

XML Technology Overview

Jon Warbrick
University of Cambridge Computing Service

Administrivia

Administrivia

- Fire escapes

Administrivia

- Fire escapes
- Who am I?

Administrivia

- Fire escapes
- Who am I?
- Pink sheets

Administrivia

- Fire escapes
- Who am I?
- Pink sheets
- Green sheets

Administrivia

- Fire escapes
- Who am I?
- Pink sheets
- Green sheets
- Timing.

This course

This course

- What we will (and won't) be covering

This course

- What we will (and won't) be covering
- The handouts

This course

- What we will (and won't) be covering
- The handouts
- Course website:

<http://www-uxsup.csx.cam.ac.uk/~jw35/courses/xml/>.

XML itself

In the beginning...

In the beginning...

- SGML

In the beginning...

- SGML
 - ◆ Invented in the 1970's at IBM

In the beginning...

- SGML
 - ◆ Invented in the 1970's at IBM
 - ◆ Now ISO standard 8879

In the beginning...

- SGML
 - ◆ Invented in the 1970's at IBM
 - ◆ Now ISO standard 8879
 - ◆ A "semantic and structural markup language for text documents"

In the beginning...

- SGML
 - ◆ Invented in the 1970's at IBM
 - ◆ Now ISO standard 8879
 - ◆ A "semantic and structural markup language for text documents"
- HTML is the most famous 'application' of SGML

In the beginning...

- SGML
 - ◆ Invented in the 1970's at IBM
 - ◆ Now ISO standard 8879
 - ◆ A "semantic and structural markup language for text documents"
- HTML is the most famous 'application' of SGML
- XML is a reformulation of SGML

In the beginning...

- SGML
 - ◆ Invented in the 1970's at IBM
 - ◆ Now ISO standard 8879
 - ◆ A "semantic and structural markup language for text documents"
- HTML is the most famous 'application' of SGML
- XML is a reformulation of SGML
 - ◆ Missing out the complicated and redundant features

In the beginning...

- SGML
 - ◆ Invented in the 1970's at IBM
 - ◆ Now ISO standard 8879
 - ◆ A "semantic and structural markup language for text documents"
- HTML is the most famous 'application' of SGML
- XML is a reformulation of SGML
 - ◆ Missing out the complicated and redundant features
 - ◆ A W3C-endorsed standard

In the beginning...

- SGML
 - ◆ Invented in the 1970's at IBM
 - ◆ Now ISO standard 8879
 - ◆ A "semantic and structural markup language for text documents"
- HTML is the most famous 'application' of SGML
- XML is a reformulation of SGML
 - ◆ Missing out the complicated and redundant features
 - ◆ A W3C-endorsed standard
 - ◆ Designed for easy parsing

In the beginning...

- SGML
 - ◆ Invented in the 1970's at IBM
 - ◆ Now ISO standard 8879
 - ◆ A "semantic and structural markup language for text documents"
- HTML is the most famous 'application' of SGML
- XML is a reformulation of SGML
 - ◆ Missing out the complicated and redundant features
 - ◆ A W3C-endorsed standard
 - ◆ Designed for easy parsing
 - ◆ A "meta-markup language for text documents"

In the beginning...

- SGML
 - ◆ Invented in the 1970's at IBM
 - ◆ Now ISO standard 8879
 - ◆ A "semantic and structural markup language for text documents"
- HTML is the most famous 'application' of SGML
- XML is a reformulation of SGML
 - ◆ Missing out the complicated and redundant features
 - ◆ A W3C-endorsed standard
 - ◆ Designed for easy parsing
 - ◆ A "meta-markup language for text documents"
- *XML* is simple

In the beginning...

- SGML
 - ◆ Invented in the 1970's at IBM
 - ◆ Now ISO standard 8879
 - ◆ A "semantic and structural markup language for text documents"
- HTML is the most famous 'application' of SGML
- XML is a reformulation of SGML
 - ◆ Missing out the complicated and redundant features
 - ◆ A W3C-endorsed standard
 - ◆ Designed for easy parsing
 - ◆ A "meta-markup language for text documents"
- *XML* is simple
 - ◆ it's the rest of the technology that's powerful

In the beginning...

