Programmieren 3 – Grundlagen der Webprogrammierung

Vorlesung an der Hochschule Bremerhaven

Prof. Dr. Thomas Umland

Wintersemester 2023/2024

Stand: 16. Januar 2024

Allgemeines

1. Vorlesungsmaterialien, insbesondere Folien und Aufgabenblätter stehen im pdf-Format auf dem ELLI-Server der Hochschule
2. Kontakt:
   - E-Mail: Thomas.Umland@hs-bremerhaven.de
   - Raum Z2030
   - Telefon: (0471) 4823-406
   - Sprechzeiten: nach Vereinbarung

Lernziele

- Umgang mit XML üben
- DOM verstehen und dynamische Änderungen vornehmen können
- Programmierpraxis mit XML-APIs vertiefen
- Erfahrungen mit JavaScript sammeln

Vorkenntnisse

- **nicht notwendig** (aber hilfreich):
  - HTML-Erfahrung
  - Erfahrungen mit einer (Java-)IDE: IntelliJ oder Eclipse (+ oXygen XML-Plugin)
- **ausreichend:**
  - fundierte Java-Kenntnisse

Organisatorisches

- **Umfang**: 5 CP[1] (4 SWS: 2 Std. Vorl. + 2 Std. Üb.)
- **Termine:**
  - **Vorlesung:** Di., 08.00 – 09.30 Uhr (S 201)
  - **Übungsgruppe 1:** Di., 09.45 – 11.15 Uhr (Z 2320)
  - **Übungsgruppe 2:** Di., 13.45 – 15.15 Uhr (Z 2320)
  - **Übungsgruppe 3:** Di., 15.30 – 17.00 Uhr (Z 2320)

[1] Mit Zusatzaufgabe auch 6 CP möglich.

- Es wird wöchentlich **Übungsaufgaben** geben
  - Die Übungszeiten dienen vorrangig dem **Besprechen der Übungsaufgaben**
  - Pflicht: **Jeder stellt** mind. drei Lösungen zu den Übungsaufgaben **vor** (mind. drei versch. Aufgabenblätter)!
- Zur Erlangung eines **Leistungsnachweises** sind im Verlauf des Semesters größere Aufgaben („Assignments“) zu lösen und termingerecht abzugeben.

Erwartungen

Was erwarten Sie von der Vorlesung?

Welche inhaltlichen Wünsche haben Sie?

- ...
- ...
- ...

Gliederung

A Literaturhinweise

[Bal01] H. Balzert, *Lehrbuch der Software-Technik*, Spektrum Akademischer Verlag, Heidelberg, 2. Auflage (2001).

[Bra02] N. Bradley, *The XML Companion*, Addison-Wesley, London (2002).

[Cro08] D. Crockford, *JavaScript: The Good Parts*, O'Reilly, E-Book (2008).

[Ecl23] Eclipse Foundation, *Jakarta XML Binding*. https://eclipse-ee4j.github.io/jaxb-ri (09.10.2023).

[Fit04] M. Fitzgerald, *Learning XSLT*, O'Reilly, Sebastopol, CA (2004).

[GHJV96] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Entwurfsmuster*, Addison-Wesley, München (1996).

[Har04] E. R. Harold, *XML 1.1 Bible*, Wiley, Indianapolis (2004).

[Hol01] S. Holzner, *Inside XML*, New Riders Publishing, Indianapolis (2001).

[Jak23] Jakarta Platform Team, *The Jakarta EE Tutorial*. https://eclipse-ee4j.github.io/jakartaee-tutorial (09.10.2023).

[JSO23] JSON.org, *Introducing JSON*. https://www.json.org (09.10.2023).

[Kan23] I. Kantor, *The Modern JavaScript Tutorial*. https://javascript.info (09.10.2023).

[Koc23] P.-P. Koch, *QuirksMode*. https://www.quirksmode.org (09.10.2023).

[Lar98] C. Larman, *Applying UML and Patterns*, Prentice Hall, Upper Saddle River, New Jersey (1998).

[Meg23] D. Megginson, *About SAX*. http://www.saxproject.org (09.10.2023).

[Moz23] Mozilla Foundation, *HTML Drag and Drop API*. https://developer.mozilla.org/en-US/docs/Web/API/HTML_Drag_and_Drop_API (09.10.2023).

[Ora23a] Oracle, *The Java EE 7 Tutorial*. https://docs.oracle.com/javaee/7/tutorial (09.10.2023).

[Ora23b] Oracle, *The Java Web Services Tutorial*. https://docs.oracle.com/.../tutorial/doc/index.html (09.10.2023).

[Ora23c] Oracle, *Streaming API for XML*. https://docs.oracle.com/.../tutorial/jaxp/stax/api.html (09.10.2023).

[Ray03] E. T. Ray, *Learning XML*, O'Reilly, Sebastopol, CA (2003).

[SEL23] SELFHTML e. V., *SELFHTML – Wiki*. https://wiki.selfhtml.org/wiki (09.10.2023).

[TK08] F. Thiesing, S. Kortemeyer, Entwicklung moderner Web-Anwendungen mit Open-Source-Bausteinen, *Informatik-Spektrum*, (2) **31** (2008), 115–132.

[Web23] Web Hypertext Application Technology Working Group (WHATWG), *DOM – Living Standard*. https://dom.spec.whatwg.org (09.10.2023).

[Wor23a] World Wide Web Consortium, *Extensible Markup Language (XML)*. https://www.w3.org/XML/Core (09.10.2023).

[Wor23b] World Wide Web Consortium, *Extensible Markup Language (XML) 1.0*. https://www.w3.org/TR/2008/REC-xml-20081126 (09.10.2023).

[Wor23c] World Wide Web Consortium, *HTTP - Hypertext Transfer Protocol*. https://www.w3.org/Protocols (09.10.2023).

[Wor23d] World Wide Web Consortium, *W3C Math Home*. https://www.w3.org/Math (09.10.2023).

[Wor23e] World Wide Web Consortium, *W3C XML Schema Definition Language (XSD) 1.1 Part 1: Structures*. https://www.w3.org/TR/xmlschema11-1 (09.10.2023).

[Wor23f] World Wide Web Consortium, *W3C XML Schema Definition Language (XSD) 1.1 Part 2: Datatypes*. https://www.w3.org/TR/xmlschema11-2 (09.10.2023).

[Wor23g] World Wide Web Consortium, *XML Path Language (XPath)*. https://www.w3.org/TR/xpath (09.10.2023).

[Wor23h] World Wide Web Consortium, *XML Schema Part 1: Structures Second Edition*. https://www.w3.org/.../structures.html (09.10.2023).

[Wor23i] World Wide Web Consortium, *The XMLHttpRequest Object (Working Draft)*. https://www.w3.org/TR/XMLHttpRequest (09.10.2023).

[Wor23j] World Wide Web Consortium, *XSL Transformations (XSLT)*. https://www.w3.org/TR/xslt (09.10.2023).

[Wor23k] World Wide Web Consortium, *XSL Transformations (XSLT) Version 2.0*. https://www.w3.org/TR/xslt20 (09.10.2023).

[Zak12] N. C. Zakas, *Professional JavaScript for Web Developers*, John Wiley & Sons, E-Book, 3. Auflage (2012).

Teil I

XML-Grundlagen

1 Allgemeines über XML

- „XML“ ist ein Akronym für „Extensible Markup Language“
- XML ist eine Metasprache, d. h. mit deren Hilfe können speziellere Beschreibungssprachen für bestimmte Anwendungsgebiete (sogenannte „XML-Anwendungen“) definiert werden (z. B. XHTML, X3D, SVG, SMIL, RDF, . . .)
- XML ist kein proprietäres „Spezialformat“ sondern ein vom „World Wide Web Consortium“ verabschiedeter offener Standard (vgl. [Wor23a])
- XML legt nur die Struktur der Dokumente/Daten fest, nicht die Wiedergabe der Inhalte!

Ziele von XML

- direkt einsetzbar im Internet-Umfeld
- einheitliches (strukturiertes) Format für Electronic Publishing und allgemeinen Datenaustausch
- basiert auf formalen Regeln ($\rightarrow$ einfaches Erstellen und automatisches Verarbeiten von XML-Dokumenten)
- erweiterbar/anpassbar an spezielle Bedürfnisse
- „Kompaktheit“ der XML-Beschreibung ist von untergeordneter Bedeutung

Beispiel: Formale Definition des XML-Prologs

vgl. Definition unter [Wor23b]:

```
prolog       ::= XMLDecl? Misc* (doctypedecl Misc*)?
XMLDecl      ::= '<?xml' VersionInfo EncodingDecl?
                 SDDecl? S? '?>'
VersionInfo ::= S 'version' Eq
                 ("'" VersionNum "'" | '"' VersionNum '"')
Misc         ::= Comment | PI | S
...
```

Konkrete Beispiele:

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

oder

```
<?xml version="1.1"?>
<!DOCTYPE greeting SYSTEM "hello.dtd">
```

XML-Historie

- Vorläufer „SGML“, ISO-Standard, 1986
- 1996: W3C beginnt Arbeit an XML: Ziel Datenaustausch über das Internet
- 1998: XML 1.0 verabschiedet
- Nov. 2008: XML 1.0 (Fifth Edition)
- Aug. 2006: XML 1.1 (Second Edition)
  - wichtige Änderungen gegenüber Version 1.0: Verwendung neuer Versionen von Unicode, Änderung der erlaubten Zeichen in Namen, zusätzliche Zeilenende-Zeichen erlaubt, . . .
  - Verwendung von XML 1.0 ist weiter erlaubt

2 Aufbau von XML-Dokumenten

Grundsätzliches: Das XML-Format ist textbasiert

- Text erweitert um textuelle Auszeichnungselemente („Markups“)
- ähnlich den Textverarbeitungsformaten RTF oder LaTeX

Bemerkung: Die Ausdrücke XML-Dokument, XML-Daten oder XML-Datei werden weitgehend synonym verwendet.

Jedes XML-Dokument besteht aus den folgenden Bestandteilen:

1. Prolog:
   (a) XML-Deklaration (optional)
   (b) Document type declaration (optional)
2. Wurzelelement (obligarotisch)

Diese Bestandteile werden im folgenden näher erläutert.

2.1 Elemente

Wichtigster Bestandteil von XML-Dokumenten sind die sogenannten „Elemente“.

- Jedes Element besitzt genau einen Namen.
- Elementnamen bestehen aus einem Buchstaben gefolgt von einer beliebig langen Zeichenkette aus Buchstaben, Ziffern und den Zeichen „.“, „-“, „:“, „_“.

  Beispiele:

  - Erlaubte Elementnamen: `a`, `b:c`, `einVorname`
  - Nicht erlaubte Namen: `a+b`, `2Tausend`, `Ernie&Bert`

- Groß- und Kleinschreibung wird unterschieden!

- Es gibt keine vordefinierten Elementnamen (alle möglichen Elementnamen sind damit auch nutzbar)
- Jedes Element kann beliebig oft in einem XML-Dokument vorkommen
- Alle Elemente gleichen Namens gehören zum selben „Elementtyp“;

  jedes Auftreten des Elements wird als „Instanz“ des zugehörigen Elementtyps bezeichnet
- Die Reihenfolge der Instanzen eines Elementtyps ist signifikant

- Es werden zwei Arten von Elementen unterschieden:

  Nichtleeres Element (container element): Nichtleere Elemente enthalten Daten, die durch sie identifiziert werden.

  Sie bestehen aus:

  1. einem Start-Tag (<*name*>)
  2. dem Inhalt des Elements (Daten)
  3. einem Ende-Tag (</*name*>)

  Beispiel: `<einElement>`... Inhalt ... `</einElement>`

  Leeres Element (empty element): Leere Elemente enthalten keinerlei Daten

  Sie bestehen nur aus einem Tag (<*name*/>)

  Beispiel: `<einLeeresElement/>`

Elementhierarchien

- Nichtleere Elemente können als Daten enthalten:
  1. Text („text content“)
  2. weitere Elemente („element content“)
  3. eine Mischung aus Text und weiteren Elementen („mixed content“)
- Elemente dürfen verschachtelt sein, dürfen sich aber *nicht überlappen*
- Jedes XML-Dokument enthält auf der obersten Ebene genau ein Element („Wurzelelement“)

  $\rightarrow$ XML-Dokumente besitzen eine baumartige Struktur.

Beispiel: Verschiedene Elementinhalte:

```
<!-- file: sample.xml -->
<book>
  <abstract>
    Ein sehr gutes Buch über <topic>XML</topic>
    für <skill>Programmierer</skill>.
  </abstract>
  <chapter>
    <chapterName>Einführung</chapterName>
    <chapterContent>...Text...</chapterContent>
  </chapter>
  <chapter>
    <chapterName>Fazit</chapterName>
    <chapterContent>kurz, aber gut</chapterContent>
  </chapter>
  <sparePages/>
</book>
```

Beispiel: Nicht erlaubte Überlappung zweier Elemente:

```
<!-- file: ueberlappen.xml -->
<book>
  <abstract>
    Ein sehr gutes Buch über <topic>XML</topic>
    für <skill>Designer</skill> und
    <topic>Java-<skill>Programmierer</topic></skill>
  </abstract>
</book>
```

Rekursionen

Rekursive Strukturen von Elementen sind erlaubt:

Beispiel:

```
<!-- file: recursion.xml -->
<list>
  <item>1</item>
  <item>2</item>
  <item>3
    <list>
      <item>3 a</item>
      <item>3 b</item>
      <item>3 c</item>
    </list>
  </item>
  <item>4</item>
</list>
```

2.2 Attribute

Elemente können neben ihrem Namen zusätzlich Informationen – sogenannte „Attribute“ – enthalten.

- Die Attribute werden im Start-Tag von nichtleeren Elementen bzw. im Tag von leeren Elementen angeben
- Jedes Attribut besteht aus einem „Attributnamen“ und einem „Attributwert“
  - Für Attributnamen gelten dieselben Regeln wie für Elementnamen. Jedoch sind Attributnamen, die mit `xml:` beginnen, reserviert. Bislang sind folgende reservierte Attribute definiert: `xml:lang`, `xml:space` und `xml:base`

  - Attributwerte bestehen aus beliebigen Strings, die entweder in einfache oder doppelte Anführungszeichen eingeschlossen sind.
    Der Wert selbst darf entsprechend doppelte bzw. einfache Anführungszeichen enthalten!
    Beispiel:
    ```
    <login user="tom's pub" passwd='Eisb"ar'/>
    ```

- Jedes Element kann beliebig viele Attribute enthalten
- Alle Attributnamen eines Elementes müssen verschieden sein

Beispiele: Elemente mit Informationen über die verwendete Sprache (gemäß ISO 639).

- ```
  <instruction xml:lang="de">
    Hier steht es auf Deutsch
  </instruction>
  ```

- ```
  <instruction xml:lang="en">
    This is in English.
  </instruction>
  ```

- ```
  <instruction xml:lang="en-GB">
    Take the lift to the restaurant.
  </instruction>
  ```

- ```
  <instruction xml:lang="en-US">
    Take the elevator to the restaurant.
  </instruction>
  ```

Über Leerzeichen

- In einem XML-Dokument enthaltene Leer-, Zeilenende- oder Tabulatorzeichen dienen in der Regel nur der Lesbarkeit und haben keinen Einfluss auf den Inhalt des Dokuments
- Es werden zwei Arten von Leerzeichen unterschieden:

  Signifikante Leerzeichen (significant whitespace):
  Leerzeichen innerhalb von Elementen die Text enthalten

  Nicht signifikante Leerzeichen (insignificant whitespace):
  Leerzeichen innerhalb von Elementen die weitere Elemente enthalten

- Nicht signifikante Leerzeichen können durch die Anwendung ignoriert werden
- Aufeinander folgende signifikante Leerzeichen können durch die Anwendung zu einem einzelnen Leerzeichen zusammengefasst werden
- Über das Attribut `xml:space="preserve"` kann der Anwendung der Wunsch mitgeteilt werden, alle Leerzeichen innerhalb eines Elementes zu erhalten

Beispiel: „whitespaces“

```
<!-- file: whitespaces.xml -->
<list>
    <item>1</item>

    <item>2</item>
    <item>3 <list> <item>3 a</item>
<item>3 b</item> <item>3      c </item> </list>
        </item>
    <item>4 <program xml:space="preserve">
         public class test {

            public static void main (String[] x) {
              System.out.println("test");
            }
         }
        </program> </item>
</list>
```

2.3 Entitäten

- Unter dem Begriff „Entität (entity)“ werden hier Platzhalter für beliebige XML-Inhalte verstanden.
- Entitäten werden einmal deklariert und können beliebig oft innerhalb des XML-Dokuments verwendet werden.
- Bei der Verarbeitung des XML-Dokuments wird jedes Auftreten einer Entität durch seinen entsprechenden Inhalt ersetzt.
- Eine Entität mit dem Namen *name* wird über die Zeichenfolge `&name;` angesprochen.

Vordefinierte Zeichen-Entitäten

Es sind folgende Definitionen für Zeichen vorhanden:

Name	Zeichen
amp	&
apos	'
gt	>
lt	<
quot	"
`#n`	n-tes Unicodezeichen

Bemerkung: Deutsche Umlaute können als interne Entitäten selbst definiert werden, falls z. B. die direkte Eingabe von Umlauten nicht möglich/erwünscht ist.

Beispiel:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- file: entities.xml -->
<example>
    <text>
        Text mit Zeichen-Entitäten:
        &amp;, &gt;, &quot;, &apos;, &#71;
    </text>
</example>
```

2.4 Sonstige Bestandteile

Verarbeitungsanweisungen (processing instructions):

Dienen zur Steuerung der verarbeitenden Anwendung und haben keinen Einfluss auf den Inhalt des Dokuments:

Form: `<? ... ?>`

Beispiel:

```
<?xml-stylesheet href="doit.xsl" type="text/xsl"?>
```

XML-Deklaration:

Spezielle Verarbeitungsanweisung, die am Anfang des Dokuments steht und Angaben über das XML-Dokument enthält; sie sollte stets angegeben werden.

Form: `<?xml ... ?>`

Beispiel:

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

Weitere Beispiele zur XML-Deklaration siehe einführende Beispiele.

Markup-Deklarationen:

Markup-Deklarationen haben die Form `<! ... >`.

Es werden drei Arten unterschieden:

1. `<!DOCTYPE ... >` „Document type declaration“: Enthält Angaben zur näheren Beschreibung des XML-Dokuments (siehe später); sie folgt unmittelbar auf die XML-Deklaration
2. `<![CDATA[ ... ]]>` „Character data section“: Kann beliebigen Text (auch reservierte Zeichen!) enthalten; wird anstelle von Text in nichtleeren Elementen angegeben
3. `<!-- ... -->`: „Comment“: Kann beliebigen Kommentar enthalten; wird nicht als Bestandteil des XML-Dokuments interpretiert.

Beispiel:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- file: allMarkup.xml -->
<!DOCTYPE example SYSTEM "example.dtd">
<!-- Ein Dokument mit allen optionalen Bestandteilen
     und diversen Markup-Deklarationen -->
<?xml-stylesheet href="doit.xsl" type="text/xsl"?>
<example>
    <text>
        Text mit Sonderzeichen:
        &amp;, &gt;, &quot;, &apos;, &#xE4;, &#xC4;,
        &#xF6;, &#xD6;, &#xFC;, &#xDC;, &#xDF;, &#231;
    </text>
    <text>
        <![CDATA[Text mit Sonderzeichen:
                 &, >, ", ', ä, Ä,
                 ö, Ö, ü, Ü, ß, ...]]>
    </text>
</example>
```

Interne und externe Entitäten

Neben den vordefinierten Entitäten können weitere innerhalb des `DOCTYPE`-Abschnittes definiert werden.

Sie werden entweder innerhalb des Dokuments („interne Entitäten“) oder außerhalb in anderen Dokumenten definiert („externe Entitäten“).

Bemerkung: Wird eine Entität mehrfach definiert, so gilt die erste Definition.

Bemerkung: Eine extern definierte Entität kann durch eine identisch benannte intern definierte Entität überdeckt werden (interne Entitäten werden vor externen ausgewertet).

Beispiel: interne und externe Entitäten

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- file: entitiesInternalExternal.xml -->
<!-- Ein Beispiel für interne und externe Entities -->
<!DOCTYPE example [
  <!ENTITY auml "ä" >

  <!ENTITY part0 "<text>Teil 0</text>">
  <!ENTITY part1 "<text>Teil 1 (intern definiert)</text>">
  <!ENTITY part2 SYSTEM "part2.xml">
]>
<example>
  <text> &auml; oder  ä</text>

  &part0; &part1; &part2;
</example>
```

2.5 Zusammenfassung

- XML-Dokumente sind textbasiert
- XML-Dokumente werden aufgebaut aus:
  1. Elementen
  2. Verarbeitungsanweisung
  3. Markup-Deklarationen
- Elemente können geschachtelt werden
- Elemente können um Attribute erweitert werden
- Jedes XML-Dokument enthält genau ein Wurzelelement
- Jedes XML-Dokument definiert eine eindeutige baumartige Struktur

3 Document type definition – DTD

Mit dem Wissen des vorigen Abschnitts lassen sich bereits brauchbare Dokumente für XML-Anwendungen entwerfen.

Problem: Wie wird sichergestellt, dass die Syntax/Struktur des XML-Dokuments, die sich die Autoren überlegt haben, wirklich eingehalten wird?

Antwort: Definiere die erlaubten Elementnamen und die Struktur in geeigneter Weise und prüfe automatisch deren Einhaltung in den Dokumenten der XML-Anwendung.

→ „Document type definition (DTD)“

Beispiel: Dokument `courseDoc` – informell

Jedes XML-Dokument, das den Inhalt eines Vorlesungsskriptes modelliert, soll bestehen aus:

1. Einem Wurzelelement `courseDoc` mit dem Titel des Skripts
2. Genau einer textuellen Zusammenfassung `abstract`
3. Mindestens einem Kapitel `chapter` (mit Überschrift) und mindestens einem Unterelement `para`, das einen Textabsatz modelliert
4. Einem optionalen Raum für persönliche Notizen am Ende jedes Kapitels (`sparePages`)

Frage: Wie sehen mögliche XML-Dokumente aus?

Beispiel: script01

```
<?xml version="1.0" encoding="UTF-8"?>
<courseDoc>
    <title>1. Testen von XML-Dokumenten</title>
    <abstract>Kurze Zusammenfassung</abstract>
    <chapter>Erstes Kapitel <para> Absatz eins</para>
        <para> Absatz zwei</para>
    </chapter>
    <chapter>Zweites Kapitel <para> Absatz eins</para>
    </chapter>
    <para>noch ein Absatz</para>
    <chapter>Drittes Kapitel <para> Absatz eins</para>
       <sparePages/>
    </chapter>
</courseDoc>
```

Beispiel: script02

```
<?xml version="1.0" encoding="UTF-8"?>
<courseDoc>
    <title>2. Testen von XML-Dokumenten</title>
    <abstract>Zusammenfassung...</abstract>
    <chapter>
        <heading>Erstes Kapitel</heading>
        <para> Absatz eins</para><para> Absatz zwei</para>
    </chapter>
    <chapter>
        <heading>Zweites Kapitel</heading>
        <para> Absatz eins</para><sparePages/>
    </chapter>
    <chapter><heading>Drittes Kapitel</heading>
        <para> Absatz eins</para><sparePages> </sparePages>
    </chapter>
</courseDoc>
```

Beispiel: script03

```
<?xml version="1.0" encoding="UTF-8"?>
<courseDoc title="3. Testen von XML-Dokumenten">
    <abstract>Zusammenfassung...</abstract>
    <chapter heading="Erstes Kapitel">
        <para> Absatz eins</para>
        <para> Absatz zwei</para>
    </chapter>
    <chapter heading="Zweites Kapitel">
        <para> Absatz eins</para>
        <sparePages/>
    </chapter>
    <chapter heading="Drittes Kapitel">
        <para> Absatz eins</para>
    </chapter>
</courseDoc>
```

Bemerkungen zu den Beispielen `script01`, `script02` und `script03`

- Es handelt sich um syntaktisch korrekte XML-Dokumente!
- Stimmen sie mit den Vorgaben überein?
  - Titelangabe des Skripts o. k. ?
  - Überschrift der Kapitel o. k. ?
  - Verwendung der Absätze o. k. ?
  - Raum für Notizen o. k. ?

  Im Prinzip schon, aber es gibt Unterschiede im Detail! Die Dokumente sind so nicht austauschbar!

  → formelle Definition des Aufbaus der Skript-Dokumente mithilfe einer verbindlichen DTD festlegen!

3.1 Element-Deklarationen

Für jedes Element der XML-Anwendung wird in der DTD Name und Inhaltstyp festgelegt:

```
<!ELEMENT name content >
```

name bezeichnet das Element entsprechend den Namenskonventionen für Elemente

content bestimmt den möglichen Inhalt des Elements:

1. Schlüsselwort `EMPTY` definiert ein leeres Element
2. Schlüsselwort `#PCDATA` („Parsable Character Data“) definiert ein Textelement

3. Schlüsselwort `ANY` definiert ein allgemeines Container Element, das jedes andere in der DTD definierte Element als Inhalt aufnehmen darf
4. Ein regulärer Ausdruck definiert ein spezielles Container Element mit einer vorgeschriebenen Anordnung der Elemente („content model“):
   - Ausdrücke werden von runden Klammern eingeschlossen: `( ... )`
   - Wiederholung von Ausdrücken bzw. Elementen:
     (a) beliebig oft: `( ... )*`
     (b) mindestens einmal: `( ... )+`
     (c) höchstens einmal: `( ... )?`
   - Reihung von Ausdrücken bzw. Elementen: `,`
   - Auswahl von Ausdrücken bzw. Elementen: `|`

5. Gemischter Inhalt ist möglich durch Ausdrücke der Form `(#PCDATA | name1 | ...| namen)*`

Skript-DTD

Eine mögliche DTD für das Skript-Beispiel:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- file: script01.dtd -->
<!-- einfache DTD fuer Skript-Dateien -->
<!ELEMENT courseDoc (title, abstract, chapter+)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT abstract (#PCDATA)>
<!ELEMENT chapter (heading, para+, sparePages?)>
<!ELEMENT heading (#PCDATA)>
<!ELEMENT para (#PCDATA)>
<!ELEMENT sparePages EMPTY>
```

Frage: Welche der vorherigen Skript-Dokumente `script01`, `script02` oder `script03` erfüllen die Regeln dieser DTD?

Interne oder externe DTD-Deklaration

Eine DTD kann auf zwei Arten deklariert werden (vgl. interne und externe Entitäten):

Interne DTD: Definition direkt im `DOCTYPE`-Abschnitt des entsprechenden XML-Dokuments:

```
<?xml version="1.0"?>
<!DOCTYPE courseDoc [
  <!-- hier steht die DTD des Dokuments, welche die
    Struktur des Wurzelelements 'courseDoc' definiert -->
  <! ... >
  ] >
<courseDoc>
  ...
</courseDoc>
```

Externe DTD: Einlesen der Definition aus einer separaten Datei oder über einen URL:[2]

```
<?xml version="1.0"?>
<!DOCTYPE courseDoc SYSTEM "script.dtd">
<courseDoc>
   ...
</courseDoc>
```

[2] Eine externe DTD kann auch aus einer sogenannten „öffentlichen DTD“ bestehen – mehr dazu später.

Bemerkungen

- Externe und interne DTDs dürfen nebeneinander verwendet werden:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- file: internExtern.xml -->
<!DOCTYPE root SYSTEM "internExtern.dtd"
  [<!ELEMENT third (#PCDATA)>] >
<root>
    <first>foo</first>
    <second>
        <third>test</third>
    </second>
</root>
```

- Jedes Element darf nur entweder in einer internen oder in einer externer DTD definiert werden. Eine Überdeckung wie bei Entitäten ist *nicht* erlaubt.

3.2 Attribut-Deklarationen

Attribute werden in der DTD nach folgendem Muster deklariert.

1. *Ein* Attribut pro Deklaration:

```
<!ATTLIST element attribute type default >
```

2. *Mehrere* Attribute für dasselbe Element in einer Deklaration:

```
<!ATTLIST element
          attribute_1 type_1 default_1
          attribute_2 type_2 default_2
          ...
          attribute_n type_n default_n >
```

Bemerkung: Die Mehrfachdeklaration kann auch gleichwertig durch n einzelne Attributdeklarationen ersetzt werden.

Bestandteile der Attribut-Deklaration

element bezeichnet das Element, für das ein Attribut deklariert werden soll

attribute Name des Attributs entsprechend den Regeln

type Type des Attributs (hier ist nur eine Auswahl angegeben):

CDATA: („Character Data“) Beliebiger Text

NMTOKEN, NMTOKENS: („Name Token“) Einzelner XML-Name oder Liste von XML-Namen

ID: („Identity“) Innerhalb des Dokuments eindeutig vergebener XML-Name

IDREF, IDREFS („Identity Reference“) Verweis auf eine ID

ENTITY, ENTITIES Bezeichnung einer innerhalb der DTD definierten Entität

***Enumeration*:** Explizite Aufzählung der erlaubten Werte

default Standardwert für das Attribut:

***„Immediate Value“*:** Der angegebene Wert wird als Standardwert des Attributs verwendet; er kann im XML-Dokument überschrieben werden

#REQUIRED: Es wird kein Standardwert vorgegeben; im XML-Dokument ist das Attribut jedoch mit einem Wert anzugeben

#IMPLIED: Es wird kein Standardwert vorgegeben; im XML-Dokument kann das Attribut auch weggelassen werden

#FIXED *value*: Definiert einen fest vorgegebenen, nicht änderbaren Wert für das Attribut

Beispiel: Skript-DTD mit Attributen

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- file: script02.dtd -->
<!-- DTD fuer Skript-Dateien mit Attributen-->
<!ELEMENT courseDoc (abstract, chapter+)>
<!ELEMENT chapter (para+)>
<!ELEMENT abstract (#PCDATA)>
<!ELEMENT para (#PCDATA)>

<!ATTLIST courseDoc title CDATA #REQUIRED>
<!ATTLIST chapter heading CDATA #REQUIRED>
<!ATTLIST chapter sparePages (1 | 2) #IMPLIED>
<!ATTLIST chapter important (yes | no) "yes">
<!ATTLIST chapter interest CDATA #FIXED "high">
```

Frage: Wie sieht ein passendes XML-Dokument aus?

