



Werkzeuge der Informatik

XML - Extensible Markup Language

Prof. Dr. Sven Hartmann

TU Clausthal

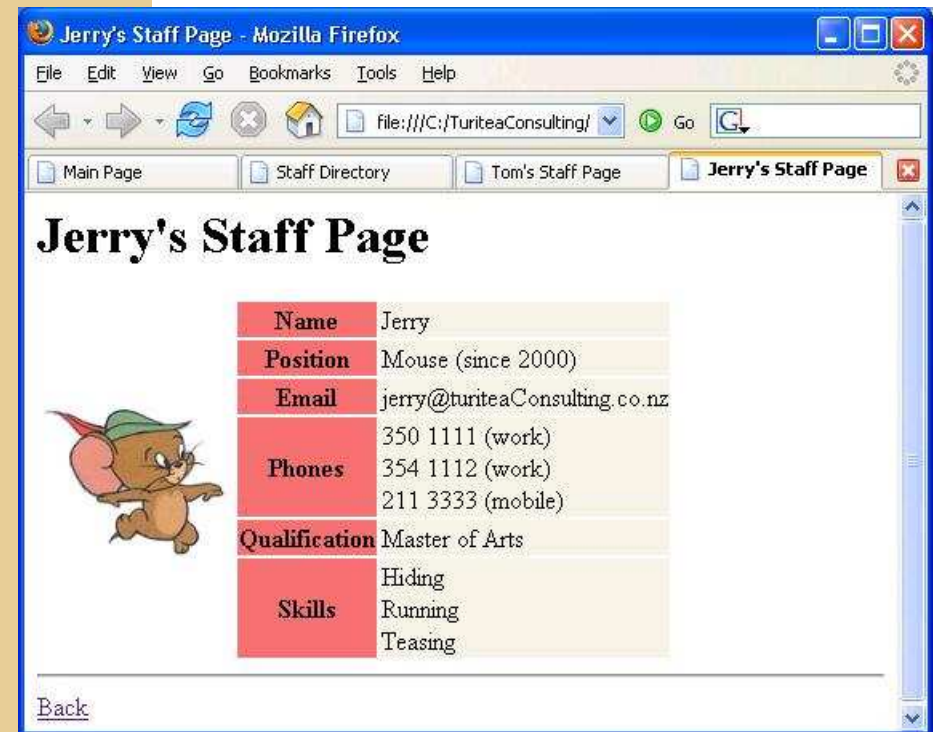
Institut für Informatik

Lehrstuhl für Datenbanken und Informationssysteme

XML Data

- A semi-structured data tuple (and a possible visualisation through a web browser):

```
<Employee>
  <Name>Jerry</Name>
  <Position>Mouse</Position>
  <Email>jerry@turiteaConsulting.co.nz</Email>
  <Phones>
    <Phone>350 1111</Phone>
    <Phone>354 1112</Phone>
    <Phone>211 3333</Phone>
  </Phones>
  <Qualification>Master of Arts</Qualification>
  <Skills>
    <Skill>Hiding</Skill>
    <Skill>Running</Skill>
    <Skill>Teasing</Skill>
  </Skills>
  <Photo>figures/jerry.jpg</Photo>
</Employee>
```



XML Elements

- XML stands for *Extensible Markup Language*, describing data with XML is sometimes called *XML-ification*
- We have chosen *markup tags* to specify the logical structure of the data
 - the staff details of an employee consist of a name, a position, etc.
 - hence we have chosen the corresponding tags to markup the respective data items
- The essential information is the text between the tags, while the tags represent *meta-information* that helps to understand the text
- Any piece of XML code is called an *XML fragment*
 - however, there are certain rules for forming XML code
- Markup tags usually come in pairs and markup *XML elements*, such as

```
⟨Skill⟩Hiding⟨/Skill⟩
```

- herein, ⟨Skill⟩ is the *start tag*, and ⟨/Skill⟩ the *end tag*
- the text in between is the *content* of the XML element

XML Elements

- The content of an XML element might be
 - pure text
 - a mixture of pure text and markup
 - further XML elements
 - nothing
- XML elements may be *nested* into one another, such as

```
<Phones>
  <Phone>350 1111</Phone>
  <Phone>354 1112</Phone>
  <Phone>211 3333</Phone>
</Phones>
```

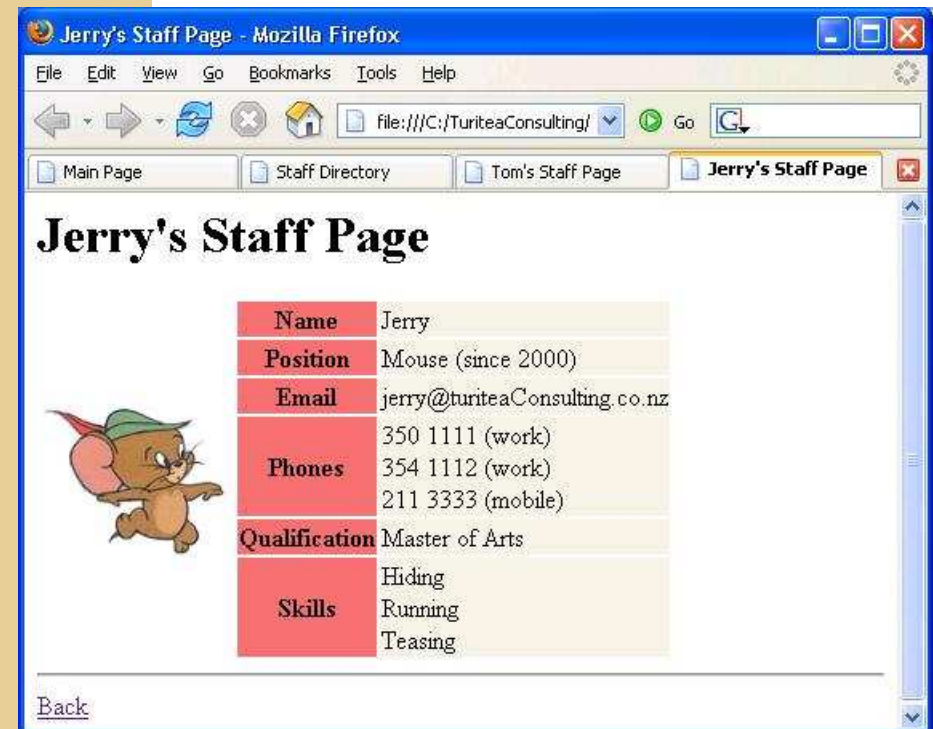
- An XML element without content is called an *empty XML element*
 - in this case, we use only a single markup tag, such as

```
<Retired/>
```