- SGML
 - ◆ Invented in the 1970's at IBM
 - ◆ Now ISO standard 8879
 - ◆ A "semantic and structural markup language for text documents"
- HTML is the most famous 'application' of SGML
- XML is a reformulation of SGML
 - ◆ Missing out the complicated and redundant features
 - ◆ A W3C-endorsed standard
 - ◆ Designed for easy parsing
 - ◆ A "meta-markup language for text documents"
- *XML* is simple
 - ◆ it's the rest of the technology that's powerful
 - ◆ and in places complicated

In the beginning...

- SGML
 - ◆ Invented in the 1970's at IBM
 - ◆ Now ISO standard 8879
 - ◆ A "semantic and structural markup language for text documents"
- HTML is the most famous 'application' of SGML
- XML is a reformulation of SGML
 - ◆ Missing out the complicated and redundant features
 - ◆ A W3C-endorsed standard
 - ◆ Designed for easy parsing
 - ◆ A "meta-markup language for text documents"
- *XML* is simple
 - ◆ it's the rest of the technology that's powerful
 - ◆ and in places complicated
- XML isn't just a web technology.

XML Documents

XML Documents

- XML documents contain text, never binary data

XML Documents

- XML documents contain text, never binary data
- These can be manipulated by any tool that understand text

XML Documents

- XML documents contain text, never binary data
- These can be manipulated by any tool that understand text
- An XML document could be a disk file

XML Documents

- XML documents contain text, never binary data
- These can be manipulated by any tool that understand text
- An XML document could be a disk file
 - ◆ but it could as easily be a field in a database

XML Documents

- XML documents contain text, never binary data
- These can be manipulated by any tool that understand text
- An XML document could be a disk file
 - ◆ but it could as easily be a field in a database
 - ◆ or delivered over a network connection

XML Documents

- XML documents contain text, never binary data
- These can be manipulated by any tool that understand text
- An XML document could be a disk file
 - ◆ but it could as easily be a field in a database
 - ◆ or delivered over a network connection
- When delivered by a web server, they will probably have a media type of `text/xml` or `application/xml`

XML Documents

- XML documents contain text, never binary data
- These can be manipulated by any tool that understand text
- An XML document could be a disk file
 - ◆ but it could as easily be a field in a database
 - ◆ or delivered over a network connection
- When delivered by a web server, they will probably have a media type of `text/xml` or `application/xml`
- However the approved modern usage is to use something more like `application/svg+xml`.

Elements

Elements

- XML documents mainly consist of *elements*

Elements

- XML documents mainly consist of *elements*
- Have a *start-tag* and an *end-tag*

```
<name>  
  Computing Service  
</name>
```

Elements

- XML documents mainly consist of *elements*
- Have a *start-tag* and an *end-tag*

```
<name>  
  Computing Service  
</name>
```

- Everything between the tags is the element's *content*

Elements

- XML documents mainly consist of *elements*
- Have a *start-tag* and an *end-tag*

```
<name>  
  Computing Service  
</name>
```

- Everything between the tags is the element's *content*
- Whitespace is part of the content, though applications may ignore it

Elements

- XML documents mainly consist of *elements*
- Have a *start-tag* and an *end-tag*

```
<name>  
  Computing Service  
</name>
```

- Everything between the tags is the element's *content*
- Whitespace is part of the content, though applications may ignore it
- Empty elements can be written: `<name />`

Elements

- XML documents mainly consist of *elements*
- Have a *start-tag* and an *end-tag*

```
<name>  
  Computing Service  
</name>
```

- Everything between the tags is the element's *content*
- Whitespace is part of the content, though applications may ignore it
- Empty elements can be written: `<name />`
- ...but not `<name>`.

Tag names

Tag names

- Have no intrinsic meaning

Tag names

- Have no intrinsic meaning
- Are case sensitive

Tag names

- Have no intrinsic meaning
- Are case sensitive
- Can contain any alphanumeric character, underscore(_), hyphen(-), and dot (.)

