unsere XML-Objekte.

XML.java

```
public default List<XML> child(){
  return child(new ArrayList<>());
}
```

XML.java

```
public default List<XML> child(List<XML> result){
  //TODO
  return result;
}
```

c) Implementieren die Achse *descendent* für unsere XML-Objekte.

XML.java

```
public default List<XML> descendant(){
  return descendant(new ArrayList<>());
}
```

XML.java

```
public default List<XML> descendant(List<XML> result){
  //TODO
  return result;
}
```

d) Implementieren die Achse *descendentOrSelf* für unsere XML-Objekte.

XML.java

```
public default List<XML> descendantOrSelf(){
  return descendantOrSelf(new ArrayList<>());
}
```

XML.java

```
public default List<XML> descendantOrSelf(List<XML> result){
  //TODO
  return result;
}
```

e) Implementieren die Achse *parent* für unsere XML-Objekte.

XML.java

```
public default List<XML> parent(){
  return parent(new ArrayList<>());
}
```

XML.java

```
public default List<XML> parent(List<XML> result){
  //TODO
  return result;
}
```

f) Implementieren die Achse *ancestor* für unsere XML-Objekte.

XML.java

```
public default List<XML> ancestor(){
  return ancestor(new ArrayList<>());
}
```

XML.java

```
public default List<XML> ancestor(List<XML> result){
  //TODO
  return result;
}
```

g) Implementieren die Achse *ancestorOrSelf* für unsere XML-Objekte.

XML.java

```
public default List<XML> ancestorOrSelf(){
  return ancestorOrSelf(new ArrayList<>());
}
```

XML.java

```
public default List<XML> ancestorOrSelf(List<XML> result){
  //TODO
  return result;
}
```

h) Implementieren die Achse *followingSibling* für unsere XML-Objekte.

XML.java

```
public default List<XML> followingSibling(){
  return followingSibling(new ArrayList<>());
}
```

XML.java

```
public default List<XML> followingSibling(List<XML> result){
  //TODO
  return result;
}
```

i) Implementieren die Achse *precedingSibling* für unsere XML-Objekte.

XML.java

```
public default List<XML> precedingSibling(){
  return precedingSibling(new ArrayList<>());
}
```

XML.java

```
public default List<XML> precedingSibling(List<XML> result){
  //TODO
  return result;
}
```

j) Implementieren die Achse *following* für unsere XML-Objekte.

XML.java

```
public default List<XML> following(){
  return following(new ArrayList<>());
}
```

Die zweite bekommt die Ergebnisliste als Parameter übergeben und fügt in diese die Knoten ein.

XML.java

```
public default List<XML> following(List<XML> result){
  //TODO
  return result;
}
```

k) Implementieren die Achse *preceding* für unsere XML-Objekte.

XML.java

```
public default List<XML> preceding(){
  return preceding(new ArrayList<>());
}
```

XML.java

```
public default List<XML> preceding(List<XML> result){
  //TODO
  return result;
}
```

Aufgabe 3 In dieser Aufgabe sollen ein paar Funktionen geschrieben werden, die durch die Baumstruktur von DOM-Objekten traversieren.

XML.java

```
public static class DOMUtil {
```

a) Schreiben Sie eine Funktion, die die maximale Pfadlänge von der Wurzel bis zu einem Blatt berechnet. Ein Dokument, das nur aus einem Knoten besteht, hat dabei die Länge 1.

XML.java

```
public static long height(Node n){
  return 0;
}
```

b) Schreiben Sie eine Funktion, in der alle Texte von Textknoten des Dokuments `n` dem `StringBuffer` in der Dokumentenreihenfolge angehängt werden. Sie dürfen dabei nicht die DOM-Methode getTextContent() verwenden.

XML.java

```
public static  String text(Node n){
  StringBuffer result = new StringBuffer();
  collectText(n, result);
  return result.toString();
  }
```

XML.java

```
public static void collectText(Node n, StringBuffer result){
}
```

c) Schreiben Sie eine Funktion, die testet, ob ein Element mit dem übergebenen Tagnamen im Dokument existiert.

