

# DATABASE SYSTEMS

## THE COMPLETE BOOK

SECOND EDITION

Hector Garcia-Molina

## Chapter 11

© CourseSmart

# The Semistructured-Data Model

Stanford University

Stanford  
ONLINE  
Lagunita

Databases: DB2 XML Data

everestso

Courseware

Course Info

Discussion

Wiki

Progress

Readings

Software Guide

▶ Getting Started

▼ XML Data

Well-formed XML

DTDs, IDs, and IDREFs

XML Schema

XML Quiz  
Quiz



DTD Exercises  
Exercise



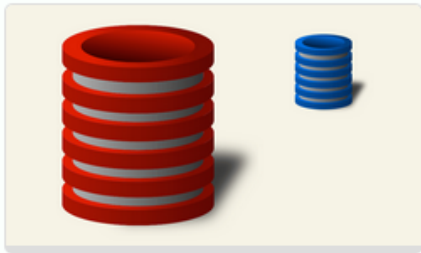
▶ Course Completion



Each multiple-choice quiz problem is based on a "root question," from which the system generates different correct and incorrect choices each time you take the quiz. Thus, you can test yourself on the same material multiple times. We strongly urge you to continue testing on each topic until you complete the quiz with a perfect score at least once. Simply click the "Reset" button at the bottom of the page for a new variant of the quiz.

After submitting your selections, the system will score your quiz, and for incorrect answers will provide an "explanation" (sometimes for correct ones too). These explanations should help you get the right answer the next time around. To prevent rapid-fire guessing, the system enforces a minimum of 10 minutes between each submission of solutions.

# Stanford Online: DB2 XML Data



## XML Data

Databases - DB2

Started - Jun 09, 2014



[View Course](#)

Your final grade: **100%.**

[Download Statement \(PDF\)](#)

Stanford University

Stanford  
Online  
Lagunita

Databases: DB2 XML Data

everestso

Courseware

Course Info

Discussion

Wiki

Progress

Readings

Software Guide

▸ Getting Started

▾ XML Data

Well-formed XML

DTDs, IDs, and IDREFs

XML Schema

XML Quiz  
Quiz



DTD Exercises  
Exercise



▸ Course Completion



Each multiple-choice quiz problem is based on a "root question," from which the system generates different correct and incorrect choices each time you take the quiz. Thus, you can test yourself on the same material multiple times. We strongly urge you to continue testing on each topic until you complete the quiz with a perfect score at least once. Simply click the "Reset" button at the bottom of the page for a new variant of the quiz.

After submitting your selections, the system will score your quiz, and for incorrect answers will provide an "explanation" (sometimes for correct ones too). These explanations should help you get the right answer the next time around. To prevent rapid-fire guessing, the system enforces a minimum of 10 minutes between each submission of solutions.

# Semistructured Data

- Does not enforce a rigid structure
- Self-Describing
  - Does not have a separate schema declaration
- Lack of structure may make finding data more expensive
- Offers significant advantages for users.
  - Much simpler

# Sec 11.1 Semistructured Data

- Semistructured-Data is going to play an important role in DB.
  - It'll be used in integrating multiple database
    - Providing a format to describe data from multiple database w/ multiple schemas.
    - It provides the model for XML that's used to share data w/ Web.

# Semistructured Data Representation

- A database of semistructured data is a collection of nodes.
- Nodes are either:
  - Leaf
    - Atomic types like string or number
  - Interior
    - One or more arc out
    - Arcs are labeled
  - Root node
    - Top level
    - Represents all the data

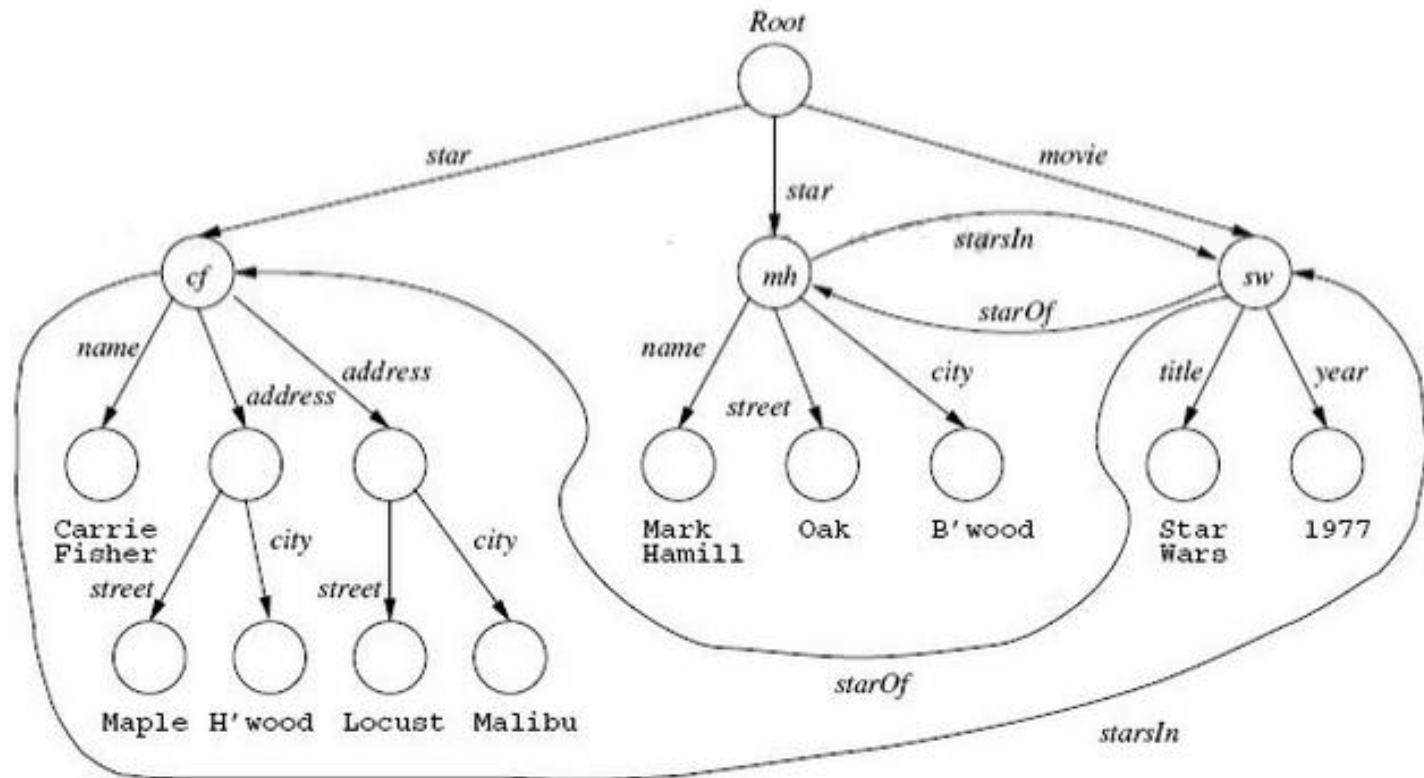


Figure 11.1: Semistructured data representing a movie and stars

- Semistructured data for Stars of Show
- Label  $L$  on arc from node  $N$  to node  $M$  can represent:
  - $N$  is an object, and  $M$  is an attribute,  $L$  is attribute name
  - $N$  and  $M$  objects and  $L$  is relationship between them.

