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
  
  ```xml
  <Skill>Hiding</Skill>
  ```

- within, <Skill> is the *start tag*, and </Skill> the *end tag*
- the text in between is the *content* of the XML element

XML Data

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

```xml
<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>
```
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>
  <PositionSince="2000">Mouser</PositionSince>
  <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

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>` `/apple`

XML is case-sensitive (this is different from HTML)

The following XML fragment is not well-formed:

```
<Apple/>
```

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

XML Repositories

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

```
<?xml version="1.0" encoding="UTF-8"?>
<Employee>
  <Name>Jerry</Name>
  <PositionSince="2000">Mouser</PositionSince>
  <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>
```

Similarly, create an XML document for each staff member
XML Repositories

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

- We use `(#PCDATA)` if the content is pure text
- `#PCDATA` stands for parsed, or better, parsable character data

- We use `{ANY}` if the content may be anything
- this is very convenient, but not very informative ...

- We use `{EMPTY}` if there is no content
- but wait, till we can add attributes ...

- We use `{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

\[
\langle \text{ELEMENT } \text{element-name reg-expression} \rangle
\]

- 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 practice, the regular expressions used for XML elements are often rather simple

Our Example

- We indicate that Qualification is only optional:
  \[
  \langle \text{ELEMENT Employee (Name, Position, Email, Phones, Qualification}, \text{Skills, Photo}) \rangle
  \]

- We declare data types for the child elements Names, Positions, etc.
  \[
  \langle \text{ELEMENT Name (} \#\text{PCDATA}) \rangle
  \langle \text{ELEMENT Position (} \#\text{PCDATA}) \rangle
  \langle \text{ELEMENT Email (} \#\text{PCDATA}) \rangle
  \langle \text{ELEMENT Phones (Phone*)} \rangle
  \langle \text{ELEMENT Qualification (} \#\text{PCDATA}) \rangle
  \langle \text{ELEMENT Skills (Skill*)} \rangle
  \langle \text{ELEMENT Photo (} \#\text{PCDATA}) \rangle
  \]

- We declare data types for the grand child elements Phone and Skill
  \[
  \langle \text{ELEMENT Phone (} \#\text{PCDATA}) \rangle
  \langle \text{ELEMENT Skill (} \#\text{PCDATA}) \rangle
  \]

Declaring Child Elements

- Here are some easy-to-follow rules of thumb:
  - To describe a sequence of elements of types child\(_1\), \ldots, child\(_n\), use
    \[
    \langle \text{ELEMENT element-name (} \text{child}_1, \ldots, \text{child}_n) \rangle
    \]
  - To describe the alternative of elements of types child\(_1\), \ldots, child\(_n\), use
    \[
    \langle \text{ELEMENT element-name (} \text{child}_1 | \ldots | \text{child}_n) \rangle
    \]
  - 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 check the suitability of the data type:
  \[
  \langle \text{Employee} \rangle
  \langle \text{Name} \rangle \text{Tom} \langle /\text{Name} \rangle \\
  \langle \text{Position} \rangle \text{Cat} \langle /\text{Position} \rangle \\
  \langle \text{Email} \rangle \text{tom@turiteaConsulting.co.nz} \langle /\text{Email} \rangle \\
  \langle \text{Phones} \rangle \\
  \langle \text{PhoneKind} \rangle \text{work} \langle /\text{PhoneKind} \rangle \text{3502222} \langle /\text{Phone} \rangle \\
  \langle \text{PhoneKind} \rangle \text{home} \langle /\text{PhoneKind} \rangle \text{3542222} \langle /\text{Phone} \rangle \\
  \langle /\text{Phones} \rangle \\
  \langle \text{Skills} \rangle \\
  \langle \text{Skill} \rangle \text{Constructing mousetraps} \langle /\text{Skill} \rangle \\
  \langle \text{Skill} \rangle \text{Eating} \langle /\text{Skill} \rangle \\
  \langle /\text{Skills} \rangle \\
  \langle \text{Photo} \rangle \text{figures/tom.gif} \langle /\text{Photo} \rangle \\
  \langle /\text{Employee} \rangle
  \]

- We indicate that Qualification is only optional:
  \[
  \langle \text{ELEMENT Employee (Name, Position, Email, Phones, Qualification}, \text{Skills, Photo}) \rangle
  \]

- We declare data types for the child elements Names, Positions, etc.
  \[
  \langle \text{ELEMENT Name (} \#\text{PCDATA}) \rangle
  \langle \text{ELEMENT Position (} \#\text{PCDATA}) \rangle
  \langle \text{ELEMENT Email (} \#\text{PCDATA}) \rangle
  \langle \text{ELEMENT Phones (Phone*)} \rangle
  \langle \text{ELEMENT Qualification (} \#\text{PCDATA}) \rangle
  \langle \text{ELEMENT Skills (Skill*)} \rangle
  \langle \text{ELEMENT Photo (} \#\text{PCDATA}) \rangle
  \]

- We declare data types for the grand child elements Phone and Skill
  \[
  \langle \text{ELEMENT Phone (} \#\text{PCDATA}) \rangle
  \langle \text{ELEMENT Skill (} \#\text{PCDATA}) \rangle
  \]
**Attribute Declaration**

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

\[
\langle \text{ATTLIST } \text{element-name attribute-specifications} \rangle
\]

- An attribute declaration has the general form:

\[
\langle \text{ATTLIST } \text{element-name attribute-specifications} \rangle
\]

- 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

\[
\text{attribute-name attribute-type attribute-constraint}
\]

- 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

---

**Our Example**

- We check the suitability of the data type again:

\[
\langle \text{Employee} \rangle
\]

\[
\text{Name}/Tom/\text{Name}
\]

\[
\text{Position Since}: \text{"2000"}/\text{Cat}/\text{Position}
\]

\[
\text{Email}/\text{tom@turiteaConsulting.co.nz}/\text{Email}
\]

\[
\text{Phones}
\]

\[
\text{Phone Kind}/\text{work}/3502222/\text{Phone}
\]

\[
\text{Phone Kind}/\text{home}/3542222/\text{Phone}
\]

\[
\langle \text{Phones} \rangle
\]

\[
\text{Skills}
\]

\[
\text{Skill}/\text{Constructing mousetraps}/\text{Skill}
\]

\[
\text{Skill}/\text{Eating}/\text{Skill}
\]

\[
\langle \text{Skills} \rangle
\]

\[
\text{Photo}/\text{figures/tom.gif}/\text{Photo}
\]

\[
\langle \text{Employee} \rangle
\]
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 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

Validation of XML Documents

- An XML document is said to be
  - be **well-formed** if it 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 form 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 are 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 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

Nodes of XML Trees
**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.

**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.

**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.

**Translating XML Trees into DTDs**

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

```
<![CDATA[
  <!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)>
]]>
```