Attributes of XML Elements

- An XML element may have *attributes* to capture further properties
 - they are stored as *attribute-value pairs* in the start tag

```
<Employee>
  <Name>Jerry</Name>
  <Position Since= "2000" >Mouse</Position>
  <Email>jerry@turiteaConsulting.co.nz</Email>
  <Phones>
    <Phone Kind= "work" >350 1111</Phone>
    <Phone Kind= "work" >354 1112</Phone>
    <Phone Kind= "mobile" >211 3333</Phone>
  </Phones>
  <Qualification>Master of Arts</Qualification>
  <Skills>
    <Skill>Hiding</Skill>
    <Skill>Running</Skill>
    <Skill>Teasing</Skill>
  </Skills>
  <Photo>figures/jerry.jpg</Photo>
</Employee>
```



XML Documents

- XML code is stored in *XML documents*
- An XML document consists of three parts:
 - its XML declaration
 - its processing instructions
 - its root element
- An XML document must have a root element, while XML declaration and the processing instructions are optional
- Usually, the *XML declaration* looks as follows:

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

- it indicates the version of XML being used, here 1.0
- and it states in which encoding the document is written
- The *processing instructions* could be declarations of style sheets, etc.
- For the *root element*, just choose a name and form it like any other XML element:

```
<DB>...</DB>
```

XML Documents

- An XML document must be *well-formed*, that is,
 - there is exactly one root element
 - start and end tags must match
 - start and end tags must nest properly
- The following XML fragments are not well-formed:

```
⟨apple⟩⟨/pear⟩
```

```
⟨apple⟩⟨pear⟩⟨/apple⟩⟨/pear⟩
```

- XML is case-sensitive (this is different from HTML)
 - The following XML fragment is not well-formed:

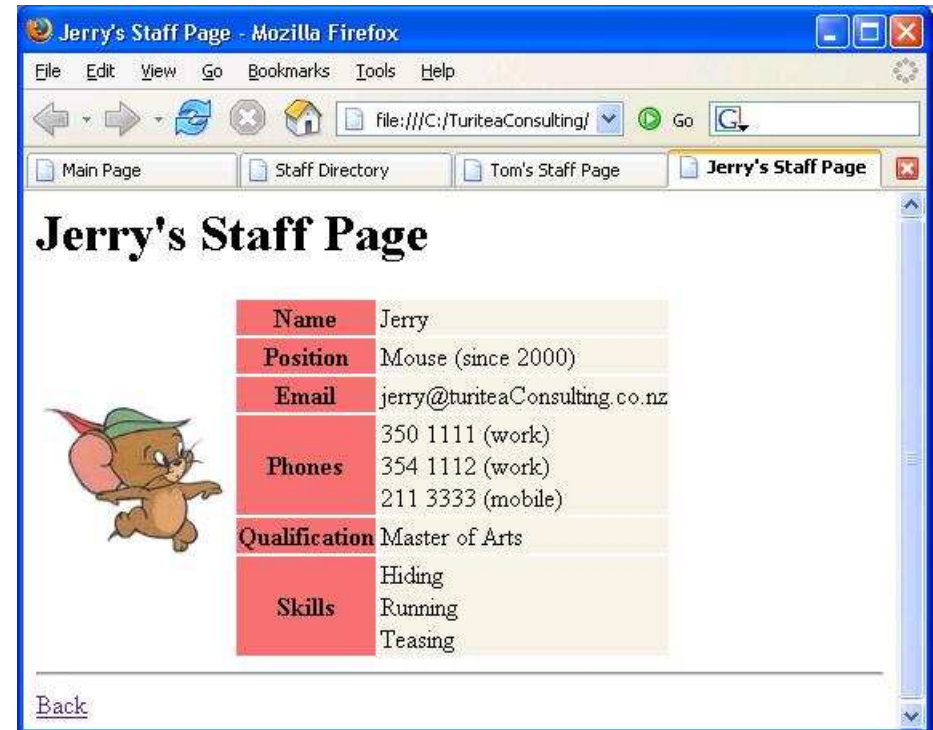
```
⟨Apple⟩⟨/apple⟩
```

- In future, whenever we talk about an XML document, we mean a well-formed one

XML Repositories

```
<?xml version="1.0" encoding="UTF-8"?>
<Employee>
  <Name>Jerry</Name>
  <Position Since="2000">Mouse</Position>
  <Email>jerry@turiteaConsulting.co.nz</Email>
  <Phones>
    <Phone Kind="work">350 1111</Phone>
    <Phone Kind="work">354 1112</Phone>
    <Phone Kind="mobile">211 3333</Phone>
  </Phones>
  <Qualification>Master of Arts</Qualification>
  <Skills>
    <Skill>Hiding</Skill>
    <Skill>Running</Skill>
    <Skill>Teasing</Skill>
  </Skills>
  <Photo>figures/jerry.jpg</Photo>
</Employee>
```