# Semistructured Data is Everywhere

- Data Exchange
- Legacy Database Integration
- NoSQL Systems
- We'll discuss two variants:
  - XML
  - JSON

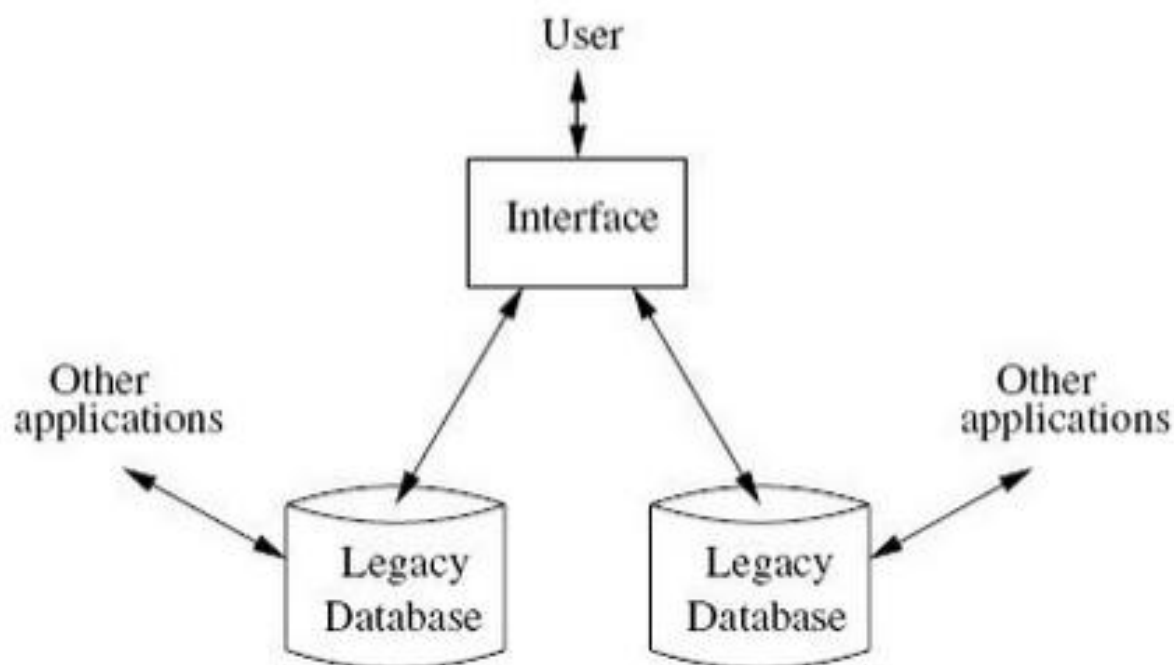


Figure 11.2: Integrating two legacy databases through an interface that supports semistructured data



# XML

- XML stands for:
  - The **E**xtensible **M**arkup **L**anguage.
- The XML Specification is defined and published by the World Wide Web Consortium,
  - <http://www.w3.org/>, W3C.
- XML is a *self-describing* format.
- It is a subset of ISO 8879,
  - Standard Generalized Markup Language
  - also the parent of HTML
- Origins in technical publishing and manuals.

# Semistructured Data

- Lets look at storing two actors from Glee in a text file:
  - Corey Monteith
  - Chris Colfer
- We know they are stars, but the computer would now.

```
<Stars updated="2014-07-18">  
  <Star>Corey Monteith</Star>  
  <Star>Chris Colfer</Star>  
</Stars>
```

- XML lets us tag the data for understanding.

# Elements, Attributes, and Content

- The three most common parts of an XML document are
  - *elements*,
  - *attributes*
  - *content*.
- Again:  

```
<Stars updated="2014-07-18">  
  <Star>Chris Colfer</Star>  
</Stars>
```
- Here, Stars & Star are XML elements;
- *updated* is an XML attribute whose value, 2014-07-18, applies to the <Stars> element;
- “Chris Colfer” is **content**, the actual text.

# Well-Formed XML

- An XML document that satisfies the minimum requirements is *well formed*.
- “well-formed” distinguishes basic XML documents from “valid” documents;
  - XML DTD validation is a stricter level of conformance.
- A document not well-formed is, strictly speaking, NOT an XML document at all.

# The XML Declaration

- XML documents start with an optional *XML Declaration* which:
  - Tells software the document is trying to be in XML;
  - Identifies the version of XML in use;
  - Identifies the character encoding;
  - Tells the XML processor reading the document whether it must first fetch an external “Document Type Definition” (DTD) resource before continuing.

# XML Declaration

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

- **<?xml . . . . ?>**
  - contains the XML Declaration.
- **version="1.0"**
  - says the document is using XML version 1.0
- **encoding="UTF-8"**
  - says the document is in the Unicode UTF-8 encoding.

# XML Declaration

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

- **standalone="yes"**
  - is used only when there is a *Document Type Definition*, or DTD, present.
- So a more usual XML declaration looks like this:

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

- The XML declaration must be at the very start of the XML document
  - do not put blank lines or comments before it.

# XML Elements

- ELEMENTS are the basic building block of XML.
  - Every XML document contains at least one element.
- The beginning of an element is represented by a *start tag*:
  - an angle bracket immediately followed by the name of the element
  - and then a right angle bracket ">"
- The end of an element is
  - </ followed by the name followed by > like this:

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



# XML Tags

- There are actually *three* kinds of XML tag:
  - A start tag, also called an open tag: `<stars>`
  - An end tag, also called a close tag: `</stars>`
  - A self-closing tag, also called an empty tag: `<Stars/>`

# XML Tags

- There must be no space between the < or </ and the name.
- You can have space and newlines after the name and before the closing >-sign:
  - `<Stars >`
- The self-closing tag is used instead of the start and end tags, and can be used only when an element has no content.
  - `<tag id="foo" />`
  - `<tag id="foo"></tag>`

```
<? xml version = "1.0" encoding = "utf-8" standalone = "yes" ?>
<StarMovieData>
  <Star>
    <Name>Carrie Fisher</Name>
    <Address>
      <Street>123 Maple St.</Street>
      <City>Hollywood</City>
    </Address>
    <Address>
      <Street>5 Locust Ln.</Street>
      <City>Malibu</City>
    </Address>
  </Star>
  <Star>
    <Name>Mark Hamill</Name>
    <Street>456 Oak Rd.</Street>
    <City>Brentwood</City>
  </Star>
  <Movie>
    <Title>Star Wars</Title>
    <Year>1977</Year>
  </Movie>
</StarMovieData>
```

Figure 11.3: An XML document about stars and movies

# Example

```
1 <?xml version="1.0" ?>
2
3 <Course_Catalog>
4
5   <Department Code="CS">
6
7     <Title>Computer Science</Title>
8
9     <Chair>
10       <Professor>
11         <First_Name>Jennifer</First_Name>
12         <Last_Name>Widom</Last_Name>
13       </Professor>
14     </Chair>
15
16     <Course Number="CS106A" Enrollment="1070">
17       <Title>Programming Methodology</Title>
18       <Description>Introduction to the engineering of computer applications emphasizing modern software engineering princip
19       <Instructors>
20         <Lecturer>
21           <First_Name>Jerry</First_Name>
22           <Middle_Initial>R.</Middle_Initial>
23           <Last_Name>Cain</Last_Name>
24         </Lecturer>
25         <Professor>
26           <First_Name>Eric</First_Name>
27           <Last_Name>Roberts</Last_Name>
28         </Professor>
29         <Professor>
30           <First_Name>Mehran</First_Name>
31           <Last_Name>Sahami</Last_Name>
32         </Professor>
33       </Instructors>
34     </Course>
35
36     <Course Number="CS106B" Enrollment="620">
37       <Title>Programming Abstractions</Title>
```

# XML Names

- An XML name must start with one of:
  - an underscore (\_);
  - any Unicode upper or lower case letter;
- Characters after the first one can also be a dash or digit.
- The colon “:” is also allowed but only for use with XML namespaces
  - as explained later.
- You can never include spaces or punctuation in XML names.
- You can’t start names with “XML” (in any mixture of UPPER or lower case)

# Root Element

- All XML documents have exactly one element called the *root* element.
  - The root element start tag is the first start tag in the document,
  - its close tag is at the very end of the document.
- All the rest of the document is thus “inside” the root element.
- The term *root* comes because an XML document is often viewed as representing a “tree” of elements.