Tag names

- Have no intrinsic meaning
- Are case sensitive
- Can contain any alphanumeric character, underscore(_), hyphen(-), and dot (.)
- Colon (:) should be avoided

Tag names

- Have no intrinsic meaning
- Are case sensitive
- Can contain any alphanumeric character, underscore(_), hyphen(-), and dot (.)
- Colon (:) should be avoided
 - ◆ it has a special meaning which we'll come to shortly

Tag names

- Have no intrinsic meaning
- Are case sensitive
- Can contain any alphanumeric character, underscore(_), hyphen(-), and dot (.)
- Colon (:) should be avoided
 - ◆ it has a special meaning which we'll come to shortly
- Must start with a letter or underscore

Tag names

- Have no intrinsic meaning
- Are case sensitive
- Can contain any alphanumeric character, underscore(_), hyphen(-), and dot (.)
- Colon (:) should be avoided
 - ◆ it has a special meaning which we'll come to shortly
- Must start with a letter or underscore
- Names starting 'xml...' (in any case) are reserved.

Elements within elements

Elements within elements

- Consider

```
<institution>  
  <name>Computing Service</name>  
  <address>New Museums Site, Pembroke Street</address>  
  <website>  
    <url>http://www.cam.ac.uk/cs/</url>  
    <url>http://www-uxsup.csx.cam.ac.uk/</url>  
  </website>  
</institution>
```

Elements within elements

- Consider

```
<institution>  
  <name>Computing Service</name>  
  <address>New Museums Site, Pembroke Street</address>  
  <website>  
    <url>http://www.cam.ac.uk/cs/</url>  
    <url>http://www-uxsup.csx.cam.ac.uk/</url>  
  </website>  
</institution>
```

- The `<institution>` element contains 3 'children': a `<name>` element, an `<address>` element and a `<website>` element

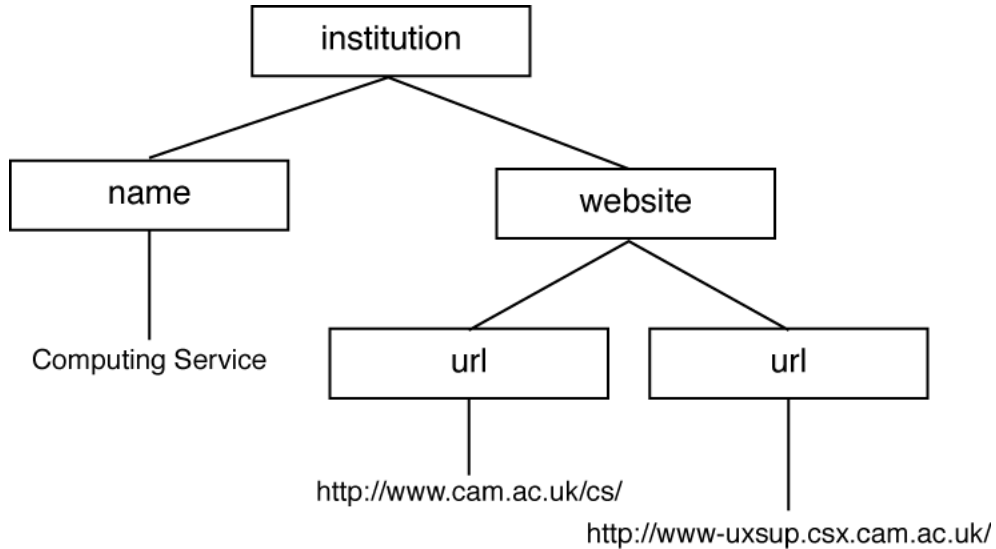
Elements within elements

- Consider

```
<institution>
  <name>Computing Service</name>
  <address>New Museums Site, Pembroke Street</address>
  <website>
    <url>http://www.cam.ac.uk/cs/</url>
    <url>http://www-uxsup.csx.cam.ac.uk/</url>
  </website>
</institution>
```

- The `<institution>` element contains 3 'children': a `<name>` element, an `<address>` element and a `<website>` element
- The `<website>` element itself contains 2 `<url>` elements.