Beispiel eines passenden XML-Dokuments:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- file: script03b.xml -->
<!DOCTYPE courseDoc SYSTEM "script02.dtd">
<courseDoc title="3. Testen von XML-Dokumenten">
    <abstract>Zusammenfassung...</abstract>
    <chapter heading="Erstes Kapitel">
        <para> Absatz eins</para>
        <para> Absatz zwei</para>
    </chapter>
    <chapter heading="Zweites Kapitel" sparePages="1">
        <para> Absatz eins</para>
    </chapter>
    <chapter heading="Drittes Kapitel">
        <para> Absatz eins</para>
    </chapter>
</courseDoc>
```

Bemerkung: Wird innerhalb eines `DOCTYPE`-Abschnittes ein Attribut für dasselbe Element mehrfach deklariert, so gilt die erste Deklaration.

Bemerkung: Innerhalb eines `DOCTYPE`-Abschnittes werden interne Deklarationen vor externen gelesen und verarbeitet.

Attribute versus Elemente

Der Entwurf einer DTD für eine neue XML-Anwendung ist in der Regel nicht eindeutig. Eine „gewisse Erfahrung“ in der Modellierung von Daten ist hilfreich . . .

Tipps:

- Für Elemente und Attribute sinnvolle, „menschenlesbare“ Namen verwenden
- Hierarchiestufen zur Gliederung der Informationen einsetzen
- aber „zu tiefe“ Hierarchien, die keine Information enthalten, vermeiden

- Elemente, die das vorherige Element näher bezeichnen, evtl. als Attribut auslagern (Beispiel: Titel des Skripts als Attribut)

  Regel: Elemente enthalten Informationen, Attribute bezeichnen ein Element näher (vgl. Skript-DTD mit und ohne Verwendung von Attributen)

3.3 Parameter-Entitäten

Die Verwendung der bisher bekannten Entitäten ist innerhalb des `DOCTYPE`-Abschnittes nur beschränkt möglich.

Für die ausschließliche Verwendung innerhalb des `DOCTYPE`-Abschnittes werden die sogenannten „Parameter-Entitäten“ eingesetzt.

Deklaration:

```
<!ENTITY % name definition>
```

Verwendung:

```
... %name; ...
```

Beispiel: Parameter-Entitäten

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- file: script02_para.dtd -->

<!ENTITY % pageValues "1 | 2">
<!ENTITY test "very">
<!ENTITY % interestValue "'&test; high'">

<!ELEMENT courseDoc (abstract, chapter+)>
<!ELEMENT abstract (#PCDATA)>
<!ELEMENT chapter (para+)>
<!ELEMENT para (#PCDATA)>

<!ATTLIST courseDoc title CDATA #REQUIRED>
<!ATTLIST chapter heading CDATA #REQUIRED>
<!ATTLIST chapter sparePages (%pageValues;) #IMPLIED>
<!ATTLIST chapter important (yes | no) "yes">
<!ATTLIST chapter interest CDATA #FIXED %interestValue;>
```

Fragen:

1. Warum ist die Parameter-Entität `interestValue` in doppelte Anführungszeichen gesetzt?
2. Lässt sich die Entität `test` auch als Parameter-Entität definieren?

Bedingte Verarbeitung

Unter Verwendung von Parameter-Entitäten ist das bedingte Einfügen oder Auslassen von `DOCTYPE`-Abschnitten möglich.

Einfügen:

```
<![ INCLUDE [DTD-Abschnitt]]>
```

Auslassen:

```
<![ IGNORE [DTD-Abschnitt]]>
```

Beispiel: Bedingte Verarbeitung

```
<!ENTITY % selector "INCLUDE">
<!ENTITY % selector "IGNORE">

<!ELEMENT courseDoc (abstract, chapter+)>
<!ELEMENT abstract (#PCDATA)>
<!ELEMENT chapter (para+)>
<!ELEMENT para (#PCDATA)>

<!ATTLIST courseDoc title CDATA #REQUIRED>
<!ATTLIST chapter heading CDATA #REQUIRED>
<!ATTLIST chapter sparePages (1 | 2) #IMPLIED>
<!ATTLIST chapter important (yes | no) "yes">
<![ %selector;[
  <!ATTLIST chapter interest CDATA #FIXED "very high">
]]>
<!ATTLIST chapter interest CDATA #FIXED "infinitely high">
```

Fragen:

1. Welche Definition des Attributs `interest` ist im vorigen Beispiel gültig?[3]
2. Wie lässt sich erreichen, dass die andere Definition verwendet wird?

[3] **Tipp:** Suche mit XPath-Ausdrücken verwenden (siehe später)

3.4 Öffentliche DTDs

Bisher wurden DTDs entweder direkt im XML-Dokument deklariert `<!DOCTYPE rootElement [DTD]>` (interne DTD) oder aus einer anderen Datei oder über einen URL eingelesen `<!DOCTYPE rootElement SYSTEM location>` (externe DTD).

Außerdem besteht die Möglichkeit, DTDs von „allgemeinem Interesse“, die als sogenannte öffentliche DTDs zur Verfügung gestellt wurden, als externe DTDs zu importieren:

```
<!DOCTYPE rootElement PUBLIC fpi url>
```

rootElement Wurzelelement des Dokuments, das in der DTD deklariert wird

url URL, unter dem die öffentliche DTD verfügbar ist

fpi „formal public identifier“ enthält vier nähere Angaben zur DTD; diese sind durch „// “ voneinander getrennt:

1. Angaben zum Standard:
   (a) „–“ für eigene (inoffizielle) Veröffentlichungen
   (b) „+“ für anerkannte, aber nicht standardisierte DTDs
   (c) Kürzel der Standardisierungsorganisation
2. Angaben zum Autor
3. Typ des Dokuments und Versionsnummer
4. Sprache, die in der DTD verwendet wird

Beispiele öffentlicher DTDs

DTD für XHMTL:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
```

DTD für die 3D-Beschreibungssprache X3D:

```
<!DOCTYPE X3D PUBLIC "ISO//Web3D//DTD X3D 3.0//EN"
 "http://www.web3d.org/specifications/x3d-3.0.dtd">
```

Eigene Test-DTD für das Skript-Beispiel:

```
<!DOCTYPE courseDoc
 PUBLIC "-//Thomas Umland//Testen von PUBLIC DTDs 1.0//DE"
 "http://www1.hs-.../umland/.../xml/testPublic.dtd">
```

3.5 Wann sind DTDs sinnvoll?

Das Erstellen einer DTD zu einer XML-Anwendung bedeutet Mehraufwand: Beim Entwurf der Anwendung, bei der Fehlersuche, bei der Pflege, ...

Dennoch ist dieser Mehraufwand meistens gerechtfertigt:

- Eine DTD kann als veröffentlichbare Spezifikation der XML-Anwendung dienen (Aufbau erinnert an BNF-Grammatiken zur Beschreibung von Programmiersprachen).

- Anhand der DTD kann automatisch überprüft, ob ein Dokument die geforderte Struktur und Syntax einhält. DTDs sind plattformunabhängig (im Gegensatz zu eventuellen Softwaretests der Dokumente).
- Eine DTD erleichtert das gemeinsame Arbeiten mehrerer Entwickler/Autoren. Schreibfehler oder fehlende Daten können schon beim Erstellen der Dokumente erkannt werden.
- DTDs sind erweiterbar. Bei der Verwendung ähnlicher XML-Anwendungen können Teile der DTDs evtl. gemeinsam benutzt/wieder verwendet werden.

Fazit

- Eine DTD sollte immer dann eingesetzt werden, wenn XML-Dokumente manuell (insbesondere von mehreren Autoren) erstellt werden.
- Auch bei der automatischen Verarbeitung von XML-Dokumenten kann durch die Verwendung einer DTD eine Vorab-Syntax-Überprüfung durchgeführt werden.
- Nur bei sehr kleinen XML-Anwendungen, deren Dokument ausschließlich von *einer* Software automatisch verarbeitet wird, kann evtl. auf den Einsatz einer DTD verzichtet werden.

4 Wohlgeformtheit und Gültigkeit von XML-Dokumenten

Im Zusammenhang mit XML-Dokumenten werden häufig die Begriffe Wohlgeformtheit („well-formedness“) und Gültigkeit („validity“) gebraucht.

Diese Begriffe werden in der XML-Spezifikation definiert (für XML 1.0 z. B. in den Abschnitten 2.1 bzw. 2.8).

4.1 Wohlgeformtheit

- Jedes Dokument, das die in der XML-Spezifikation festgelegten Regeln für den Aufbau von XML-Dokumenten erfüllt, wird als wohlgeformt bezeichnet.
- Es gibt mehr als 100 solcher Regeln; die wichtigsten sind:
  - Einhalten der Namensregeln für Tags,
  - Beachten der Groß- und Kleinschreibung bei Tags,
  - Verwenden korrespondierender Start- und Endetags,
  - kein Überlappen der Elemente,
  - genau ein Wurzelelement im Dokument
  - ...

- Wohlgeformtheit ist die minimale Eigenschaft, die ein Dokument erfüllen muss, um von einem XML-Prozessor[4] erfolgreich verarbeitet werden zu können.
- Jede Verletzung der Wohlgeformtheit eines Dokuments ist vom XML-Prozessor an die Anwendung zu melden.
- Nach einem festgestellten Fehler darf keine weitere Verarbeitung des Dokuments mehr erfolgen (nur noch zur Fehleranalyse).
- Insbesondere darf nicht versucht werden, fehlerhafte Dokumente „automatisch“ zu korrigieren (anders als z. B. bei HTML-Browsern üblich).

[4] Ein „XML-Prozessor“ ist ein Softwaremodul, das XML-Dokumente liest und einer XML-Anwendung Zugriff auf Inhalt und Struktur des Dokuments ermöglicht.

4.2 Gültigkeit

- Neben den sehr allgemeinen Regeln der Wohlgeformtheit, kann jede XML-Anwendung weitergehende Anforderungen an die Struktur und den Inhalt ihrer XML-Dokumente stellen.
- Diese werden in der Regel anhand einer DTD oder eines Schemas (siehe später) definiert.
- Jedes wohlgeformte Dokument, das die dort definierten Einschränkungen erfüllt, heißt gültig.
- Ein XML-Prozessor wird validierender Prozessor genannt, wenn er ein XML-Dokument auf Gültigkeit überprüft.

- Validierende Prozessoren haben neben Verletzungen der Wohlgeformtheit auch Verletzungen der Gültigkeit an die Anwendung zu melden. Nach der Feststellung eines Fehlers ist die weitere Verarbeitung des Dokuments zu unterbinden.

Bemerkung: Das Überprüfen der Wohlgeformtheit und der Gültigkeit lässt sich mit speziellen XML-Editoren schon beim Erstellen der Dokumente durchführen.

5 Namensräume (Namespaces)

- XML-Namensräume stellen einen Mechanismus zur Gruppierung oder Identifizierung von Elementen dar
- Durch Namensräume lassen sich Elemente aus verschiedenen Anwendungen eindeutig unterscheiden
- Unter Verwendung von Namensräumen können Elemente aus unterschiedlichen XML-Anwendungen in *einem* Dokument verwendet werden.

Definition eines Namensraumes

Für ein XML-Element wird ein Namensraum durch Angabe eines Attributs der folgenden Form deklariert:

1. Impliziter, unbenannter Namensraum:

   `xmlns="uri"`

2. Explizit benannter Namensraum:

   `xmlns:name="uri"`

name lokaler Bezeichnung des Namensraumes (frei wählbar)

uri String zum Identifizieren des Namensraumes;

Konvention: (Eindeutigen) URI mit Bezug zum Herausgeber des Namensraumes verwenden (siehe Beispiele).

Bemerkung: Aus der gewählten Bezeichnung des Namensraumes (*name*) wird der für dessen Elemente zu verwendende Präfix (`name:`) abgeleitet.

Bemerkung: Die Bezeichnung *xml* sollte für einen Namensraum nicht verwendet werden, da er für den internen Gebrauch reserviert ist (vgl. `xml:lang` oder `xml:space`).

Bemerkung: Bei unbenannten Namensräumen werden die Elemente nicht mit einem Präfix versehen, sondern bleiben in ihrer ursprünglichen Schreibweise erhalten. Ein unbenannter Namensraum wird in seinem Gültigkeitsbereich auch als Standard-Namensraum („default namespace“) bezeichnet.

Bemerkungen zu XHTML

- **Konvention:** Web-Browser stellen XHTML-Dokumente nur dann dar, wenn das Wurzelelement den Namen `html` besitzt und für es außerdem ein Namensraum mit der Identifikation `http://www.w3.org/1999/xhtml` deklariert ist (Voraussetzung: Dateiendung deutet nicht auf HTML hin; ist z. B. `.xml`).
- Über die Angabe einer entsprechenden DTD können XHTML-Dokumente[5] zusätzlich validiert werden.

[5] Die Syntax von XHTML wird hier nicht weiter vermittelt. Sie kann im Selbststudium – z. B. unter Verwendung der Anleitung SELFHTML-Wiki (vgl. [SEL23]) – erlernt werden.

Beispiel: Einfaches XHTML-Dokument

```
<?xml version="1.0"?>
<!-- file: html01.xml -->
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
  <head><title>HTML Example</title></head>
  <body>
    <h2>Beispiel</h2>
    <p>Die Geradengleichung lautet:
      <em>hier soll eine Gleichung stehen</em></p>
    <p>Dabei ergibt sich die Steigung zu<br/>
      <em>hier soll eine Formel stehen</em>.</p>
    <p>Hier geht es mit <b>HTML</b> weiter...</p>
  </body>
</html>
```

- Die Elemente im vorigen XML-Dokument liegen alle im implizit deklarierten Namensraum mit der Identifikation `xmlns="http://www.w3.org/1999/xhtml`.
- Ohne Angabe des Namensraumes stellen die Browser nur den Inhalt des Dokuments dar, ohne eine HTML-Formatierung vorzunehmen.
- Auf die Angabe der DTD kann verzichtet werden, wenn keine Validierung des Dokuments gewünscht wird.

Beispiel: XHTML-Dokument mit explizitem Namensraum

```
<?xml version="1.0"?>
<!-- file: html02.xml -->
<h:html xmlns:h="http://www.w3.org/1999/xhtml">
  <h:head></h:head>
  <h:body>
    <h:h2>Beispiel</h:h2>
    <h:p>Die Geradengleichung lautet:
      <h:em>hier soll eine Gleichung stehen</h:em></h:p>
    <h:p>Dabei ergibt sich die Steigung zu<h:br/>
      <h:em>hier soll eine Formel stehen</h:em>.</h:p>
    <h:p>Hier geht es mit <h:b>HTML</h:b> weiter...</h:p>
  </h:body>
</h:html>
```

Aufgabe: In den XHTML-Dokumenten sollen an den markierten Stellen mathematische Formeln eingebunden und dargestellt werden!

Lösung: Verwende die XML-Anwendung MathML (vgl. [Wor23d]) zur Beschreibung der Formeln.

Beispiel: Einfaches MATHML-Dokument

```
<?xml version="1.0"?>
<!-- file: math01.xml -->
<!DOCTYPE math PUBLIC "-//W3C//DTD MathML 2.0//EN"
        "http://www.w3.org/Math/DTD/mathml2/mathml2.dtd">
<math xmlns="http://www.w3.org/1998/Math/MathML">
    <mi>m</mi> <mo>=</mo>
    <mfrac>
        <mrow> <msub><mi>y</mi><mi>1</mi></msub>
               <mo>-</mo>
               <msub><mi>y</mi><mi>0</mi></msub>
        </mrow>
        <mrow> <msub><mi>x</mi><mi>1</mi></msub>
               <mo>-</mo>
               <msub><mi>x</mi><mi>0</mi></msub>
        </mrow>
    </mfrac>
</math>
```

Bemerkung: Ohne Angabe des Namensraumes werden mathematische Formeln im Browser i. allg. nicht formatiert.

Bemerkung: Die Angabe der DTD ist optional.

Frage: Wie können XHTML- und MATHML-Elemente kombiniert werden?

Antwort: Unterschiedliche Namensräume für die entsprechenden Elemente verwenden!

Kombination von XHTML und MATHML in einem Dokument (1)

```
<?xml version="1.0"?>
<!-- file: mathHtml01.xml -->
<!DOCTYPE html
  PUBLIC "-//W3C//DTD XHTML 1.1 plus MathML 2.0//EN"
  "http://www.w3.org/Math/DTD/mathml2/xhtml-math11-f.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
    <head><title>MathML in HTML (01)</title></head>
    <body>
        <h2>Beispiel 01</h2>
        <p>Die Geradengleichung lautet:
          <math xmlns="http://www.w3.org/1998/Math/MathML">
              <mi>y</mi> <mo>=</mo>
              <mi>m</mi><mo>*</mo><mi>x</mi>
                        <mo>+</mo><mi>b</mi>
          </math> </p>
```

```
        <p>Dabei ergibt sich die Steigung zu <br/>
          <math xmlns="http://www.w3.org/1998/Math/MathML">
             <mi>m</mi><mo>=</mo>
             <mfrac>
                <mrow> <msub><mi>y</mi><mi>1</mi></msub>
                       <mo>-</mo>
                       <msub><mi>y</mi><mi>0</mi></msub>
                </mrow>
                <mrow> <msub><mi>x</mi><mi>1</mi></msub>
                       <mo>-</mo>
                       <msub><mi>x</mi><mi>0</mi></msub>
                </mrow>
             </mfrac>
          </math>. </p>
        <p>Hier geht es mit <b>HTML</b> weiter...</p>
    </body>
</html>
```

Bemerkung: Im vorigen Beispiel wird in den HTML- bzw. MATHML-Abschnitten jeweils der Standard-Namensraum gewechselt.

Bemerkung: Es ist auch ein mehrfacher Wechsel des Standard-Namensraumes möglich.

Mehrfacher Wechsel des default namespace

```
<?xml version="1.0"?>
<!-- file: mathHtml02.xml -->
<html xmlns="http://www.w3.org/1999/xhtml">
    <head><title>MathML in HTML (02)</title></head>
    <body>
        <h2>Beispiel 02</h2>
        <p>Die Steigung ist <br/>
          <math xmlns="http://www.w3.org/1998/Math/MathML">
              <mi>m</mi><mo>=</mo>
              <p  xmlns="http://www.w3.org/1999/xhtml">
                Liste innerhalb einer Gleichung:
                <ul><li>eins</li><li>zwei</li></ul> </p>
              <mfrac> ... </mfrac>
          </math>.
        </p>
    </body>
</html>
```

Explizite Namensräume

```
<h:html xmlns:h="http://www.w3.org/1999/xhtml"
        xmlns:m="http://www.w3.org/1998/Math/MathML">
  <h:head><h:title>MathML in HTML (03)</h:title></h:head>
  <h:body>
    <h:h2>Beispiel 03</h:h2>
    <h:p>Die Steigung ist <h:br/>
      <m:math>
        <m:mi>m</m:mi><m:mo>=</m:mo>
        <h:p>
          Liste innerhalb einer Gleichung:
          <h:ul><h:li>eins</h:li><h:li>zwei</h:li></h:ul>
        </h:p>
        <m:mfrac>...</m:mfrac>
      </m:math>
    </h:p>
  </h:body>
</h:html>
```

Gemischte Namensräume

```
<h:html xmlns:h="http://www.w3.org/1999/xhtml"
        xmlns="http://www.w3.org/1998/Math/MathML">
  <h:head><h:title>MathML in HTML (04)</h:title></h:head>
  <h:body>
    <h:h2>Beispiel 04</h:h2>
    <h:p>Die Steigung ist <h:br/>
      <math>
        <mi>m</mi><mo>=</mo>
        <h:p>
          Liste innerhalb einer Gleichung:
          <h:ul><h:li>eins</h:li><h:li>zwei</h:li></h:ul>
        </h:p>
        <mfrac> ... </mfrac>
      </math>
    </h:p>
  </h:body>
</h:html>
```

Namensräume und DTDs

Da das Konzept der Namensräume dem XML-Standard erst später hinzugefügt wurde, treten Probleme im Zusammenspiel mit DTDs auf:

- Soll ein explizit deklarierter Namensraum mithilfe einer DTD validiert werden, sind i. allg. Änderungen an der DTD notwendig!
- Der Präfix des Namensraums muss explizit den in der DTD deklarierten Elementnamen hinzugefügt werden.
- Werden zusätzliche Elemente eines anderen Namensraumes verwendet (Beispiel: MATHML innerhalb von XHTML) müssen diese Elemente der ursprünglichen DTD hinzugefügt werden.

6 XML-Schema

Mithilfe der sogenannten „Document type definition (DTD)“ lassen sich XML-Dokumente validieren (d. h. auf die Einhaltung einer vorgegebenen Struktur überprüfen).

DTDs weisen aber zwei wesentliche Nachteile auf:

1. Es gibt Probleme im Zusammenspiel mit Namensräumen
2. DTDs werden in einer eigenen, von XML verschiedenen Notation deklariert und können deshalb nicht mit XML-Mitteln automatisch verarbeitet werden.

$\rightarrow$ Abhilfe schaffen sogenannte Schemata

Ansätze für Schemata

Es gibt unterschiedliche Ansätze zur Definition von Schemata für XML-Dokumente (als Ersatz für DTDs).

Ziel aller Ansätze sind neben der gewünschten XML-Syntax die Integration von Namensräumen und die flexible Definition von Datentypen (für Attribute *und* Elemente):

RELAX NG (wird „relaxing“ ausgesprochen)

Schematron

(W3C) XML-Schema: Dieser von W3C koordinierte Ansatz hat sich durchgesetzt und wird im Folgenden näher behandelt.

6.1 Erstellen eines XML-Schemas

Für die Deklaration eines XML-Schemas ist ein Wurzelelement namens `schema` vorgesehen sowie ein spezieller Namensraum reserviert.

Dieser Namensraum kann innerhalb des Schemadokuments mit einem expliziten Präfix versehen oder implizit verwendet werden:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
```

oder

```
<schema xmlns="http://www.w3.org/2001/XMLSchema">
```

Schemadeklaration und Namensräume

Innerhalb der Schemadeklaration werden mindestens zwei verschiedene Namensräume benötigt:

1. Ein Namespace für das W3C-Vokabular zur Festlegung des Schemas
2. Ein Namespace für die Namen der zu deklarierenden XML-Anwendung (kann auch implizit sein)
3. evtl. weitere Namensräume für eingebundene zusätzliche Elemente

Aufbau der Schemadeklaration

Im folgenden wird der Aufbau einer Schemadeklaration am Beispiel eines zur Skript-DTD vergleichbaren Schemas schrittweise beschrieben.

Es werden dabei nur ausgewählte Sprachelemente verwendet! Für eine ausführliche Beschreibung sei auf die Literatur verwiesen: z. B. [Wor23e, Bra02, Ray03, Hol01, Har04].

Im Vergleich zur DTD stehen im XML-Schema deutlich mehr Datentypen sowohl für Attribute als auch für Elemente zur Verfügung. Bei der Validierung findet eine Typüberprüfung statt!

Auswahl von Datentypen in XML-Schema

Typ	Wertebereich
`xs:string`	beliebiger Text
`xs:boolean`	`true`, `false` bzw. `1`, `0`
`xs:int`, `xs:long`	vergleichbar mit Java `int`, `long`
`xs:float`	Zahl in Gleitkommandarstellung
`xs:decimal`	Dezimalzahl (mit Nachkommastellen)
`xs:integer`	ganzzahliger Wert
`xs:time`	`12:59:00`
`xs:date`	`2004-02-29`
`xs:token`	beliebiger XML-Name
`xs:ID`	wie in DTD
`xs:IDREF`, `xs:IDREFS`	wie in DTD

Für eine ausführliche Beschreibung aller Typen siehe [Wor23f]

Element-Deklaration

1. Element mit beliebigem Inhalt (keine Typfestlegung):

```
<xs:element name="example"/>
```

2. Element mit „einfachem“ Inhalt (erlaubt ist jeder einfache oder explizit definierte Datentyp, hier z. B. `xs:time`):

```
<xs:element name="example" type="xs:time"/>
```

3. Element mit „komplexem“ Inhalt (es ist eine explizite Definition des komplexen Inhalts notwendig):

```
<xs:element name="example">
    <xs:complexType>
        ...
    </xs:complexType>
</xs:element>
```

Spezialfälle von komplexen Typen

(a) Leerer Inhalt:

```
<xs:element name="example">
    <xs:complexType/>
</xs:element>
```

(b) Element-Inhalt (vorgegebene Reihenfolge):

```
<xs:element name="example">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="one"/>
            <xs:element name="two" minOccurs="2"
                                   maxOccurs="3"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
```

oder als Auswahl:

```
<xs:element name="example">
    <xs:complexType>
        <xs:choice>
            <xs:element name="one"/>
            <xs:element name="two" minOccurs="2"
                        maxOccurs="unbounded"/>
        </xs:choice>
    </xs:complexType>
</xs:element>
```

oder Variante der Auswahl mit *explizit benanntem Typ* (statt anonymem) und *referenziertem Element*.

Vorteil: Wiederverwendung möglich; (vgl. z. B. Parameterentitäten in DTD oder anonyme Klassen in Java):

```
<xs:element name="two"/>

<xs:complexType name="myChoice">
    <xs:choice>
        <xs:element name="one"/>
        <xs:element ref="two" minOccurs="2"
                    maxOccurs="unbounded"/>
    </xs:choice>
</xs:complexType>

<xs:element name="example" type="myChoice"/>
```

oder alle Elemente, aber in beliebiger Reihenfolge:

```
<xs:element name="one"/>
<xs:element name="two"/>

<xs:element name="example">
    <xs:complexType>
        <xs:all>
            <xs:element ref="one"/>
            <xs:element ref="two"/>
        </xs:all>
    </xs:complexType>
</xs:element>
```

(c) Gemischter Element-Inhalt („Mixed Content“):

```
<xs:element name="one"/>
<xs:element name="two"/>

<xs:element name="example">
    <xs:complexType mixed="true">
        <xs:choice minOccurs="0"
                   maxOccurs="unbounded">
            <xs:element ref="one"/>
            <xs:element ref="two"/>
        </xs:choice>
    </xs:complexType>
</xs:element>
```

Attribut-Deklaration

Beispiel eines Text-Attributs und eines mit speziellem Wertebereich:

```
<xs:element name="example">
    <xs:complexType>
        <xs:attribute name="title" type="xs:string"/>
        <xs:attribute name="number" type="specialValues"/>
    </xs:complexType>
</xs:element>

<xs:simpleType name="specialValues">
  <xs:restriction base="xs:integer">
    <xs:minInclusive value="0"/>
    <xs:maxInclusive value="3"/>
  </xs:restriction>
</xs:simpleType>
```

Bemerkung: Attribute zu einem Element werden innerhalb eines `xs:complexType`-Elements vereinbart.

Bemerkung: Innerhalb eines `xs:complexType`-Elements können sowohl Attribute als auch enthaltene Unterelemente deklariert werden.

Beispiel:

```
<xs:complexType>
    <xs:sequence>
        <xs:element name="foo1"/>
        <xs:element name="foo2"/>
    </xs:sequence>
    <xs:attribute name="att1" type="xs:integer"/>
    <xs:attribute name="att2" type="xs:date"/>
</xs:complexType>
```

Modifikation bestehender Datentypen

Auf der Basis bestehender Datentypen lassen sich neue z. B. durch Einschränkungen des Wertebereiches – sogenannte Facetten (facets) – definieren.

Bezeichnung der Facette (facet)	Bedeutung
`xs:minInclusive`, `xs:minExclusive`	untere Schranke für Werte
`xs:maxInclusive`, `xs:maxExclusive`	obere Schranke für Werte
`xs:enumeration`	explizite Aufzählung der Werte
`xs:length`	erlaubte Länge/Anzahl
`xs:minLength`, `xs:maxLength`	Schranken für Länge/Anzahl
`xs:totalDigits`	Anzahl Ziffern
`xs:fractionDigits`	max. Anzahl Nachkommastellen
...	

Bemerkung: Nicht sämtliche Facetten lassen sich auf alle Datentypen anwenden – z. B. ist die Facette `xs:fractionDigits` für den Datentyp `xs:string` offensichtlich unsinnig!