# Nested Elements

- Inside the root element you can have any mixture of free text and more elements.
- Elements must always “nest” properly. You cannot “overlap” tags like this:

```
<verse>  
  <line>And they said, <quote>“We must</line>  
  <line>take away his shoes!”</quote> but,</line>  
</verse>
```

- Remember that there must always be a single outermost root element.

# XML Attributes

- ATTRIBUTES are name-value pairs associated with elements.
  - You put them inside start tags or inside self-closing tags.
- Attribute values *must* be in quotes. You can use " or '
- You can only have *one* attribute of a given name in any tag.
- There must always be a name, an = sign, and a value:
  - `<option selected = "selected"> . . . </option>`



# Entity References

- An *XML entity* is a named string that can be defined in a DTD and used anywhere.
- You use an entity by referencing it: an entity reference is an &-sign followed by a name and then a semicolon.
- **There are five built-in XML entities: &amp; (&), &apos; ('), &quot; ("), &lt; (<) and &gt; (>). You can always use these.**

# Numeric Character References

- You can refer to any legal Unicode character by codepoint, either in decimal or Hex.
- Numeric Character References are different from entities in that you don't have to declare them.
- Example:
  - `&#160;` or `&#xA0;` is a non-breaking space.
- It is possible to define your own entities.

# XML Comments

- XML comments are passed back to the parser, but applications usually ignore them.
  - Use `<!--` and `-->` to start and end comments.

`<!-- This is a comment -->`

- Comments are not allowed to contain “--” inside them.
- Comments do not nest.
- One comment convention uses stars for more visibility:

```
<!--* This is a comment
    * over multiple
    * lines
    *-->
```

Bookstore-noDTD.xml - D:\Documents\GitHub\c040f15 - Atom

File Edit View Selection Find Packages Help

c040f15

- .git
- cpp
  - ch1
    - ch1.3.2.cpp
    - ch1.4.cpp
    - hw.cpp
  - ch2
  - ch3
  - ch4
    - ch4.ex.1.cpp
    - ch4.ex.2.cpp
    - ch4.ex.2.spaces.cpp
    - ch4.ex.3.cpp
  - ch5
    - ch5.ex.1.cpp
    - ch5.ex.2.cpp
    - ch5.ex.2.txt
    - ch5.ex.3.cpp
    - ch5.ex.3.txt
    - ch5.ex.3b.cpp
  - addTable1.cpp
  - addTable2.cpp
  - cos.cpp
  - g.bat
  - noise.rand.cpp
  - sin.cpp
  - wumpus.1.cpp
  - zb.pa.1.2.1.cpp
- doc
  - coding.md
  - cygwin.md
  - git.md
  - README.md

REA... x git.md x wum... x zb.pa... x ch.3... x ch.4... x codi... x ch.5... x ch.5... x hw.cpp x wum... x Bookstore-no... x sin.cpp x cygw... x

```

1  <?xml version="1.0" ?>
2  <!--Bookstore with no DTD-->
3
4  <Bookstore>
5      <Book ISBN="ISBN-0-13-713526-2" Price="85" Edition="3rd">
6          <Title>A First Course in Database Systems</Title>
7          <Authors>
8              <Author>
9                  <First_Name>Jeffrey</First_Name>
10                 <Last_Name>Ullman</Last_Name>
11             </Author>
12             <Author>
13                 <First_Name>Jennifer</First_Name>
14                 <Last_Name>Widom</Last_Name>
15             </Author>
16         </Authors>
17     </Book>
18     <Book ISBN="ISBN-0-13-815504-6" Price="100">
19         <Remark>
20             Buy this book bundled with "A First Course" - a great deal!
21         </Remark>
22         <Title>Database Systems: The Complete Book</Title>
23         <Authors>
24             <Author>
25                 <First_Name>Hector</First_Name>
26                 <Last_Name>Garcia-Molina</Last_Name>
27             </Author>
28             <Author>
29                 <First_Name>Jeffrey</First_Name>
30                 <Last_Name>Ullman</Last_Name>
31             </Author>
32             <Author>
33                 <First_Name>Jennifer</First_Name>
34                 <Last_Name>Widom</Last_Name>
35             </Author>
36         </Authors>
37     </Book>
38 </Bookstore>
  
```

D:\Documents\GitHub\Fall14\226\L2\Bookstore-noDTD.xml 2:1

Unix(LF) UTF-8 XML 1 update

## 11.1.4 Exercises for Section 11.1

**Exercise 11.1.1:** Since there is no schema to design in the semistructured-data model, we cannot ask you to design schemas to describe different situations. Rather, in the following exercises we shall ask you to suggest how particular data might be organized to reflect certain facts.

- a) Add to Fig. 11.1 the facts that *Star Wars* was directed by George Lucas and produced by Gary Kurtz.
- b) Add to Fig. 11.1 information about *Empire Strikes Back* and *Return of the Jedi*, including the facts that Carrie Fisher and Mark Hamill appeared in these movies.
- c) Add to (b) information about the studio (Fox) for these movies and the address of the studio (Hollywood).