XML documents as a tree



XML document styles

XML document styles

- Record orientated

```
<institution>  
  <name>Computing Service</name>  
  <address>New Museums Site, Pembroke Street</address>  
  <website>  
    <url>http://www.cam.ac.uk/cs/</url>  
    <url>http://www-uxsup.csx.cam.ac.uk/</url>  
  </website>  
</institution>
```

XML document styles

- Record orientated

```
<institution>
  <name>Computing Service</name>
  <address>New Museums Site, Pembroke Street</address>
  <website>
    <url>http://www.cam.ac.uk/cs/</url>
    <url>http://www-uxsup.csx.cam.ac.uk/</url>
  </website>
</institution>
```

- Mixed content

```
<handbook>
  <para>
    The <inst>Computing Service</inst> provides
    services, including <service>Hermes</service>
    and <service>Raven</service>. It is <em>really
    important</em> that you find out how to access
    these services.
  </para>
</handbook>
```

Attributes

Attributes

- Elements can have *attributes*

Attributes

- Elements can have *attributes*
- Name/value pairs in the start tag

Attributes

- Elements can have *attributes*
- Name/value pairs in the start tag
- Name and value separated by '=' and optional white space

Attributes

- Elements can have *attributes*
- Name/value pairs in the start tag
- Name and value separated by '=' and optional white space
- Value enclosed in single or double quotes. Always

Attributes

- Elements can have *attributes*
- Name/value pairs in the start tag
- Name and value separated by '=' and optional white space
- Value enclosed in single or double quotes. Always
- Pairs separated by white space

Attributes

- Elements can have *attributes*
- Name/value pairs in the start tag
- Name and value separated by '=' and optional white space
- Value enclosed in single or double quotes. Always
- Pairs separated by white space

```
<institution type="non"    key = 'ucs'>  
  <name>  
    Computing Service  
  </name>  
</institution>
```

Attributes

- Elements can have *attributes*
- Name/value pairs in the start tag
- Name and value separated by '=' and optional white space
- Value enclosed in single or double quotes. Always
- Pairs separated by white space

```
<institution type="non"    key = 'ucs'>  
  <name>  
    Computing Service  
  </name>  
</institution>
```

- Each attribute can appear only once in any particular tag

Attributes

- Elements can have *attributes*
- Name/value pairs in the start tag
- Name and value separated by '=' and optional white space
- Value enclosed in single or double quotes. Always
- Pairs separated by white space

```
<institution type="non"    key = 'ucs'>  
  <name>  
    Computing Service  
  </name>  
</institution>
```

- Each attribute can appear only once in any particular tag
- Attribute names follow the same rules as element names

Attributes

- Elements can have *attributes*
- Name/value pairs in the start tag
- Name and value separated by '=' and optional white space
- Value enclosed in single or double quotes. Always
- Pairs separated by white space

```
<institution type="non"    key = 'ucs'>  
  <name>  
    Computing Service  
  </name>  
</institution>
```

- Each attribute can appear only once in any particular tag
- Attribute names follow the same rules as element names
- When to use attribute values, when content?.

Character References

Character References

- Some characters can't appear as themselves in character data

Character References

- Some characters can't appear as themselves in character data
 - ◆ e.g. < and & are never allowed

Character References

- Some characters can't appear as themselves in character data
 - ◆ e.g. < and & are never allowed
 - ◆ Some characters can't be typed easily, e.g. Â¥

Character References

- Some characters can't appear as themselves in character data
 - ◆ e.g. < and & are never allowed
 - ◆ Some characters can't be typed easily, e.g. Â¥
- They can be represented as

Character References

- Some characters can't appear as themselves in character data
 - ◆ e.g. < and & are never allowed
 - ◆ Some characters can't be typed easily, e.g. Â¥
- They can be represented as
 - ◆ an entity reference, e.g. <

Character References

- Some characters can't appear as themselves in character data
 - ◆ e.g. `<` and `&` are never allowed
 - ◆ Some characters can't be typed easily, e.g. `Â¥`
- They can be represented as
 - ◆ an entity reference, e.g. `<`;
 - ◆ a numeric character reference, e.g. `<`;