- Store the XML element Employee in an XML document (**jerry.xml**)



- Similarly, create an XML document for each staff member

XML Repositories

```
<?xml version="1.0" encoding="UTF-8" ?>
<Employee>
  <Name>Tom</Name>
  <Position Since="2000">Cat</Position>
  <Email>tom@turiteaConsulting.co.nz</Email>
  <Phones>
    <Phone Kind="work">350 2222</Phone>
    <Phone Kind="home">354 2222</Phone>
  </Phones>
  <Skills>
    <Skill>Constructing mousetraps</Skill>
    <Skill>Eating</Skill>
  </Skills>
  <Photo>figures/tom.gif</Photo>
</Employee>
```



- An *XML repository* is a collection of XML documents (that are somehow related)

Describing Data Types

- We observe:
 - there are lots of employees having different staff details, but in all cases the structure of their staff details looks similar
 - classification abstraction means to describe the common structure
 - we aim to describe the common *data type* (as far as possible)
 - then, this data type can serve as a schema for the XML data tuples, which will be *instances* of the data type
- After analysing the structure of the Employee elements, we declare:

```
⟨!ELEMENT Employee (Name, Position, Email, Phones, Qualification, Skills, Photo)⟩
```

- this may serve as a common data model for all staff
- We observe:
 - this is a complex data type, so we also need to declare data types for Names, Positions, etc.
 - Qualification is only optional, so we need to indicate this

XML Element Declarations

- An *element declaration* has the general form:

```
⟨!ELEMENT element-name content-model⟩
```

- The *element name* is the name inside the start and end tag
 - it must be a valid XML name, that is,
 - start with an alphabetical character or an underscore _
 - but not with the string “xml”
 - it may contain any alphanumerical character or _ or - or .
 - but no blanks, no reserved symbols such as < or > or & or ”
- The *content model* specifies what may occur between the start and end tag:
 - pure text
 - anything (any mixture of pure text and markup)
 - further XML elements
 - nothing

XML Element Declaration

- We use `<!ELEMENT element-name (#PCDATA)>` if the content is pure text
 - `#PCDATA` stands for parsed, or better, parsable character data
- We use `<!ELEMENT element-name ANY>` if the content may be anything
 - this is very convenient, but not very informative ...
- We use `<!ELEMENT element-name EMPTY>` if there is no content
 - but wait, till we can add attributes ...
- We use `<!ELEMENT element-name child-elements>` if the content are further XML elements
 - these elements are referred to as *child elements* or *children*
 - as an example, we recall our data type for the staff details:

```
<!ELEMENT Employee (Name, Position, Email, Phones, Qualification, Skills, Photo)>
```

Declaring Child Elements

- Recall, that we need to indicate that Qualification is an optional child
- We use regular expressions to describe the permitted combinations of child elements

```
⟨!ELEMENT element-name reg-expression⟩
```

- Regular expressions can be build as follows:
 - start with #PCDATA, EMPTY or any valid XML names
 - form sequences
 - form alterations
 - indicate optionality
 - indicate iteration
 - indicate non-empty iteration
 - add braces
- In practise, the regular expressions used for XML elements are often rather simple

Declaring Child Elements

- Here are some easy-to-follow rules of thumb:

- To describe a sequence of elements of types $child_1, \dots, child_n$, use

```
⟨!ELEMENT element-name ( child1, . . . , childn )⟩
```

- To describe the alternative of elements of types $child_1, \dots, child_n$, use

```
⟨!ELEMENT element-name ( child1 | . . . | childn )⟩
```

- To indicate an *option*, attach a **?** to one or more child elements
 - such an element may or may not appear
- To indicate an *iteration*, attach a ***** to one or more child elements
 - such an element may occur a finite number of times (or not at all)
- To indicate a *non-empty iteration*, attach a **+** to one or more child elements
 - such an element may occur a non-zero, finite number of times

Our Example

- We indicate that Qualification is only optional:

```
⟨!ELEMENT Employee (Name, Position, Email, Phones, Qualification?, Skills, Photo)⟩
```

- We declare data types for the child elements Names, Positions, etc.

```
⟨!ELEMENT Name (#PCDATA)⟩
```

```
⟨!ELEMENT Position (#PCDATA)⟩
```

```
⟨!ELEMENT Email (#PCDATA)⟩
```

```
⟨!ELEMENT Phones (Phone*)⟩
```

```
⟨!ELEMENT Qualification (#PCDATA)⟩
```

```
⟨!ELEMENT Skills (Skill*)⟩
```

```
⟨!ELEMENT Photo (#PCDATA)⟩
```

- We declare data types for the grand child elements Phone and Skill

```
⟨!ELEMENT Phone (#PCDATA)⟩
```

```
⟨!ELEMENT Skill (#PCDATA)⟩
```

Our Example

- We check the suitability of the data type:

⟨Employee⟩

⟨Name⟩Tom⟨/Name⟩

⟨Position Since= “2000” ⟩Cat⟨/Position⟩

⟨Email⟩tom@turiteaConsulting.co.nz⟨/Email⟩

⟨Phones⟩

⟨Phone Kind= “work” ⟩350 2222⟨/Phone⟩

⟨Phone Kind= “home” ⟩354 2222⟨/Phone⟩

⟨/Phones⟩

⟨Skills⟩

⟨Skill⟩Constructing mousetraps⟨/Skill⟩

⟨Skill⟩Eating⟨/Skill⟩

⟨/Skills⟩

⟨Photo⟩figures/tom.gif⟨/Photo⟩

⟨/Employee⟩

```
⟨!ELEMENT Employee (Name, Position, Email,  
    Phones, Qualification?, Skills, Photo)⟩
```

```
⟨!ELEMENT Name (#PCDATA)⟩
```

```
⟨!ELEMENT Position (#PCDATA)⟩
```

```
⟨!ELEMENT Email (#PCDATA)⟩
```

```
⟨!ELEMENT Phones (Phone*)⟩
```

```
⟨!ELEMENT Phone (#PCDATA)⟩
```

```
⟨!ELEMENT Qualification (#PCDATA)⟩
```

```
⟨!ELEMENT Skills (Skill*)⟩
```

```
⟨!ELEMENT Skill (#PCDATA)⟩
```

```
⟨!ELEMENT Photo (#PCDATA)⟩
```


Attribute Declaration

- XML elements can have attributes to capture particular properties of these elements, such as

```
⟨!ATTLIST Position Since CDATA #REQUIRED⟩
```

- An *attribute declaration* has the general form:

```
⟨!ATTLIST element-name attribute-specifications⟩
```

- the element name specifies the element whose attributes we want to declare
- the list of attribute specifications contains exactly one for each attribute, each *attribute specification* has the form

```
attribute-name attribute-type attribute-constraint
```

- the *attribute name* is the name chosen for this attribute
 - the attribute name must be a valid XML name (as explained above)
 - naturally, any two attributes of the same element should have distinct names

Attribute Declaration

- There are three kinds of attribute values: strings, enumerated, and tokens
- *Strings*: the attribute's value is a character string
 - we use the simple data type CDATA
 - blanks are allowed
 - any text is allowed except for reserved symbols
- *Enumerated*: the attribute's value must be chosen from a user-specified list

```
<!ELEMENT Car EMPTY>
<!ATTLIST Car Make CDATA #REQUIRED
             Colour CDATA #REQUIRED
             New ( yes | no ) #REQUIRED>
```

- *Tokens*: the attribute's value is a special-purpose character string
 - NMTOKEN can be used for a valid XML name
 - ENTITY can be used for a reference to an external file
 - ID, IDREF and IDREFS are explained later on

Attribute Declaration

- The *attribute constraint* is one of
 - #REQUIRED if the attribute must occur in every element
 - #IMPLIED if the attribute is optional
 - a default value for the attribute
 - #FIXED value
 - #CURRENT if the attribute takes the value most recently assigned to this attribute
- For our example
 - we can simply choose:

```
<!ATTLIST Position Since CDATA #REQUIRED>
```

```
<!ATTLIST Phone Kind CDATA #IMPLIED>
```
 - thus, Since is a compulsory attribute, and Kind is an optional attribute
 - alternatively we could also choose:

```
<!ATTLIST Phone Kind (work | home | mobile) #IMPLIED>
```

Our Example

- We check the suitability of the data type again:

```
<Employee>
  <Name>Tom</Name>
  <Position Since="2000">Cat</Position>
  <Email>tom@turiteaConsulting.co.nz</Email>
  <Phones>
    <Phone Kind="work">350 2222</Phone>
    <Phone Kind="home">354 2222</Phone>
  </Phones>
  <Skills>
    <Skill>Constructing mousetraps</Skill>
    <Skill>Eating</Skill>
  </Skills>
  <Photo>figures/tom.gif</Photo>
</Employee>
```

```
<!ELEMENT Employee (Name, Position, Email,
  Phones, Qualification?, Skills, Photo)>
<!ELEMENT Name (#PCDATA)>
<!ELEMENT Position (#PCDATA)>
<!ATTLIST Position Since CDATA #REQUIRED>
<!ELEMENT Email (#PCDATA)>
<!ELEMENT Phones (Phone*)>
<!ELEMENT Phone (#PCDATA)>
<!ATTLIST Phone Kind CDATA #IMPLIED>
<!ELEMENT Qualification (#PCDATA)>
<!ELEMENT Skills (Skill*)>
<!ELEMENT Skill (#PCDATA)>
<!ELEMENT Photo (#PCDATA)>
```

Document Type Definitions

- We store all the XML element declarations and their attribute declarations in a separate document (**staff.dtd**)

```
<!ELEMENT Employee (Name, Position, Email, Phones, Qualification?, Skills, Photo)>
<!ELEMENT Name (#PCDATA)>
<!ELEMENT Position (#PCDATA)>
<!ELEMENT Email (#PCDATA)>
<!ELEMENT Phones (Phone*)>
<!ELEMENT Phone (#PCDATA)>
<!ELEMENT Qualification (#PCDATA)>
<!ELEMENT Skills (Skill*)>
<!ELEMENT Skill (#PCDATA)>
<!ELEMENT Photo (#PCDATA)>
<!ATTLIST Position Since CDATA #REQUIRED>
<!ATTLIST Phone Kind CDATA #IMPLIED>
```

- We observe:
 - this document is called a *Document Type Definition* or *DTD*, for short
 - this is not XML code, hence a DTD is not an XML document
 - we used the DTD language as a separate language for describing data types

Document Type Definitions

- Finally, we need to link the DTD and the respective XML documents together
 - an DTD contains a data type
 - an XML document contains an instance of the data type
 - usually, there are many XML documents that correspond to a single DTD
- Add an document type declaration after the XML declaration in an XML document

```
⟨!DOCTYPE Employee SYSTEM "staff.dtd" ⟩
```

- In general, the *document type declaration* has the form

```
⟨!DOCTYPE root-name SYSTEM uri ⟩
```