## 11.2.8 Exercises for Section 11.2

**Exercise 11.2.1:** Repeat Exercise 11.1.1 using XML.

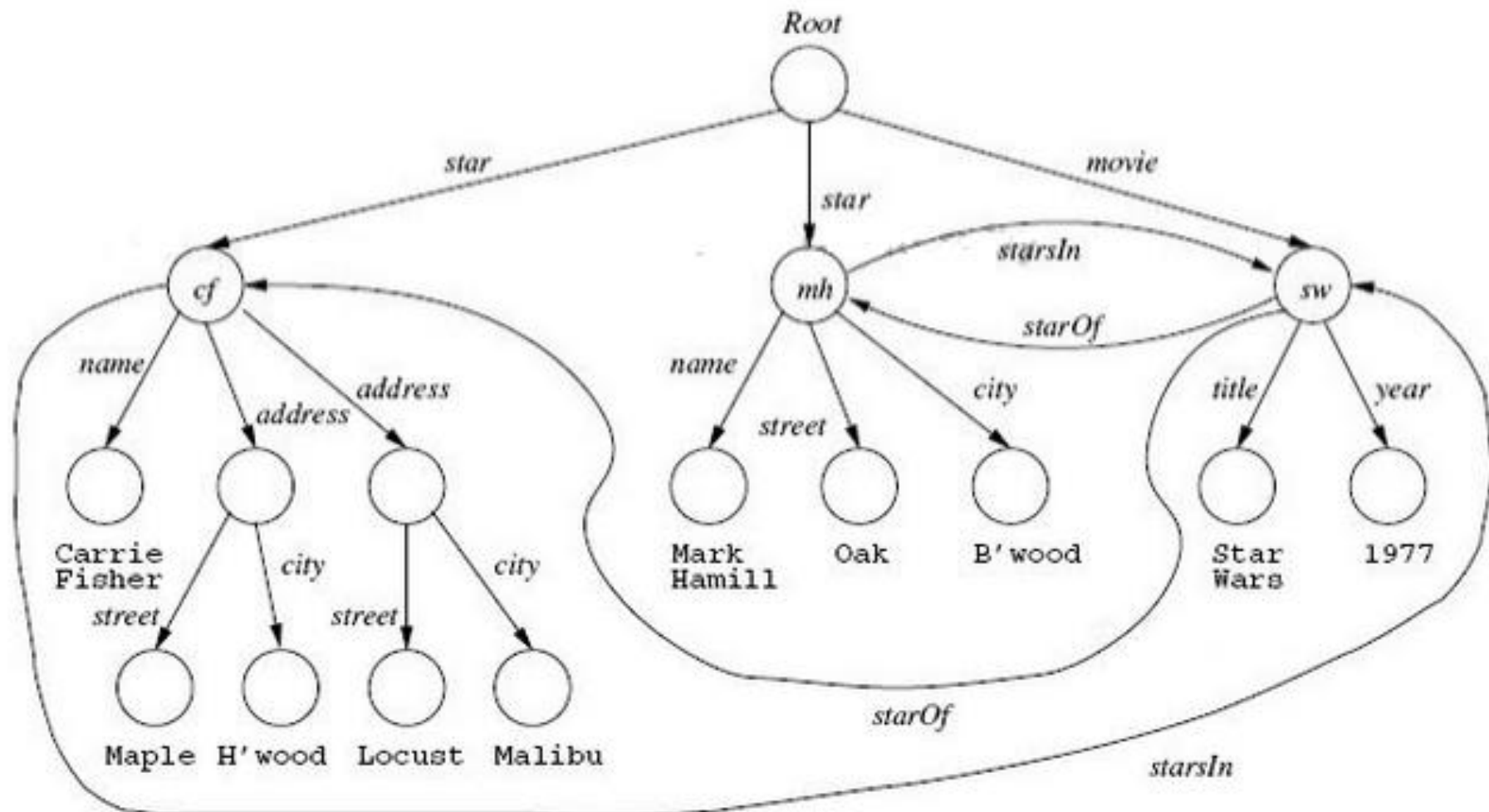


Figure 11.1: Semistructured data representing a movie and stars

- well-formed XML that satisfies the following conditions:
  - It has a root element “diner”
  - The root element has 3 “item” subelements
  - Each of the “item” subelements has an attribute named "name"
  - The values of the "name" attributes for the 3 tasks are “coffee”, “burger”, and “fries”
- Does the XML below meet the above requirements?

```
<diner>  
  <item name=coffee/>  
  <item name=burger/>  
  <item name=fries/>  
</diner>
```

- well-formed XML that satisfies the following conditions:
  - It has a root element “diner”
  - The root element has 3 “item” subelements
  - Each of the “item” subelements has an attribute named "name"
  - The values of the "name" attributes for the 3 tasks are “coffee”, “burger”, and “fries”
- Does the XML below meet the above requirements.

```
<diner>  
  <item name="coffee"/>  
  <item name="burger"/>  
  <item name="fries"/>  
<diner>
```



- well-formed XML that satisfies the following conditions:
  - It has a root element “diner”
  - The root element has 3 “item” subelements
  - Each of the “item” subelements has an attribute named "name"
  - The values of the "name" attributes for the 3 tasks are “coffee”, “burger”, and “fries”
- Does the XML below meet the above requirements?

```
<diner>  
  <item name="coffee"/>  
</diner>  
<diner>  
  <item name="burger"/>  
</diner>  
<diner>  
  <item name="fries"/>  
</diner>
```

# NAMESPACES

*11.2. XML*

493

**11.2.6 Namespaces**

# XML Namespaces

- A way to group some elements and attributes together under a common label
- Example: you hear people talking about a table:
  - Could be a relational database table...
  - Or a table in a Web page, with a border round it...
  - Or a wooden dining room table with extensible flaps!
- Namespaces give you some context.

# Some Sample Namespaces

- Probably hundreds of thousands of XML namespaces in use, with a few hundred common ones.
- The next few slides show some of the more common XML namespaces.
- People working in one field (aircraft documentation, say) may never encounter namespaces very common in some other field (open source software package descriptions, for example) and so this list is fairly arbitrary.

# XML, XMLNS, XSD

- The *xml* prefix is always bound to `http://www.w3.org/XML/1998/namespace` - you don't need to declare it.
- The *xmlns* prefix is always bound to `http://www.w3.org/2000/xmlns/` (note the trailing slash); this one *must not* be declared in a document.
- W3C XML Schema uses `http://www.w3.org/2001/XMLSchema` - by convention usually bound to “xs” as a prefix.

# SOAP, Web Services

- SOAP is used as an XML-based way for computers to communicate over the Web:
  - companies often provide a “web service API” to use their services remotely, such as obtaining a stock quote or a fragment of a Google map.
- Web services are often being replaced by JSON services on the Internet, but are still heavily used inside Intranets.
- There are lots of Web services namespaces, such as:  
`http://schemas.xmlsoap.org/soap/envelope/`

# XML Soap

[< Previous](#)[Next >](#)

- SOAP stands for **S**imple **O**bject **A**ccess **P**rotocol
- SOAP is an application communication protocol
- SOAP is a format for sending and receiving messages
- SOAP is platform independent
- SOAP is based on XML
- SOAP is a W3C recommendation

## Why SOAP?

It is important for web applications to be able to communicate over the Internet.

The best way to communicate between a browser and servers is by using a standard protocol. The best way to communicate between a browser and servers. SOAP was created for this purpose.

SOAP provides a way to communicate between different technologies and programming languages.

## Skeleton SOAP Message

```
<?xml version="1.0"?>

<soap:Envelope
  xmlns:soap="http://www.w3.org/2003/05/soap-envelope/"
  soap:encodingStyle="http://www.w3.org/2003/05/soap-encoding">

  <soap:Header>
    ...
  </soap:Header>

  <soap:Body>
    ...
    <soap:Fault>
      ...
    </soap:Fault>
  </soap:Body>

</soap:Envelope>
```

# RSS and Atom

- The Really Simple Syndication format, RSS, is used for news feeds (“channels”);
  - most RSS formats do not use namespaces, but a newer variant called Atom uses: `http://purl.org/atom/ns#`
- The trailing # is common in RDF (a non-XML linked data / Semantic Web format), but has no XML-defined meaning.
  - The RDF convention is that names associated with a namespace can be appended to the namespace URI, and a # or / at the end of the URI makes that work better.



# XSLT

- The XSL Transformation language, XSLT, and the XML Path Language, XPath, are the most widely used XML-based languages today, with XQuery a distant third.
- XSLT uses `http://www.w3.org/1999/XSL/Transform` and this is very commonly bound to an *xml* prefix.
- XPath and XQuery do not use XML syntax, and do not have their own XML namespaces in the same way.
- XSLT also uses a *version* attribute, currently one of "1.0", "1.1", "2.0" or "3.0" in practice.

# XHTML

- XHTML is an XML-based version of HTML, the markup language used for the World Wide Web.
- XHTML uses `http://www.w3.org/1999/xhtml` as its namespace URI.
- XHTML documents can be served as HTML or as XML, but when served as HTML they must use a default namespace, not a prefix.

# Namespaces and Vocabularies

- It's common to mix different vocabularies (tagsets) in a single document;
  - namespaces prevent problems if two vocabularies define the same name.
- Example: embedding fragments of XHTML inside another format,
  - perhaps a person's résumé inside a business document,
  - or an item description in a catalog.
  - What if there's more than one <title> element?
- XML is about sharing data and using other people's vocabularies, so you can't simply change the names.

# What's in a Name?

- That shared name, the namespace name, must be unique, so that someone else can't accidentally use the same name.
  - “bread-prices” would not be a good choice.
- The name *should* be in a space you own, so that you know no-one else will use it without your permission.
- The name *should* be at least vaguely descriptive.
- The name *must* be a URI or URN.

$\text{xmlns:}name = \text{“URI”}$

- Attribute “xmlns” is used to state namespace in use.
- “name” is then used to modify tags to note from namespace:
  - “name:tag” is the new namespace tag

**Example 11.7:** Suppose we want to say that in element `StarMovieData` of Fig. 11.5 certain tags belong to the namespace defined in the document `infolab.stanford.edu/movies`. We could choose a name such as `md` for the namespace by using the opening tag:

```
<md:StarMovieData xmlns:md=  
    "http://infolab.stanford.edu/movies">
```

- `StarMovieData` is also part of this namespace.