Character References

- Some characters can't appear as themselves in character data
 - ◆ e.g. `<` and `&` are never allowed
 - ◆ Some characters can't be typed easily, e.g. `Â¥`
- They can be represented as
 - ◆ an entity reference, e.g. `<`;
 - ◆ a numeric character reference, e.g. `<`;
 - ◆ a hexadecimal numeric character reference, e.g. `<`;

Character References

- Some characters can't appear as themselves in character data
 - ◆ e.g. `<` and `&` are never allowed
 - ◆ Some characters can't be typed easily, e.g. `Â¥`
- They can be represented as
 - ◆ an entity reference, e.g. `<`;
 - ◆ a numeric character reference, e.g. `<`;
 - ◆ a hexadecimal numeric character reference, e.g. `<`;
- XML pre-defines only 5 entity references

Character References

- Some characters can't appear as themselves in character data
 - ◆ e.g. `<` and `&` are never allowed
 - ◆ Some characters can't be typed easily, e.g. `Â¥`
- They can be represented as
 - ◆ an entity reference, e.g. `<`;
 - ◆ a numeric character reference, e.g. `<`;
 - ◆ a hexadecimal numeric character reference, e.g. `<`;
- XML pre-defines only 5 entity references
 - ◆ `<` for the less-than symbol: `<`

Character References

- Some characters can't appear as themselves in character data
 - ◆ e.g. `<` and `&` are never allowed
 - ◆ Some characters can't be typed easily, e.g. `Â¥`
- They can be represented as
 - ◆ an entity reference, e.g. `<`;
 - ◆ a numeric character reference, e.g. `<`;
 - ◆ a hexadecimal numeric character reference, e.g. `<`;
- XML pre-defines only 5 entity references
 - ◆ `<` for the less-than symbol: `<`
 - ◆ `&` for the ampersand: `&`

Character References

- Some characters can't appear as themselves in character data
 - ◆ e.g. `<` and `&` are never allowed
 - ◆ Some characters can't be typed easily, e.g. `Â¥`
- They can be represented as
 - ◆ an entity reference, e.g. `<`;
 - ◆ a numeric character reference, e.g. `<`;
 - ◆ a hexadecimal numeric character reference, e.g. `<`;
- XML pre-defines only 5 entity references
 - ◆ `<` for the less-than symbol: `<`
 - ◆ `&` for the ampersand: `&`
 - ◆ `>` for the greater-than symbol: `>`

Character References

- Some characters can't appear as themselves in character data
 - ◆ e.g. `<` and `&` are never allowed
 - ◆ Some characters can't be typed easily, e.g. `Â¥`
- They can be represented as
 - ◆ an entity reference, e.g. `<`;
 - ◆ a numeric character reference, e.g. `<`;
 - ◆ a hexadecimal numeric character reference, e.g. `<`;
- XML pre-defines only 5 entity references
 - ◆ `<` for the less-than symbol: `<`
 - ◆ `&` for the ampersand: `&`
 - ◆ `>` for the greater-than symbol: `>`
 - ◆ `"` for straight, double quotation marks: `"`

Character References

- Some characters can't appear as themselves in character data
 - ◆ e.g. `<` and `&` are never allowed
 - ◆ Some characters can't be typed easily, e.g. `Â¥`
- They can be represented as
 - ◆ an entity reference, e.g. `<`;
 - ◆ a numeric character reference, e.g. `<`;
 - ◆ a hexadecimal numeric character reference, e.g. `<`;
- XML pre-defines only 5 entity references
 - ◆ `<` for the less-than symbol: `<`
 - ◆ `&` for the ampersand: `&`
 - ◆ `>` for the greater-than symbol: `>`
 - ◆ `"` for straight, double quotation marks: `"`
 - ◆ `'` for the apostrophe, a.k.a the straight quote: `'`.