Beispiele der Anwendung von Facetten

- String mit genau fünf Zeichen:

```
<xs:simpleType name="fixedString">
    <xs:restriction base="xs:string">
        <xs:length value="5"/>
    </xs:restriction>
</xs:simpleType>
```

- Zahl mit maximal zwei Nachkommastellen:

```
<xs:simpleType name="maxTwoFractions">
    <xs:restriction base="xs:decimal">
        <xs:fractionDigits value="2"/>
    </xs:restriction>
</xs:simpleType>
```

Beispiel: Werteliste als neuer Datentyp

```
<xs:simpleType name="fixedString"> s.o. </xs:simpleType>

<!-- Liste mit Zeichenketten der Laenge 5 -->
<xs:simpleType name="fixedStringList">
    <xs:list itemType="fixedString"/>
</xs:simpleType>

<!-- obige Liste mit genau 3 Elementen -->
<xs:simpleType name="fixedStringFixedList">
   <xs:restriction base="fixedStringList">
       <xs:length value="3"/>
   </xs:restriction>
</xs:simpleType>

<xs:element name="example" type="fixedStringFixedList"/>
```

Vereinigung von Basistypen

Beispiel: Vereinigung der Typen `fixedString` und `maxTwoFractions`

```
<xs:simpleType name="myUnion">
    <xs:union memberTypes="fixedString maxTwoFractions"/>
</xs:simpleType>

<xs:simpleType name="myUnionList">
    <xs:list itemType="myUnion"/>
</xs:simpleType>
```

Wie sehen mögliche Werte aus?

Typdeklaration mithilfe regulärer Ausdrücke

Als Facette ist außerdem der Wert `xs:pattern` möglich. Damit können die erlaubten Werte mithilfe regulärer Ausdrücke definiert werden.

> **Aufgabe:** Eine Preisangabe soll in XML notiert werden durch eine Währungsbezeichnung gefolgt von einem Leerzeichen und dem eigentlichen Preis.
>
> Als Währungsbezeichnung sind die Zeichenketten `EUR`, `USD` oder ein Währungssymbol erlaubt.
>
> Der Preis besteht aus mindestens einer Zahl vor dem *Komma* und genau zwei Nachkommastellen.

Frage: Wie sehen der entsprechende reguläre Ausdruck und die zugehörige Typdeklaration dafür aus?

Tipp: Ein Währungssymbol kann über die Sequenz `\p{Sc}` und eine Dezimalziffern kann direkt oder über `\p{Nd}` bezeichnet werden.

Lösung:

```
<xs:simpleType name="myPrice">
    <xs:restriction base="xs:string">
        <xs:pattern
        value="(EUR |USD |\p{Sc} )[0-9]+,[0-9]{2}"/>
    </xs:restriction>
</xs:simpleType>
```

Varianten

- Wie kann in `myPrice` ein Preis in Britischen Pfund oder japanischen Yen angegeben werden (außer in der Langform `GBP` oder `YEN`)?
- Wie lassen sich in `myPrice` „führende Nullen“ vermeiden?
- Was passiert, wenn der Typ `myPrice` von `xs:integer`, `xs:token` oder `xs:float` abgeleitet wird?

Umsetzung der Skript-DTD als XML-Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- file: script02.xsd -->
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <!-- <!ELEMENT abstract (#PCDATA)> -->
  <!-- <!ELEMENT para (#PCDATA)> -->
  <xs:element name="abstract" type="xs:string"/>
  <xs:element name="para" type="xs:string"/>

  <!-- <!ELEMENT chapter (para+)> -->
  <xs:element name="chapter">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="para"
            minOccurs="1" maxOccurs="unbounded"/>
      </xs:sequence>
```

```
    <!-- <!ATTLIST chapter heading CDATA #REQUIRED>
         <!ATTLIST chapter sparePages (1 | 2) #IMPLIED>
         <!ATTLIST chapter important (yes | no) "yes">
         <!ATTLIST chapter interest CDATA #FIXED "high"> -->
      <xs:attribute name="heading" use="required"/>
      <xs:attribute name="sparePages" type="pageValues"/>
      <xs:attribute name="important" default="yes">
        <xs:simpleType> <!-- anonymer Typ -->
          <xs:restriction base="xs:token">
            <xs:enumeration value="yes"/>
            <xs:enumeration value="no"/>
          </xs:restriction>
        </xs:simpleType>
      </xs:attribute>
      <xs:attribute name="interest" fixed="high"/>
    </xs:complexType>
  </xs:element>
```

```
<!-- spezieller Typ fuer das Attribut sparePages -->
<xs:simpleType name="pageValues">
  <xs:restriction base="xs:integer">
    <xs:minInclusive value="1"/>
    <xs:maxInclusive value="2"/>
  </xs:restriction>
</xs:simpleType>

<!-- <!ELEMENT courseDoc (abstract, chapter+)> -->
<xs:element name="courseDoc">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="abstract"/>
      <xs:element ref="chapter" maxOccurs="unbounded"/>
    </xs:sequence>
```

```
        <!-- <!ATTLIST courseDoc title CDATA #REQUIRED> -->
        <xs:attribute name="title" use="required"/>
      </xs:complexType>
    </xs:element>
</xs:schema>
```

Variante 1: Explizite Typdeklarationen

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- file: script02.xsd -->
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:simpleType name="abstractType">
    <xs:restriction base="xs:string"/>
  </xs:simpleType>

  <xs:simpleType name="paraType">
    <xs:restriction base="xs:string"/>
  </xs:simpleType>
```

```
<xs:complexType name="chapterType">
  <xs:sequence>
    <xs:element name="para" minOccurs="1"
      maxOccurs="unbounded" type="paraType"/>
  </xs:sequence>
  <xs:attribute name="heading" use="required"/>
  <xs:attribute name="sparePages" type="pageValues"/>
  <xs:attribute name="important" default="yes">
    <xs:simpleType>
      <xs:restriction base="xs:token">
        <xs:enumeration value="yes"/>
        <xs:enumeration value="no"/>
      </xs:restriction>
    </xs:simpleType>
  </xs:attribute>
  <xs:attribute name="interest" fixed="high"/>
</xs:complexType>
```

```
  <!-- spezieller Typ fuer das Attribut sparePages -->
  <xs:simpleType name="pageValues">...</xs:simpleType>

  <xs:element name="courseDoc">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="abstract" type="abstractType"/>
        <xs:element name="chapter" maxOccurs="unbounded"
          type="chapterType"/>
      </xs:sequence>
      <xs:attribute name="title" use="required"/>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Vorteil: Wurzelelement ist eindeutig!

6.2 Verwenden eines XML-Schemas

- Die bisher verwendeten Schemadeklarationen haben für das Schema keinen expliziten Namensraum definiert.

  Es wurde ein einfacher Standardnamensraum ohne Präfix verwendet!

- Daher kann bei der Verwendung der im Schema definierten Elemente auch kein Namensraum deklariert werden.

  Das Schema wird in dem Dokument über das Attribut `xsi:noNamespaceSchemaLocation` unter Angabe des Ablage-URLs identifiziert.

Beispiel: Einfaches Schema zum Testen des Datentyps `date`

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- file: testDate.xsd -->
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:element name="example" type="xs:date"/>
</xs:schema>
```

und ein passendes XML-Dokument:

```
<?xml version="1.0" encoding="UTF-8"?>
<example
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="testDate.xsd">
    2004-02-29
</example>
```

Beispiel: XML-Schema mit explizitem Namensraum

Das vorige Beispiel erweitert um einen expliziten Namensraum:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- file: testDateNamespace01.xsd -->
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
    targetNamespace="testDateNamespace01">
    <xs:element name="example" type="xs:date"/>
</xs:schema>
```

`targetNamespace` definiert den Namenssraum für das Element `example` dieses Schemas (hier kein URI).

Beispielvariante: XML-Schema mit explizitem Namensraum (Langform)

Bei der Festlegung des Namensraumes sind evtl. weitere Attribute nützlich:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- file: testDateNamespace01a.xsd -->
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
    targetNamespace="testDateNamespace01"
    xmlns="NamespaceInSchema"
    elementFormDefault="qualified"
    attributeFormDefault="unqualified">
    <xs:element name="example" type="xs:date"/>
</xs:schema>
```

Genauer . . .

`targetNamespace`: Namensraum für die durch `name` deklarierten Elemente

`xmlns`: Standardnamensraum für das Schema über den Elemente ohne Namenspräfix *innerhalb der Schemadeklaration* angesprochen werden.

(Kann auch weggelassen werden; wird in der Regel identisch zu `targetNamespace` gewählt)

`elementFormDefault`: legt fest, dass die definierten Elemente nur „qualifiziert“ über den Namensraum (`targetNamespace`) angesprochen werden dürfen

`attributeFormDefault`: dito für die Attribute

. . . und die entsprechende Verwendung

```
<?xml version="1.0" encoding="UTF-8"?>
<example
    xmlns="testDateNamespace01"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation=
        "testDateNamespace01 testDateNamespace01.xsd">
    2004-02-29
</example>
```

`xmlns`: Standardnamensraum wie im Schema deklariert

`xsi:schemaLocation`: Liste mit Wertepaaren *namespace uri* zum Auffinden der verwendeten Schemadeklarationen

. . . Variante mit explizitem Namensraum

```
<?xml version="1.0" encoding="UTF-8"?>
<test:example
    xmlns:test="testDateNamespace01"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation=
        "testDateNamespace01 testDateNamespace01.xsd">
    2004-02-29
</test:example>
```

XML-Schema: Referenzen + Namensraum (1)

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- file: testElementContentNamespace01.xsd -->
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  targetNamespace="Namespace02"
  xmlns="Namespace02">
  <xs:element name="one"/>
  <xs:element name="two"/>
  <xs:element name="example">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="one"/>
        <xs:element ref="two"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

XML-Schema: Referenzen + Namensraum (2)

Frage: Was passiert, wenn im vorigen Beispiel ein *expliziter* Namensraum eingeführt wird – statt des Standardnamensraumes z. B. `xmlns:test="testNamespace02"`?

Antwort: Die Namen der Referenzen `ref="one"` und `ref="two"` sind nicht mehr gültig!

$\rightarrow$ Auch dort muss der explizite Namenspräfix verwendet werden!

XML-Schema: Referenzen + Namensraum (3)

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- file: testElementContentNamespace02.xsd -->
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  targetNamespace="Namespace02"
  xmlns:test="Namespace02">
  <xs:element name="one"/><xs:element name="two"/>
  <xs:element name="example">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="test:one"/>
        <xs:element ref="test:two"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

6.3 Noch mehr zum XML-Schema

Hier werden weitere „Spezialitäten“ von XML-Schemata behandelt:

- Mehrfache Namensräume
- Erweiterung eigener Elementdeklarationen
- Primär- und Fremdschlüssel

Mehrfache Namensräume

Durch die Verwendung von Namensräumen lassen sich in einem XML-Schema Elemente eines anderen Schemas verwenden (auch wenn diese identisch bezeichnet sind).

> **Aufgabe:** In dem vorher definierten Schema mit dem Namensraum `Namespace02` soll die Referenz auf das Element `two` durch eine Referenz auf das Element `example` aus dem Namensraum `testDateNamespace01` ersetzt werden.

Vorgehen bei mehrfachen Namensräumen

1. Für die externen Elemente einen Namensraum (evtl. mit Präfix) festlegen
2. Die entsprechenden Schemata einbinden
3. Die externen Elemente (evtl. über den Präfix) ansprechen
4. Bei der Verwendung des Schemas: Auch die weiteren Schemata in der Liste `schemaLocation` bekannt machen.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- file: testElementContentNamespace03.xsd -->
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
    targetNamespace="Namespace03"
    xmlns="Namespace03"
    xmlns:date="testDateNamespace01">
    <xs:import namespace="testDateNamespace01"
        schemaLocation="testDateNamespace01.xsd"/>
    <xs:element name="one"/>
    <xs:element name="example">
        <xs:complexType>
            <xs:sequence>
                <xs:element ref="one"/>
                <xs:element ref="date:example"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
</xs:schema>
```

Verwenden des Schemas . . .

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- file: testElementContentNamespace03.xml -->
<example
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="Namespace03"
    xmlns:date="testDateNamespace01"
    xsi:schemaLocation=
        "Namespace03 testElementContentNamespace03.xsd
         testDateNamespace01 testDateNamespace01.xsd">
    <one>text</one>
    <date:example>2005-02-28</date:example>
</example>
```

Frage: Darf hier für den Namensraum `testDateNamespace01` ein anderer Präfix gewählt werden?

Beispiel: Mehrfache Namensräume im Skript-Schema

Aufgaben:

- Das Skript-Schema soll einen eigenen Namensraum `targetNamespace="http://www.courseDoc.de"` festlegen
- Außerdem sollen in den `para`-Elementen neben Text auch MATHML-Ausdrücke aus dem Namensraum `xmlns:m="http://www.w3.org/1998/Math/MathML"` erlaubt sein (Mixed Content).

```
<?xml version="1.0" encoding="UTF-8" ?>
<!-- file: script01_namespace.xsd -->

<xs:schema
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:m="http://www.w3.org/1998/Math/MathML"

  xmlns="http://www.courseDoc.de"
  targetNamespace="http://www.courseDoc.de"
  attributeFormDefault="unqualified">

  <!-- import schema declaration for MATHML -->
  <xs:import namespace="http://www.w3.org/1998/Math/MathML"
    schemaLocation=
    "http://www.w3.org/Math/XMLSchema/mathml2/mathml2.xsd"/>
```

```
  <!-- change para declaration to allow mixed content -->
  <xs:element name="para">
    <xs:complexType mixed="true">
      <xs:sequence minOccurs="0" maxOccurs="unbounded">
        <xs:element ref="m:math"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  ...
</xs:schema>
```

Beispiel einer dazu passenden Instanz

```
<?xml version="1.0" encoding="UTF-8" ?>
<!-- file: script01_namespace.xml -->
<s:courseDoc
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:m="http://www.w3.org/1998/Math/MathML"
  xmlns:s="http://www.courseDoc.de"
  xsi:schemaLocation=
    "http://www.courseDoc.de script01_namespace.xsd"

  title="Dies und das aus der Informatik">

  <s:abstract>Dinge, die man wissen sollte</s:abstract>
  <s:chapter heading="Erstes Kapitel" important="no">
    <s:para>Grundlagen</s:para>
    <s:para>Noch mehr</s:para>
  </s:chapter>
```

```
  <s:chapter heading="Nette Dinge">
    <s:para>Chromatische Zahl</s:para>

    <s:para>Eulersche Polyederformel: <m:math>
        <m:mi>p</m:mi><m:mo>+</m:mo><m:mi>f</m:mi>
        <m:mo>=</m:mo>
        <m:mi>q</m:mi><m:mo>+</m:mo><m:mi>2</m:mi>
      </m:math></s:para>

    <s:para><m:math>
        <m:mi>y</m:mi><m:mo>=</m:mo><m:mi>m</m:mi>
        <m:mo>&#8290;</m:mo> <!-- &InvisibleTimes; -->
        <m:mi>x</m:mi><m:mo>+</m:mo><m:mi>b</m:mi>
      </m:math></s:para>
  </s:chapter>
</s:courseDoc>
```

Abschließendes Beispiel zum Skript-Schema: Erweiterung und Eindeutigkeit von Elementen

Aufgaben:

1. Falls für ein Kapitel beliebte Prüfungsfragen bekannt sind, sollen diese in einem erweiterten `chapterExam`-Element zusätzlich zu den Standardangaben von `chapter` angegeben werden.

   Außerdem kann optional die Erfolgsquote für diesen Abschnitt notiert werden

2. Außerdem sollen die Überschriften der `chapter`-Elemente eindeutig sein.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- file: script02a.xsd -->
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

<xs:complexType name="chapterExamType">
  <xs:complexContent>
    <xs:extension base="chapterType">
      <xs:sequence>
        <xs:element name="question" minOccurs="1"
          maxOccurs="unbounded" type="paraType" />
      </xs:sequence>
      <xs:attribute name="success" use="optional"
        type="percentage"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

```
<xs:element name="courseDoc">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="abstract" type="abstractType"/>
       <xs:choice maxOccurs="unbounded">
  <xs:element name="chapter" type="chapterType"/>
  <xs:element name="chapterExam" type="chapterExamType"/>
       </xs:choice>
    </xs:sequence>
    <xs:attribute name="title" use="required"/>
  </xs:complexType>

  <xs:unique name="uniqueChapters">
    <xs:selector xpath="./chapter | ./chapterExam"/>
    <xs:field xpath="@heading"/>
  </xs:unique>
</xs:element>
```

Bemerkung: Neben der Eindeutigkeit von Attributen mittels `xs:unique` können über `xs:key` Schlüsselattribute definiert werden.

Bemerkung: Werden Schlüsselattribute in anderen Elementen referenziert, kann die referentielle Integrität innerhalb des XML-Dokuments über `xs:keyref` sichergestellt werden.

Bemerkung: Ausführliche Beispiele zur Verwendung von `xs:unique`, `xs:key` oder `xs:keyref` sind auf [Wor23h] zu finden.

Bemerkung: Die Abfragesprache XPath wird ausführlich in Kapitel 7 behandelt.

7 Suche in XML-Dokumenten mit XPath

- In XML-Dokumenten können (große) Datenmengen in strukturierter Form abgelegt werden
- Jedes XML-Dokument definiert eine baumartige Struktur der Daten
- Der Pfad von der Wurzel des Dokuments zu jedem Datum/Element ist eindeutig

Frage: Wie lassen sich Daten/Elemente mit bestimmten Eigenschaften innerhalb des Dokuments finden?

$\rightarrow$ Abfragesprache XPath verwenden!

Grundsätzliches zu XPath

- Ziel von XPath ist es, eine einheitliche Syntax und Semantik zur Abfrage in XML-Dokumenten zu schaffen;

  XPath-Ausdrücke werden z. B. benutzt in XSLT, XPointer oder in der dom-API (siehe Abschnitt 9.3).
- Jedes XML-Dokument wird als Baum interpretiert; dabei werden neben den Elementen auch die verwendeten Attribute, Texte, Kommentare, Verarbeitungsanweisungen und Namensräume als Knoten des Baumes betrachtet.
- Durch XPath-Ausdrücke wird beschrieben, wie ein XML-Dokument durchlaufen werden soll („Path“), um die gesuchten Knoten zu finden.
- XPath wird vom W3C spezifiziert (vgl. [Wor23g]).

Bemerkung: Jedes beliebige XML-Dokument lässt sich mithilfe von XPath durchsuchen.

Bemerkung: DTDs lassen sich *nicht* mittels XPath durchsuchen, da sie nicht in XML-Syntax notiert sind!

Wichtige Bezeichnungen

Die folgenden Begriffe werden in Bezug auf einen beliebigen, aber festen Knoten eines XML-Dokuments gebraucht:

Vorfahren („ancestor“): Menge der Knoten, die auf dem Weg von der Wurzel zu dem Knoten passiert werden

Nachkommen („descendant“): Menge der Knoten, für die der Knoten zu den Vorfahren gehört

Kind („child“): Direkter Nachkomme eines Knotens

Vater („parent“): Falls vorhanden, eindeutig bestimmter direkter Vorfahre

Vorgänger („preceding“): Menge der Knoten, die im XML-Dokument vorher notiert sind (mit Ausnahme der Vorfahren)

Nachfolger („following“): Menge der Knoten, die im XML-Dokument nachher notiert sind (mit Ausnahme der Nachkommen)

Geschwister („siblings“): Menge der Knoten, die Kinder desselben Vaters sind (ohne den betrachteten Knoten selbst)

Suchanfragen in XPath

Grundbaustein aller Suchanfragen ist der sogenannte Lokalisierungsschritt („location step“); er hat die Form

Suchachse`::`*Knotentest*`[`*Prädikat*`]...[`*Prädikat*`]`

mit den folgenden Bedeutungen:

Suchachse („axis“): richtet die Suche ausgehend vom aktuellen Knoten in einen bestimmten Teil des Baumes

Knotentest („node test“): beschränkt die Suche entlang der Achse auf bestimmte Knotentypen oder -namen

Prädikat („predicate“): boolescher Ausdruck zum Filtern der Ergebnismenge des Knotentests in Bezug auf die Achse (optional, aber beliebig viele Prädikate erlaubt)

7.1 Suchachsen

Bezeichnung der Achse	betroffene Knoten	Abkürzung
`self`	betrachteter Knoten	`.`
`parent`	Vater des Knotens	`..`
`child`	Menge der Kinder	weglassen
`attribute`	Attribute des Knotens	`@`
`namespace`	Namensräume des Knotens	
`ancestor`	Vorfahren des Knotens	
`ancestor-or-self`	Vorfahren oder selbst	
`descendant`	Nachkommen des Knotens	
`descendant-or-self`	Nachkommen oder selbst	`//`
`following`	Nachfolger des Knotens	
`following-sibling`	Nachfolger (nur Geschwister)	
`preceding`	Vorgänger des Knotens	
`preceding-sibling`	Vorgänger (nur Geschwister)	

Bemerkung: Attribute und Namensräume werden zwar im Dokumentenbaum als Knoten abgebildet, sie werden aber *nicht* über die Suchachse `child` erreicht, sondern besitzen jeweils eine eigene Achse.

7.2 Knotentests

Der Knotentest liefert in Bezug auf die angegebene Achse eine Knotenmenge als Ergebnis:

Bezeichnung des Knotentests	Beschränkung auf
name	Elemente mit entsprechendem Namen
`*`	sämtliche Elemente
`node()`	sämtliche Knoten
`text()`, `comment()`	sämtliche Textknoten bzw. Kommentare
`processing-instruction()`	sämtliche Verarbeitungsanweisungen

Bemerkung: Eine Beschränkung auf Verarbeitungsanweisungen mit einer bestimmten Bezeichnung ist durch `processing-instruction(`*`bezeichnung`*`)` möglich.

7.3 Prädikate

Durch Prädikate wird die Ergebnismenge gefiltert; es können vordefinierte Funktionen verwendet werden, z. B.

Bezeichnung der Funktion	Beschreibung
`=, !=, <, <=, ...`	Vergleichsoperatoren
`+, -, *, div, ...`	übliche arithmetische Operatoren
`true(), false(), not(...)`	boolesche Funktionen
`name()`	Name des Elements
`position()`	Index des Elements in Knotenmenge
`last()`	Anzahl Elemente in der Knotenmenge
`count(`*`set`*`)`	Anzahl Elemente in bestimmter Menge
`contains(`*`str, sub`*`)`	Auftreten einer Teilzeichenkette testen
`starts-with(`*`str, sub`*`)`	Anfang einer Zeichenkette testen

Bemerkung: Wird als Prädikat eine Nummer *num* angegeben, so ist es gleichwertig mit dem Ausdruck `position()=`*num*.

Kombination von Lokalisierungsschritten

- Lokalisierungsschritte können über das Zeichen / zu einem sogenannten Lokalisierungspfad („location path“) kombiniert werden
- In jedem Lokalisierungspfad bildet die Ergebnismenge des vorherigen Lokalisierungsschrittes die Ausgangsmenge des nachfolgenden Schrittes
- Beginnt ein Lokalisierungspfad mit einem /, so bildet die Wurzel des Dokuments die Ausgangsmenge des ersten Lokalisierungsschrittes (absoluter Pfad)
- Ansonsten ergibt sich die Ausgangsmenge des ersten Lokalisierungsschrittes aus dem Zusammenhang (relativer Pfad)

Beispiel: XML-Dokument ...

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- file: script03b.xml -->
<!DOCTYPE courseDoc SYSTEM "script02.dtd">
<courseDoc title="3. Testen von XML-Dokumenten">
    <abstract><!-- Abstr. -->Zusammenfassung...</abstract>
    <chapter heading="Erstes Kapitel">
        <para> Absatz eins</para>
        <para> Absatz zwei</para>
    </chapter>
    <chapter heading="Zweites Kapitel" sparePages="2">
        <para> Absatz eins</para>
    </chapter>
    <chapter heading="Drittes Kapitel">
        <!-- 3. Kapitel -->
        <para> Absatz eins</para>
    </chapter>
</courseDoc>
```

. . . und einige Suchausdrücke

1. Suche den `courseDoc`-Knoten:

   ```
   /descendant::courseDoc
   ```

2. Suche alle `para`-Knoten und gib deren Textinhalt aus:

   ```
   /descendant::para/child::text()
   ```

   dito, vereinfachte Form:

   ```
   //para/text()
   ```

3. Suche alle Kapitel mit mindestens zwei Absätzen:

   ```
   //chapter[count(para)>=2]
   ```

4. Suche die Knoten, die ein `abstract`-Element als Kind besitzen:

   `//*[abstract]`, `//*/abstract/..` oder `//abstract/..`

5. Bestimme alle im Dokument auftretenden Attribute:

   `/descendants::node()/attribute::*` oder `//@*`

6. Bestimme alle im Dokument auftretenden Attribute, die einen Titel (`title`) oder eine Überschrift (`heading`) bezeichnen:

   ```
   //attribute::*[name()="title" or name()="heading"]
   ```

7. Suche die Kapitel, die ein `sparePages`-Attribut besitzen:

   ```
   //chapter[@sparePages]
   ```

   ```
   //chapter/attribute::sparePages/..
   ```

   Bestimme deren Vorgänger- bzw. Nachfolger-Geschwister:

   ```
   //chapter[@sparePages]/preceding-sibling::*
   ```

   ```
   //chapter[@sparePages]/following-sibling::*
   ```

8. Bestimme die Anzahl der Kommentare im Dokument:

   ```
   count(//comment())
   ```

   Gib alle Kommentare der Wurzel aus;

   ```
   /comment()
   ```

9. Bestimme alle Kinder des dritten `chapter`-Elements:

   ```
   //chapter[position()=3]/node()
   ```

10. Bestimme von dem Vater des `abstract`-Knotens diejenigen Nachkommen zwischen der vierten und siebten Position, die einen `para`-Knoten als Kind besitzen:

    ```
    //abstract/parent::node()/descendant::*
    [position()>=4 and position()<=7][para]
    ```

11. Bestimme den Inhalt des letzten `para`-Knotens jedes Kapitels:

    ```
    //chapter/para[position()=last()]/text()
    ```

oder, wenn `para`-Knoten nur innerhalb von Kapiteln auftreten:

```
//*[self::para and position()=last()]/text()
```

12. Bestimme den Namen aller Elemente, die ein Attribut mit dem Wert `2` enthalten

    `//*[@*="2"]/name()` oder `//@*[.="2"]/../name()`

13. Suche alle Elemente, die den leeren String `""` als Namensraum besitzen

    ```
    //*[namespace-uri(.)=""]
    ```

14. Beachte den Unterschied von `*` (sämtliche **Elemente**) und `node()` (sämtliche **Knoten**):

    Vergleiche `/child::*` und `/child::node()`

Bemerkungen zu XPath-Ausdrücken

Bemerkung: Ergebnismengen können durch `|` zu einer neuen Menge zusammengefasst werden.

Beispiel: Bestimme alle Knoten der ersten und zweiten Ebene des Dokuments:

```
/* | /*/*
```

Bemerkung: Auch auf Ergebnismengen können Prädikate angewendet werden.

Beispiel: Bestimme alle Knoten der ersten und zweiten Ebene des Dokuments, deren Name ein „p“ enthält:

```
(/* | /*/*)[contains(name(),"p")]
```

Anwendung innerhalb von XML-Schema

- Sicherstellen der Eindeutigkeit (siehe Beispiel voriges Beispiel zu `xs:unique`)
- Definition von Primärschlüsseln (`xs:key`) und Fremdschlüsseln (`xs:keyref`)

8 Transformation von XML-Dokumenten mit XSLT

- XSLT ist eine Abkürzung für „XSL Transformations“, wobei XSL für „Extensible Stylesheet Language“ steht
- XSLT wird vom W3C spezifiziert (vgl. [Wor23j, Wor23k])
- XSLT stellt eine (Programmier-)Sprache zur automatischen Umwandlung von XML-Dokumenten in andere XML-Dokumente dar
- Bei der Umwandlung können z. B. die Struktur, die Markup-Regeln oder auch der Inhalt geändert werden

Beispiele hierfür sind:

- Aufbereiten der Inhalte eines XML-Dokumentes für eine HTML-Darstellung
- Extraktion von Daten aus einem XML-Dokument und deren Zusammenstellung in einem neuen Dokument
- Änderung der Darstellungform von Daten (z. B. ganzzahlige Werte in Gleitkommadarstellung überführen, Datumsformate anpassen etc.)

  Zur Erinnerung: XML dient auch als Datenaustauschformat – unterschiedliche XML-Anwendungen können die gleichen Daten nur in anderer Form enthalten.

- Zur ausführlichen Behandlung von XSLT siehe z. B. [Wor23j] oder [Fit04]

Arbeitsweise von XSLT

Ein XSLT-Prozessor bekommt als Eingaben

1. ein zu transformierendes Quelldokument („source tree“)
2. die Verarbeitungsregeln („XSLT Stylesheet“)

und produziert daraus ein Zieldokument („result tree“)

Beispiele für XSLT-Prozessoren:

- Saxon, Xalan (Java; allgemein)
- `xsltproc` unter Unix/Linux
- „gängige Web-Browser“ (z. B. Firefox für HTML als Zieldokument)

Genauer . . .

- Der XSLT-Prozessor kann als Zustandsmaschine aufgefasst werden, d. h. er befindet sich zu jedem Zeitpunkt in genau einem Zustand und es gibt Regeln, um ihn (deterministisch) von einem Zustand in einen anderen zu überführen
- Die Regeln werden im XSLT Stylesheet festgesetzt und enthalten Angaben über den Folgezustand und über den in das Zieldokument zu schreibenden Inhalt
- Gesteuert werden die Regeln vom aktuellen Zustand des XSLT-Prozessors und dem nächsten Element des zu transformierenden XML-Dokuments
- Die Bearbeitung kann in jedem Schritt rekursiv erfolgen

- Die einzelnen Regeln werden in sogenannten „Templates“ formuliert
- Die Auswahl des anzuwendenden Templates erfolgt über einen XPath-Ausdruck im Template, der auf das nächste Element im Source Tree passen muss
- In den XPath-Ausdrücken der Templates sind nur Suchachsen erlaubt, die im Suchbaum absteigen (`child`, `attribute`, `...`); Kommentare und Verarbeitungsanweisungen werden ignoriert
- Treffen XPath-Ausdrücke aus *mehreren* Templates auf das nächste Element des Source Trees zu, greifen Prioritätsregeln zur Auswahl („je genauer ein XPath-Ausdruck die Elemente spezifiziert, desto höher ist seine Priorität“ oder explizit zugewiesene Priorität)

Aufbau eines XSLT-Stylesheets

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:stylesheet
        xmlns:xs="http://www.w3.org/1999/XSL/Transform"
        version="1.0">
        <xs:output method="xml"/>

        <!-- Angabe der Templates -->
        <xs:template> ... </xs:template>
</xs:stylesheet>
```

`stylesheet`: Name des Wurzelelements in einem Stylesheet; alternativ ist der Name `transform` möglich

`xmlns`: Angabe des Namensraumes zur Identifikation des Dokuments als XSLT-Stylesheet

`version`: XSLT-Version; aktuell ist Version 3.0[6]

`output`: optionales erstes Unterelement beschreibt das Ausgabeformat (`xml`, `xhtml`[7], `html` oder `text`)

Beispiel: `<xs:output method="xml"/>`

`template`: Angabe der Transformationsregeln (Templates) für die zu transformierenden Knoten der Quelldatei.