# URI, URL, URN

- A URL is a *Uniform Resource Locator*; a Web address you could type into a Web browser and “visit.”
- But what if you just used a URL as a name for something, an identifier?
  - That would be a Uniform Resource Identifier, a URI.
- A URI can actually be either a URL or a URN;
  - a URN is a name, such as the ISBN of a book, used to stand in for real-life objects that have no URL:
- urn:isbn:9781118162132 is “Beginning XML.”

# XML Namespaces and URIs

- XML Namespaces use URIs for namespace names.
- Two namespace URIs are considered the same if they are written the same, without unquoting them.
- An XML namespace name does not “point” to anything: it’s just a name.
- URIs were used because there’s already a good distributed mechanism for people to create their own URIs without any fear of conflicts.



# Namespace Example

- The namespace identifies who owns the format:

```
<Months xmlns="http://wrox.com/ns/months/">  
  <Month index="1">January</Month>  
  <Month index="2">February</Month>  
  . . .  
</Months>
```

# The Default Namespace

- When you use *xmlns = value* like this, you are setting a “default namespace.”
- All elements inside the one with the namespace declaration are said to be “in” that namespace.
- Attributes are not in any namespace.
- In the absence of *xmlns*, elements and attributes are not in any namespace.
- **Note:** Do not think of *xmlns* as an XML attribute. It looks like an attribute, but it is not an attribute.

# The Namespace is Part of the Name

- When an element or attribute has an associated namespace URI, *that namespace name becomes part of the element's name.*
- This is very important when you're using an API or language such as XPath that lets you fetch elements from XML documents based on the element name.
- You always have to pay attention: if elements are in a namespace, code handling the XML (e.g. XSLT, XQuery, Java) must account for that namespace.

# Multiple Namespaces

- Very often you need to refer to more than one namespace in a document:
  - part of the purpose of namespaces in XML is so you mix elements from different vocabularies.
- You can change the default namespace on any element, for all the elements beneath (inside),
  - but what if you need two namespaces in one element, or if you need to use an attribute from a particular namespace?
- The solution is the *Namespace Prefix*, a short string that lets you work with multiple namespaces.

# Reserved Names

- As with XML itself, names starting with XML in any combination of upper and lower case (XML, xml, Xml, xMl...) are reserved, and you can't use them for a prefix.
- The namespace prefix *xml* is bound automatically to `http://www.w3.org/XML/1998/namespace` and you can declare it as long as you use exactly that value.
- The namespace prefix *xmlns* is bound automatically to `http://www.w3.org/2000/xmlns/` but you must not declare it yourself; the value is available for programmers.

# Example Using Prefixes

```
<m:Months  
  xmlns:h="http://example.org/holidays/"  
  xmlns:m="http://wrox.com/ns/months/">  
  <m:Month index="1">  
    <m:Name>January</m:Name>  
    <h:Holiday day="Hangover Day"</h:Holiday>  
  </m:Month>  
</m:Months>
```

- The prefixes (here *h* and *m*) are *not* part of the element names; they stand for the full namespace name.
- Using a prefix for Months has the same result as using the default namespace, and you can mix the two styles.

# Namespaces on Attributes

- Attributes are in no “no namespace” unless they are explicitly prefixed.
- Usually this is what you want, but some vocabularies define attributes to be used by other vocabularies: here is an example using XLink:

```
<svg version="1.0" xmlns="http://www.w3.org/2000/svg"
      xmlns:xlink="http://www.w3.org/1999/xlink">
  <image x="10" y="10" width="300px" height="310px"
        xlink:href="isaac-newton-300x310.jpg">
    <title>Sir Isaac Newton</title>
  </image>
</svg>
```

# Prefixes or Default Namespace?

- Many XML documents use explicit namespace prefixes on every element, even though a single default namespace declaration on the root element would do the same thing.
- The prefixes make it obvious that namespaces are in use;
- They also make it easier to use copy and paste between documents (as long as you remember to declare the namespace prefix in the resulting document).
- The prefixes may make the document harder to read, but it's really a matter of preference.



# In-Scope Namespaces

- A default namespace declaration affects all unprefixes elements inside it, as well the element with the declaration.
- A namespace declaration with a prefix (sometimes called a *binding*) makes that prefix available to all elements inside it as well as to the element with the declaration.
- For any element, the set of available namespaces is called the *in-scope namespaces*.
- The scope rules for namespaces are similar to variable names in programming languages.

# Rebinding and Unbinding

- You can't have two namespace bindings for the same prefix on an element
  - it would be like having two attributes with the same name, `xmlns:fred` and `xmlns:george`.
- If an inner element binds a namespace to a prefix, *arthur* for example, any binding to the same prefix by outer elements higher in the document tree are hidden.
- You can get rid of the default namespace like this:

```
<colin xmlns="">  
  <arthur>not in any namespace</arthur>  
</colin>
```
- You cannot unbind a namespace prefix except in XML 1.1, which is not widely supported.

# QNames in Content

- An XML name with a namespace prefix and a colon is called a Qualified Name,
  - QName to its friends.
- An XML name without a namespace prefix is a No-Colon Name, or NCName.
- Some languages use NCNames inside attributes or elements (e.g. XSLT uses them in XPath fragments).
- You have to be careful when processing such documents, as if the prefix is changed in the namespace binding, the QNames may have to be changed inside the text too!

```
<xsl:copy select="//bbc:character[bbc:name = 'The Doctor']"/>
```

# When to use Namespaces

- A namespace can help to identify a document format and who is responsible for it.
- Namespaces help disambiguate elements of the same name in different vocabularies.
- Namespaces support “mixins”—elements and attributes designed to be used in other languages.
- Namespaces can also help with validation using W3C XML Schema or RelaxNG, as shown later in this course.

# When *not* to use Namespaces

- Namespaces make all XML processing more complicated.
- Namespaces are frequently misunderstood: most common is to think that there should be something - perhaps a schema - “at” the namespace URI. It’s really just a name.
- Not all XPath systems work properly with namespaces.
- If your XML is just for your own use, not to be shared, it’s often best to avoid namespaces.
- XML for AJAX (between Web server and browser) is best without namespaces: it’s not worth the complexity.

# Namespaces and Versioning

- Namespace names are treated as strings with no internal structure: a namespace matches or it doesn't. Therefore,

`"http://buy.sheep.now/api/1.0"`

and,

`"http://buy.sheep.now/api/1.1"`

are seen by XML software as two completely unrelated namespaces.

- Usually this is not what you want, so it's better to have a separate version attribute:
- `<bsn:order xmlns:bsn="http://buy.sheep.now/api/" version="1.1">`

# Document Type Definitions

## 11.3 Document Type Definitions

For a computer to process XML documents automatically to be something like a schema for the documents. It is kinds of elements can appear in a collection of documents can be nested. The description of the schema is given by rules, called a *document type definition*, or DTD. It is in or communities wishing to share data will each create a form(s) of the data they share, thus establishing a shared of their elements. For instance, there could be a DTD structures, a DTD for describing the purchase and sale on.

### 11.3.1 The Form of a DTD

**Stanford**  
ONLINE  
Lagunita

Databases: DB2 XML Data

Home Course Discussion Wiki

Bookmarks


▶ Getting Started


▼ XML Data

Well-formed XML

DTDs, IDs, and IDREFs

XML Schema

XML Quiz  
Quiz 

DTD Exercises  
Exercise 

▶ Course Completion

# Document Type Definitions

- A Document Type Definition defines an XML vocabulary, sometimes also loosely called a tagset.
- A DTD is just a plain text file with a particular syntax.
- XML files can contain a pointer to the DTD they use.
- The DTD defines elements, attributes and entities that can appear inside such an XML file.