Character sets and encodings

Character sets and encodings

- XML documents are 'text documents' containing 'characters'

Character sets and encodings

- XML documents are 'text documents' containing 'characters'
- Internally, XML processors work in Unicode, a.k.a ISO 10646

Character sets and encodings

- XML documents are 'text documents' containing 'characters'
- Internally, XML processors work in Unicode, a.k.a ISO 10646
- But computers can only process sequences of octets

Character sets and encodings

- XML documents are 'text documents' containing 'characters'
- Internally, XML processors work in Unicode, a.k.a ISO 10646
- But computers can only process sequences of octets
- Characters are mapped to octets by two-stage process

Character sets and encodings

- XML documents are 'text documents' containing 'characters'
- Internally, XML processors work in Unicode, a.k.a ISO 10646
- But computers can only process sequences of octets
- Characters are mapped to octets by two-stage process
 - ◆ A *character set* maps characters to numbers

Character sets and encodings

- XML documents are 'text documents' containing 'characters'
- Internally, XML processors work in Unicode, a.k.a ISO 10646
- But computers can only process sequences of octets
- Characters are mapped to octets by two-stage process
 - ◆ A *character set* maps characters to numbers
 - ◆ An *encoding* maps those numbers to bytes

Character sets and encodings

- XML documents are 'text documents' containing 'characters'
- Internally, XML processors work in Unicode, a.k.a ISO 10646
- But computers can only process sequences of octets
- Characters are mapped to octets by two-stage process
 - ◆ A *character set* maps characters to numbers
 - ◆ An *encoding* maps those numbers to bytes
- The name of an encoding refers to a combination of these, for example

Character sets and encodings

- XML documents are 'text documents' containing 'characters'
- Internally, XML processors work in Unicode, a.k.a ISO 10646
- But computers can only process sequences of octets
- Characters are mapped to octets by two-stage process
 - ◆ A *character set* maps characters to numbers
 - ◆ An *encoding* maps those numbers to bytes
- The name of an encoding refers to a combination of these, for example
 - ◆ **iso-8859-1**, a.k.a ISO Latin-1, defines a sub-set of characters, mainly European, mapped to numbers on the range 0-255 which are directly encoded as octets

Character sets and encodings

- XML documents are 'text documents' containing 'characters'
- Internally, XML processors work in Unicode, a.k.a ISO 10646
- But computers can only process sequences of octets
- Characters are mapped to octets by two-stage process
 - ◆ A *character set* maps characters to numbers
 - ◆ An *encoding* maps those numbers to bytes
- The name of an encoding refers to a combination of these, for example
 - ◆ **iso-8859-1**, a.k.a ISO Latin-1, defines a sub-set of characters, mainly European, mapped to numbers on the range 0-255 which are directly encoded as octets
 - ◆ **ucs-2** consists of the first 65,536 characters from Unicode encoded as a pair of bytes

Character sets and encodings

- XML documents are 'text documents' containing 'characters'
- Internally, XML processors work in Unicode, a.k.a ISO 10646
- But computers can only process sequences of octets
- Characters are mapped to octets by two-stage process
 - ◆ A *character set* maps characters to numbers
 - ◆ An *encoding* maps those numbers to bytes
- The name of an encoding refers to a combination of these, for example
 - ◆ **iso-8859-1**, a.k.a ISO Latin-1, defines a sub-set of characters, mainly European, mapped to numbers on the range 0-255 which are directly encoded as octets
 - ◆ **UCS-2** consists of the first 65,536 characters from Unicode encoded as a pair of bytes
 - ◆ **UTF-8** encodes all the characters from Unicode using a variable number of bytes. Unicode characters 0-127 (ASCII) encode to the same single byte as ASCII.

The XML declaration

The XML declaration

- XML documents *should* start with an XML declaration

The XML declaration

- XML documents *should* start with an XML declaration

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


The XML declaration

- XML documents *should* start with an XML declaration

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

- If present, it *must* be the *very* first thing in the document

The XML declaration

- XML documents *should* start with an XML declaration

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

- If present, it *must* be the *very* first thing in the document
- In the absence of other information it is used to guess the character encoding

The XML declaration

- XML documents *should* start with an XML declaration

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

- If present, it *must* be the *very* first thing in the document
- In the absence of other information it is used to guess the character encoding
- It contains 3 things that look like attributes (though they aren't):