Die Auswahl der Regel für den nächsten Knoten der Quelldatei erfolgt über einen entsprechenden XPath-Ausdruck:

```
<xs:template match="XPath-Ausdruck">
    ... Anweisungen
</xs:template>
```

[6] Version 3.0 vom 8. Juni 2017; Browser unterstützen teilweise nur Version 1.0!

[7] Ausgabeformat `xhtml` erst ab XSLT Version 2.0

8.1 Standardregeln

In XSLT sind für eine Reihe von Knotentypen Standardregeln vordefiniert, die das Entwickeln von Stylesheets vereinfachen sollen:

Wurzel: „Gib nichts aus, bearbeite alle Kinder“:

```
<xs:template match="/">
    <xs:apply-templates/>
</xs:template>
```

Bemerkung: Durch den Aufruf `apply-templates` wird die rekursive Anwendung der Templates auf jedes der Kinder des entsprechenden Knotens gestartet.

(Container-)Element: „Gib nichts aus, bearbeite alle Kinder“:

```
<xs:template match="*">
    <xs:apply-templates/>
</xs:template>
```

Attribut: „Gib nichts aus“:

```
<xs:template match="@*"/>
```

Text: „Gib den Text aus“:

```
<xs:template match="text()">
    <xs:value-of select="."/>
</xs:template>
```

Verarbeitungsanweisung: „Gib nichts aus“:

```
<xs:template match="processing-instruction()"/>
```

Kommentar: „Gib nichts aus“:

```
<xs:template match="comment()"/>
```

Bemerkung: Innerhalb eines XML-Dokuments wird ein anzuwendendes Stylesheet direkt über eine Verarbeitungsanweisung angegeben:

```
<?xml-stylesheet type="text/xsl" href="XSLT-Datei"?>
```

Bemerkung: Das Ergebnis der Transformation kann mithilfe eines XSLT-Prozessors kontrolliert werden (z. B. unter Unix mithilfe von `xsltproc`).

Erste Übungen zu XSLT

- Testen der Standardregeln z. B. anhand `script.xml`
- Wie lassen sich die Aktionen der Standardregeln für das Wurzelelement und die Containerelemente in *einem* Template zusammenfassen?
- Wie können auch die Kommentare der Eingabedatei ausgegeben werden?
- Wie wirken sich Änderungen der Ausgabemethode (`html` oder `text` statt `xml`) aus?[8]

[8] Tipp: Auf XML-Deklaration und XML-Kommentare achten.

8.2 Diverse XSLT-Sprachkonstrukte

Im folgenden soll aus dem bekannten XML-Dokument script.xml mithilfe von XSLT eine HTML-Darstellung gewonnen werden.

Dabei werden diverse XSLT-Sprachkonstrukte am Beispiel vorgestellt.

Zuerst muss das Zielformat festgelegt werden: HTML, xhtml oder xml?

Wie geht das?

HTML: Ausgabemethode entsprechend wählen:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:stylesheet version="1.0"
    xmlns:xs="http://www.w3.org/1999/XSL/Transform">
    <xs:output method="html"/>
    ...
</xs:stylesheet>
```

xhtml: Ausgabemethode `xml` wählen und vorgeschriebenen Namensraum für xhtml festlegen (wird in das Wurzelelement des Zieldokuments übernommen:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:stylesheet version="1.0"
    xmlns:xs="http://www.w3.org/1999/XSL/Transform"
    xmlns="http://www.w3.org/1999/xhtml">
    <xs:output method="xml"/> <!-- no xhtml in browsers!-->
    ...
</xs:stylesheet>
```

Rahmen der HTML-Seite erzeugen

- Bei Auftreten des Wurzelknotens der Quelldatei den Rahmen der HTML-Seite in die Zieldatei schreiben
- Titel der Seite aus `courseDoc`-Element lesen

```
<xs:template match="/">
  <html>
    <head>
      <title>
        <xs:value-of select="courseDoc/attribute::title"/>
      </title>
    </head>
    <body>
      <xs:apply-templates/>
    </body>
  </html>
</xs:template>
```

Umsetzen der restlichen Elemente ...

```
<xs:template match="courseDoc">
  <h1><xs:value-of select="@title"/></h1>
  <xs:apply-templates/>
</xs:template>

<xs:template match="abstract">
  <h2>Zusammenfassung</h2>
  <p>
    <xs:copy-of select="text()"/>
    <!-- alternativ: <xs:value-of select="text()"/> -->
  </p>
</xs:template>

<xs:template match="chapter">
  <h2><xs:value-of select="@heading"/></h2>
  <xs:apply-templates/>
</xs:template>
```

```
<xs:template match="para">
  <p>
    <xs:copy-of select="text()"/>
    <!-- alternativ: <xs:value-of select="."/> -->
  </p>
</xs:template>
```

- Der Inhalt eines Knotens der XML-Quelldatei wird durch

  `<xs:value-of select="`*XPath-Ausdruck*`"/>`

  bestimmt. Der XPath-Ausdruck bestimmt den Knoten.
- Ein Teilbaum des zur Quelldatei gehörenden Dokumentenbaumes wird durch

  `<xs:copy-of select="`*XPath-Ausdruck*`"/>`

  direkt in die Zieldatei kopiert. Der XPath-Ausdruck bestimmt die Wurzel des Teilbaumes.

Benannte Templates

- Bisher wurde die Anwendung eines Templates *implizit* über einen passenden XPath-Ausdruck gesteuert
- Es ist außerdem möglich, Templates mit einem Namen zu versehen, und diese Templates *explizit* aufzurufen:
  - dadurch kann mehrfach benötigter Code ausgelagert und zentral gewartet werden
  - rekursive Aufrufe sind auch mit benannten Templates möglich
  - Konzept ist mit Funktionsaufrufen in Programmiersprachen vergleichbar

- Die Deklaration eines benannten Templates geschieht durch die Anweisung

```
<xs:template name="foo">
  ... Anweisungen
</xs:template>
```

- Die Verwendung eines benannten Templates erfolgt durch

```
<xs:call-template name="foo"/>
```

Beispiel zu benannten Templates

Aufgabe: In die HTML-Seite des Skriptes sollen zwischen den Kapiteln waagerechte Trennlinien eingefügt werden.

Lösung: Lagere das Zeichnen der Linien in ein benanntes Template aus und rufe es bei Bedarf auf.

```
<!-- Deklaration des benannten Templates -->
<xs:template name="separator">
  <hr/> <!-- oder aufwändiger ... -->
</xs:template>

<!-- Verwendung des benannten Templates -->
<xs:template match="chapter">
  <h2><xs:value-of select="@heading"/></h2>
  <xs:apply-templates/>
  <xs:call-template name="separator"/>
</xs:template>
```

Textknoten

- Bisher wurde in die Zieldatei auszugebender Text einfach direkt in die entsprechenden Templates geschrieben.

  Nachteil: Leerzeichen werden komprimiert und Sonderzeichen (z. B. <) erscheinen in der Zieldatei nur durch ihre Entität!

  Abhilfe: Textknoten verwenden!

- Die Deklaration eines Textknotens geschieht durch die Anweisung

```
<xs:text>... hier steht der Text ...</xs:text>

<!-- mit Umsetzung von Sonderzeichen -->
<xs:text disable-output-escaping="yes">...</xs:text>
```

Beispiel zu Textknoten

```
<xs:template match="abstract">
  <h2>
    &lt;
    <xs:text disable-output-escaping="yes">
      &lt;Zusammenfassung&gt;
    </xs:text>
    &gt;
  </h2>
  <p><xs:copy-of select="text()"/></p>
  <xs:call-template name="separator"/>
</xs:template>
```

Frage: Wie sieht der Inhalt der Zieldatei bzw. die HTML-Ausgabe aus?

Element- und Attributknoten

- Bisher wurden die in die Zieldatei auszugebenden Element-Tags direkt in die entsprechenden Templates geschrieben.

  Nachteil: Tagnamen sind fest und können nicht dynamisch – z. B. abhängig vom Inhalt der Quelldatei – erzeugt werden!

  Abhilfe: Elementknoten verwenden!

- Die Deklaration eines XML-Elements in der Zieldatei geschieht durch die Anweisung

```
<xs:element name="foo">
  ... Inhalt des Elements foo in der Zieldatei
</xs:element>
```

- **Vorteile:** Durch die Verwendung von Elementknoten ...
  - wird automatisch die Wohlgeformtheit des Zieldokuments gesichert (d. h. es werden automatisch korrespondierende öffnende und schließende Tags erzeugt)
  - sind dynamische Tagnamen in Abhängigkeit der Daten im Quelldokument möglich.
- Ebenso lassen sich den Elementen in der Zieldatei Attribute hinzufügen

```
<xs:attribute name="bar">
  ... Wert des Attributs bar
</xs:attribute>
```

Beispiel zu Element- und Attributknoten (1)

Aufgabe: Die Ausgabe der HTML-Überschrift „Zusammenfassung“ soll zentriert erfolgen.

```
<xs:template match="abstract">
  <!-- <h2 align="center">Zusammenfassung</h2> -->
  <xs:element name="h2">
    <xs:attribute name="align">center</xs:attribute>
    <xs:text>Zusammenfassung</xs:text>
  </xs:element>
  <xs:element name="p">
    <xs:copy-of select="text()"/>
  </xs:element>
  <xs:call-template name="separator"/>
</xs:template>
```

Beispiel zu Element- und Attributknoten (2)

Bemerkung: Wenn XPATH-Ausdrücke, XSLT-Parameter, oder -Variablen außerhalb von `select`-, `match`- oder `test`-Attributen verwendet werden, müssen sie in geschweifte Klammern `{...}` gesetzt werden – ansonsten werden die Ausdrücke nicht ausgewertet sondern wörtlich übernommen.

Frage: Was bewirkt die folgende Änderung des Standard-templates für Containerelemente?

```
<xs:template match="*">
  <xs:element name="{name()}">
    <xs:apply-templates/>
  </xs:element>
</xs:template>
```

Bedingte Verarbeitung

Es sind zwei Arten der bedingten Verarbeitung von XSLT-Anweisungen möglich:

`if`-Anweisung:

```
<xs:if test="XPath-Prädikat">
  ... bedingt ausgeführte Anweisungen
</xs:if>
```

Auswahlanweisung:

```
<xs:choose>
  <xs:when test="Prädikat 1"> 1. Fall </xs:when>
  <xs:when test="Prädikat 2"> 2. Fall </xs:when>
  ... weitere Fälle
  <xs:otherwise> ... Standardfall </xs:otherwise>
</xs:choose>
```

Bemerkung: Die `if`-Anweisung in XSLT besitzt *keinen* `else`-Zweig!

Bemerkung: In der Auswahlanweisung wird der *erste* der angegebenen `when`-Zweige ausgeführt, dessen `test`-Bedingung erfüllt ist. Ist keine der Bedingungen erfüllt, so werden die Anweisungen im Standardzweig `otherwise` ausgeführt – sofern dieser vorhanden ist.

Beispiel zur bedingten Verarbeitung

Aufgaben:

1. Es sollen nur die „wichtigen“ Kapitel (abhängig vom `important`-Attribut) auf der HTML-Seite ausgegeben werden.
2. Falls ein `sparePages`-Attribut vorhanden ist, soll eine entsprechende Anzahl an Leerzeilen auf der HTML-Seite ausgegeben werden.

Lösung: Verwende `if`- bzw. Auswahlanweisungen!

Frage: Wie sehen die entsprechenden `test`-Bedingungen aus?

```
<xs:template match="chapter">
  <!-- unwichtige Kapitel auslassen -->
  <xs:if test="not(@important='no')">
    <h2><xs:value-of select="@heading"/></h2>
    <xs:apply-templates/>
    <!-- evtl. Platz für Notizen einbauen -->
    <xs:choose>
      <xs:when test="@sparePages=1">
        <br/><br/>
      </xs:when>
      <xs:when test="@sparePages=2">
        <br/><br/><br/><br/>
      </xs:when>
      <xs:otherwise/> <!-- kann auch weggelassen werden -->
    </xs:choose>
    <xs:call-template name="separator"/>
  </xs:if>
</xs:template>
```

Aufgabe: Die explizite Abfrage der Werte des `sparePages`-Attribut im vorigen Beispiel ist sehr unschön!

Wie lässt sich die Ausgabe der Leerzeilen in Abhängigkeit des `sparePages`-Wertes besser implementieren?

Leider gibt es in XSLT keine `while`-Schleifen!

Lösung: Rekursive Aufrufe von benannten Templates mit Parameterübergabe benutzen!

Frage: Wie werden Variablen/Parameter in XSLT verwendet?

Variablen und Parameterübergabe

- Variablen werden in XSLT durch folgenden Ausdruck deklariert

  ```
  <xs:variable name="foo">...Wertangabe...</xs:variable>
  ```

  bzw. deren Wert verwendet

  ```
  <xs:value-of select="$foo"/>
  ```

- Parameter für benannte Templates werden analog deklariert und verwendet

  ```
  <xs:param name="bar">default-Wert</xs:param>
  ```

  Die Parameterübergabe beim Aufruf des Templates erfolgt durch

  ```
  <xs:call-template name="myTemplate">
    <xs:with-param name="bar">Wert</xs:with-param>
  </xs:call-template>
  ```

Beispiel zur Verwendung von Parametern

```
<!-- Deklaration des "rekursiven Zähl-Templates" -->
<xs:template name="makeSparePages">
  <xs:param name="sparePages">0</xs:param>

  <xs:if test="$sparePages>0">
    <br/><br/>
    <xs:call-template name="makeSparePages">
      <xs:with-param name="sparePages">
        <xs:value-of select="($sparePages)-1"/>
      </xs:with-param>
    </xs:call-template>
  </xs:if>
</xs:template>
```

```
<!-- Verwendung des "rekursiven Zähl-Templates" -->
<xs:template match="chapter">
  <!-- unwichtige Kapitel auslassen -->
  <xs:if test="not(@important='no')">
    <h2><xs:value-of select="@heading"/></h2>
    <xs:apply-templates/>
    <!-- evtl.Platz für Notizen einbauen -->
    <xs:call-template name="makeSparePages">
      <xs:with-param name="sparePages">
        <xs:value-of select="@sparePages"/>
      </xs:with-param>
    </xs:call-template>
    <xs:call-template name="separator"/>
  </xs:if>
</xs:template>
```

Formatierung von Zahlen

Für die formatierte Ausgabe von Zahlen gibt es in XSLT das Element `number`:

```
<xs:number value="Ausdruck"/>
```

- Bedeutung: Der Wert *Ausdruck* des `value`-Attributs wird ausgewertet, zu einer Integerzahl gerundet und als Zeichenkette ausgegeben.
- Eine Formatierung der auszugebenden Zahl ist über das optionale Attribut `format` möglich

- Ein innerhalb des Attributs auftretendes „Formatierungstoken“ bestimmt das Nummerierungsschema des Zahlenwertes:

Formatierungstoken	Nummerierungsschema
`1`	1, 2, 3, 4, ...
`0`	0, 1, 2, 3, ...
`01`	01, 02, 03, 04, ..., 09, 10, ...
`I`	I, II, III, IV, ...
`i`	i, ii, iii, iv, ...
`A`	A, B, C, D, ...
`a`	a, b, c, d, ...
...	

- Wird keine explizite Formatierung vorgegeben, so wird `format="1"` angenommen

Beispiel:

Ausgabe eines Zahlenwertes (mit mindestens drei Stellen):

```
<xs:number value="93" format="001 "/>
```

Iterationen über Mengen

```
<xs:for-each select="XPath-Ausdruck">
   ... Anweisungen für jedes Element der Menge
</xs:for-each>
```

Innerhalb der `for-each`-Anweisung kann der durch das jeweilige Element der Menge als Wurzel gebildete Teilbaum wiederum mit XPath-Ausdrücken durchsucht werden.

Aufgabe: Es sollen die Kapitel der Quelldatei gesucht und deren Überschriften in einem separaten Inhaltsverzeichnis in HTML-Code ausgegeben werden.

Lösung: Variante (1)

```
<xs:template match="courseDoc">
  ...
  <xs:variable name="chapters" select="//chapter"/>
  <xs:element name="h2">Inhalt</xs:element>
  <xs:for-each select="$chapters">
    <xs:element name="p">
      <xs:number value="position()" format="1. "/>
      <xs:value-of select="@heading"/>
    </xs:element>
  </xs:for-each>
  ...
</xs:template>
```

Bemerkungen zur Variante (1)

An der vorigen Lösung fällt auf, dass die Inhalte der Kapitel implizit über die Anwendung einer entsprechenden Regel geschrieben werden, während die Kapitelangaben im Inhaltsverzeichnis durch explizite Anweisungen innerhalb der Regel für den `courseDoc`-Knoten geschehen.

Diese unterschiedlichen Verfahrensweisen im Umgang mit Kapitelknoten sind unschön!

Besser wäre ein einheitliches Konzept, d. h. eine Regel für Kapitel wie bisher, die die Angaben als ausführlichen Text ausgibt, und eine weitere Regel für Kapitel, die die Angaben für das Inhaltsverzeichnis erzeugt.

Frage: Wie lässt sich solch ein Ansatz in XSLT realisieren?

Antwort: Modi verwenden!

Dazu Deklaration und Aufruf der Templates jeweils um das Attribut `mode` erweitern und alle Angaben zum Inhaltsverzeichnis vom `courseDoc`- in das neue `chapter`-Template auslagern.

Lösung: Variante (2)

```
<xs:template match="chapter" mode="toc">
  <!-- Kapitelangaben für Inhaltsverzeichnis ausgeben -->
  <xs:if test="position()=1">
    <xs:element name="h2">Inhalt</xs:element>
  </xs:if>
  <xs:element name="p">
    <xs:number value="position()" format="1. "/>
      <xs:value-of select="@heading"/>
  </xs:element>
  <xs:if test="position()=last()">
    <xs:call-template name="separator"/>
  </xs:if>
</xs:template>
```

```
<xs:template match="chapter"><!-- default mode -->
                             <!-- wie bisher   -->
  <!-- Kapitelangaben für ausführlichen Text ausgeben -->
  <xs:element name="h2">
    <xs:value-of select="@heading"/>
  </xs:element>
  <xs:apply-templates/>
  <!-- evtl.Platz für Notizen einbauen -->
  <xs:call-template name="makeSparePages">
    <xs:with-param name="sparePages">
      <xs:value-of select="@sparePages"/>
    </xs:with-param>
  </xs:call-template>
  <xs:call-template name="separator"/>
</xs:template>
```

Einschränkung von `apply-templates`

Die Anweisung `apply-templates` kann gezielt nur auf bestimmte Elemente angewendet werden:

```
<xs:apply-templates select="XPath-Ausdruck">
```

Nur die „wichtigen“ Kapitel verarbeiten

```
<xs:template match="courseDoc">
  <!-- Verarbeiten des Wurzelelements -->
  <xs:element name="h1">
    <xs:value-of select="@title"/>
  </xs:element>
  <xs:call-template name="separator"/>

  <xs:apply-templates select="//abstract"/>

  <!-- Kapitel suchen, dabei unwichtige auslassen -->
  <xs:variable name="chapters"
    select="//chapter[not(@important='no')]"/>
  <xs:apply-templates select="$chapters" mode="toc"/>
  <xs:apply-templates select="$chapters"/>
</xs:template>
```

Rechnen in XSLT

Aufgabe: Am Ende der Zusammenfassung soll die Anzahl der Kapitel, die Summe der vorgesehenen Leerseiten (`sparePages`) und die durchschnittliche Anzahl Absätze pro Kapitel ausgegeben werden.

- Die Elemente einer Menge lassen sich durch `count(`*`XPath-Ausdruck`*`)` zählen
- Die Werte von Knoten des Dokumentenbaumes lassen sich durch `sum(`*`XPath-Ausdruck`*`)` aufsummieren
- Es gelten die arithmetischen Operationen aus XPATH

- Gleitkommazahlen können über die Funktion `format-number` formatiert werden:

  `format-number(`*`number`*`, `*`pattern`*`)`

  `format-number(`*`number`*`, `*`pattern`*`, `*`specialFormat`*`)`

  number: zu formatierende Gleitkommazahl

  pattern: Muster zur Formatierung (analog zur Java-Klasse `DecimalFormat`)

  specialFormat (optional): Bezeichnung eines vom Standard abweichenden Formats, das z. B. die Zeichen für das Dezimalkomma und die Tausendergruppierung neu festlegt (siehe folgendes Beispiel).

Kleines Statistikbeispiel

```
<xs:decimal-format name="myFormat" decimal-separator=","
                   grouping-separator="."/>

<xs:template name="statistics">
  <xs:comment>Es folgen Statistikdaten</xs:comment>
  <xs:element name="p">
    Anzahl der in allen Kapiteln vorgesehenen Leerseiten:
      <xs:number value="sum(//chapter/@sparePages)"/>
  </xs:element>
  <xs:element name="p">
    Durchschnittliche Absatzanzahl pro Kapitel:
      <xs:value-of
        select="format-number(count(//para)
          div count(//chapter), '0,00', 'myFormat')"/>
  </xs:element>
</xs:template>
```

Sortieren von Mengen

- Vor der Abarbeitung durch `apply-templates` oder `for-each` können die betroffenen Elemente durch folgende Anweisung sortiert werden:

  ```
  <xs:sort select="XPath-Ausdruck"/>
  ```

- Durch den XPath-Ausdruck wird der Sortierschlüssel bestimmt;

  durch aufeinander folgende `sort`-Anweisungen können mehrstufige Sortierschlüssel definiert werden

- Durch das Attribut `order` kann die Sortierreihenfolge festgelegt werden (`ascending` oder `descending`)
- Weitere Attribute zur Steuerung der Sortierung siehe z. B. [Wor23j] oder [Fit04]

Beispiel zur Sortierung

Die Kapitel des Skripts sollen nach ihrer Überschrift alphabetisch sortiert ausgegeben werden:

```
<xs:template match="courseDoc">
  ...
  <!-- Kapitel suchen, dabei unwichtige auslassen -->
  <xs:variable name="chapters"
    select="//chapter[not(@important='no')]"/>
  <xs:apply-templates select="$chapters" mode="toc">
    <xs:sort select="@heading"/>
  </xs:apply-templates>
  <xs:apply-templates select="$chapters">
    <xs:sort select="@heading"/>
  </xs:apply-templates>
</xs:template>
```

Beispiel: HTML-Ausgabe eines Skript-Dokuments mit MATHML-Ausdrücken

1. Vorbereiten der XML-Datei durch Einfügen der Verarbeitungsanweisung:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!-- file: script_namespace.xml -->
<?xml-stylesheet type="text/xsl"
                 href="trans06_namespace.xsl"?>
<s:courseDoc xmlns:s="http://www.courseDoc.de"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation=
    "http://www.courseDoc.de script01_namespace.xsd"
  xmlns:m="http://www.w3.org/1998/Math/MathML"
  title="Dies und das aus der Informatik">
  ...
</s:courseDoc>
```

2. Anpassen der XSLT-Datei:

   - Deklaration der benötigten Namensräume
   - Expl. Namensraum in XPath-Ausdrücke aufnehmen
   - `xs:copy-of` statt `xs:value-of` für `para` (Warum?)