- the root name is the name of the root element in the XML documents
- the URI is the uniform resource identifier of the DTD (usually the file name)
- Alternatively, one can include the entire DTD into the XML document

```
⟨!DOCTYPE root-name [... here goes the DTD ...] ⟩
```

- but this is not recommended for an XML repository where several XML documents share a DTD

Validation of XML Documents

- An XML document is said to be
 - be *well-formed* if has a unique, well-formed root element
 - *conforms* to a DTD if the DTD adequately describes its root element
 - be *valid* if it is linked to DTD and conforms to this DTD
- An XML document is a text file, so any text editor can be used for editing it . . .
- However, to validate it, we can use an *XML parser*:
 - ensure that all required XML elements are present
 - prevent undefined XML elements from being used
 - specify the use of attributes of XML elements and define their permitted values
- To create XML documents and data models for them (such as DTDs) we run through a data modelling process:
 - layout analysis and data access, knowledge integration, and content extraction,
 - structure analysis (recognition, visualisation, representation) of all elements,
 - testing an XML document whether it is well-formed and valid

XML Data Modelling

- Some features of XML are especially attractive for data modelling:
 - an XML document (considered as a complex data tuple) does not necessarily have a data model (such as a DTD)
 - in case it has one, we can prescribe/control the structure to exactly the extent we want to
 - but still, its structure may depart from that specified in that data model
 - the element names used for XML elements make XML documents self-explanatory
- In addition to the DTD language there exist several popular languages for describing XML data types
 - examples are XML Schema, Relax NG, DSD2, tree grammars
 - overcome some known limitations of the DTD language
 - provide more data modelling features than the DTD language
 - comparing their expressiveness is an important topic in research
 - graphical languages like the XML tree model are popular, too

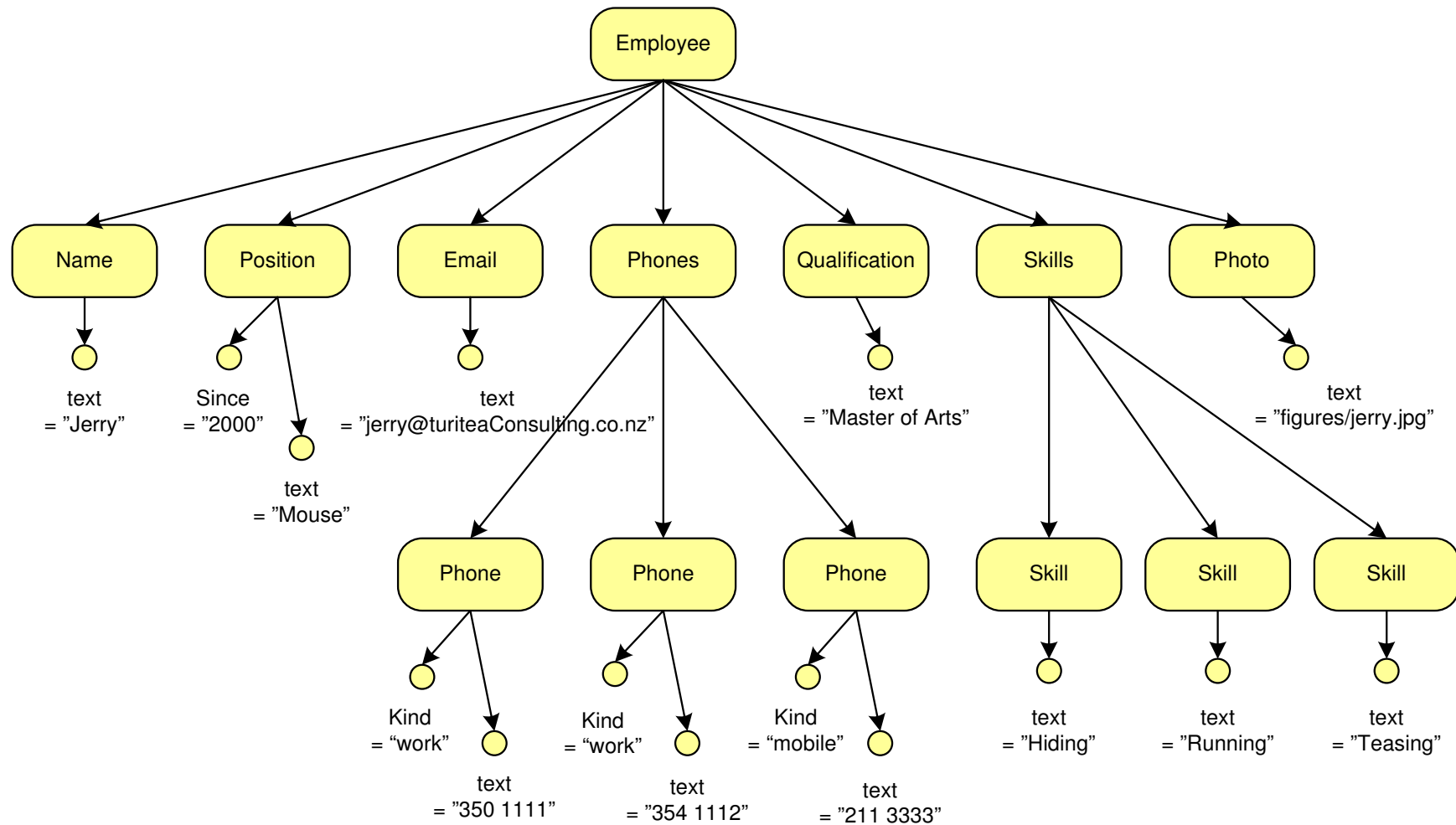
Who owns XML?