The XML declaration

- XML documents *should* start with an XML declaration

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

- If present, it *must* be the *very* first thing in the document
- In the absence of other information it is used to guess the character encoding
- It contains 3 things that look like attributes (though they aren't):
 - ◆ version: 1.0 or *perhaps* 1.1

The XML declaration

- XML documents *should* start with an XML declaration

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

- If present, it *must* be the *very* first thing in the document
- In the absence of other information it is used to guess the character encoding
- It contains 3 things that look like attributes (though they aren't):
 - ◆ version: 1.0 or *perhaps* 1.1
 - ◆ encoding: the character encoding used in the document. Optional, default from external metadata

The XML declaration

- XML documents *should* start with an XML declaration

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

- If present, it *must* be the *very* first thing in the document
- In the absence of other information it is used to guess the character encoding
- It contains 3 things that look like attributes (though they aren't):
 - ◆ version: 1.0 or *perhaps* 1.1
 - ◆ encoding: the character encoding used in the document. Optional, default from external metadata
 - ◆ standalone. Optional, default no.

Processing instructions

Processing instructions

- Intended for passing information to particular parsers

Processing instructions

- Intended for passing information to particular parsers
- Look like a tag starting <? immediately followed by an XML name, and ending ?>

Processing instructions

- Intended for passing information to particular parsers
- Look like a tag starting <? immediately followed by an XML name, and ending ?>
- The rest is arbitrary, but often looks like a sequence of attributes

Processing instructions

- Intended for passing information to particular parsers
- Look like a tag starting <? immediately followed by an XML name, and ending ?>
- The rest is arbitrary, but often looks like a sequence of attributes

```
<?xml-stylesheet href="person.css" type="test/css" ?>
```

Processing instructions

- Intended for passing information to particular parsers
- Look like a tag starting <? immediately followed by an XML name, and ending ?>
- The rest is arbitrary, but often looks like a sequence of attributes

```
<?xml-stylesheet href="person.css" type="test/css" ?>
```

- They are not entities: no end tag; no nesting

Processing instructions

- Intended for passing information to particular parsers
- Look like a tag starting <? immediately followed by an XML name, and ending ?>
- The rest is arbitrary, but often looks like a sequence of attributes

```
<?xml-stylesheet href="person.css" type="test/css" ?>
```

- They are not entities: no end tag; no nesting
- XML declarations are not processing instructions.

CDATA

CDATA

- Raw characters can appear between '`<![CDATA['` and '`']>`'

CDATA

- Raw characters can appear between '<![CDATA[' and '']>'
- To a parser this is identical to the equivalent text expressed using entities

CDATA

- Raw characters can appear between '`<![CDATA['` and '`']>`'
- To a parser this is identical to the equivalent text expressed using entities
- Very useful for including XML examples in XML!

CDATA

- Raw characters can appear between '<![CDATA[' and ']]>'
- To a parser this is identical to the equivalent text expressed using entities
- Very useful for including XML examples in XML!

```
<![CDATA[
  <tag1>
    <!-- comment here -->
    <tag2>foo</tag2>
  </tag1>
]]>
```

CDATA

- Raw characters can appear between '<![CDATA[' and ']]>'
- To a parser this is identical to the equivalent text expressed using entities
- Very useful for including XML examples in XML!

```
<![CDATA[  
  <tag1>  
    <!-- comment here -->  
    <tag2>foo</tag2>  
  </tag1>  
]]>
```

- Beware that the sequence ']]>' can not itself appear in an XML document - use ']]>'

Comments

Comments

- XML documents can contain *comments*

Comments

- XML documents can contain *comments*
- They start with `<!--`

Comments

- XML documents can contain *comments*
- They start with `<!--`
- and end `-->`

Comments

- XML documents can contain *comments*
- They start with `<!--`
- and end `-->`
- They may not contain `--`

Comments

- XML documents can contain *comments*
- They start with `<!--`
- and end `-->`
- They may not contain `--`
- XML parsers are not required to preserve comments

Comments

- XML documents can contain *comments*
 - They start with `<!--`
 - and end `-->`
 - They may not contain `--`
 - XML parsers are not required to preserve comments
- ```
<!-- insert example here -->
```