```
<xs:stylesheet version="1.0"
    xmlns:xs="http://www.w3.org/1999/XSL/Transform"
    xmlns="http://www.w3.org/1999/xhtml"
    xmlns:m="http://www.w3.org/1998/Math/MathML"
    xmlns:s="http://www.courseDoc.de">
    ...
    <xs:template match="s:para">
        <p> <!-- <xs:value-of select="text()"/>-->
            <xs:copy-of select="node()"/> </p>
    </xs:template>
    ...
</xs:stylesheet>
```

9 Programmierschnittstellen für XML

XML-Dokumente lassen sich sehr einfach automatisch verarbeiten:

- XML ist sehr gut dokumentiert; alle Spezifikationen sind öffentlich zugänglich [Wor23a]
- Die XML-Syntax ist einfach zu verstehen und zu parsen
- Das Einhalten der Regeln zur Wohlgeformtheit und Gültigkeit von XML-Dokumenten wird durch viele Tools im Vorfeld überprüft; dadurch kann der notwendige Programmieraufwand zur Überprüfung derartiger Fehler reduziert werden.

Beschränkungen des Einsatzes von XML

XML sollte trotz der einfachen automatischen Verarbeitung nur dann eingesetzt werden, wenn die Anforderungen nicht den XML inhärenten Beschränkungen widersprechen:

- XML ist nicht für Zugriffsgeschwindigkeit optimiert:
  - XML-Dokumente sollen erst vollständig geladen und dann verarbeitet werden
  - Bei jeder Verarbeitung soll eine Syntaxüberprüfung durchgeführt werden
- XML-Dokumente sind nicht „kompakt“; XML-Parser erwarten unkomprimierte Textdokumente als Eingabe
- XML ist für strukturierte Textdaten und nicht für die Behandlung von Binärdaten gedacht (oder doch?)

Verarbeitungstechniken für XML-Dokumente

Zur automatischen Verarbeitung von XML-Dokumenten werden am häufigsten die folgenden Techniken eingesetzt:

Ereignisströme („event streams“): Aus dem XML-Dokument wird ein sequentieller Strom von Tokens erzeugt, die in der Reihenfolge ihres Auftretens verarbeitet werden müssen (vgl. 9.1 und 9.2).

Objektbäume („object trees“): Die Bestandteile des XML-Dokuments werden in einer baumartigen Datenstruktur abgelegt; dieser Baum kann mit entsprechenden Methoden durchlaufen werden (vgl. 9.3).

„Data binding“: Jedes Element des XML-Dokuments wird auf eine entsprechende Instanz eines Objektes abgebildet.

Dadurch kann auf die Daten direkt mit den Mitteln der Programmiersprache zugegriffen werden, ohne die XML-Strukturen berücksichtigen zu müssen (vgl. 9.4).

Zu jeder dieser Techniken wird im folgenden eine Umsetzung beispielhaft für die Programmiersprache Java vorgestellt.

9.1 SAX („Simple API for XML“)

- SAX ist eine Programmierschnittstelle zur ereignisgesteuerten Verarbeitung von XML-Dokumenten
- SAX gilt als „Quasi-Standard“ und ist frei verfügbar; SAX wurde von David Megginson veröffentlicht [Meg23]
- SAX wurde ursprünglich für die Programmiersprache Java entwickelt;

  es gibt auch Implementierungen in Python, Perl, Pascal, C/C++, Visual Basic, . . .
- SAX Version 1.0 wurde im Januar 1998 veröffentlicht; aktuell ist die Version 2.0.2 vom April 2004 (diese Version wird im folgenden verwendet)

Arbeitsweise von SAX

- Ein SAX-Treiber („SAX driver“) liest das zu verarbeitende XML-Dokument sequentiell, zerlegt es mithilfe eines Parsers in seine Bestandteile und erzeugt daraus eine Folge von Ereignissen (z. B. „Start-Tag gelesen“, „End-Tag gelesen“, „White spaces gelesen“, „Text gelesen“, . . .)
- Beim Eintreten jedes Ereignisses wird eine durch den Anwendungsprogrammierer zur Verfügung gestellte Methode aufgerufen („Call-back-Mechanismus“), die das Ereignis bzw. die damit zusammenhängenden Daten verarbeitet

- Durch die rein sequentielle Verarbeitung der Eingabe braucht der Treiber keine Informationen zwischenzuspeichern („zustandslose Verarbeitung“)

  $\rightarrow$ geringer Speicherbedarf für den Treiber auch bei sehr großen Dokumenten

- Alle Ereignisse müssen in der Reihenfolge ihres Auftretens verarbeitet werden; es findet kein „look-ahead“ oder „look-behind“ statt.

- Durch diese sequentielle, zustandslose Verarbeitung kann der Treiber seine Eingabe nicht nur aus einer XML-Datei, sondern z. B. auch direkt von der Ausgabe einer anderen Anwendung beziehen (Pipeline-Verarbeitung)

- In SAX werden die für diese Arbeitsweise benötigten Interfaces definiert; die wichtigsten sind
  - `XMLReader`: Interface des SAX-Treibers zur Verarbeitung von XML-Daten;
    es gibt verschiedene Implementierungen, die untereinander austauschbar sind/sein sollen (z. B. Apache Xerces, Apache Crimson, Oracle, GNU Aelfred, . . .)
  - `Locator`: enthält Methoden, um dem aufgetretenen Ereignis die Position innerhalb der Eingabe zuordnen zu können
  - `Attributes`: Methoden zum Zugriff auf Attribute
  - `ContentHandler`: Call-back-Interface, das die Methoden zur Verarbeitung der auftretenden Ereignisse implementiert (vom Anwender zu implementieren)

  - `ErrorHandler`: analog für Fehlerereignisse
  - `DTDHandler`: analog für DTD bezogene Ereignisse
  - `EntityResolver`: analog für den Zugriff auf externe Entitäten

- Eine vollständige SAX-Implementierung ist z. B. in der „Java Platform Standard Edition (JDK)“ ab JDK 5.0 enthalten (ansonsten das Paket „Java API for XML Processing (JAXP)“ installieren)

Anwendung von SAX

Um SAX zur Verarbeitung eines XML-Dokuments zu verwenden, sind im wesentlichen vier Schritte notwendig:

1. Call-back-Interfaces implementieren
2. Instanz eines SAX-Treibers erzeugen
3. Call-back-Interface beim Treiber registrieren
4. Verarbeitung der XML-Eingabe starten

Call-back-Interfaces implementieren

Beachte: Jede der im folgenden aufgeführten Methoden liefert den Wert `void` zurück und kann bei der Abarbeitung eine `SAXException` werfen.

- `startDocument()`: Benachrichtigung über den Anfang des XML-Dokuments
- `endDocument()`: dito für das Ende des Dokuments

- `startElement(String uri, String localName, String qName, Attributes atts)`: Benachrichtigung über ein Start-Tag mit der Angabe eines eventuellen Namensraumes (`uri`), dem Namen des Elements ohne und mit Präfix (`localName` bzw. `qName`) und eventuell vorhandenen Attributen (`atts`).

  Das Interface `Attributes` stellt Methoden zum Zugriff auf die Attribute zur Verfügung, z. B. `getValue(int index)`, `getType(int index)`, ...

- `endElement(String uri, String localName, String qName)`: dito für ein Ende-Tag

- `characters(char[] ch, int start, int length)`: Benachrichtigung über das Einlesen eines Blocks von Zeichen (character data); die aktuell gelesenen `length` Zeichen sind im Array `ch` ab der Position `start` aufgeführt
- `ignorableWhitespace(char[] ch, int start, int length)`: dito für nicht signifikante Leerzeichen
  Achtung: Diese Methode muss nur von validierenden Parsern verwendet werden!
- `startPrefixMapping(String prefix, String uri)`: Anfang des Gültigkeitsbereiches eines Präfix' für einen Namensraum
- `endPrefixMapping(String prefix)`: dito für das Ende

- `processingInstruction(String target, String data)`: Benachrichtigung über das Einlesen einer Verarbeitungsanweisung
- `skippedEntity(String name)`: Benachrichtigung für eine nicht berücksichtigte Entität (z. B. falls der Parser keine externen DTDs einliest)
- `setDocumentLocator(Locator locator)`: Methode zum Setzen eines `Locator`-Objektes, über das die aktuelle Position des nächsten Ereignisses im Eingabestrom bestimmt werden kann; diese Methode wird *vor* allen anderen Methoden des Interface aufgerufen.

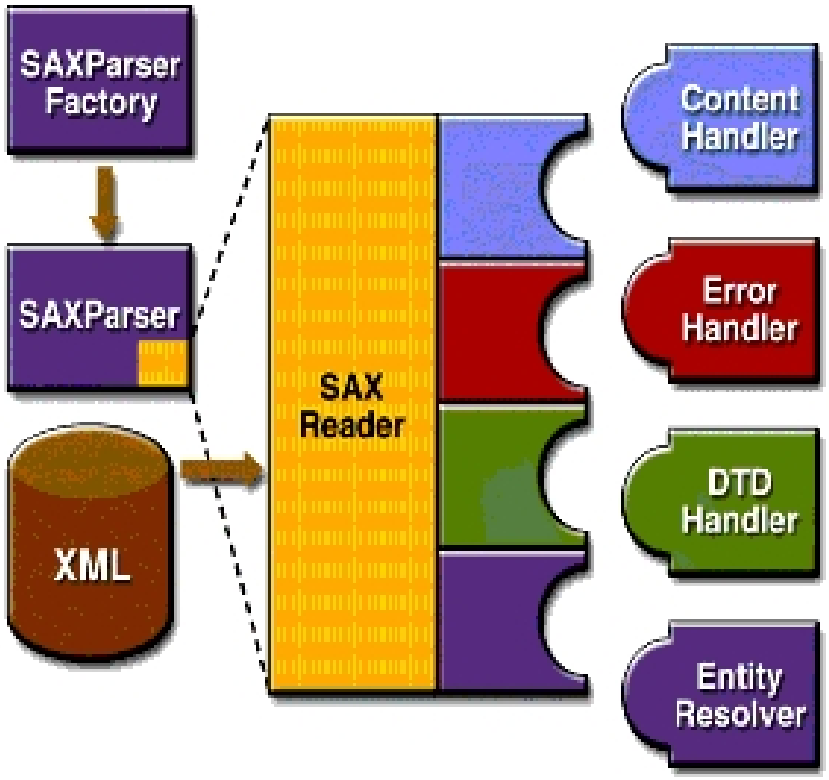
Für die anderen Call-back-Interfaces siehe die API-Dokumentation.

Standardimplementierung der Call-back-Interfaces

In der Klasse `org.xml.sax.helpers.DefaultHandler` sind Standardimplementierungen aller Methoden der Schnittstellen `ContentHandler`, `DTDHandler`, `EntityResolver` und `ErrorHandler` vorhanden.

Wenn von dieser Klasse abgeleitet wird, brauchen nur die konkret benötigten Methoden überschrieben zu werden.

Anwendung der SAX-APIs im Überblick



Schematische Darstellung zur Anwendung der SAX-APIs (vgl. [Ora23a])

Konkretes Beispiel

Aufgabe: Unter Verwendung von SAX soll ein Java-Programm für `courseDoc` entwickelt werden, dass ...

- Anfang und Ende der XML-Verarbeitung meldet,
- für jedes Element das Start- und Ende-Tag meldet,
- für jedes Start-Tag außerdem die vorhandenen Attribute mit Namen und Wert ausgibt,
- das Auftreten von nicht signifikanten Leerzeichen sowie von Zeichenketten meldet
- die Vorkommen aller `para`-Elemente innerhalb des Dokuments zählt und am Ende der Verarbeitung ausgibt.

Lösung: Erstelle eine Klasse `Inspector`, die von `DefaultHandler` abgeleitet ist und die Methoden für die geforderten Ereignisse überschreibt

```
public class Inspector extends DefaultHandler {
  private int paraCount;

  /* methods of org.xml.sax.ContentHandler */
  public void startDocument() throws SAXException {
    System.out.println("Start document detected");
    paraCount = 0;
  }

  public void endDocument() throws SAXException {
    System.out.println("End document detected; " +
      "total number of para elements found: " + paraCount);
  }
```

```
public void startElement(String uri, String localName,
    String qName, Attributes attributes)
    throws SAXException {
  System.out.println("start of element detected: " +
                     qName);
  // Attribute ausgeben
  if (attributes.getLength() > 0) {
    System.out.print("containing attributes:");
    for (int i = 0; i < attributes.getLength(); i++) {
      System.out.print(" " + attributes.getQName(i) +
        "=" + attributes.getValue(i));
    }
    System.out.println();
  }
  // evtl. para-Zaehler erhoehen
  if (localName.equals("para")) paraCount++;
}
```

```
  public void endElement(String uri, String localName,
      String qName) throws SAXException {
    System.out.println("end of element detected: " +
                       qName);
  }

  public void ignorableWhitespace(char[] ch, int start,
      int length) throws SAXException {
    System.out.println("Ignorable white spaces detected");
  }

  public void characters(char[] ch, int start, int length)
      throws SAXException {
    System.out.println("Charcters detected: "
        + new String(ch, start, length));
  }
}
```

Erzeugen eines SAX-Treibers

- Erzeugen eines SAX-Treibers über eine Fabrik[9]:

```
SAXParserFactory saxParserFactory =
                    SAXParserFactory.newInstance();
saxParserFactory.setNamespaceAware(true);
saxParserFactory.setValidating(true);
SAXParser parser = saxParserFactory.newSAXParser();
XMLReader xmlReader = parser.getXMLReader();
```

- Eine Instanz von `Inspector` muss nun als Call-back-Implementierung beim SAX-Treiber registriert werden:

```
ContentHandler contentHandler = new Inspector();
xmlReader.setContentHandler(contentHandler);
```

[9] Zum Entwurfsmuster „Fabrik“ („Factory“) siehe z. B. [GHJV96], [Lar98] oder [Bal01].

Festlegen der Eingabe des SAX-Treibers

- Das Starten der XML-Verarbeitung erfolgt über die Methode `parse(`*`InputSource`*`)` des SAX-Treibers
- Die Datenquelle des SAX-Treibers wird durch eine Instanz der Klasse `org.xml.sax.InputSource` festgelegt.
- Für die konkrete Quelle gibt es mehrere Möglichkeiten, die z. B. über entsprechende `set`-Methoden von `InputSource` spezifiziert werden:
  - `setByteStream(InputStream byteStream)`: Die Datenquelle als byteorientierter Datenstrom
  - `setCharacterStream(Reader characterStream)`: Die Datenquelle als zeichenorientierter Datenstrom

  - `setSystemId(String systemId)`: „system identifier“ der Datenquelle – z. B. der URL der XML-Datei; Diese Angabe wird nur verwendet, wenn keine Datenströme gesetzt sind.
  - `setPublicId(String publicId)`: Optionale Angabe des „formal public identifiers“ der Datenquelle
- Alternativ zu den `set`-Methoden kann die Datenquelle auch direkt im Konstruktor von `InputSource` übergeben werden:
  - `new InputSource(InputStream byteStream)`
  - `new InputSource(Reader characterStream)`
  - `new InputSource(String systemId)`

Bemerkung: Die Java `io`-Klassen für Eingabeströme (`InputStream` für Byteströme bzw. `Reader` für Zeichenströme) müssen vor der Übergabe an die `parse`-Methode des SAX-Treibers immer durch die Klasse `InputSource` gekapselt werden!

Starten der SAX-Verarbeitung

Alle vier Schritte zur Anwendung von SAX zusammengefasst:

```
public static void main(String[] args) throws Exception {
  // 1. Instanz des ContentHandlers erzeugen
  ContentHandler contentHandler = new Inspector();

  // 2. SAX-Treiber erzeugen
  saxParserFactory = SAXParserFactory.newInstance();
  saxParserFactory.setNamespaceAware(true);
  saxParserFactory.setValidating(true);
  SAXParser parser = saxParserFactory.newSAXParser();
  XMLReader xmlReader = parser.getXMLReader();

  // 3. Call-back für ContentHandler setzen
  xmlReader.setContentHandler(contentHandler);

  // 4. Verarbeitung starten
  xmlReader.parse(args[0]);
}
```

Variante der `Inspector`-Klasse

Aufgabe: Beim Auftreten von Start-Tags und von nicht signifikanten Leerzeichen sollen auch deren Zeile und Spalte innerhalb der XML-Quelle ausgegeben werden.

Lösung: Das `Locator`-Interface verwenden!

- `org.xml.sax.Locator` stellt Methoden zum Auslesen der aktuellen Position innerhalb des Eingabestrom zur Verfügung; z. B.

  ```
  int getLineNumber(), int getColumnNumber()
  ```

- Dazu muss die Methode `setDocumentLocator(Locator locator)` im `ContentHandler` entsprechend implementiert sein.

Erweitern der `Inspector`-Klasse

```
public class Inspector extends DefaultHandler {
  private Locator locator;
  ...
  public void setDocumentLocator(Locator locator) {
    this.locator = locator;
  }
  ...
  public void ignorableWhitespace(char[] ch, int start, int l
      throws SAXException {
    System.out.println("Ignorable white spaces detected");
    System.out.println("Location is at line " +
    locator.getLineNumber() + ", column " +
    locator.getColumnNumber());
  }
  ...
}
```

Features

- Die Anwendung kann spezielle Eigenschaften („features“) des SAX-Treibers ein- und ausschalten resp. das Vorhandensein auslesen: `setFeature(`*`feature`*`,` *`bool`*`)` bzw. `boolean getFeature(`*`feature`*`)`
- Es sind unter anderem folgende Features definiert:

  http://xml.org/sax/features/validation: SAX-Parser soll die Quelle gegen eine DTD validieren (default: `false`)

  http://xml.org/sax/features/namespaces: Namensräume sollen berücksichtigt werden (default: `true`)
- Bei unbekannten oder nicht verfügbaren Features werden entsprechende Exceptions geworfen!

Beispiel zu Features

Das folgende Beispiel ist [Meg23] entnommen:

```
try {
  String id = "http://xml.org/sax/features/validation";
  if (xmlReader.getFeature(id)) {
    System.out.println("Parser is validating.");
  } else {
    System.out.println("Parser is not validating.");
  }
} catch (SAXNotRecognizedException e) {
  System.out.println("Can't tell.");
} catch (SAXNotSupportedException e) {
  System.out.println("Wrong time to ask.");
}
```

Validierung gegen ein Schema

Das Feature zur Validierung verwendet standardmäßig die im Dokument deklarierte DTD (intern oder extern)!

Zur Validierung gegen ein im Dokument deklariertes XML-Schema muss im Parser eine spezielle Property gesetzt werden:

```
parser.setProperty(
  "http://java.sun.com/xml/jaxp/properties/schemaLanguage",
  "http://www.w3.org/2001/XMLSchema");

xmlReader = parser.getXMLReader();

... (use xmlReader as before)
```

ErrorHandler

- Zur Reaktion auf Fehlerzustände kann beim SAX-Treiber eine Instanz des Interface `org.xml.sax.ErrorHandler` registriert werden: `setErrorHandler(`*`errorHandler`*`)`
- Dazu müssen folgende Methoden implementiert werden:
  - `fatalError(SAXParseException exception)`: Es ist ein schwerwiegender Fehler aufgetreten – z. B. ein Fehler in der Wohlgeformtheit der XML-Quelle
  - `error(SAXParseException exception)`: Der Sax-Treiber hat z. B. einen Fehler bei der Validierung des Dokuments festgestellt
  - `warning(SAXParseException exception)`: Der Sax-Treiber gibt eine Warnung aus, die keinen Abbruch der Abarbeitung erzwingt (Beispiel ?)

Beispiele zur Verwendung des ErrorHandlers

- Verarbeitung einer nicht wohlgeformten XML-Datei: Was passiert?
- Abweichung des XML-Dokuments von der DTD – z. B. durch Einfügen eines zusätzlichen Attributs in einem Start-Tag:

  Wie verhält sich der SAX-Treiber mit bzw. ohne `validation`-Feature?

Zusammenfassung der SAX-Arbeitsweise

- SAX arbeitet ereignisgesteuert, d. h. der SAX-Treiber und nicht die eigentliche Anwendung steuert den Ablauf der XML-Verarbeitung
- die Anwendung wird vom Eintreten bestimmter Ereignisse benachrichtigt (Observer Pattern; siehe [GHJV96])
- der SAX-Parser „schiebt“ die XML-Daten über den Callback-Mechanismus in die Anwendung (Push Parser)

Eventuelle Nachteile von SAX

Die Verwendung von SAX ist etwas umständlich, wenn . . .

- die Anwendung die Verarbeitung selbst steuern möchte.

  Abhilfe: Auftretende „Ereignisse“ im XML-Dokument sollen nicht automatisch in die Anwendung „geschoben“, sondern von ihr explizit abgefragt werden (Pull-Parser)

- die Anwendung nur bestimmte Teile des XML-Dokuments verarbeiten möchte.

  Abhilfe: Die Anwendung schiebt ein „Fenster (Cursor)“ über das XML-Dokument und sieht so nur die für sie „interessanten“ Teile

→ Streaming API for XML (StAX)

9.2 StAX („Streaming API for XML“)

- Alle Klassen und Interfaces sind im Paket `javax.xml.stream` bzw. dessen Unterpaketen enthalten
- Jedes XML-Dokument wird aufgefasst als Strom von Ereignissen (z. B. Dokumentanfang, -ende, Start-, Ende-Tag, Textelement, . . .)
- Mit der Iterator-API und der Cursor-API bietet StAX zwei unterschiedliche Verarbeitungsmöglichkeiten der XML-Daten
- In beiden Fällen wird zunächst eine Instanz der Klasse `XMLInputFactory` benötigt:

```
XMLInputFactory factory = XMLInputFactory.newInstance();
```

Iterator-API

- Mithilfe der Methoden von `XMLEventReader` wird über alle Ereignisse in einem XML-Dokument iteriert; die **Ereignisse** liegen jeweils **als Objekte** vor:
  - `boolean hasNext()` zur Abfrage, ob weitere Ereignisse im XML-Dokument vorhanden sind
  - `XMLEvent nextEvent()` liest das nächste Ereignisse; wirft evtl. eine `XMLStreamException`
  - `XMLEvent peek()` Vorschau auf das nächste Ereignis, ohne es zu lesen; wirft evtl. eine `XMLStreamException`
  - ...

- Für jedes Ereignis im XML-Dokument wird eine passende Instanz der Klasse `XMLEvent` bzw. davon abgeleiteter Klassen (`StartElement`, `EndElement`, `Attribute`, `Comment`, `...`) erzeugt
- die Instanzen von `XMLEvent` können nicht verändert werden („immutable“) und können daher gefahrlos weitergegeben oder gespeichert werden

Beispiel zur Iterator-API

Aufgabe: Mittels der Iterator-API von StAX sollen alle in einem XML-Dokument auftretenden Ereignistypen ausgeben werden.

```
public static void main(String[] args) {
  // get XML input factory
  XMLInputFactory factory = XMLInputFactory.newInstance();

  // create XML Event reader
  XMLEventReader eventReader = factory.
    createXMLEventReader(new FileInputStream("fileName"));

  // list all events
  while (eventReader.hasNext()) {
    XMLEvent currentEvent = eventReader.nextEvent();

    /* print type of current event */
    System.out.printf("got %s\n",
      currentEvent.getClass().getSimpleName());
  }
}
```

Beispiel zur Iterator-API: Variante 1

Zusammen mit jedem Ereignis soll als Vorschau der Typ des nächsten Ereignisses ausgeben werden.

```
while (eventReader.hasNext()) {
  XMLEvent currentEvent = eventReader.nextEvent();

  XMLEvent nextEvent = eventReader.peek();

  System.out.printf("got %s (next will be %s)\n",
    currentEvent.getClass().getSimpleName(),
    (nextEvent != null) ?
      nextEvent.getClass().getSimpleName() : null);
}
```

Beispiel zur Iterator-API: Variante 2

Es sollen die Namen aller Elementknoten sowie deren Attribute ausgegeben werden.

```
while (eventReader.hasNext()) {
  XMLEvent currentEvent = eventReader.nextEvent();
  if (currentEvent.isStartElement()) {
    StartElement startElement =
      currentEvent.asStartElement();
    System.out.printf("<%s>\n", startElement.getName());
    startElement.getAttributes().forEachRemaining(
      attribute -> System.out.printf("  %s: %s\n",
         attribute.getName(),  attribute.getValue())
    );
  }
}
```

Umgang mit Namespaces in der Iterator-API?

Frage: Wie lässt sich der Namespace der Elemente bestimmen?

Bemerkung zur Iterator-API

Überspringt die Anwendung große Teile des XML-Dokuments, werden evtl. sehr viele `XMLEvent`-Instanzen unnötig erzeugt (und wieder vernichtet)

$\rightarrow$ unnötiger Zeit- und Speicherverbrauch (insbesondere bei leistungsschwacher Hardware zu beachten).

Cursor-API

- Mithilfe der Methoden von `XMLStreamReader` wird ein Cursor über die Ereignisse in einem XML-Dokument bewegt; der Typ des Ereignisse an der aktuellen Cursorposition wird als int-Wert beschrieben:
  - `int getEventType()` liefert den Typ des Ereignisses an der aktuellen Cursorposition
  - `boolean hasNext()` zur Abfrage, ob weitere Ereignisse im XML-Dokument vorhanden sind
  - `int next()` bewegt den Cursor zum nächsten Ereignis

- an jeder Cursorposition kann direkt auf die jeweiligen Daten zugegriffen werden; dabei ist eine `IllegalStateException` möglich, wenn die Methode nicht zum Kontext passt:
  - `isStartElement()`, `isEndElement()`, `isWhiteSpace()` `isCharacters()` zur Abfrage der Art der Daten an der aktuellen Cursorposition
  - `String getLocalName()` liefert den Namen des aktuellen Elements
  - `String getElementText()` liefert den Inhalt eines Textelements
  - `getAttributeCount()` liefert die Anzahl der Attribute des aktuellen Elements
  - ...

Beispiel zur Cursor-API

Aufgabe: Mittels der Cursor-API von StAX soll der Cursor über alle Ereignisse in einem XML-Dokument bewegt werden; die Nummern der Ereignisse, die Namen aller XML-Elemente sowie die enthaltenen Attribute ausgeben werden.

```
public static void main(String[] args) {
  // get XML input factory
  XMLInputFactory factory = XMLInputFactory.newInstance();
  // create XML Stream reader
  XMLStreamReader strmReader = factory
    .createXMLStreamReader(new FileInputStream("fileName"));

  /* first event is always START_DOCUMENT */
  int eventType = strmReader.getEventType();
  assert eventType == XMLEvent.START_DOCUMENT :
    "type of first event must be START_DOCUMENT";

  // list all events
  while (strmReader.hasNext()) {
    /* print event type */
    eventType = strmReader.next();
    System.out.printf("got event %s\n", eventType);
```

```
      /* print element names and attributes */
      // if (eventType == XMLEvent.START_ELEMENT) {
      if (strmReader.isStartElement()) {
        String elementName = strmReader.getLocalName();
        int attributeCount = strmReader.getAttributeCount();
        System.out.printf("<%s> %s\n", elementName,
          (attributeCount > 0 ? "with attributes" : ""));
        for (int i = 0; i < attributeCount; i += 1) {
          System.out.printf("  %s: %s\n",
            strmReader.getAttributeName(i),
            strmReader.getAttributeValue(i));
        }
      }
    }
}
```

Beispiel zur Cursor-API: Variante 1

Es sollen nur die Attribute der `chapter`-Elemente ausgegeben werden

```
while (strmReader.hasNext()) {
  strmReader.next();
  /* find a chapter element and print its attributes */
  if (strmReader.isStartElement() &&
    strmReader.getLocalName().equals("chapter")) {
    int attributeCount = strmReader.getAttributeCount();
    System.out.printf("<%s>\n", strmReader.getLocalName());
    for (int i = 0; i < attributeCount; i += 1) {
      System.out.printf("  %s: %s\n",
        strmReader.getAttributeName(i),
        strmReader.getAttributeValue(i));
    }
  }
}
```

Beispiel zur Cursor-API: Variante 2

Statt der expliziten Abfrage

```
if (strmReader.isStartElement() &&
    strmReader.getLocalName().equals("chapter"))
```

kann mit

```
require(int type, String namespaceURI, String localName)
```

getestet werden, ob der Cursor auf einem Ereignis `type` im Namensraum `namespaceURI` mit dem Namen `localName` steht; `namespaceURI` und `localName` werden im Falle von `null` nicht ausgewertet; falls der Cursor auf einer anderen Stelle steht, wird eine `XMLStreamException` geworfen.

```
while (strmReader.hasNext()) {
  strmReader.next();
  try {
    strmReader.require(XMLEvent.START_ELEMENT,
                       null,
                       "chapter");

    int attributeCount = strmReader.getAttributeCount();
    ...
  } catch (XMLStreamException e) {
    System.out.printf("Ignoring event type %s\n",
      strmReader.getEventType());
  }
}
```

Beispiel zur Cursor-API: Variante 3

Nicht benötigte Ereignisse können durch einen `XMLStreamReader` mit `StreamFilter` schon im Vorfeld ausgesondert werden:

```
public XMLStreamReader createFilteredReader(
  XMLStreamReader reader, StreamFilter filter)
```

Konkret:

```
public interface StreamFilter{
  boolean accept(XMLStreamReader reader);
}
```

hier:

```
public boolean accept(XMLStreamReader strmReader) {
  return strmReader.isStartElement()
         && strmReader.getLocalName().equals("chapter");
}
```

insgesamt:

```
// create base XML Stream reader
XMLStreamReader streamReader = factory
  .createXMLStreamReader(new FileInputStream("fileName"));

// create filtering wrapper around original stream reader
XMLStreamReader filteredStrmReader = factory
  .createFilteredReader(streamReader,
    r -> r.isStartElement()
         && r.getLocalName().equals("chapter"));