XML.java

```
public static boolean containsTag(Node n, String tagname) {
  return false;
}
```

d) Schreiben Sie eine Funktion, die in der Ergebnismenge alle Tagnamen, die im Dokument auftreten, einfügt.

XML.java

```
public static Set<String> collectTagnames(Node n){
  Set<String> result = ConcurrentHashMap.newKeySet();
  collectTagnames(n,result);
  return result;
}
```

XML.java

```
public static void collectTagnames(Node n, Set<String> result) {
}
```

e) Schreiben Sie eine Funktion, die für alle Attribute im gesammten Dokument mit Schlüssel `attrKey`, die eine Zahl als Wert haben, den größten Wert berechnet.

XML.java

```
public static long getMaxHAttributeValue(Node n, String attrKey) {
  long result = 0;
  return result;
}
```

XML.java

```
}
```

Aufgabe 4 Implementieren Sie eine Handlerklasse für SAX, die alle Attributwerte des Attributs `born`, die in einem Elementknoten mit Tagnamen `element` stehen, als int-Zahl einliest und in einer Menge speichert.

XML.java

```
public static class GetBirthYears extends DefaultHandler {
  public Set<Integer> result = new HashSet<>();

  // ToDo
}
```

XML.java

```
}
```

Literatur

[BCH+04] Steven B Byrne, Mike Champion, Philippe Le Hégaret, Gavin Nicol, Jonathan Robie, Arnaud Le Hors, and Lauren Wood. Document object model

(DOM) level 3 core specification. W3C recommendation, W3C, April 2004. https://www.w3.org/TR/2004/REC-DOM-Level-3-Core-20040407/.

[BWS+18] Bogdan Brinza, Eric Willigers, David Storey, Chris Lilley, Dirk Schulze, and Amelia Bellamy-Royds. Scalable vector graphics (SVG) 2. Candidate recommendation, W3C, October 2018. https://www.w3.org/TR/2018/CR-SVG2-20181004/.

[CDSR14] Don Chamberlin, Michael Dyck, John Snelson, and Jonathan Robie. XQuery 3.0: An XML query language. W3C recommendation, W3C, April 2014. https://www.w3.org/TR/2014/REC-xquery-30-20140408/.

[DRS17] Michael Dyck, Jonathan Robie, and Josh Spiegel. XML path language (XPath) 3.1. W3C recommendation, W3C, March 2017. https://www.w3.org/TR/2017/REC-xpath-31-20170321/.

[HLTB06] Dave Hollander, Andrew Layman, Richard Tobin, and Tim Bray. Namespaces in XML 1.1 (second edition). W3C recommendation, W3C, August 2006. https://www.w3.org/TR/2006/REC-xml-names11-20060816/.

[McC18] Shane McCarron. XHTML™ basic 1.1 - second edition. W3C recommendation, W3C, March 2018. https://www.w3.org/TR/2018/SPSD-xhtml-basic-20180327/.

[MTB+12] Noah Mendelsohn, Henry Thompson, David Beech, Murray Maloney, Michael Sperberg-McQueen, and Sandy Gao. W3C xml schema definition language (XSD) 1.1 part 1: Structures. W3C recommendation, W3C, April 2012. https://www.w3.org/TR/2012/REC-xmlschema11-1-20120405/.

[PSMY+06] Jean Paoli, Michael Sperberg-McQueen, François Yergeau, John Cowan, Tim Bray, and Eve Maler. Extensible markup language (XML) 1.1 (second edition). W3C recommendation, W3C, August 2006. https://www.w3.org/TR/2006/REC-xml11-20060816/.

[TWZ+10] Joanne Tong, Norman Walsh, Henry Zongaro, Scott Boag, and Michael Kay. XSLT 2.0 and XQuery 1.0 serialization (second edition). W3C recommendation, W3C, December 2010. https://www.w3.org/TR/2010/REC-xslt-xquery-serialization-20101214/.

[Ved07] Asir Vedamuthu. Web services description language (WSDL) version 2.0 SOAP 1.1 binding. W3C note, W3C, June 2007. https://www.w3.org/TR/2007/NOTE-wsdl20-soap11-binding-20070626/.

Syntax
XPath, XSL, DTD
DOM, SAX, StAX