- Well, the *World Wide Web Consortium (W3C)* ... (though not really)
- W3C develops Web standards and guidelines (*W3C Recommendations*)
 - publishes open (non-proprietary) standards for Web languages
 - more than 90 standards since 1994
 - its mission is to lead the Web to its full potential by developing protocols and guidelines that ensure long-term growth for the Web
 - provides an open forum for discussion about the Web
- The goal is *Web interoperability*:
 - the most fundamental Web technologies must be compatible with one another and allow any hardware and software used to access the Web to work together
 - avoid market fragmentation and thus Web fragmentation
- W3C operations are
 - supported by more than 400 members worldwide (vendors, universities, etc.)
 - financed by member fees, research grants, public and private funding
 - run by about 70 full-time staff
 - administered by the MIT CS&AI Lab (CSAIL), the European Research Consortium for Informatics and Mathematics (ERCIM), and Keio University

W3C Activities

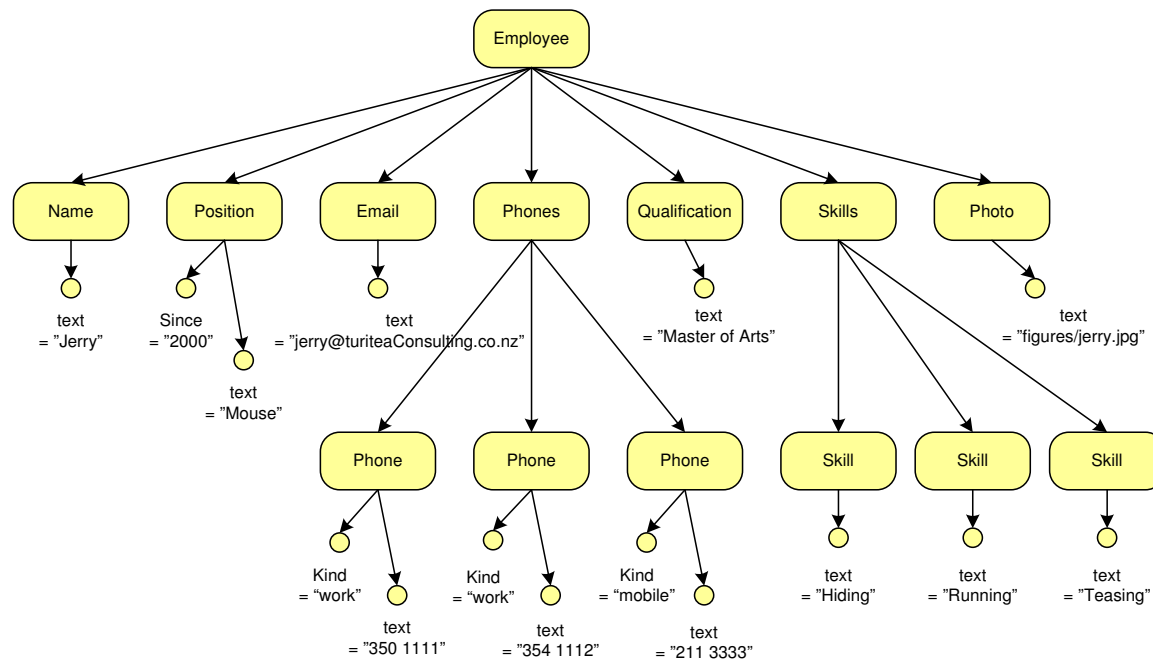
- Researchers can participate in the W3C activities
- W3C currently hosts 23 registered activities on:
 - web architecture: DOM, XML, Internationalisation, URI, Web Services
 - interaction: Graphics, HTML, Math, Rich Web Client, Style, XForms, Sync Multimedia
 - quality assurance: Quality Assurance, Incubator
 - technology and society: Patent Policy, Privacy, Semantic Web
 - ubiquitous web: Device Independence, Mobile Web, Multi-modal Interaction, Voice Browser
 - web accessibility: International Program Office, Technical Issues
- Activities are organised into groups:
 - Working Groups (WG) for technical developments
 - Interest Groups (IG) for strategy discussions
 - Coordination Groups (CG) for communication among related groups
- For the XML activity there are currently 9 groups:
 - XML Core WG, XML Processing WG, XML Query WG, XML Schema WG, XSL WG, Efficient XML Interchange WG, XML Plenary IG, XML Schema IG, XML CG

XML Trees



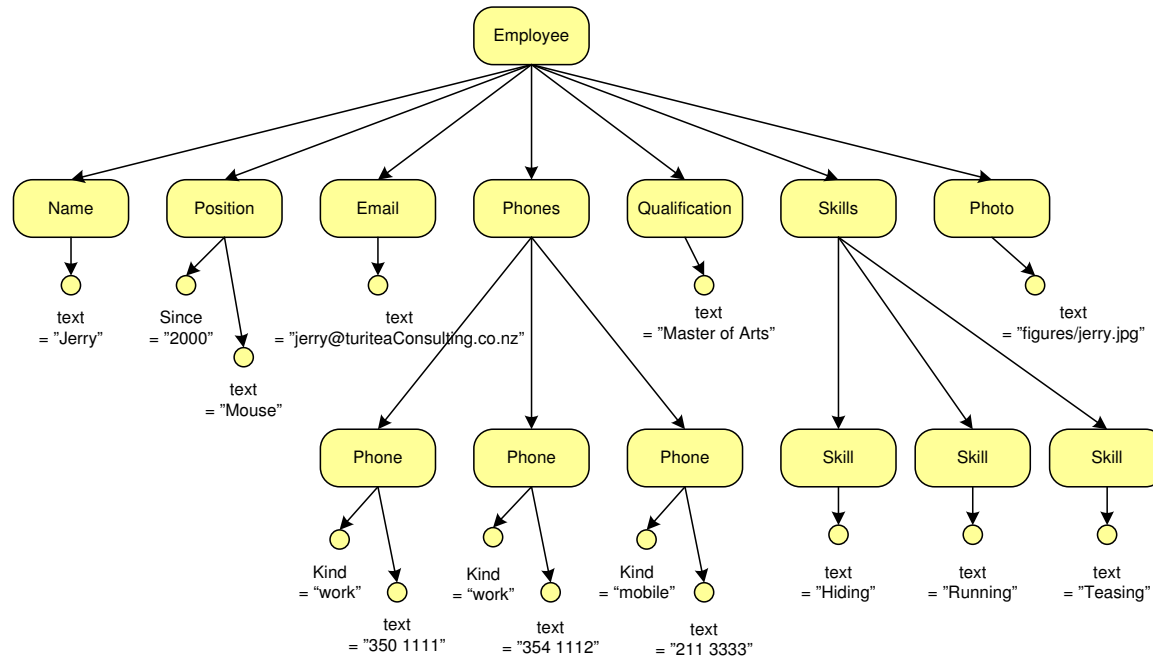
- XML elements may be visualised as XML trees
- This helps to imagine the hierarchical structure of XML elements