// list all chapter elements with attributes
while (filteredStrmReader.hasNext()) {
  filteredStrmReader.next();
  ...
}
```

Frage: Warum fehlt in der Ausgabe jetzt das erste Kapitel?

Bemerkungen

- In der Cursor-API lassen sich mittels `XMLEventAllocator` auch Instanzen von `XMLEvent` aus der Iterator-API erzeugen und damit beide Ansätze vereinen (siehe z. B. [Ora23c])
- Im Gegensatz zu SAX enthält die StaX-API auch Methoden zum Erstellen von XML-Dokumenten (siehe z. B. [Ora23c]).

9.3 DOM („Document Object Model“)

- DOM ist eine Programmierschnittstelle für den Zugriff auf die einzelnen Bestandteile von (XML-) Dokumenten
- DOM besteht aus einer Reihe von Schnittstellen, die die entsprechende Funktionalität definieren; vgl. [Web23].

  Aufteilung in sogenannte „Level“:

  1. „Level 1“ (Oktober 1998) legt die Grundfunktionalität zur Navigation durch Dokumente fest
  2. „Level 2“ (November 2000) erweitert den Level 1 um Namensräume
  3. „Level 3“ (April 2004) erweitert den Level 2 u. a. um XPATH, Schemata und Validierung
- Der Funktionsumfang jedes Levels ist in verschiedene

„Module“ unterteilt („Core“, „HTML“, „Events“, „Load and Save“, . . .);

- Die DOM-Spezifikation ist objektorientiert, aber plattform- und sprachneutral; es gibt Implementierungen z. B. in Java, Perl, C++, ECMAScript, . . .
- Ursprünglich war DOM als einheitliche Schnittstelle in Webbrowsern gedacht – um über Skripte in Webseiten auf HTML-Elemente zugreifen zu können
- Später erfolgte die Verallgemeinerung auf beliebige XML-Dokumente

Idee von DOM

- Jedes Bestandteil des XML-Dokuments wird durch eine Instanz einer entsprechenden Knotenklasse repräsentiert
- Die Knoteninstanzen sind baumartig entsprechend der Dokumentenstruktur verkettet
- In den Knoten gibt es Methoden zum Lesen, Ändern, Löschen oder Modifizieren der Daten sowie zur Navigation
- Außerdem sind Containerklassen (sogen. Collections) zur Aufnahme von Knoten definiert (z. B. Listen)

- DOM spezifiziert nur die Schnittstellen; die Implementierung erfolgt in einem sogenannten „DOM-Parser“ oder „DocumentBuilder“
- Die Implementierungen können sich in der internen Datenstruktur und den verwendeten Algorithmen unterscheiden

Im Folgenden werden aufgrund der Komplexität der gesamten API nur ausgewählte DOM-Schnittstellen am Beispiel vorgestellt. Für die vollständige Dokumentation sei auf die DOM-Spezifikation [Web23] verwiesen.[10]

[10] Als DOM-Implementierung in Java wird JAXP verwendet, das ab Version 5.0 Bestandteil des JDK ist; realisiert in Java ist DOM Level 3.

DOM-Knoten

- Oberklasse für alle in DOM abgebildeten Inhalte ist das Interface `org.w3c.dom.Node`.

  Durch entsprechende Erweiterungen von `Node` werden sämtliche Inhalte eines Dokuments abgebildet (z. B. Elemente, Texte, Attribute, Verarbeitungsanweisungen, CDATA-Abschnitte usw.)

- In `Node` sind u. a. folgende Methoden definiert:
  - `short getNodeType()` liefert den Typ des Knotens als Zahl codiert (siehe API)
  - `String getNodeName()` Ausgabe des Knotennamens (abhängig von dessen Typ)

- `NamedNodeMap getAttributes()` liefert Datenstruktur mit evtl. Attributen (falls Knoten `Element`)
- `boolean hasChildNodes()`, `Node getFirstChild()`, `Node getLastChild()`, `NodeList getChildNodes()`, `Node getParentNode()` zum Zugriff auf die direkten Vorfahren bzw. Nachkommen
- `Node getPreviousSibling()`, `Node getNextSibling()` Zugriff auf die Geschwister
- `Document getOwnerDocument()` liefert das Dokument, zu dem der Knoten gehört
- `removeChild(Node oldChild)`, `insertBefore(Node newChild, Node refChild)`, `replaceChild(Node newChild, Node oldChild)`, `appendChild(Node newChild)` zum Ändern/Hinzufügen von Kindknoten

- `normalize()` bearbeitet den Teilbaum mit dem Knoten als Wurzel so, dass benachbarte Textelemente zu einem Element zusammengefasst werden
- `...`

Wichtige Erweiterungen von `Node`

- `Element`: jede Instanz bildet einen Elementknoten ab
  - `String getTagName()`
  - `boolean hasAttribute(String name)`
  - `String getAttribute(String name)` liefert den Wert des entsprechenden Attributes
  - `Attr getAttributeNode(String name)` liefert den entsprechenden Attributknoten
  - `removeAttribute(String name)`
  - `setAttribute(String name, String value)`
  - `NodeList getElementsByTagName(String tagName)` liefert alle Nachkommen des Elementtyps `tagName` (Spezialfall: `"*"` passt auf alle Elementtypen!)

- `Attr`: jede Instanz bildet ein Attribut eines Elements ab
  - `String getName()` liefert den Namen des Attributs
  - `String getValue()` liefert den Wert
  - `Element getOwnerElement()` liefert das zugehörige Element
  - `boolean isId()` Abfrage, ob das Attribut vom Typ `ID` ist
  - `void setValue(String value)` zum Setzen eines neuen Wertes

- `CharacterData`: jede Instanz bildet ein aus einer Zeichenkette bestehendes Datum ab; wird von `Comment` und `Text` erweitert
  - `int getLength()`
  - `String getData()`
  - `insertData(int offset, String arg)`
  - `setData(String data)`
- `Comment`: jede Instanz bildet einen Kommentarknoten ab; erbt von `CharacterData`,
- `Text`: jede Instanz stellt den Textinhalt eines Elements dar; erbt von `CharacterData`
  - `String getWholeText()` liefert den Text von aufeinanderfolgenden Textknoten als Einheit

  - `replaceWholeText(String content)` ersetzt den Inhalt des Textknotens
  - `boolean isElementContentWhitespace()` liefert `true`, wenn es sich bei dem Text um „ignorable whitespace“ handelt
- `CDATASection`: jede Instanz bildet einen CDATA-Abschnit ab; erbt von `Text`

- `Document`: bildet das Dokument selbst (also die Wurzel des Dokumentenbaumes) ab
  - `Element createElement(String tagName)` Factory-Methode zum Erzeugen eines Elementknotens innerhalb des Dokuments[11]
  - weitere Factory-Methoden analog, z. B. `Comment createComment(String co)`, `Text createTextNode(String text)`, `CDATASection createCDATASection(String data)`, `Attr createAttribute(String name)`
  - `normalizeDocument()` führt das Dokument in eine „Normalform“ über (wie in der Methode `Node.normalize()` beschrieben)

[11] **Achtung:** Jedes Element gehört immer zu genau einem Dokument!

- `Element getDocumentElement()` liefert das Wurzelelement des Dokuments
- `String getDocumentURI()`
- `String getXmlVersion()`
- `Element getElementById(String idValue)` liefert das (eindeutige) Element, das ein Attribut vom Typ `ID` mit dem Wert `idValue` besitzt (oder `null`, falls keines im Dokument vorhanden ist)[12]
- `NodeList getElementsByTagName(String tagName)` liefert alle im Dokument enthaltenen Instanzen des Elementtyps `tagName` (Spezialfall: `"*"` passt auf alle Elementtypen!)

[12] **Achtung:** `ID` bezieht sich auf den *Typ* und nicht auf den *Namen* des Attributs!

- `DocumentFragment`: Hilfskonstrukt, das den Rahmen eine Dokuments zur Verfügung stellt; wird beim Erzeugen eines neuen Dokuments eingesetzt; darf im Gegensatz zu `Document` auch nicht wohlgeformten Inhalt aufnehmen; wenn eine Instanz von `DocumentFragment` einem Knoten hinzugefügt wird, so werden stattdessen seine Kinder hinzugefügt

Containerklassen

- `org.w3c.dom.NodeList`: geordnete Liste von Knoten
  - `int getLength()`
  - `Node item(int index)`

  Es werden keine Vorgaben für die zugrundeliegende Datenstruktur gemacht.

  `NodeList` ist *nicht* von `java.util.Vector` oder `java.util.List` abgeleitet.

  Es sind keine `set`-Methoden vorhanden. Außerdem ist keine Factory zum Erzeugen von Instanzen von `NodeList` in DOM definiert!

- `org.w3c.dom.NamedNodeMap`: Datenstruktur, in der Knoten über ihren Namen angesprochen werden können. Die Knoten werden nicht in einer bestimmten Ordnung gespeichert
  - `int getLength()` Anzahl der gespeicherten Elemente
  - `Node item(int i)` Zugriff auf das `i`-te Element
  - `Node getNamedItem(String name)` Zugriff auf einen Knoten über dessen Namen
  - `setNamedItem(Node arg)` Aufnahme eines Knotens in die Datenstruktur; als Name wird dessen `nodeName` verwendet; ein bereits unter dem Namen vorhandener Eintrag wird überschrieben.

  `NamedNodeMap` ist *kein* Array und ist auch *nicht* von `java.util.Map` abgeleitet!

Anwendung der DOM-API

Um ein XML-Dokument in eine interne DOM-Darstellung zu überführen, sind im wesentlichen die beiden folgenden Schritte notwendig:

1. Eine DOM-Verarbeitungsinstanz erzeugen;

   bei Bedarf können hier auch Eigenschaften für die Verarbeitung festgelegt werden (z. B. „Namensräume verwenden“ oder „Validierung des Dokuments durchführen“)

2. DOM-Verarbeitungsinstanz anwenden,

   d. h. eine interne Instanz von `org.w3c.dom.Document` mitsamt der Unterstruktur erzeugen

DOM-Verarbeitungsinstanz erzeugen

Je nach verwendeter DOM-Implementierung variiert dieser Schritt.

1. Bei Verwenden der Standardimplementierung JAXP:

   Eine `javax.xml.parsers.DocumentBuilderFactory` und mit dessen Hilfe die eigentliche Verarbeitungsinstanz `javax.xml.parsers.DocumentBuilder` erzeugen:

```
DocumentBuilderFactory factory =
  DocumentBuilderFactory.newInstance();
DocumentBuilder documentBuilder =
  factory.newDocumentBuilder();
```

In der Factory können für die zu erzeugende Verarbeitungsinstanz auch bestimmte Eigenschaft festgelegt werden:

```
// Namensraeume verwenden
factory.setNamespaceAware(true);
// Validierung durchfuehren
factory.setValidating(true);
```

2. Bei Verwendung der DOM-Implementierung von Apache-Xerces wird direkt über den Konstruktor eine Instanz des Parsers `org.apache.xerces.parsers.DOMParser` erzeugt:

```
DOMParser domParser = new DOMParser()
```

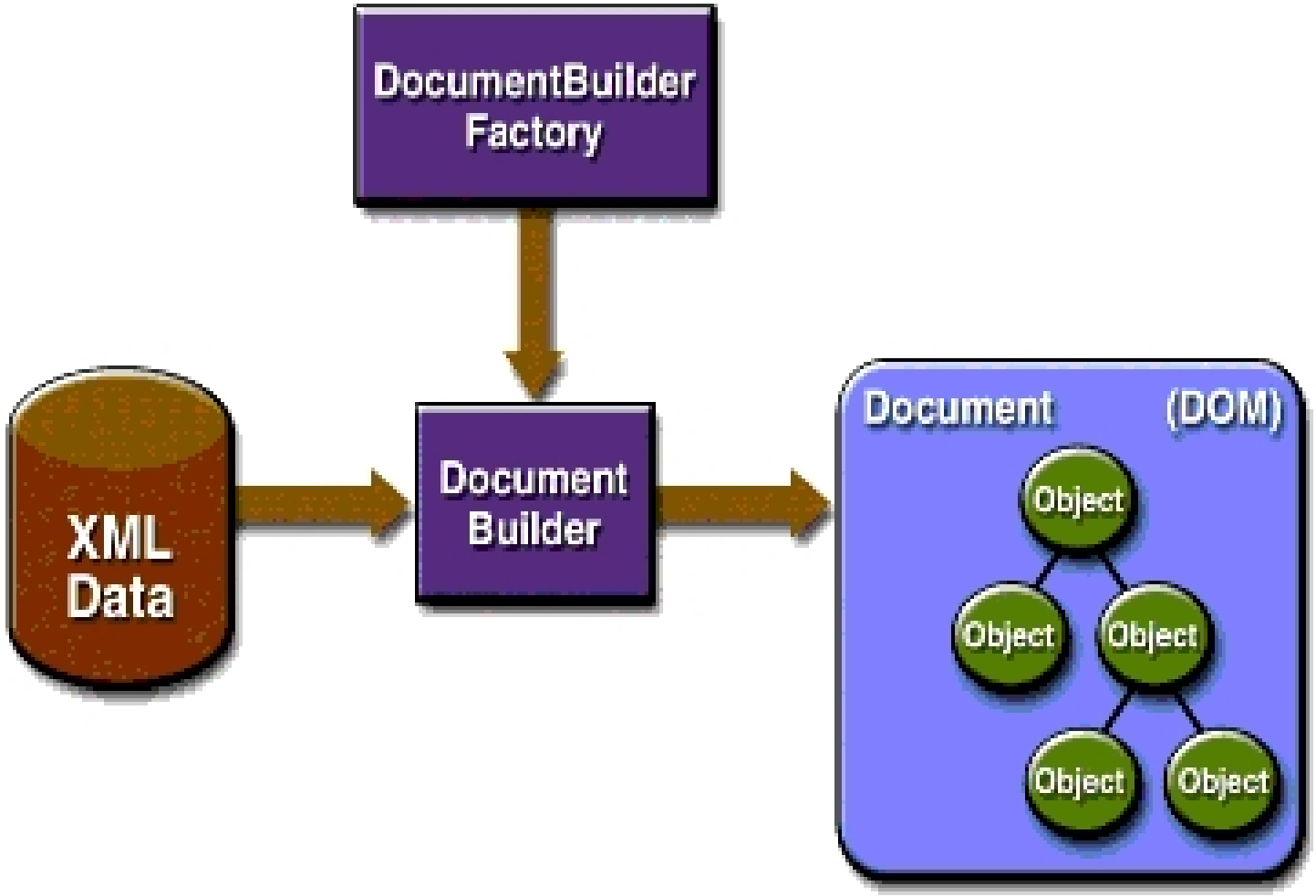
DOM-Verarbeitungsinstanz anwenden

- Die Verarbeitung des XML-Dokuments wird in beiden Fällen durch Aufruf der Methode `parse(`*`XML-Dokument`*`)` angestoßen.
- Die Angabe der Quelle des Dokuments kann dabei z. B. als Instanz von `File`, `InputSource`, `InputStream` oder als URI (`String`) erfolgen.
- **Beispiel:**

```
Document document = documentBuilder.parse(fileName);
```

bzw.

```
Document document = domParser.parse(fileName);
```



Schematische Darstellung zur Anwendung der DOM-API (vgl. [Ora23a])

Bemerkung: In der Regel wird beim Aufbau der DOM-Datenstruktur zur Verarbeitung der Eingabe intern die SAX-API verwendet. Daher können auch beim Einsatz von DOM Exceptions aus der SAX-API geworfen werden (z. B. bei Verletzungen der Wohlgeformtheit oder der Validierung).

Beispiel zu DOM (1)

Aufgabe: Aus der Eingabedatei `script.xml` soll eine interne DOM-Struktur aufgebaut und darin die Instanzen von `chapter`-Elementen gefunden und deren Überschriften ausgegeben werden.

Lösung: Implementiere zunächst eine Hilfsmethode, die eine Liste mit den `chapter`-Elementen des Dokuments liefert:

```
private static NodeList getChapters(Document document) {
    NodeList list =
      document.getElementsByTagName("chapter");
    return list;
}
```

Wende `getChapters()` z. B. in der `main`-Methode einer entsprechenden Klasse `GetChapters` an:

```
public static void main(String[] args) {
    String fileName = "script.xml";
    DocumentBuilderFactory factory =
      DocumentBuilderFactory.newInstance();
    // evtl. zusätzliche Eigenschaften setzen
    factory.setNamespaceAware(true);
    factory.setValidating(true);

    try {
      DocumentBuilder documentBuilder =
        factory.newDocumentBuilder();
      Document document =
        documentBuilder.parse(fileName);
      NodeList chapterList = getChapters(document);
```

```
      // Ergebnisse ausgeben
      for (int i = 0; i < chapterList.getLength(); i++) {
        Node node = chapterList.item(i);
        System.out.println(
          node.getAttributes().getNamedItem("heading"));
      }
    } catch (Exception e) {
      System.out.println(">>> " + e);
      System.exit(1);
    }
}
```

Bemerkung: Das Durchlaufen der Ergebnismenge über eine explizite Indexvariable ist unschön. In Java wünscht man sich die Verwendung von Streams oder zumindest von `Enumeration`, `Iterator`!

Beispiel zu DOM (2)

Aufgabe: Die Struktur eines DOM-Dokuments soll beginnend beim Wurzelelement rekursiv unter Angabe der entsprechenden Knotentypen ausgegeben werden.

Lösung: Implementiere eine rekursive Methode

```
private static void walkDepthFirst(Node node) {
  if (node != null) {
    System.out.println(node.getNodeName());
    if (node.hasChildNodes()) {
      NodeList children = node.getChildNodes();
      for (int i = 0; i < children.getLength(); i++) {
        walkDepthFirst(children.item(i));
      }
    }
  }
}
```

Einbindung z. B. in eine `main`-Methode:

```
public static void main(String[] args) {
  DocumentBuilderFactory factory = ...
  try {
    DocumentBuilder documentBuilder =
      factory.newDocumentBuilder();
    Document document =
      documentBuilder.parse(fileName);
    walkDepthFirst(document.getDocumentElement());
  } catch (Exception e) { ... }
}
```

Varianten . . .

1. Geben Sie zu jedem Element die vorhandenen Attribute aus
2. Geben Sie die Inhalte der Textknoten aus; unterscheiden Sie dazu Textknoten mit druckbaren Zeichen, Kommentare und „ignorable whitespace“-Knoten

Modifizieren und Abspeichern in DOM

- Zum Modifizieren des Dokuments stehen in den Knoten-Schnittstellen entsprechende Methoden zur Verfügung

  Beispiel: Hinzufügen von `para`-Elementen:

```
public static void modify(Document document) {
  NodeList chapterList =
    GetChapters.getChapters(document);
  for (int i = 0; i < chapterList.getLength(); i++) {
    Node node = chapterList.item(i);
    Text text = document.createTextNode
      ("Dies ist neu! (" + new Date() + ")");
    Element element = document.createElement("para");
    element.appendChild(text);
    node.appendChild(element);
  }
}
```

- Das Speichern einer DOM-Struktur kann in der Implementierung „DOM level 3“ über das Modul „load and save“ erfolgen (siehe Spezifikation)

Verwenden von „load and save“

```
/* zuerst eine Implementierung besorgen,
   die "load and save" unterstuetzt */
DOMImplementationRegistry registry =
  DOMImplementationRegistry.newInstance();
DOMImplementationLS lsImplementation =
  (DOMImplementationLS) registry.getDOMImplementation("LS");

/* dann damit die Ausgabedatei festlegen */
String outputName = "output.xml";
LSOutput output = lsImplementation.createLSOutput();
output.setByteStream(new FileOutputStream(outputName));

/* und das Dokument mithilfe eines "Serialisierers"
   schreiben */
LSSerializer serializer =
  lsImplementation.createLSSerializer();
serializer.write(document, output);
```

Validierender `DocumentBuilder`

Vor dem Erzeugen des DocumentBuilders kann die Factory angewiesen werden, nur validierende Parser zu erzeugen:

```
DocumentBuilderFactory factory =
  DocumentBuilderFactory.newInstance();
factory.setValidating(true);
```

1. Ohne Setzen weiterer „Properties“ wird angenommen, dass in dem zu parsenden XML-Dokuments eine DTD enthalten ist, die zum Validieren verwendet werden kann.

2. Wird in dem Dokument statt einer DTD ein XML-Schema verwendet, muss der Factory über eine spezielle Property zusätzlich die Art des Schemas mitgeteilt werden (hier immer W3C XML-Schema verwendet):

```
final String JAXP_SCHEMA_LANGUAGE =
  "http://java.sun.com/xml/jaxp/properties/" +
  "schemaLanguage";

DocumentBuilderFactory factory =
  DocumentBuilderFactory.newInstance();
factory.setValidating(true);
factory.setAttribute(JAXP_SCHEMA_LANGUAGE,
  XMLConstants.W3C_XML_SCHEMA_NS_URI);
```

3. Ist in dem XML-Dokument kein Schema enthalten, muss die Schemadatei explizit gesetzt werden:

```
final String JAXP_SCHEMA_LANGUAGE =
  "http://java.sun.com/xml/jaxp/properties/" +
  "schemaLanguage";
final String JAXP_SCHEMA_SOURCE =
  "http://java.sun.com/xml/jaxp/properties/" +
  "schemaSource";

DocumentBuilderFactory factory =
  DocumentBuilderFactory.newInstance();
factory.setValidating(true);
factory.setAttribute(JAXP_SCHEMA_LANGUAGE,
  XMLConstants.W3C_XML_SCHEMA_NS_URI);
factory.setAttribute(JAXP_SCHEMA_SOURCE,
  new File(SCHEMA_FILE));
```

Validieren mit `Validator`

Mithilfe der Methode `validate` der Klasse `Validator` lässt sich jederzeit die Validierung eines XML-Dokuments oder eines XML-Elements durchführen:

```
File schemaFile = new File(SCHEMA_FILE);
SchemaFactory sf = SchemaFactory.
  newInstance(XMLConstants.W3C_XML_SCHEMA_NS_URI);
Schema schema = sf.newSchema(schemaFile);
Validator validator = schema.newValidator();

Document document = ...;
validator.validate(new DOMSource(document));

Element element = ...;
validator.validate(new DOMSource(element));
```

Besonderheit der DOM-Containerklassen

1. Was gibt das folgende Programmfragment aus?
2. Was gibt das folgende Programmfragment aus, wenn die markierte Zeile auskommentiert wird?

```
DocumentBuilder documentBuilder = DocumentBuilderFactory.
  newInstance().newDocumentBuilder();

Document document = documentBuilder.newDocument();

Element root = document.createElement("root");
document.appendChild(root);

NodeList list = document.getElementsByTagName("foo");
System.out.printf("#foo elements in document: %s\n",
                  list.getLength());

for (int i = 0; i < 4; i += 1) {
  root.appendChild(document.createElement("foo"));
  list = document.getElementsByTagName("foo"); /* <-- */
  System.out.printf("#foo elements in document: %s\n",
                    list.getLength());
}
```

„live“-Eigenschaft

Die DOM-Containerklasse `NodeList` besitzt die sogenannte „live“-Eigenschaft; d. h. Änderungen in der Dokumentenstruktur sind **sofort** in allen betroffenen Instanzen von `NodeList` sichtbar!

Suchen im DOM

Das Auffinden von Knoten eines bestimmten Namens im gesamten XML-Dokument oder unterhalb eines Elementknotens gelingt mit `getElementsByTagName`.

Beispiel: Suchen *aller* `chapter`-Elemente im gesamten Dokument:

```
Document document = ...;
NodeList list = document.getElementsByTagName("chapter");
```

Frage: Was ist mit einer detaillierteren Suche?

DOM und XPath

Die Integration von XPath in DOM ist möglich!

Beispiel: Suchen aller wichtigen `chapter`-Elemente mit mindestens zwei Absätzen:

```
XPath xpath = XPathFactory.newInstance().newXPath();
try {
  Document document = ...;
  NodeList list = (NodeList)
    xpath.evaluate("//chapter[not(@important='no')]
                           [count(para)>=2]",
                   document, XPathConstants.NODESET);
} catch (EXPathExpressionException e) { ... }
```

Bemerkung: Der Rückgabetyp der `XPath`-Methode `evaluate` wird über einen Parameter anhand vordefinierter Konstanten eingestellt (`XPathConstants.STRING`, `XPathConstants.NODE`, `XPathConstants.NODESET`, `XPathConstants.BOOLEAN`, `XPathConstants.NUMBER`).

9.4 JAXB („Jakarta XML Binding“)

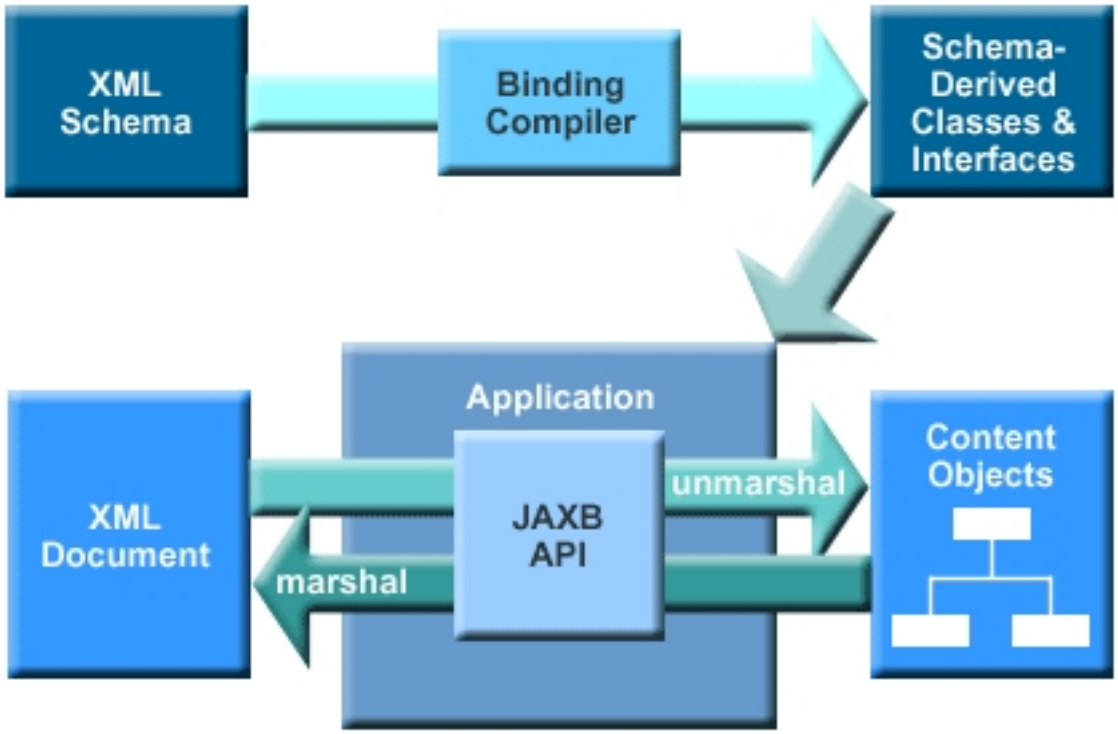
- **Nachteile der XML-APIs SAX oder DOM:**
  - sehr komplex (insbesondere für „einfache“ Anwendungsfälle)
  - aus Sicht von Java teilweise umständlich, da die APIs sprachunabhängig und damit sehr allgemein definiert sind
  - Programmierer braucht gute XML-Kenntnisse (Umgang mit den Begriffen „Element“, „Attribut“, „[ignorable] whitespace“, „parent“, „sibling“, . . .)

- **Wünschenswerte Eigenschaften einer einfacheren XML-API für Java:**
  - Verbergen der XML-Details vor dem Programmierer
  - Abbildung des XML-Dokuments in eine Java-Repräsentation
  - Zugriff auf Daten über `get`- und `set`-Methoden (Java Beans-Konzept)
  - Bei Bedarf „Validierung“ der Java-Repräsentation gegen ein Schema
  - Möglichkeit einer Speicherung der Java-Repräsentation in einem XML-Dokument

  → JAXB („Jakarta XML Binding“)[13]

[13] Frühere Bezeichnung: „Java Architecture for XML Binding“

JAXB-Architektur



Schematische Darstellung zur Anwendung von JAXB (vgl. [Ora23b])

Bestandteile einer JAXB-Anwendung

XML-Schema: Definiert die Syntax eines gültigen XML-Dokuments für die Anwendung; beschreibt die Beziehung der einzelnen Elemente zueinander

XML-Eingabedokument: XML-Dokument, aus dem die interne Java-Repräsentation erzeugt wird („Unmarshalling“)

XML-Ausgabedokument: XML-Dokument, das aus der internen Java-Repräsentation erzeugt werden kann („Marshalling“, engl. „to marshall“: anordnen, arrangieren)

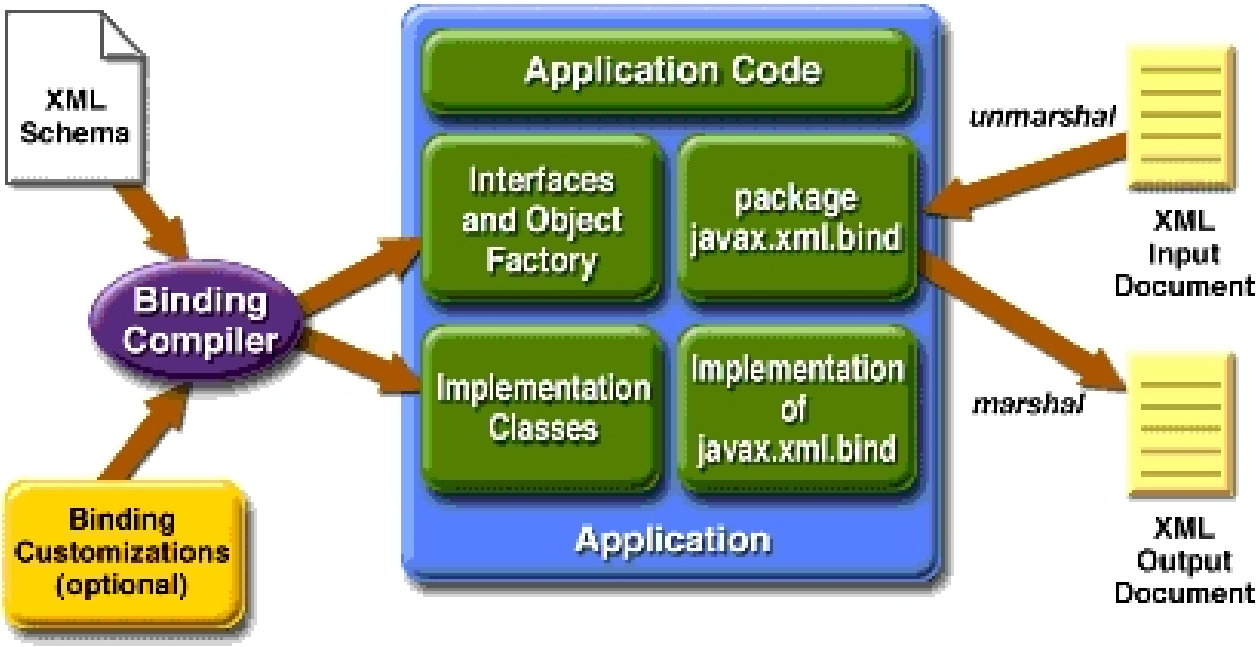
JAXB-Implementierung: Implementierung des Frameworks, das z. B. Mechanismen zum Unmarshalling, Marshalling und zur „Validierung“ der Java-Repräsentation zur Verfügung stellt

„Binding Compiler“: Umsetzer, der aus dem XML-Schema eine Menge von Datenklassen erzeugt, aus denen die Java-Repräsentation des XML-Dokuments aufgebaut wird

Erzeugte Datenklassen: Werden vom Umsetzer anhand des XML-Schemas erzeugt

Java-Anwendung: Nutzt die erzeugte Java-Repräsentation; stößt das Unmarshalling, das Marshalling oder die Validierung an

Architektur von JAXB



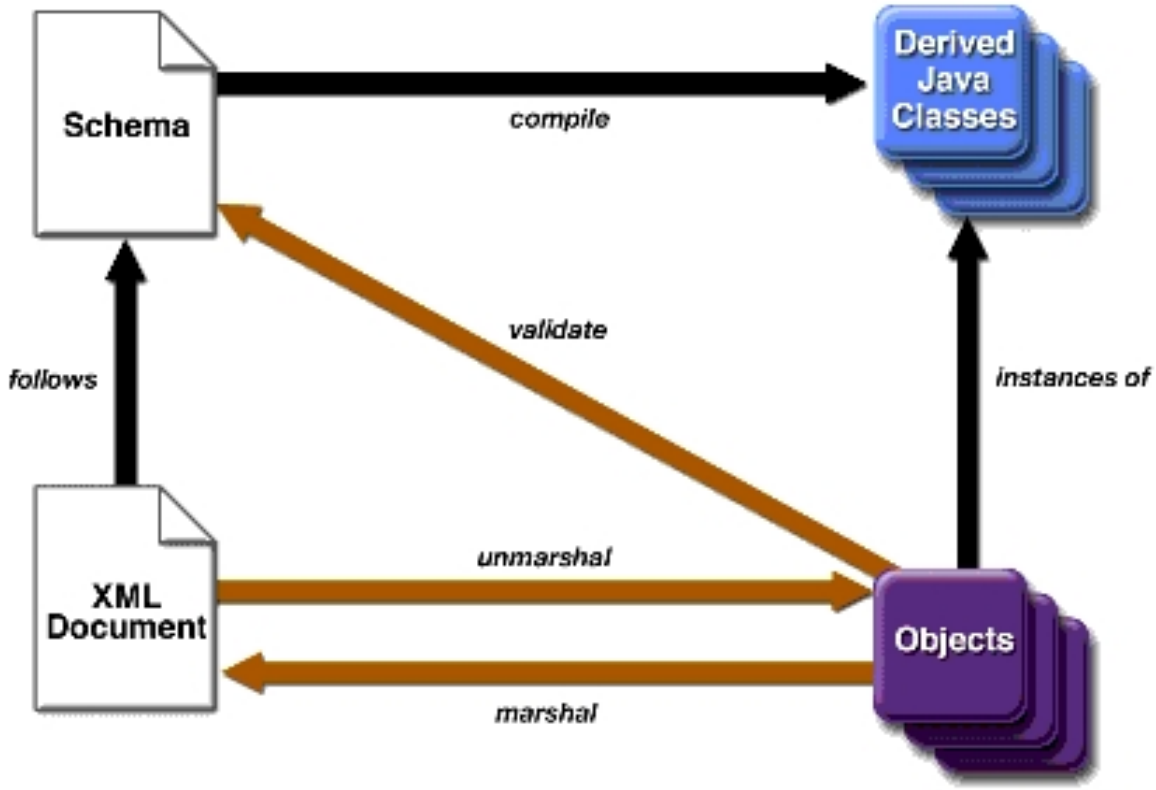
(vgl. [Ora23b])[14]

[14] Der Paketname der Implementierung hat sich zu `jakarta.xml.bind` geändert.

Anpassung der Umsetzungsregeln (optional)

Falls die Umsetzung anhand der Standardregeln den Anforderungen nicht genügt, ist eine Anpassung möglich (siehe API, wird hier nicht behandelt)

Abläufe beim Einsatz von JAXB



Schritte beim Einsatz von JAXB (Abb. vgl. [Ora23b]):

1. Vorbereitung: Erzeugen und Übersetzung der Java-Datenklassen
2. Anwendung: Unmarshalling, Benutzen der Datenklassen, Marshalling, Validierung

Vorbereiten des Einsatzes von JAXB

- Die Syntax der zu verarbeitenden XML-Dokumente muss als W3C-Schema vorliegen
- Erzeugen der Java-Datenklassen anhand des Schemas durch Aufruf des „Binding Compilers“ (z. B. durch das Kommando `xjc`, das Skript `xjc.sh` aus der Reference Implementation oder durch Einbinden einer entsprechenden Ant build-Datei in Eclipse).

  Beispiel:

  ```
  xjc -p jaxb.script -d foo -xmlschema script.xsd
  ```

  Erzeugt anhand des W3C-Schemas `script.xsd` die zugehörigen Datenklassen. Zielverzeichnis ist `foo`, die Klassen sollen zum Paket `jaxb.script` gehören.

- Neben den Datenklassen wird eine `ObjectFactory` erzeugt, über die zur Laufzeit neue Instanzen der Datenklassen erzeugt werden können;

  für jede Datenklasse ist dort eine entsprechende `create`-Methode vorhanden.

Abbildung von XML-Namen auf Java-Bezeichner

- Der Abbildungsalgorithmus versucht, die im Schema verwendeten Namen für Elemente, Attribute, Typen usw. in entsprechende Java-Bezeichner umzusetzen.
- Dabei können z. B. folgende Konflikte auftreten:
  - In XML-Namen sind mehr Sonderzeichen erlaubt
  - Als XML-Namen sind reservierte Java-Schlüsselwörtern erlaubt (z. B. `abstract`, `String`, ...)
  - In einem Namensraum kann derselbe Name mehrfach verwendet werden (z. B. für ein Element und ein Attribut oder für Attribute verschiedener Elemente)

Bemerkung: Durch Verändern der Standardregeln des JAXB-Compilers können diese Konflikte behoben werden.

Bemerkung: Es empfiehlt sich jedoch schon beim Entwurf des XML-Schemas mögliche Namenskonflikte zu berücksichtigen!

Bemerkungen zur verwendeten JAXB-Version

- JavaSE von Version 6 bis Version 10 enthält eine Implementierung von JAXB Version 2
- Ab JavaSE 11 ist JAXB ausgelagert; neben dem SDK muss daher z. B. die JAXB Reference Implementation installiert werden, oder es müssen die entsprechenden Pakete über Dependencies hinzugefügt werden (siehe Assignment 2)
- **Achtung:** Hier wird die Version 4.0.x aus dem Paket `jakarta.xml.bind` benötigt!

Abbildung einfacher XML-Datentypen

Einfache XML-Datentypen werden in entsprechende Java-typen abgebildet:

XML-Datentyp	Java-Datentyp
`xs:string`	`java.lang.String`
`xs:integer`	`java.math.BigInteger`
`xs:int`	`int`
`xs:decimal`	`java.math.BigDecimal`
`xs:float`	`float`
`xs:double`	`double`
`xs:boolean`	`boolean`
`xs:date`	`java.util.Calendar`
`xs:hexBinary`	`byte[]`

Abbildung komplexer XML-Datentypen

- Für jede Komponente des komplexen Datentyps (z. B. Elemente oder Attribute) werden entsprechende `get`- und `set`-Methoden definiert
- Wiederholungen von Elementen werden mithilfe von Listen als `java.util.List` abgebildet

  Hinweis: Zum Sicherstellen der Typsicherheit werden Generics verwendet (siehe folgende Beispiele)

Beispiel zur Abbildung komplexer Datentypen: `script.xsd`

Element `courseDoc` besitzt einen komplexen Datentyp:

```
<!-- <!ELEMENT courseDoc (abstract, chapter+)> -->
<xs:element name="courseDoc">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="abstract"/>
      <xs:element maxOccurs="unbounded" ref="chapter"/>
    </xs:sequence>
    <!-- <!ATTLIST courseDoc title CDATA #REQUIRED> -->
    <xs:attribute name="title" use="required"/>
  </xs:complexType>
</xs:element>
```

Der JAXB-Compiler erzeugt für das XML-Element `courseDoc` die Datenklasse `CourseDoc` mit den folgenden Methoden:

```
// Zugriff auf Element "abstract"
public String getAbstract();
public void setAbstract(String value);

// Zugriff auf Attribut "title"
public String getTitle();
public void setTitle(String value);

// Zugriff auf Sequenz des Elements "chapter"
public List<Chapter> getChapter();
```

Unmarshalling

- Unmarshalling bezeichnet das Umwandeln einer XML-Struktur in die interne Java-Repräsentation
- Die Umwandlung wird von der Methode `java.lang.Object unmarshall(`*`quelle`*`)` im Interface `jakarta.xml.bind.Unmarshaller` durchgeführt und liefert eine Instanz der Datenklassen des XML-Wurzelelements
- Als Quelle können in der `unmarshall`-Methode z. B. eine Instanz von `File`, von `InputStream`, eine SAX-Eingabequelle (`InputSource`) oder ein DOM-Knoten (`Node`) verwendet werden

- Instanzen von `Unmarshaller` liefert die Methode `Unmarshaller createUnmarshaller()` der Klasse `jakarta.xml.bind.JAXBContext`

Marshalling

- Marshalling bezeichnet das Umwandeln der internen Datenklassen in eine XML-Struktur
- Die Umwandlung wird von der Methode `void marshall(Object object, ziel)` im Interface `jakarta.xml.bind.Marshaller` durchgeführt
- Als Ziel können in der `marshall`-Methode z. B. eine Instanz von `Writer`, von `OutputStream`, ein SAX-`ContentHandler` oder ein DOM-Knoten (`Node`) verwendet werden
- Instanzen von `Marshaller` liefert die Methode `Marshaller createMarshaller()` der Klasse `JAXBContext`

Bemerkung: Eine formatierte Ausgabe durch den Marshaller wird durch Setzen einer entsprechenden Property erreicht:

```
marshaller.setProperty(
  Marshaller.JAXB_FORMATTED_OUTPUT, true);
```

Validierung

Die Validierung der Java-Repräsentation kann in der Standard-JAXB-Implementierung an zwei Stellen erfolgen:

Unmarshall-Validierung: Die Validierung erfolgt direkt bei der Umwandlung in die interne Repräsentation.

Dazu wird der `Unmarshaller`-Instanz über die Methode `setSchema` eine Instanz des entsprechenden W3C-Schemas übergeben (der Wert `null` schaltet die Validierung wieder aus; default: ausgeschaltet).

Marschall-Validierung: Entsprechend beim Speichern des XML-Dokuments durch vorheriges `setSchema` in der `Marshaller`-Instanz.

Beispiel zu JAXB (1)

Aufgabe: Das XML-Dokument `script.xml` soll in eine interne Java-Repräsentation überführt und von dort Angaben über die enthaltenen Kapitel ausgegeben werden.

Vorgehen:

1. Vorbereitung: Erzeugen der Datenklassen anhand des Schemas `script.xsd` in das Paket `jaxb.script`
2. Erzeugen einer Instanz von `JAXBContext` für die erzeugten Datenklassen:

```
JAXBContext jaxbContext =
  JAXBContext.newInstance("jaxb.script");
```

3. Erzeugen eines Objektes zur Umwandlung eines konkreten XML-Dokuments und Einlesen des Dokuments `script.xml`;

```
Unmarshaller unmarschaller =
  jaxbContext.createUnmarshaller();
CourseDoc courseDoc = (CourseDoc) unmarschaller.
  unmarshal(new FileInputStream(inputFile));
```

4. Zugriff auf die Liste der Kapitel des Skripts:

```
List<Chapter> chapterList = courseDoc.getChapter();
```

5. Darstellen der Kapiteldaten;

```
void printChapterList(List<Chapter> list) {
  list.forEach(chapter -> {
    System.out.printf("Chapter: \"%s\",
      important: \"%s\", " + "interest: \"%s\",
      sparepages: \"%s\"\n",
      chapter.getHeading(), chapter.getImportant(),
      chapter.getInterest(), chapter.getSparePages());
    chapter.getPara().forEach(para ->
      System.out.printf("  Para: \"%s\"\n", para));
  } );
}
```

Beispiel zu JAXB (2)

Aufgabe: Wie vorher, jedoch soll in jedem Kapitel ein neuer Absatz angehängt werden.

Lösung:

```
void modifyChapters(CourseDoc courseDoc) {
  courseDoc.getChapter().forEach(chapter ->
    chapter.getPara().
      add("Dies ist neu! (" + new Date() + ")"));
  }
}
```

Beispiel zu JAXB (3)

Aufgabe: Es soll ein neues Kapitel *vor* allen anderen Kapiteln hinzugefügt werden.

Lösung:

```
void addChapter(CourseDoc courseDoc) {
  ObjectFactory factory = new ObjectFactory();
  try {
    Chapter chapter = factory.createChapter();
    chapter.setHeading("Ein neues Kapitel");
    chapter.setSparePages(2);
    chapter.getPara().add("neuer Absatz");
    courseDoc.getChapter().add(0, chapter);
  } catch (Exception e) { /* ... */ }
}
```

Beispiel zu JAXB (4)

Aufgabe: Wie vorher, jedoch soll sowohl beim Einlesen als auch beim Speichern eine Validierung stattfinden.

Lösung:

```
Unmarshaller unmarschaller = ...
SchemaFactory sf = SchemaFactory.
  newInstance(XMLConstants.W3C_XML_SCHEMA_NS_URI);
Schema schema = sf.newSchema(new File(schemaFile));
unmarschaller.setSchema(schema);
CourseDoc courseDoc = (CourseDoc) unmarschaller
  .unmarshal(new FileInputStream(inFile));
... modify courseDoc
Marshaller marshaller = ...
marshaller.setSchema(schema);
marshaller.marshal(courseDoc,
                   new FileOutputStream(outFile));
```

Bemerkung zur Validierung

Aufgabe: Manchmal ist es sinnvoll, die veränderte Datenstruktur zu validieren *ohne* die Daten in eine XML-Datei zu schreiben. Wie geht das?

Lösung: Verwende als Ausgabe einen SAX `DefaultHandler`, der auf keine Events reagiert:

```
Marshaller tempMarshaller = jaxbContext.createMarshaller();
tempMarshaller.setSchema(schema);
try {
  tempMarshaller.marshal(courseDoc, new DefaultHandler());
} catch (Exception e) { ... }
```

oder besser im Marshaller einen `ValidationEventHandler` implementieren (Implementierung zur Übung)

Frage: Wie reagiert das Programm, wenn ...

1. in der Eingabedatei in dem letzten `para`-Element das Attribut `foo="test"` hinzugefügt wird?
2. in der obigen Methode `addChapter` der Parameter des Methodenaufrufs `chapter.setSparePages` auf `new Integer("4")` geändert wird?
3. in der obigen Methode `addChapter` der Methodenaufruf `chapter.setInterest("low");` hinzugefügt wird?

Bemerkungen

- Besteht ein XML-Element aus einer Sequenz anderer Elemente (`<xs:sequence>`), so sorgt JAXB dafür, dass die Elemente in der richtigen Reihenfolge gespeichert werden. Ein Vertauschen wie z. B. bei DOM oder bei einer manuellen Bearbeitung der Dokumente ist nicht mehr möglich.
- JAXB wird intern verwendet, um Java-Objekte als String im XML- oder JSON-Format zu „serialisieren“ (siehe RESTful Webservices in „Fortgeschrittene Webprogrammierung“).

Speichern von Binärdaten

Zu folgendem Schema werden JAXB-Klassen erzeugt:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="example">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="description" type="xs:string"/>
        <xs:element name="data" type="xs:hexBinary"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Methoden der erzeugten Klasse `Example`:

```
public class Example {
  public String getDescription()  { ... }
  public void setDescription(String value)  { ... }
  public byte[] getData() ...
  public void setData(byte[] value) { ... }
}
```

Verwendung der Klasse als Übung . . .

Wie sieht eine erzeugte XML-Instanz von `Example` aus?

Vererbung innerhalb von JAXB-Klassen

Für die Erweiterung von `chapter`-Elementen in einer vorigen `CourseDoc`-Variante wird folgender Java-Code erzeugt:

```
public class ChapterType {...}

public class ChapterExamType extends ChapterType {...}

public class CourseDoc {
   ...
   public List<ChapterType> getChapterOrChapterExam() {...}
   ...
}
```

10 Anwendung von DOM: Dynamisches HTML

- HTML-Dokumente lassen sich auch als DOM auffassen
- Web-Browser besitzen (zumeist) durch JavaScript eine Programmierschnittstelle
- In der Regel sind die spezifizierten DOM-Methoden implementiert und der Zugriff auf das dargestellte Dokument ist möglich
- Damit kann der Inhalt des dargestellten Dokuments von JavaScript-Code dynamisch verändert werden
- Diese Änderungen erfolgen direkt im DOM innerhalb des Browsers ohne Neuladen des Dokuments vom Server

Voraussetzungen für das weitere Vorgehen

- Verwendung eines „modernen“ Webbrowsers mit „ordentlicher“ JavaScript-Implementierung (d. h. Standards werden eingehalten): z. B. aktuelle Version von Firefox
- Beschreibung der HTML-Dokumente erfolgt hier mit XHTML (um Wohlgeformtheit und Validierungsmöglichkeiten aus der XML-Welt nutzen zu können)

$\Rightarrow$ Damit ist das dynamische Verändern von HTML-Dokumenten im Browser möglich

10.1 Exkurs JavaScript

- JavaScript wurde von der Firma Netscape entwickelt, um in Webseiten erweiterte Funktionalität zur Verfügung zu haben
- JavaScript ist standardisiert nach ECMA-262[15] und ISO/IEC 16262; die standardisierte Version wird auch als ECMAScript (ES) bezeichnet; aktuell ist Version ES 14 vom Juni 2023
- „JavaScript“ ist nicht mit der Programmiersprache „Java“ (ursprünglich von Sun Microsystems) zu verwechseln (obwohl der Name von JavaScript bewusst gewählt wurde, um von der Popularität von Java zu profitieren)

[15] „European Computer Manufacturers Association (ECMA)“

Einige Eigenschaften von JavaScript

JavaScript . . .

- . . . wird in der Regel im Client (d. h. im Webbrowser) zur Manipulation des DOM eingesetzt
- . . . ist eine interpretierte Programmiersprache (d. h. der Quellcode liegt zur Ausführungszeit vor und wird direkt ausgeführt)
- . . . ist eine höhere Programmiersprache und weist objektorientierte Merkmale auf
- . . . verwendet grundsätzlich ungetypte Variablen

„Historische“ oder „moderne“ Variante?

- JavaScript unterlag/unterliegt teilweise heftigen Sprachänderungen; trotzdem sollten alte Programme unverändert funktionieren
- viele „modernen“ Features sind standardmäßig ausgeschaltet
- durch die Anweisung `"use strict";` als erste Codezeile wird die moderne Variante aktiviert.

  Achtung: Dadurch ändert sich bei altem Code teilweise das Verhalten (es gelten z. B. strengere Regeln für den Zugriff auf nicht deklariert Variablen)
- in dieser Vorlesung wird die moderne Variante verwendet

Einbinden von JavaScript in HTML

Einbinden des Codes durch `script`-Elemente direkt im `head` oder durch Verweise auf externe JavaScript-Dateien.

Beispiel:

```
<?xml version="1.0" encoding="UTF-8" ?>
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>Exkurs JavaScript</title>
    <script type="text/javascript">
      function inline() {alert("calling function inline");}
    </script>
    <script type="text/javascript" src="exkursJS01.js">
    </script>
  </head>
  <body> ... <body>
</html>
```

Funktionen in JavaScript

- Die Angabe formaler Parameter erfolgt ohne Angabe eines Typs
- Die Anzahl der formalen und der aktuellen Funktionsparameter muss nicht übereinstimmen
- Aus HTML lassen JavaScript-Funktionen z. B. über spezielle `<a>`-Verweise unter Angabe des Protokolls `javascript` aufrufen:

```
<body>
  <h2>Some Experiments</h2>
  Calling the JavaScript
  function <a href="javascript:inline()">inline</a>
</body>
```

oder z. B. über ein `onclick`-Attribut eines `input`-Buttons innerhalb eines `form`-Elements:[16]

```
<body>
  <form action="#">
    <input type="button"
           value="Calling function inline()"
           onclick="inline()"/>
  </form>
</body>
```

[16] Neben `onclick` gibt es noch weitere sogenannte „EventHandler“ in JavaScript: `onmouseover`, `onmouseout`, `onblur`, ...;
die Angabe des Protokolls `javascript` ist hier optional.

Bemerkungen zu Funktionsparametern

- Der aktuelle Typ von Variablen oder Parametern lässt sich über `typeof` bestimmen
- Der Zugriff auf alle aktuellen Parameter ist z. B. über den implizit vorhandenen Parameter `arguments` möglich:

```
function test(){
  alert("called function test, first argument is " +
    arguments[0] + ", type is " + typeof arguments[0]);
}
```

Achtung: `arguments` sieht aus wie ein Array, ist aber keines (manche Eigenschaften fehlen)[17]!

[17] Da `Array` nicht in der Prototypenkette von `arguments` enthalten ist, funktioniert z. B. `arguments.sort()` nicht.

- Ist die Anzahl der aktuellen Parameter unbekannt, ist es oft besser, statt `arguments` sogenannte „Rest-Parameter“[18] zu verwenden:

```
function testRest(first, ...rest){
  alert(`first: ${first}, rest: ${rest}`);
}
```

  hier werden mit Ausnahme des ersten Parameters `first` alle weiteren Parameter des Funktionsaufrufs (falls vorhanden) in dem Array `rest` gesammelt.

[18] vergleichbar mit Varargs in Java

Beispiel: Zugriff auf die aktuellen Parameter einer Funktion (1)

```
function test(...args) {
  alert(`called function test:
        first argument is '${args[0]}'
        of type '${typeof args[0]}'`);
}
```

Was liefern die folgenden Aufrufe?

```
<body>
  Calling the above JavaScript function <code>test</code>
  with and without arguments:
  <a href="javascript:test('111')">test('111')</a>
  <a href="javascript:test(111)">test(111)</a>
  <a href="javascript:test()">test()</a>,
</body>
```

Beispiel: Zugriff auf die aktuellen Parameter einer Funktion (2)

```
function testRest(first, ...rest) {
  alert(`first: ${first}, rest: ${rest}`);
}
```

Was liefern die folgenden Aufrufe?

```
<body>
  Testing rest parameters:
  <a href="javascript:testRest(1)">?</a>
  <a href="javascript:testRest(1, 2, 3)">??</a>
  <a href="javascript:testRest()">???</a>
</body>
```

Bemerkungen zur Deklaration von Variablen

- das Schlüsselwort `let` deklariert eine Variable:

```
let foo = 7;
foo = "now it is a string";
```

- das Schlüsselwort `const` deklariert eine unveränderliche Konstante:

```
const foo = 7;
foo = 12; // Runtime Error!
```

- Variablen können außerdem durch `var` deklariert werden (aber anderer Gültigkeitsbereich als `let`; sollte seit ECMAScript 2015 (ES 6) nicht mehr verwendet werden)
- außerhalb des `strict`-Modus können Variablen auch implizit durch bloße Zuweisung vereinbart werden (sehr unübersichtlich und fehleranfällig!)

Typfestlegung erfolgt erst nach Zuweisung

Gegeben sei folgender JavaScript-Code:

```
function f(value){
 if (arguments.length === 0){
     return "called f without arguments"; }
   else { return "called f with parameter " + value
                + " of type " + typeof value; }
}
function usingVariables(){
  let i = 1; alert(i);
  i = "Text"; alert(i);
  i = f;
  alert(i()); alert(i(16)); alert(i("17")); alert(i);
}
```

Was liefert der Aufruf der Funktion `usingVariables`?

Primitive Typen in JavaScript

`number`: Nummerischer Typ sowohl für ganzzahlige Werte als auch für Gleitkommazahlen;

d. h. alle Berechnung werden als Gleitkommaoperationen durchgeführt!

```
let i = 0;
while (i != 10) {
  i += 0.1;
}
alert(`done`);
```

```
let a = 9999999999999999;
let b = 10000000000000000;

alert(`a != b: ${a!=b}`);
```

Was machen die beiden Codefragmente?

links: Endlosschleife! rechts: Ausgabe `a != b: false`

`string`: Zur Darstellung beliebig langer Zeichenketten.

In JavaScript ist `string` ein primitiver Typ, d. h. Strings sind keine Objekte (anders als in Java)!

Stringkonstanten können in `"`, `'` oder in `` ` `` eingeschlossen werden (`"foo"`, `'foo'`, `` `foo` ``).

Bei Verwendung der sogenannten „Backticks“ darf der String sich über mehrere Zeilen erstrecken und darf in `${...}` eingeschlossene Ausdrücke enthalten (siehe vorige Beispiele).

Die Länge eines Strings kann über die Property `length` (ohne runde Klammern!) abgefragt werden:

```
let s1 = "foo"; console.log(`${s1.length}`); // -> 3
let s2 = ""; console.log(`${s2.length}`); // -> 0
```

`boolean`: Zur Darstellung der beiden logischen Werte `true` und `false`

`undefined`: Wert und Typ nicht definierter Variablen und nicht zugewiesener Werte:

```
let test;
console.log(`"${test}" of type "${typeof test}"`);
// "undefined" of type "undefined"
```

`null`: Besonderer Wert (vom Typ `object`), der für Referenzen auf nicht existierende Objekt benutzt wird:

```
let head = null;
console.log(`type of head: "${typeof head}"`);
// type of head: "object"
```

JavaScript-Wrapper „String“

Welche Ausgabe liefert der JavaScript-Code:

```
function testString() {
  let a = new String();
  alert("a: \"" + a + "\"");
  let b = new String("Hallo");
  alert("b: \"" + b + "\"");
  let c = new String(42);
  alert("c: \"" + c + "\"");
  let d = String;
  alert("d: \"" + d + "\"");
  let e = new d("foo");
  alert("e: \"" + e + "\"");
}
```

JavaScript-Wrapper „Number“

Welche Ausgabe liefert der JavaScript-Code:

```
function testNumber() {
  let a = new Number();
  alert("a: " + a);
  let b = new Number(42);
  alert("b: " + b);
  let c = new Number("42");
  alert("c: " + c);
  let d = new Number("Nummer 42");
  alert("d: " + d);
  let e = new Number('a');
  alert("e: " + e);
}
```

JavaScript-Wrapper „Boolean“

Welche Ausgabe liefert der JavaScript-Code:

```
function testBoolean() {
  let a1 = new Boolean(); alert("a1: " + a1);
  let a2 = new Boolean(true); alert("a2: " + a2);

  let b1 = new Boolean("wahr"); alert("b1: " + b1);
  let b2 = new Boolean("false"); alert("b2: " + b2);
  let b3 = new Boolean("stimmt nicht"); alert("b3: " + b3);
  let b4 = new Boolean(""); alert("b4: " + b4);

  let c1 = new Boolean(1 == 0); alert("c1: " + c1);
  let c2 = new Boolean(1 == 1); alert("c2: " + c2);
}
```

Bemerkung zu „Boolean“

Vorsicht bei der Verwendung des Wrappers „Boolean“ in Ausdrücken:

- `false && true` liefert `false` (wie erwartet)
- `new Boolean(false) && true` liefert `true`!

  hier wird nicht der *Wert* des Wrapper-Objekts sondern das *Objekt selbst* ausgewertet; Objekte liefern in JavaScript immer `true`!
- Zugriff auf den eigentlichen Wert mit `valueOf()`:

  `new Boolean(false).valueOf() && true` liefert endlich `false`!

Bemerkung zu Wrapper-Objekten

Wrapper-Objekte werden in JavaScript implizit für Typanpassungen verwendet.

Aufgrund der teilweise „merkwürdigen“ Eigenschaften der Wrapper sollte in JavaScript in der Regel auf die explizite Verwendung von Wrapper-Objekten (`Number`, `Boolean`, ...) verzichtet werden!

JavaScript-Typ „Object“

Welche Ausgabe liefert der JavaScript-Code:

```
function testObject() {
  let b1 = new Object();

  b1.eins = "eins";
  b1.zwei = 2;
  b1.drei = new Object;

  alert("b1: " + b1);
  alert("b1.eins: " + b1.eins +
      ", b1.zwei: " + b1.zwei +
      ", b1.drei: " + b1.drei);
}
```

Zugriff auf Objekteigenschaften

Auf Eigenschaften (Properties) von Objekten kann über Punkt- oder Klammernotation zugegriffen werden:

```
function testObjectAccess() {
  let a1 = new Object();
  a1.foo = "Eigenschaft foo";
  a1.gnats = 42;
  a1.bar = [10, 11, "test", true, {}];

  try {
    alert(a1.foo);
    alert(a1["gnats"]);
    alert(a1[gnats]);
  } catch (ex) { alert("Exception: " + ex); }
    finally { alert("finally done"); }
```

`for`-Schleifen

Neben `for`-Schleifen mit expliziten Laufvariablen gibt es in JavaScript z. B. folgende Kurzformen:

Iterieren über die Einträge eines Arrays:

```
for (let value of a1.bar){
  alert (`${value}`);
}
```

Iterieren über die Namen der Eigenschaften eines Objektes:

```
for (let member in a1){
  alert("member: \"" + member + "\", value: \"" +
        a1[member] + "\"");
}
```

Iterieren über die Werte der Eigenschaften eines Objektes mit `for each`: (**Achtung:** deprecated!)