# Examples

- There are tens of thousands of XML vocabularies, a great many of which have DTDs.
- Some examples:
  - XMLTV (TV Listings)
  - DocBook (technical documentation)
  - XHTML (XML HyperText Markup Language)
  - ATA Spec 2000 (Aircraft manuals and parts)
  - Text Encoding Initiative (TEI) for humanities documents

Search

Log in

<xmltv>

page

discussion

view source

history

XMLTVFormat

XMLTV File format

helpful links

■ Main Page

■ Sourceforge Project

■ Download Info

■ Nightly Status

■ Release Status

■ Wiki Changes

search

Go

Search

toolbox

■ What links here

■ Related changes

■ Special pages

■ Printable version

■ Permanent link

The format used differs from most other XML-based TV listings formats in that it is written from the user's point of view, rather than the broadcaster's. It doesn't divide listings into channels, instead all the channels are mixed together into a single unified listing. Each programme has details such as name, description, and credits stored as supplements, but metadata like broadcast details are stored as attributes. There is support for listings in multiple languages and each programme can have 'language' and 'original language' details.

The XMLTV File Format was originally created by Ed Avis, and is currently maintained by the [XMLTVProject](#). The current CVS version of the DTD is available [here](#).

There are additional requirements on grabbers if they want to be "baseline compliant". See [XmltvCapabilities](#)

Since the [DTD](#) is available, you can also use [XmltvValidation](#).

Details

An XMLTV file has 2 types of records

■ 'channel' records, store information about channels

■ 'program' records, store information about individual episodes

Most of the information is optional and may not be available from all sources

This is what a sample xmltv file looks like

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE tv SYSTEM "xmltv.dtd">

<tv source-info-url="http://www.schedulesdirect.org/" source-info-name="Schedules Direct" generator-info-name="XMLTV/$Id: tv_grab_na_dd.in,v
  <channel id="I10436.labs.zap2it.com">
    <display-name>13 KERA</display-name>
    <display-name>13 KERA TX42822:-</display-name>
    <display-name>13</display-name>
    <display-name>13 KERA fcc</display-name>
    <display-name>KERA</display-name>
    <display-name>KERA</display-name>
```

66

```

<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE tv SYSTEM "xmltv.dtd">

<tv source-info-url="http://www.schedulesdirect.org/" source-info-name="Schedules Direct" generator-info-name="XMLTV/$Id: tv_grab_na_dd.in,v
<channel id="I10436.labs.zap2it.com">
  <display-name>13 KERA</display-name>
  <display-name>13 KERA TX42822:-</display-name>
  <display-name>13</display-name>
  <display-name>13 KERA fcc</display-name>
  <display-name>KERA</display-name>
  <display-name>KERA</display-name>
  <display-name>PBS Affiliate</display-name>
  <icon src="file:///C:/Perl/site/share/xmltv/icons/KERA.gif" />
</channel>
<channel id="I10759.labs.zap2it.com">
  <display-name>11 KTVT</display-name>
  <display-name>11 KTVT TX42822:-</display-name>
  <display-name>11</display-name>
  <display-name>11 KTVT fcc</display-name>
  <display-name>KTVT</display-name>
  <display-name>KTVT</display-name>
  <display-name>CBS Affiliate</display-name>
  <icon src="file:///C:/Perl/site/share/xmltv/icons/KTVT.gif" />
</channel>
<programme start="20080715003000 -0600" stop="20080715010000 -0600" channel="I10436.labs.zap2it.com">
  <title lang="en">NOW on PBS</title>
  <desc lang="en">Jordan's Queen Rania has made job creation a priority to help curb the staggering unemployment rates among youths in the
  <date>20080711</date>
  <category lang="en">Newsmagazine</category>
  <category lang="en">Interview</category>
  <category lang="en">Public affairs</category>
  <category lang="en">Series</category>
  <episode-num system="dd_progid">EP01006886.0028</episode-num>
  <episode-num system="onscreen">427</episode-num>
  <audio>
    <stereo>stereo</stereo>
  </audio>
  <previously-shown start="20080711000000" />
  <subtitles type="teletext" />
</programme>
<programme start="20080715010000 -0600" stop="20080715023000 -0600" channel="I10436.labs.zap2it.com">
  <title lang="en">Mystery!</title>
  <sub-title lang="en">Foyle's War, Series IV: Bleak Midwinter</sub-title>
  <desc lang="en">Foyle investigates an explosion at a munitions factory, which he comes to believe may have been premeditated.</desc>

```

```
<!-- The root element, tv.
```

Date should be the date when the listings were originally produced in whatever format; if you're converting data from another source, then use the date given by that source. The date when the conversion itself was done is not important.

To indicate the source of the listings, there are three attributes you can define:

'source-info-url' is a URL describing the data source in some human-readable form. So if you are getting your listings from SAT.1, you might set this to the URL of a page explaining how to subscribe to their feed. If you are getting them from a website, the URL might be the index of the site or at least of the TV listings section.

'source-info-name' is the link text for that URL; it should generally be the human-readable name of your listings supplier. Sometimes the link text might be printed without the link itself, in hardcopy listings for example.

'source-data-url' is where the actual data is grabbed from. This should link directly to the machine-readable data files if possible, but it's not rigorously defined what 'actual data' means. If you are parsing the data from human-readable pages, then it's more appropriate to link to them with the source-info stuff and omit this attribute.

To publicize your wonderful program which generated this file, you can use 'generator-info-name' (preferably in the form 'progname/version') and 'generator-info-url' (a link to more info about the program).

```
-->
```

```
<!ELEMENT tv (channel*, programme*)>
```

```
<!ATTLIST tv date CDATA #IMPLIED
```

```
source-info-url CDATA #IMPLIED
```

```
source-info-name CDATA #IMPLIED
```

```
source-data-url CDATA #IMPLIED
```

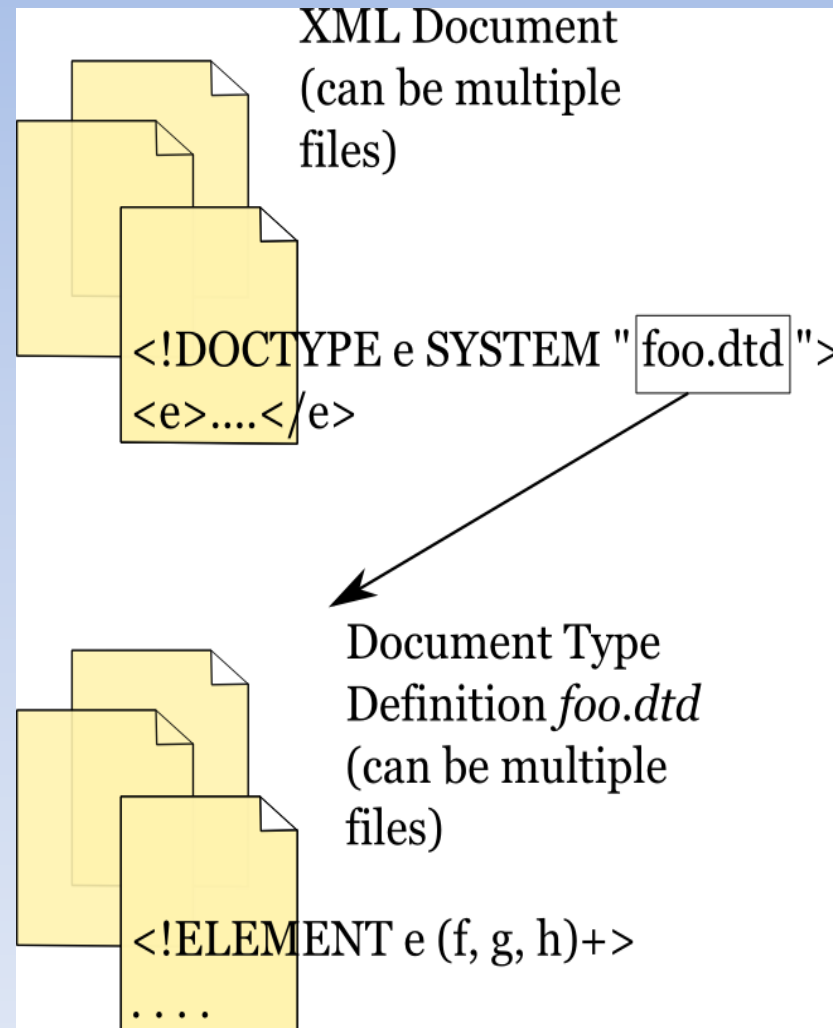
```
generator-info-name CDATA #IMPLIED
```