Nodes of XML Trees



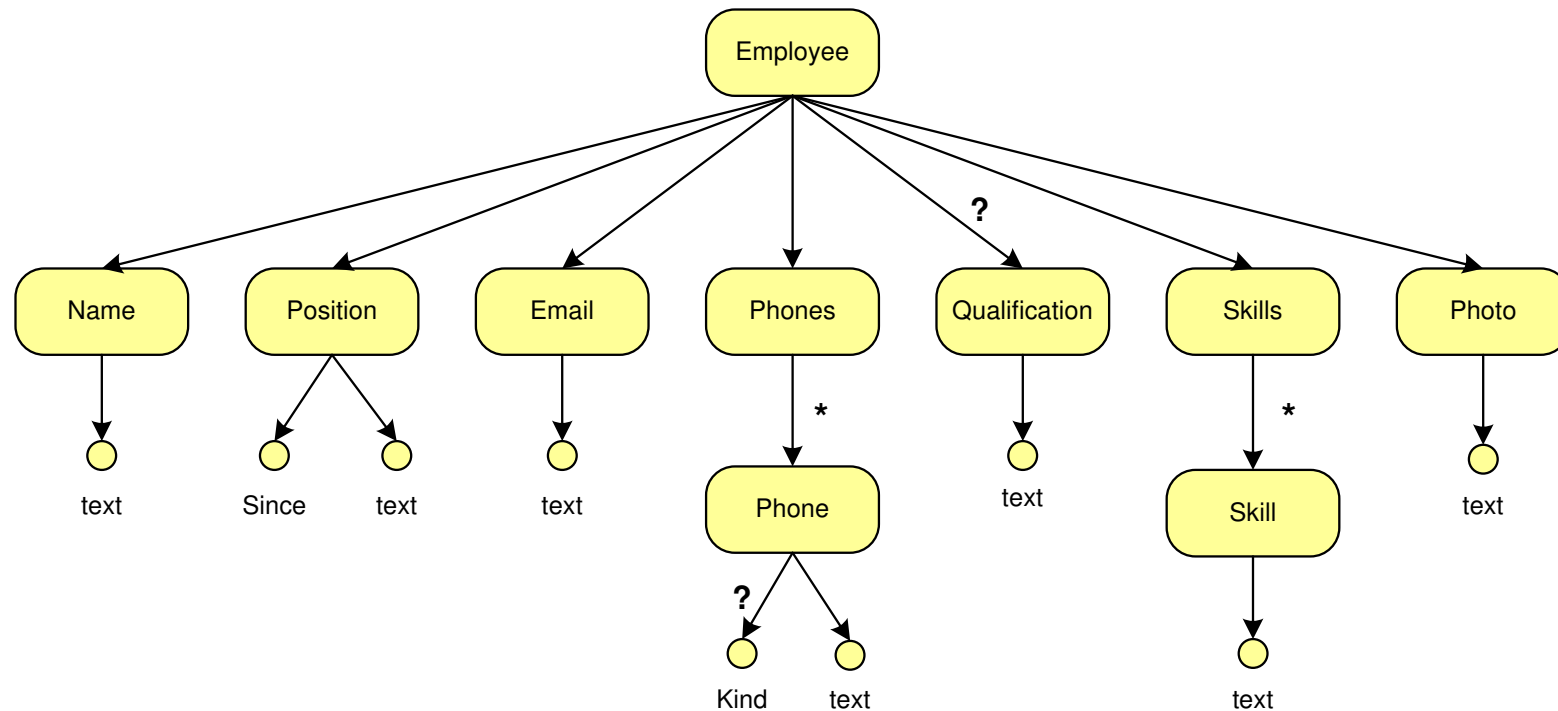
- *Element nodes* are visualised as boxes
 - they represent XML elements
- *Attribute nodes* are visualised as circles
 - they represent attributes of XML elements
- *Text nodes* are visualised as circles, too
 - they represent pure text content of XML elements