```
    for each (let val in a1){
      alert("object a1 contains value \"" + val + "\"");
    }
}
```

JavaScript-Typ „Array“

Gegeben sei folgender JavaScript-Code:

```
function testArray() {
  let a = new Array();
  for (let i = 0; i < 5; i++) a[a.length] = i;
  for (let i = 0; i < a.length; i++)
    alert("a[" + i + "]: " + a[i]);
}
```

- Was liefert der Aufruf von `testArray`?
- Was bewirkt folgende Änderung in der Deklaration `let a = new Array(5);`?
- Was bewirkt folgende Änderung in der Zuweisung `a[a.length + 1] = i;`?

Bemerkung: Die Größe von Arrays wird in JavaScript in Abhängigkeit der Schreibzugriffe dynamisch erweitert.

JSON (JavaScript Object Notation)

JSON ist ein textbasiertes Austauschformat für JavaScript-Objekte (siehe z. B. [JSO23]):

- Objektdefinitionen werden in geschweifte Klammern „`{ }`“ eingeschlossen:
  - die Bestandteile der Objekte werden durch „`,`“ getrennt
  - Member und ihre Werte werden durch „`:`“ von einander getrennt
- die Bestandteile von Arrays werden in eckige Klammern „`[ ]`“ eingeschlossen

Beispiele zu JSON

```
function construction01() {
   let a = { "foo" : "1" };
   alert("a.foo: " + a.foo);

   let b = [ 1, "1", true ];
   alert("b: " + b);
   alert("type of b[2] is " + typeof b[2]);

   let c = { "foo" : { "gnu":"GNU", "gnats":[1, 2, 3] },
             "gnu" : [ 1, { "a":"A" }] };
   alert("c: " + c);
   alert("c.foo.gnats[1]: " + c.foo.gnats[1]);
   alert("c.gnu[1].a: " + c.gnu[1].a);
}
```

Bemerkungen zum Erzeugen von Objekten und Arrays

In JavaScript wird in der Regel auf `new` zum Erzeugen von leeren `Object`- oder `Array`-Instanzen verzichtet!

Stattdessen wird die JSON-Notation verwendet:

`let a = {}` statt `let a = new Object()`

oder

`let a = []` statt `let a = new Array()`

Ebenso verwende besser

`let a = 'foo'` statt `let a = new String('foo')`

JavaScript-Objekte aus Strings erzeugen

- Objekte in JSON-Notation können als Strings abgelegt werden
- Mithilfe der Methode `JSON.parse` können Strings, die einer JSON-Notation entsprechen, direkt in JavaScript-Objekte umgewandelt werden[19]
- Mithilfe der Methode `JSON.stringify` können beliebige Objekte als Strings in JSON-Notation codiert werden.

[19] Gleiches gilt für die Methode `eval`; aus Sicherheitsgründen wird aber i. allg. von ihrer Anwendung abgeraten, um keinen „Schadcode“ in die Anwendung einzuschleusen.

Beispiele zu `JSON.parse`

(vgl. Beispiele zu JSON (1))

```
function parsing01() {
    let as = '{ "foo" : "1" }';
    alert("as: " + as);
    let a = JSON.parse(as);
    alert("a.foo: " + a.foo);

    let bs = '[ 1, "1", true ]';
    alert("bs: " + bs);
    let b = JSON.parse(bs);
    alert("b: " + b);
    alert("type b[2] is " + typeof b[2]);
}
```

Beispiele zu `JSON.stringify`

```
function stringify01(){
  let a1 = {};
  a1.foo = "FOO";
  a1.gnats = 42;
  a1.bar = [10, 11, "test", true, {}];
  a1.xxx = undefined; // null;

  alert(`${JSON.stringify(a1)}`);
}
```

Ausgabe:

```
{"foo":"FOO","gnats":42,"bar":[10,11,"test",true,{}]}
```

Wo ist `a1.xxx` geblieben?

Bemerkungen zu JSON

- JSON-Strings können z. B. auch von anderen Quellen (z. B. Web-Services) geladen und mittels `JSON.parse` in JavaScript-Objekte umgewandelt werden.
- Mit JSON lassen sich nur Daten (keine Funktionen) darstellen.

  Insbesondere gehen eventuell vorhandene Funktionen sowie Properties, die den Wert `undefined` enthalten, bei der Anwendung von `JSON.stringify` verloren!

Bemerkungen zu globalen Variablen

- alle globalen Variablen werden in demselben globalen Kontext deklariert (“global Object“)!
- dadurch können Objekte desselben Namens aus anderen Paketen/JavaScript-Dateien (unbeabsichtigt) überschrieben werden!
- **Abhilfe:** Definiere in jedem Paket/jeder JavaScript-Datei ein Containerobjekt[20] und lege die Variablen, Funktionen etc. dort ab!

[20] oder eine entsprechende Klasse (siehe später im Abschnitt 14)

Bemerkungen zum Einsatz von JavaScript

- Durch den fehlenden Compilerlauf und die nicht vorhandene Typsicherheit werden viele Fehler erst zur Laufzeit erkannt!

  Ausgiebiges Testen ist unbedingt erforderlich!
- Der Umstieg von compilierten OO-Sprachen zu JavaScript ist „gewöhnungsbedürftig“
- JavaScript ist aufgrund er fehlenden Typsicherheit für die Umsetzung größerer Vorhaben nur bedingt geeignet?!

Weitere Details zur JavaScript-Programmierung siehe entsprechende Literatur z. B. [Kan23] oder auch [Zak12, Cro08, SEL23].

10.2 JavaScript und DOM

Die Browser bieten in der Regel eine DOM-Schnittstelle zur Manipulation des geladenen HTML-Dokuments.

Aufgaben:

1. Ein fester Bereich eines HTML-Dokuments soll durch Anklicken um einen zufällig gewählten Wert horizontal verschoben werden
2. Eine Textpassage soll durch Anklicken ihre Farbe ändern
3. Einer Liste sollen durch Anklicken neue Elemente hinzugefügt werden

HTML-Seite

```
<?xml version="1.0" encoding="UTF-8" ?>
<html xmlns="http://www.w3.org/1999/xhtml">
  <head> <script type="text/javascript"
          src="first01.js"/></head>
  <body>
    <form action="#"><input type="button" value="Move it"
          onclick='$$.moveAround()'></input></form>
    <p><span id="colorP">
      <a href="javascript:$$.changeColor()">
        change the color</a> of this text</span></p>
    <div id="moveIt">We will move around.
      <a href="javascript:$$.addItem()">Add</a>
        another item to the list:
      <ul id="list"><li>me</li><li>and me too</li></ul>
    </div>
  </body>
</html>
```

JavaScript-Code

Zur Abkürzung verwende für das Containerobjekt innerhalb der Webseite den gültigen JavaScript-Namen `$$`:

```
/* use a container object instead of the global object */
let First01 = {};
window.$$ = First01;

First01.moveAround = function() {
  let x = Math.round(Math.random() * 85);
  //document.getElementById("moveIt").setAttribute("style",
  //    "position: relative; left: " + x + "%;");
  let target = document.getElementById("moveIt").style;
  target.position = "relative";
  target.left = x + "%";
}
```

```
First01.colors = [ "#FFFFFF", "#FFA0A0", "#A0FFA0" ];
First01.colorIndex = 0;

First01.changeColor = function() {
  First01.colorIndex = (First01.colorIndex + 1) %
                       First01.colors.length;
  document.getElementById("colorP").
    setAttribute("style", "background-color:" +
                 First01.colors[First01.colorIndex]);
}

First01.addItem = function() {
  let li = document.createElement("li");
  li.appendChild(document.
     createTextNode(Math.round(Math.random() * 1000)));
  document.getElementById("list").appendChild(li);
}
```

Teil II

Weitere Aspekte von JavaScript

11 Events

Fakten:

- Events werden ausgelöst, wenn im Ablauf der Anwendung „bestimmte“ Ereignisse auftreten
- Für Events können sogenannte „Listener“ oder „Eventhandler“ registriert werden („Observer Pattern“ vgl. [GHJV96])

Die Anwendungsmöglichkeiten sollen jetzt gegenüber der einfachen Anwendung aus Abschnitt 10 erweitert werden.

11.1 HTML-Eventhandler

Zu jedem Browser-Event (z. B. `click`, `load` oder `mouseover`) gehört ein Eventhandler, dessen Namen ein `on` vorangestellt wird (vgl. Beispiel in Abschnitt 10):

```
<form method="post" action="">
  <input type="text" name="foo" value="type here..." />
  <input type="button" value="Click me"
    onclick='alert(`${this}
      value: "${value}"
      event: "${event}"
      foo.value: "${foo.value}"
      method: "${method}"
      action: "${action}"
      document: "${document.title}"`)' />
</form>
```

Interna

Intern wird eine neue Funktion definiert, die

- mit `this` das HTML-Element ansprechen kann
- Zugriff auf das Attribut `value` hat
- über die lokale Variable `event` Zugriff auf den Event hat
- auf andere Member desselben `form`-Elements und auf `form` selbst zugreifen kann
- Zugriff auf die Member von `document` hat

Nachteile von HTML-Eventhandlern

- Timing: Das HTML-Element könnte dargestellt werden, obwohl der JavaScript-Code noch nicht ausführbar ist (wenn z. B. auf eine weiter unten auf der Seite definierte Funktion zugegriffen würde)

  Abhilfe: HTML-Eventhandler in `try/catch`-Block einbetten

- Zugriff auf die Daten der umgebenen Elemente hängt von der konkreten JavaScript-Implementierung ab
- Es besteht starke Kopplung zwischen HTML und JavaScript (wenig änderungsfreundlich; „MVC-Paradigma“ bzw. „Separation of Concerns“ so nicht umsetzbar)

$\Rightarrow$ keinen JavaScript-Code direkt in HTML-Seite einbetten!

11.2 Dom Level 0-Eventhandler

```
<form method="post" action="">
  <input type="text" name="foo" value="type here..." />
  <input id="myButtonLevel0"
         type="button" value="Click me" />
</form>
<!-- Wo ist der Eventhandler? -->
```

Zuweisung des Eventhandlers erfolgt in separatem Code:

1. Über die `id` eine Referenz auf das Element besorgen
2. Code des Eventhandlers in Funktion implementieren (auch anonyme Funktion ist möglich)
3. Eventhandler-Funktion an die zum Event gehörende Property des Elements zuweisen

```
function init() {
  /* init DOM Level 0 event handler */
  let button = document.getElementById("myButtonLevel0");
  button.onclick = function(event) {
  alert(`${this}
    this.id: "${this.id}"
    this.value: "${this.value}"
    event: "${event}"
    this.form.foo.value: "${this.form.foo.value}"
    this.form.method: "${this.form.method}"
    this.form.action: "${this.form.action}"
    document.title: "${document.title}"`);
  }
}
```

Frage: Wie/wann wird `init()` aufgerufen?

`init`-Code aufrufen

Die Funktion `init` wird z. B. nach dem Laden der Seite über einen `load`-Event aufgerufen:

```
<body onload="init()">
```

(eigentlich schlecht, da wieder JavaScript-Code/Eventhandler in HTML-Seite)

oder besser durch Setzen des Eventhandlers innerhalb des JavaScript-Codes:

```
window.onload = init;
```

Bemerkungen

Der Eventhandler . . .

- steht erst nach dem Ausführen der `init`-Funktion zur Verfügung
- wird in dem Scope des Elements ausgeführt (`this` verweist auf das auslösende HTML-Element)
- kann durch Zuweisen von `null` an die entsprechende Property des Elements wieder entfernt werden

11.3 Eventfluss (Event Flow)

Frage: Zu welchem Teil der Webseite gehört der Click-Event auf den rot markierten Link?

```
<html>
  <head>...</head>
  <body>
    <h1>Testing Events in JavaScript</h1>
    <h2>Event Flow</h2>
    <div id="eventflow">
      <p>First Paragraph</p>
      <ul>
        <li>one</li>
        <li><a href="javascript:click()">a link</a></li>
      </ul>
    </div>
  </body>
</html>
```

Antwort: Neben dem Link werden auch die umschließenden Elemente `li`, `ul`, `div`, `body` und das `html`-Dokument selbst angeklickt!

Welches der Elemente soll den Klick auswerten?

Der Eventfluss (Event Flow) beschreibt die Reihenfolge, in der ein Event die betroffenen Elemente erreicht!

Es gibt die beiden entgegengesetzten Vorschläge Event Bubbling und Event Capturing, die aber beide in der Spezifikation von DOM Level 2 zusammengefasst wurden.

Event Bubbling

Idee des Event Bubbling: Jeder Event startet an dem tiefst möglichen Element des Dokuments und steigt dann zum obersten Element (dem Dokument selbst) auf.

Event Bubbling des Beispiels:

1. `a`
2. `li`
3. `ul`
4. `div`
5. `body`
6. `html`-Dokument

Event Capturing

Idee des Event Capturing: Jeder Event wird zuerst von dem obersten umschließenden Element gefangen und dann zum innersten Element weitergereicht.

Dadurch ist ein Abfangen des Events und Auslösen von Aktionen möglich, bevor der Event beim eigentlichen Zielknoten ankommt!

Event Capturing des Beispiels:

1. `html`-Dokument
2. `...`
3. `li`
4. `a`

DOM Eventfluss

DOM Level 2 (siehe 11.4) spezifiziert für den Eventfluss drei Phasen:

1. Event Capturing (Absteigen im Dokument)
2. Target (Bearbeiten des Zielelements)
3. Event Bubbling (Aufsteigen im Dokument)

Aktuelle Browser sollten diesen Eventfluss implementieren.

11.4 Dom Level 2-Eventhandler

Nachteile des bisherigen HTML- und Dom Level 0-Eventhandlers (siehe 11.1 und 11.2):

- jedem Event kann maximal eine Funktion zugewiesen werden
- keine gezielte Ausnutzung des Eventflusses möglich (nur „Target“-Phase wird verwendet)

Abhilfe: Das Zuweisen/Entfernen von Eventhandlern erfolgt nicht mehr durch das direkte Zuweisen an die Property.

Stattdessen verfügt jedes Element über die beiden Methoden `addEventListener` und `removeEventListener`.

Parameter der Funktionen `addEventListener` und `removeEventListener`:

1. Name des Events (als Zeichenkette; ohne Präfix `on`)
2. auszuführende Funktion (`function`)
3. boolescher Wert zur Angabe des Eventflusses:

   true: Behandlung während der Capturing-Phase
   false: Behandlung während der Bubbling-Phase (default)

Beispiel:

```
let button = document.getElementById("myButtonLevel2");
let eventHandler = function(event) { ... };
button.addEventListener("click", eventHandler, false);
```

`init`-Code

Zum Aufrufen einer `init`-Funktion ersetze den HTML-Eventhandler innerhalb von Html

```
<body onload="init()">
```

bzw. den Level 0-Eventhandler

```
window.onload = init;
```

durch den Level 2-Eventhandler außerhalb von HTML

```
window.addEventListener("load", init);
```

Beachte

- `load` in `document` wird ausgelöst, wenn das Dokument (DOM) geladen ist; evtl. sind Bilder aber noch nicht (vollständig) geladen.
- `load` in `window` wird ausgelöst, wenn die gesamte Seite (einschließlich z. B. aller Bilder) geladen wurde.
- `load` besitzt keine bubble-Phase!

Beachte: Die Identifizierung des Eventhandlers in `addEventListener` und `removeEventListener` geschieht über eine Referenz, auf die als Parameter übergebene `function`.

```
let eventHandler = function(event) { ... };
button.addEventListener("click", eventHandler, false);
...
button.removeEventListener("click", eventHandler, false);
```

Wenn der Eventhandler nach der Zuweisung wieder entfernt werden soll, darf er **nicht als anonyme Funktion** übergeben werden:

```
button.addEventListener("click", function(){ ..}, false);
...
button.removeEventListener("click", function(){...}, false);
```

11.5 Das `Event`-Objekt

Wichtige *Properties* von `Event`:

`target`: das Element, das den Event ausgelöst hat

`currentTarget`: das Element, dessen Eventhandler gerade ausgeführt wird

`eventPhase`: aktuelle Phase im Eventfluss; 1: Capturing Phase, 2: „at target“, 3: Bubbling Phase

`type`: Typ des ausgelösten Events

`view`: Verweis auf das `window`-Object, in dem der Event ausgelöst wurde

Wichtige *Funktionen* von `Event`:

`preventDefault()`: das Standardverhalten des Events wird aufgehoben (siehe späteres Beispiel zum Dragging)

`stopPropagation()`: der Event wird wird zu keinem anderen als dem aktuellen Objekt weitergereicht

`stopImmediatePropagation()`: der Event wird weder zu anderen Objekten noch zu anderen Eventhandlern desselben Objekts weitergereicht

Aufgabe: Der Eventfluss in der Capturing- und der Bubbling-Phase soll durch entsprechende Ausgaben verfolgt werden!

Lösungsidee: Versehe die gewünschten Elemente mit eigenen Ids und registriere einen geeigneten Eventhandler.

Umsetzung als Übung. . .

11.6 Weitere Event-Typen

UI-Events

UI-Events werden vom Benutzer oder vom System ausgelöst, z. B. :

`load`: das Laden einer Seite ist abgeschlossen, betrifft `window`

`error`: JavaScript-Fehler, betrifft `window`

`resize`: Ändern der Größe, betrifft `window`

`select`: Text wurde ausgewählt in `input` oder `textarea`

Fokus-Events

`blur`: Fokus des Elements wird verlassen

`focus`: Element bekommt den Fokus

Maus-Events

`click`, `dblclick`: Einzel- oder Doppelkick

`mousedown`, `mouseup`: Drücken bzw. Loslassen einer Maustaste

`mouseenter`, `mouseover`, `mouseout`: Berühren oder Verlassen eines Elements

`mousemove`: Bewegen der Maus über einem Element

Noch mehr Events

Weitere Events sind vorhanden z. B. für:

- Tastaturaktionen
- Aktionen auf Touch Screens
- Aktionen in Kontextmenüs
- Gestenerkennung
- Lage- und Beschleunigungsänderungen (`deviceorientation`, `devicemotion`)
- ...

12 Selektieren von Dokumentknoten

Bisher wurden Knoten innerhalb des Dokuments anhand des `id`-Attributs mithilfe von `getElementById()` oder anhand des Elementnames mittels `getElementsByTagName()` selektiert.

Beispiele:

```
let ul = document.getElementById("liste");
...
let items = document.getElementsByTagName("li");
for (let item of items){
  doSomethingWith(item);
}
```

12.1 Bemerkungen zu `NodeList`

Die Funktion `getElementsByTagName()` liefert eine Containerklasse vom Typ `NodeList`:

- `NodeList` besitzt das Attribut `length` zur Abfrage der Anzahl der Einträge und die Methode `item(i)` zum Auslesen des i-ten Elements (in JavaScript auch Zugriff über Arraynotation `[i]` möglich).
- `NodeList` besitzt eine sogenannte „live“-Eigenschaft[21]; d. h. Änderungen in der Dokumentenstruktur sind sofort in allen betroffenen Instanzen von `NodeList` sichtbar.

[21] Siehe voriges DOM-Beispiel in Java

Beispiel zur „live“-Eigenschaft

Aufgabe:

1. Erstelle ein HTML-Dokument, das eine Liste mit mehreren `li`-Elementen enthält.
2. Erstelle mittels `getElementsByTagName()` eine `NodeList` mit allen `li`-Elementen der Seite und gib sie aus.
3. Füge der Liste ein weiteres `li`-Element hinzu und gib die Einträge der vorigen `NodeList`-Instanz erneut aus.

Ergebnis: Auch ohne einen erneuten Aufruf von `getElementsByTagName()` enthält die `NodeList` das neu hinzugefügte Listenelement!

Nachteile von `getElementById()` und `getElementsByTagName()`

- Teilweise „umständliche“ Notation im Vergleich zu CSS-Selektoren; z. B. für Klassen (Notation: `.foo` für `class="foo"`) oder Ids (Notation: `#foo` für `id="foo"`)
- „live“-Eigenschaft wird gar nicht immer benötigt; das Update der Listen kostet aber Rechenzeit

12.2 Die Selector-API

- Unterstützt CSS-Selektoren (siehe folgende Beispiele)
- Stellt Methoden `querySelector` (liefert nur ersten Treffer) und `querySelectorAll` (liefert alle Treffer) bereit
- Die Ergebnismenge von `querySelectorAll` besitzt *keine* „live“-Eigenschaft
- Auf das i-te Element der Ergebnismenge von `querySelectorAll` kann wie gewohnt über `item(i)` oder über die Arraynotation `[i]` zugegriffen werden
- `length` liefert die Anzahl der selektierten Elemente

Beispiele zur Selector-API

- das `body`-Element des Dokuments:

```
let body = document.querySelector("body");
```

- das erste Element mit der ID `foo`:

```
let idFoo =  document.querySelector("#foo");
```

- alle Elemente der Klasse `foo` (`class="foo"`):

```
let classFoo = document.querySelectorAll(".foo");
```

- das erste Bild der Klasse `gnats` im `body`:

```
let imgGnats = document.body.querySelector("img.gnats");
```

- alle `strong`-Elemente innerhalb von `p`-Elementen:

```
let strongP = document.querySelectorAll("p strong");
```

12.3 XPath API

Beispiel: Suche das Element mit dem `id`-Attribut `mylist`; das Ergebnis soll ein einzelner Knoten (Ergebnistyp FIRST_ORDERED_NODE_TYPE) sein:

```
let mylist = document.evaluate("//*[@id='mylist']",
             document, null,
             XPathResult.FIRST_ORDERED_NODE_TYPE, null);

alert(mylist.singleNodeValue);
```

Beispiel: Suche alle `li`-Elemente; die Ergebnismenge soll iterierbar (Ergebnistyp ORDERED_NODE_ITERATOR_TYPE) sein. Wird der DOM während der Iteration verändert, wird ein NS_ERROR_DOM_INVALID_STATE_ERR ausgelöst:

```
let items = document.evaluate("//li",
                document, null,
                XPathResult.ORDERED_NODE_ITERATOR_TYPE, null);
try{
  let thisItem = items.iterateNext();
  while (thisItem) {
    alert(thisItem.textContent);
    thisItem = items.iterateNext();
  }
} catch (ex) {
  alert(ex);
}
```

Beispiel: Suche alle `li`-Elemente; die Ergebnismenge soll als statischer Schnappschuss (Ergebnistyp ORDERED_NODE_SNAPSHOT_TYPE) gespeichert werden. Der DOM darf während der Bearbeitung des Schnappschusses verändert werden:

```
let items = document.evaluate("//li",
                document, null,
                XPathResult.ORDERED_NODE_SNAPSHOT_TYPE, null);

for (let i = 0; i < items.snapshotLength; i++) {
  alert(items.snapshotItem(i).textContent);
}
```

13 „Drag and Drop“ auf Webseiten

Aktuelle Browser (z. B. Firefox) unterstützen automatisch das Verschieben von Texten und Bildern innerhalb der Webseite (und zwischen Webseiten).

Ausgelöste Events für das *verschobene Objekt*:

`dragstart`: Das Verschieben des betroffenen Elements beginnt

`drag`: Das Verschieben des Elements läuft. . .

`dragend`: Das Verschieben des betroffenen Elements ist beendet

Ausgelöste Events für das *Zielobjekt*:

`dragenter`: Über das betroffene Element wird „etwas“ geschoben

`dragover`: Das Verschieben über dem Element geht weiter. . .

`dragleave`: Das Verschieben von „etwas“ über das betroffene Element wird dadurch beeendet, dass es wieder von dem Element weggeschoben wird

`drop`: Das Verschieben von „etwas“ über das betroffene Element wird durch „Fallenlassen“ auf dem Element beendet

Ändern des Standardverhaltens

- Standardmäßig werden für verschobene Texte und Bilder alle Events ausgelöst, Änderungen im Dokument finden jedoch nicht statt, da das Standardverhalten kein `drop` erlaubt (außer auf Eingabefelder).
- Durch Modifizieren der entsprechenden Eventhandler kann das Verhalten geändert werden!
- Durch Setzen des Attributs `draggable="true"` kann jedes Element der Webseite verschoben werden (nicht nur Texte oder Bilder).

Das `dataTransfer`-Objekt

Die zu verschiebenden Daten werden in einem `dataTransfer`-Objekt abgelegt; der Zugriff darauf erfolgt über zwei Methoden:

`setData(type, data)`: speichert die Daten des verschobenen Objektes;

`type` spezifiziert den Mime Type der Daten (z. B. `text/plain`), `data` enthält die entsprechenden Daten (z. B. den Text oder eine ID der Quelle)

`getData(type)`: liest die Daten in dem entsprechenden Mime Type aus

Aufgabe (1)

Auf einer Webseite sollen die Inhalte von markierten Elementen (z. B. durch eigene Klasse `class="draggable"`) kopiert und im „Zielbereich“ der Seite in eine Liste (z. B. `id="destination"`) eingefügt werden.

Vorbereiten der Webseite

```
<html>
  <head>
    <link rel="stylesheet" href="dragAndDrop.css"></link>
    <script type="text/javascript" src="dragAndDrop01.js"></script>
  </head>
  <body>
    <h2 class="draggable">Dragging Text</h2>
    <ul>
      <li class="draggable">A first text </li>
      <li class="draggable">Second text </li>
      <li class="draggable">Another text </li>
    </ul>
    <h2>Destination</h3>
    <ul id="destination"> <span>drop it here...</span></ul>

    <h2>Output Division</h3>
    <div id="output"></div>
  </body>
</html>
```

Optional: Etwas CSS

Verwende CSS, zum Kennzeichnen . . .

- aller Elemente mit dem Attribut `draggable="true"`
- des Zielobjekts

```
/* file dragAndDrop.css */

*[draggable=true] {
  color: red
}

#destination {
  color: blue
}
```

Ändern der Eventhandler

```
window.addEventListener("load", init);

function init() {
  /* init event handler for all sources */
  let draggables =
       document.querySelectorAll(".draggable");
  for (let element of draggables) {
    element.draggable = true;
    element.addEventListener("dragstart", function(event) {
      event.dataTransfer.effectAllowed = "copy";  // specify effect
      event.dataTransfer.setData("text/plain", event.target.innerHTML);

      appendToOutput(
        `${event.type} (${event.target}: ${event.target.innerHTML})`
      );
    }, false);
```

```
  element.addEventListener("dragend", function(event) {
    event.preventDefault();
    appendToOutput(`${event.type} (${event.target}
      effect performed: "${event.dataTransfer.dropEffect}")`);
  }, false);
}
```

```
/* init event handlers for the destination */
let destination = document.getElementById("destination");

destination.addEventListener("dragover", function(event) {
  event.preventDefault(); // allow dropping here!
  event.dataTransfer.dropEffect = "copy"; // confirm the effect
  appendToOutput(event.type);
}, false);

destination.addEventListener("dragenter", function(event) {
  event.preventDefault(); // allow dropping here!
  event.dataTransfer.dropEffect = "copy"; // confirm the effect
  appendToOutput(event.type);
}, false);

destination.addEventListener("dragleave", function(event) {
  event.preventDefault();
  appendToOutput(event.type);
}, false);
```

```
  destination.addEventListener("drop", function(event) {
    event.preventDefault();
    let data = event.dataTransfer.getData("text/plain");
    // insert new li element
    let li = document.createElement("li");
    li.appendChild(document.createTextNode(data));
    destination.appendChild(li);
    appendToOutput(`${event.type} (${data})`);
  }, false);
}

/* utility function */
function appendToOutput(text) {
  let output = document.getElementById("output");
  if (output) {
    output.innerHTML += text + "<br>";
  }
}
```

Aufgabe (2)

Wie vorige Aufgabe, jedoch sollen die bewegten Elemente nicht kopiert, sondern komplett in den „Zielbereich“ verschoben werden.

Ändern der Eventhandler für das Verschieben

```
const MY_TYPE = "text/mytype";

function init() {
  /* init event handler for all sources */
  let draggables =
       document.querySelectorAll(".draggable");
  for (let element of draggables) {
    element.draggable = true;
    element.addEventListener("dragstart", function(event) {
      event.dataTransfer.effectAllowed = "move";
      event.dataTransfer.setData(MY_TYPE, "event.target.outerHTML");
      appendToOutput(`${event.type} (${event.target}`);
    }, false);
  }

  ...
```

```
destination.addEventListener("drop", function(event) {
  event.preventDefault();
  let data = event.dataTransfer.getData(MY_TYPE);
  if (data) {
    let template = document.createElement('template');
    template.innerHTML = data.trim();
    let dataElement = template.content.firstChild;
    dataElement.draggable = false;
    destinationUL.appendChild(dataElement);
    appendToOutput(`${event.type} (${data})`);
  } else {
    appendToOutput(`${event.type}; no data found data transfer`);
  }
}, false);

...
```

```
  //  finally remove the dragged item from its parent
  element.addEventListener("dragend", function(event) {
    event.preventDefault();
    if (event.dataTransfer.dropEffect == "move") {
      let element = event.target;
      element.parentNode.removeChild(element);
      console.log(`dragend: element removed from its parent`);
    }
  }, false);
}
```

Bemerkungen

- Eine gute Übersicht über den Ablauf von Drag and Drop ist auf [Moz23] zu finden.
- Bei Chrome oder Opera funktioniert das Verschieben auch zwischen zwei Browserinstanzen (in/ab Version 96.0 aber wieder nicht . . .).

14 Klassen/Vererbung in JavaScript

Die Sprachkonstrukte `class` und `extends` sind in JavaScript seit ECMAScript 2015 (ES 6) vorhanden.

```
class A {
  constructor(something = "default") {
    this.something = something;
  }
  foo() {
    let s = `${this.className}.foo called;
      something: "${this.something}"`;
    return s;
  }
  get className() {
    return this.constructor.name;
  }
}
```

```
class B extends A {
  constructor (...args) {
    super(...args); // spread operator
  }

  // overrides A.foo()
  foo () {
    let s = "42: " + super.foo();
    return s;
  }

  // a new function only for class B
  bar () {
    let s = `${this.className}.bar called`;
    return s;
  }
}
// TODO: add useful output statements!
```

HTML-Seite zum Testen der Klassen `A` und `B`

```
<html>
<head><title>JS Inheritance</title>
<script type="text/javascript" src="classes01.js"></script>
</head>
<body>
  <form action="#">
    <input id="aFoo" type="button" value="a.foo()" />
    <input id="aBar" type="button" value="a.bar()" />
    <input id="bFoo" type="button" value="b.foo()" />
    <input id="bBar" type="button" value="b.bar()" />
  </form>

  <form action="#">
    <input id="aFooUnboundThis" type="button"
       value="a.foo() with unbound 'this'" />
  </form>
</body>
</html>
```

Event Listener für die Buttons

```
window.addEventListener("load", function () {
  let a = new A("dies");
  let b = new B("das"); // new B() or new B("das", "mehr")

  document.querySelector("#aFoo")
    .addEventListener("click", () => a.foo());
  document.querySelector("#aBar")
    .addEventListener("click", () => a.bar());
  document.querySelector("#bFoo")
    .addEventListener("click", () => b.foo());
  document.querySelector("#bBar")
    .addEventListener("click", () => b.bar());

  // unbound context of this!
  document.querySelector("#aFooUnboundThis")
    .addEventListener("click", a.foo);
});
```

Bemerkungen zu Klassen und Vererbung in JavaScript

- Diese Art der Instanziierung von Klassen bzw. der Vererbung sollte unbedingt der „alten“ Möglichkeit mittels Konstruktorfunktionen resp. Prototypen vorgezogen werden.
- Da JavaScript die Übergabe beliebig vieler Parameter beliebigen Typs erlaubt, ist kein Überladen von Konstruktoren oder Methoden wie z. B. in Java möglich.