```
generator-info-url CDATA #IMPLIED >
```

# Validation

- A DTD defines a formal grammar, a set of rules for documents to follow.
  - If a document follows the rules correctly it is said to validate against the DTD.
- The DTD constrains which XML element names can be used, and where
- The DTD can constrain where text goes but does not constrain what that text can be;
  - for that you need a Schema instead (e.g. XSD or RelaxNG)
- DTDs are part of the XML specification and are very widely supported.

# Validation



# Linking XML to DTD

- When an XML document uses a DTD, it starts with a DOCTYPE declaration.

`<!DOCTYPE contacts>`

- This document must start with an element called contacts but has no explicit DTD.

`<!DOCTYPE contacts SYSTEM "contacts.dtd">`

- This document references contacts.dtd using a relative URI reference: the DTD is in the same place as the document, in a file called contacts.dtd.

`<!DOCTYPE contacts PUBLIC "-//Beginning XML//DTD Contact Example//EN" "contacts.dtd">`

- This document uses the same contacts.dtd file but gives it a name, a “public identifier.”

# Validation

- A validating processor starts by reading the XML document.
- It then encounters the DOCTYPE declaration and finds the link to the DTD.
- It downloads the DTD and reads it into memory.
- The processor then goes back to the XML document and applies the rules in the DTD to the rest of the XML document.



# SYSTEM identifiers

- A “SYSTEM Identifier” is a URI Reference; unlike a namespace, a system identifier can be a relative URI, in which case it’s resolved relative to the resource that contains it.

```
<!DOCTYPE contacts SYSTEM "contacts.dtd">
```

```
<!DOCTYPE contacts
```

```
    SYSTEM "http://www.example.org/contacts.dtd">
```

- The identifier is like a filename: the contents of the resource identified by the URI Reference are fetched and processed if the XML processor is doing DTD validation.
- Reminder: the separator in a URI is / and not \
- The resource pointed to be a system identifier must be a valid XML DTD; the syntax will be shown in this module.
- Later you will meet other uses for XML SYSTEM identifiers.

# PUBLIC identifiers

- A “Public Identifier” is a name. Public identifiers can’t be dereferenced directly, so they are less useful than system identifiers.

If you supply a public identifier for the DTD you must *also supply a system identifier to be used as a fall-back*:

```
<!DOCTYPE contacts
```

```
    PUBLIC "-//Wiley Inc//DTD for contacts//EN"
```

```
    "http://www.example.org/contacts.dtd">
```

- A public identifier has the syntax

```
-//owner//class description//language//version
```

- The class for a Document Type Definition is always DTD, and the //Version part is optional.
- Use an XML Catalog to tell your processor how to dereference a public DTD; XML Catalogs will be covered later.

# DTD Syntax

- Once the XML processor has read the DOCTYPE declaration and used the PUBLIC and/or SYSTEM identifier to fetch the DTD, it reads and parses the DTD.
- The DTD contains:
  - element declarations
  - attribute declarations
  - entity declarations
  - notation declarations
  - comments and processing instructions

# Element Declarations

- An element declaration has three parts:
  - The !ELEMENT keyword
  - the name of the element being declared
  - what can go inside the element
- Example:
- `<!ELEMENT student (name, age, height)>`
- This declares an element called *student* that must contain three sub-elements, *name*, *age* and *height*, in that order.
- The part inside parentheses is called the *content model* and is a kind of regular expression.

# Element Content Models

- There are four kinds of content model:
  - Element Content
  - Mixed Content
  - Empty
  - Unrestricted (ANY)
- The following slides cover each type of content model in turn.

# Element Content 1:

- An XML element that is declared to contain only other elements is said to have *element content*.
- Whitespace is also allowed in element content, and is ignored by the XML parser if it is in validating mode.
- Element content models list the elements that are allowed inside the element being declared and specify their order using a regular expression:
- `<!ELEMENT student (name, location, phone)>`

# Element Content 2:

- `<!ELEMENT student (name, location, phone)>`
- Here, the element “student” must match the regular expression (name, location, phone).

`<student>`

`<name>Susan</name>`

`<location>G51</location>`

`<phone>555-9125</phone>`

`</student>`

- The whitespace between the name, location and phone elements is ignored by a validating parser.

# Content Model Syntax 1

- Content models are made of *particles*, each of which can be any of the following:
  - an element name, a
  - a sequence: a, b, c, meaning the elements must follow each other with nothing except whitespace between them
  - a choice, a | b | c, meaning the input XML document must contain exactly one of an “a” element, a “b” element or a “c” element at that point; a, b and c
  - a particle inside ( parentheses )



# Content Model Syntax 2

- Particles can be followed by an *occurrence indicator*:
  - (none): the particle must occur exactly once in the input document;
  - \* (a star): zero or more times;
  - + (plus): one or more times
  - ? (question mark) zero or one times.

# Sequences

- Sequences use commas between particles:
- `<!ELEMENT name (first, middle, last)>`
- Note: This example is used in the coursebook, but in real applications you should use family-name and given-name, because not all cultures use the same order for their names!

`<name>`

`<first>Liam</first>`

`<middle>R E</middle>`

`<last>Quin</last>`

`</name>`

- A name element will only validate if it has *exactly one each of first, middle and last elements inside it.*

# Choices

- Choices use vertical bars between particles:
- `<!ELEMENT location (address | GPS)>`
- A location element will validate if it has *exactly one address element or exactly one GPS element: it must have one or the other.*
- You can have long choices as there is no fixed limit to the number of choices:
- `<!ELEMENT a (b | c | d | e | f)>`
- Some DTDs have over 100 elements in a choice, but this can make them very difficult for authors to use.

# Combining Particles

- Content models particles can be combined, using (parentheses) where necessary.
- `<!ELEMENT location`
- `(address | (latitude, longitude))>`
- Here, a location element must contain *either exactly one address element or one latitude element followed by one longitude element*.
- If you make the pattern too complicated the XML parser might be happy but people creating documents will hate you, so be careful! It's often best to stick to simple sequences or choices.

# More than One

- Suppose an author has two middle initials:

`<!ELEMENT name (first, middle, middle, last)>`

- This works for that author, but a more general approach is usually better in computing:

`<!ELEMENT name (given, middle*, family)>`

- The star \* means the name element can contain any number of middle elements, from none at all to a million or more, as long as they are all between the given name and family name.
- There is no direct way to say you can have between zero and ten middle names (but there really *are people with 20 or more middle names!*)

# More than One

- You can also use occurrence indicators on groups:

`<!ELEMENT name (given, (middle|initial)*, family)>`

- This will match any number of middle elements or initial elements intermixed in any order (including none at all).
- You can also use `+` and `?` after parentheses or after a name, to mean one or more, or zero or more.
- `a+` is the same as `(a, a*)`

# Mixed Content 1

- Mixed Content is the second of the four content model types (element, mixed, empty and any). Any element that can contain text is called *mixed*.

<!ELEMENT description (#PCDATA)>

- This allows just text (#PCDATA, *parsed character data*) and no sub-elements. The next example allows sub-elements too:

<!ELEMENT p (#PCDATA|em|strong|a|i|b|span)\*>

- The #PCDATA must be first in the content model, there must be a simple choice and there must be a \* at the end.
- All whitespace is significant inside the p element, because it matches #PCDATA.

# Mixed Content 2

`<!ELEMENT p (#PCDATA|em|strong|i|b|span)*>`

- Example:

`<p>This p element has <em>embedded</em> sub-elements interspersed with <i>really</i> awesome text</p>`

- Most document-like text will end up allowing sub-elements, e.g. for emphasis;
- Some languages (Japanese, Chinese and Korean especially) need sub-elements for “ruby” annotation elements, e.g. even in book titles.



# Mixed Content Summary

- Must use the choice separator (|, vertical bar) to separate elements, not the comma.
- #PCDATA must appear first in the list of elements
- There must be no inner content models: no (, ), \*, ?, + or comma inside the outer brackets
- If there are child elements, the \* must appear at the end of the model, outside the brackets.