Edges of XML Trees



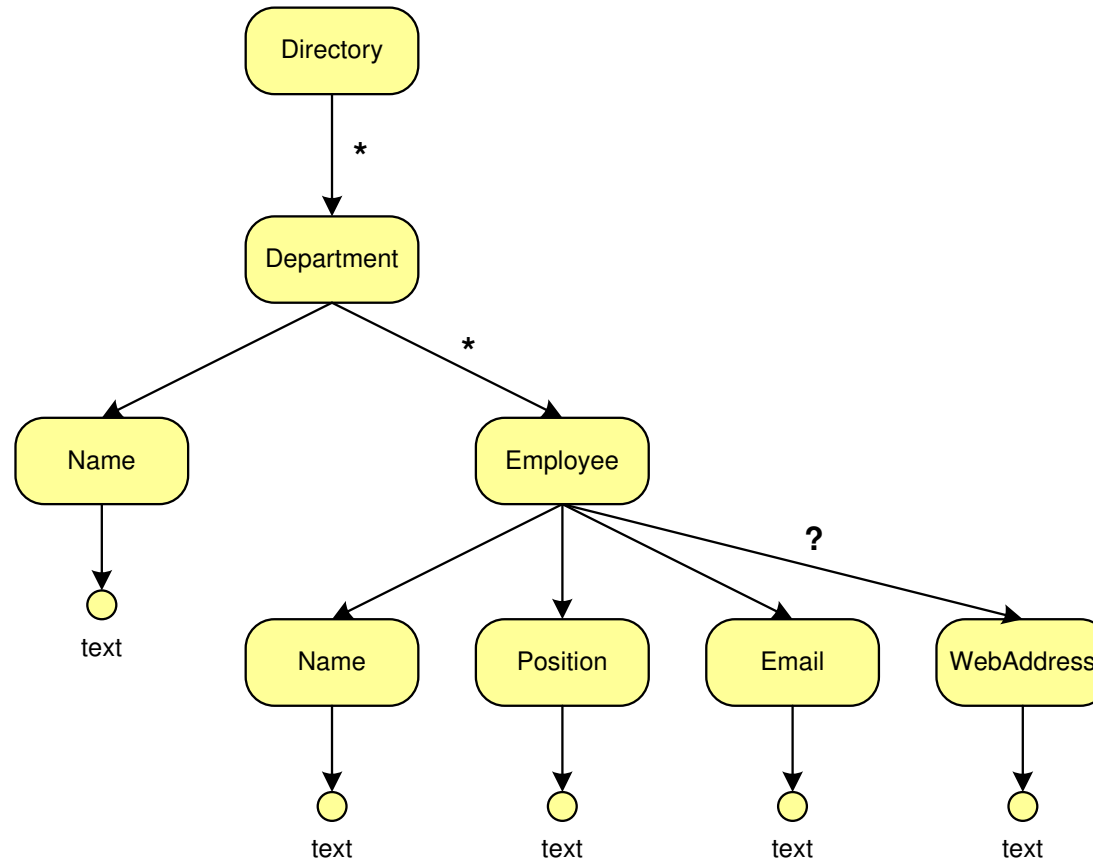
- *Edges* connect the node for an XML element to the nodes for its attributes, its child elements and its pure text content
- The top-most element node is the node of the root element or *root node*, for short
- Nodes without outgoing edges (attribute nodes, text nodes, empty element nodes) are *leaves*
 - well, yes, XML trees stand upside-down

XML Trees and Data Types



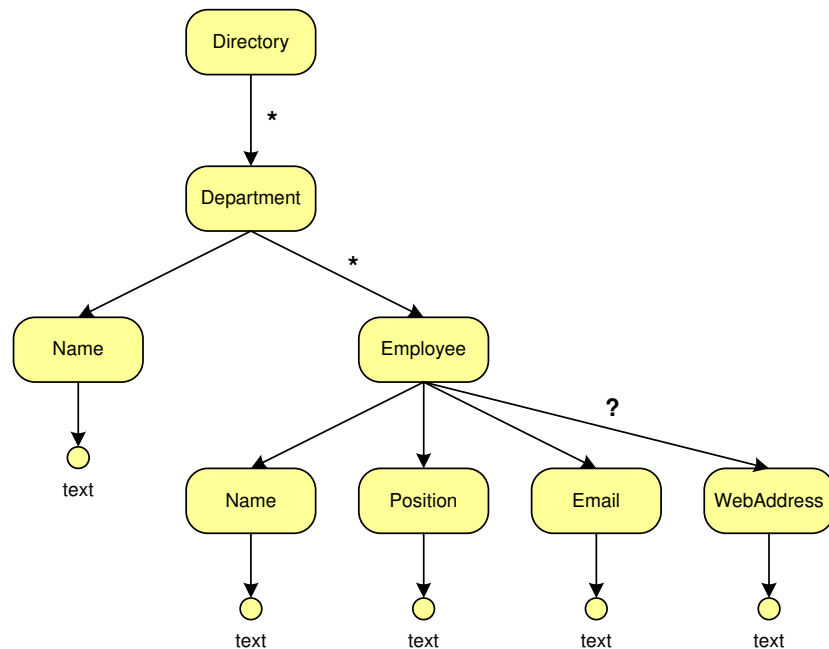
- XML trees can also be used to visualise data types
 - edges can be marked with ?, * or + to visualise optionality, iteration or non-empty iteration
- It is often convenient to draw an XML tree first before writing down a DTD
 - XML trees provide a rather intuitive way towards data modelling for XML

An Example



- We create a data type for a staff directory
 - we chose element types Directory, Department, Employee and a few others
 - this time we assemble less staff details in the Employee type
 - however, we include a new (optional) child WebAddress

Translating XML Trees into DTDs



- Now we translate the XML tree for the Directory type to the DTD language:

```
<!ELEMENT Directory (Department*)>
<!ELEMENT Department (Name, Employee*)>
<!ELEMENT Employee (Name, Position, Email, WebAddress?)>
<!ELEMENT Name (#PCDATA)>
<!ELEMENT Position (#PCDATA)>
<!ELEMENT Email (#PCDATA)>
<!ELEMENT WebAddress (#PCDATA)>
```