```
<!ELEMENT a (#PCDATA)>
```

```
<!ELEMENT b (#PCDATA)*>
```

```
<!ELEMENT c (#PCDATA|d|e|f|g)*>
```

# Empty Content

- You can say that a particular element is not allowed to contain anything at all:

`<!ELEMENT img EMPTY>`

- Note: there are no parentheses - the keyword EMPTY replaces the usual content model.
- The keyword EMPTY (like #PCDATA and ELEMENT) must be in upper case.
- Empty elements can appear either as `<img/>` or as `<img></img>`
- Whitespace and text are not allowed inside an EMPTY element.

# Unrestricted Content

- An element declared with content model of ANY can contain any mix of elements and text:

`<!ELEMENT trash ANY>`

- Any elements that do occur inside an ANY element must themselves be declared in the DTD.
- It's usually a bad idea to use ANY: it can make it harder to write software to process the XML, and it can lead to mistakes in document creation.

# Attributes 1:

- Elements are ideal for containing human-readable content; sometimes you need a place for computer-readable information, or information about elements (metadata), and that's where attributes are best used.

```
<!ELEMENT a (#PCDATA)*>  
<!ATTLIST a  
    href CDATA #IMPLIED  
>
```

- This declaration allows, e.g.:

```
<a href="I am an attribute">content here</a>
```

# Attributes 2:

- Attribute names have the same rules as element names, except that an element can't have two attributes with the same name.
- There are several attribute types; CDATA attributes (character data) contain text, and are the most common type.
- **Attributes can never contain elements.**
- Attributes can occur in any order on elements.

# Attribute Types

- Attributes can have a **default value** that will be supplied by DTD validation; most XML processing is done without DTDs today, so avoid this.
- Attributes can have a **fixed value**, e.g. to identify a version or even a namespace URI.
- Otherwise, an attribute is either required (**#REQUIRED**) or optional (**#IMPLIED**).

```
<!ATTLIST img
    src CDATA #REQUIRED
    style CDATA #IMPLIED
>
```

# Attributes Revisited 1

- There are nine attribute types
  - Attribute values are *normalized* by turning each sequence of whitespace (including newlines) into a single space.
- 1. CDATA: the value is text.**
  - 2. ID: the value is an XML name, an identifier, which must be unique in the whole document's ID attributes.**
  - 3. IDREF: the value is an identifier that also occurs as an ID value; the IDREF is said to point to, or refer to, that ID.**

# Attributes Revisited 2

4. **IDREFS**: a whitespace-separated list of IDREF values.
5. **ENTITY**: the attribute's value must be the *name* of an external unparsed entity.
6. **ENTITIES**: a whitespace-separated list of entity names.
7. **NMTOKEN**: a *name token*, like an ID but not required to be unique.
8. **NMTOKENS**: a whitespace-separated list of NMTOKENs;
9. An enumeration (see next slide).



# Attributes Revisited 3

- An *enumeration* is a list of allowed values:

```
<!ELEMENT phone (#PCDATA)*>
```

```
<!ATTLIST phone
```

```
    kind (home|work|mobile|fax|unknown) "unknown"
```

```
>
```

- This allows instances like:

```
<phone kind="fax">+33 12 23 45 67 99</phone>
```

```
<phone>extension 7</phone> (kind is "unknown" here)
```

# DTDs: Why and Why Not

- Document Type Definitions are part of the XML standard and very widely supported.
- The content model notation is easy to understand.
- **But You can't control content (e.g. require digits in a phone number) with DTDs.**
- They are not namespace aware.
- No data typing (integer, hatsize...) so they are not useful for connecting XML to programming languages.
- As a result, DTDs are mostly replaced by XSD and RNG.

```
<!DOCTYPE Stars [  
    <!ELEMENT Stars (Star*)>  
    <!ELEMENT Star (Name, Address+, Movies)>  
    <!ELEMENT Name (#PCDATA)>  
    <!ELEMENT Address (Street, City)>  
    <!ELEMENT Street (#PCDATA)>  
    <!ELEMENT City (#PCDATA)>  
    <!ELEMENT Movies (Movie*)>  
    <!ELEMENT Movie (Title, Year)>  
    <!ELEMENT Title (#PCDATA)>  
    <!ELEMENT Year (#PCDATA)>  

```

Figure 11.6: A DTD for movie stars

**Exercise 11.3.1:** Add to the document of Fig. 11.10 the following facts:

- a) Carrie Fisher and Mark Hamill also starred in *The Empire Strikes Back* (1980) and *Return of the Jedi* (1983).
- b) Harrison Ford also starred in *Star Wars*, in the two movies mentioned in (a), and the movie *Firewall* (2006).
- c) Carrie Fisher also starred in *Hannah and Her Sisters* (1985).
- d) Matt Damon starred in *The Bourne Identity* (2002).

**Exercise 11.3.2:** Suggest how typical data about banks and customers, as was described in Exercise 4.1.1, could be represented as a DTD.

**Exercise 11.3.3:** Suggest how typical data about players, teams, and fans, as was described in Exercise 4.1.3, could be represented as a DTD.

Here is a DTD:

```
<!DOCTYPE code [  
  <!ELEMENT code (text1+, subtext)>  
  <!ELEMENT text1 (#PCDATA)>  
  <!ELEMENT subtext (text1?, text2)>  
  <!ELEMENT text2 (#PCDATA)>  

```

Does the code below meet the DTD above:

```
<code>  
  <text1> </text1> <text1> </text1>  
  <subtext> <text1> </text1> <text2> </test2>  
</subtext> </code>
```

Here is a DTD:

```
<!DOCTYPE code [  
  <!ELEMENT code (text1+, subtext)>  
  <!ELEMENT text1 (#PCDATA)>  
  <!ELEMENT subtext (text1?, text2)>  
  <!ELEMENT text2 (#PCDATA)>  

```

Does the code below meet the DTD above:

```
<code>  
  <text1> </text1> <text1> </text1>  
  <subtext> </subtext>  
</code>
```

Here is a DTD:

```
<!DOCTYPE code [  
    <!ELEMENT code (text1+, subtext)>  
    <!ELEMENT text1 (#PCDATA)>  
    <!ELEMENT subtext (text1?, text2)>  
    <!ELEMENT text2 (#PCDATA)>  

```

Does the code below meet the DTD above:

```
<code>  
    <text1> </text1> <text1> </text1>  
    <subtext> <text2> </test2> <text1> </text1> </subtext>  
</code>
```

Here is a DTD:

```
<!DOCTYPE code [  
    <!ELEMENT code (text1+, subtext)>  
    <!ELEMENT text1 (#PCDATA)>  
    <!ELEMENT subtext (text1?, text2)>  
    <!ELEMENT text2 (#PCDATA)>  
>
```

Does the code below meet the DTD above:

```
<code>  
    <subtext> <text1></text1> <text2> </test2></subtext>  
</code>
```