# Well-formedness

# Well-formedness

- XML documents are required to be 'well formed'

# Well-formedness

- XML documents are required to be 'well formed'
- Every start-tag must have an end-tag

# Well-formedness

- XML documents are required to be 'well formed'
- Every start-tag must have an end-tag
- Elements must not overlap

# Well-formedness

- XML documents are required to be 'well formed'
- Every start-tag must have an end-tag
- Elements must not overlap
- One and only one root element

# Well-formedness

- XML documents are required to be 'well formed'
- Every start-tag must have an end-tag
- Elements must not overlap
- One and only one root element
- Attribute values must be quoted



# Well-formedness

- XML documents are required to be 'well formed'
- Every start-tag must have an end-tag
- Elements must not overlap
- One and only one root element
- Attribute values must be quoted
- No more than one attribute with the same name in any element

# Well-formedness

- XML documents are required to be 'well formed'
- Every start-tag must have an end-tag
- Elements must not overlap
- One and only one root element
- Attribute values must be quoted
- No more than one attribute with the same name in any element
- No comments or processing instructions inside tags

# Well-formedness

- XML documents are required to be 'well formed'
- Every start-tag must have an end-tag
- Elements must not overlap
- One and only one root element
- Attribute values must be quoted
- No more than one attribute with the same name in any element
- No comments or processing instructions inside tags
- No un-escaped '<' or '&' in character data.

# **XML: Summary**

# XML: Summary

- A meta-markup language

# XML: Summary

- A meta-markup language
- XML documents are text, processed internally in Unicode

# XML: Summary

- A meta-markup language
- XML documents are text, processed internally in Unicode
- They contain

# XML: Summary

- A meta-markup language
- XML documents are text, processed internally in Unicode
- They contain
  - ◆ *elements* (surrounded by *tags*)



# XML: Summary

- A meta-markup language
- XML documents are text, processed internally in Unicode
- They contain
  - ◆ *elements* (surrounded by *tags*)
  - ◆ an *XML declaration*

# XML: Summary

- A meta-markup language
- XML documents are text, processed internally in Unicode
- They contain
  - ◆ *elements* (surrounded by *tags*)
  - ◆ an *XML declaration*
  - ◆ *comments*

# XML: Summary

- A meta-markup language
- XML documents are text, processed internally in Unicode
- They contain
  - ◆ *elements* (surrounded by *tags*)
  - ◆ *an XML declaration*
  - ◆ *comments*
  - ◆ *processing instructions*

# XML: Summary

- A meta-markup language
- XML documents are text, processed internally in Unicode
- They contain
  - ◆ *elements* (surrounded by *tags*)
  - ◆ an *XML declaration*
  - ◆ *comments*
  - ◆ *processing instructions*
- Elements can have *attributes* and can nest

# XML: Summary

- A meta-markup language
- XML documents are text, processed internally in Unicode
- They contain
  - ◆ *elements* (surrounded by *tags*)
  - ◆ *an XML declaration*
  - ◆ *comments*
  - ◆ *processing instructions*
- Elements can have *attributes* and can nest
- Character data can contain *references*

# XML: Summary

- A meta-markup language
- XML documents are text, processed internally in Unicode
- They contain
  - ◆ *elements* (surrounded by *tags*)
  - ◆ *an XML declaration*
  - ◆ *comments*
  - ◆ *processing instructions*
- Elements can have *attributes* and can nest
- Character data can contain *references*
- Two general styles: *record orientated* vs. *mixed content*

# XML: Summary

- A meta-markup language
- XML documents are text, processed internally in Unicode
- They contain
  - ◆ *elements* (surrounded by *tags*)
  - ◆ an *XML declaration*
  - ◆ *comments*
  - ◆ *processing instructions*
- Elements can have *attributes* and can nest
- Character data can contain *references*
- Two general styles: *record orientated* vs. *mixed content*
- XML documents must be well formed.

# **Document Type Definitions**



# Defining XML documents

# Defining XML documents

- XML is used to create languages - XML *applications*

# Defining XML documents

- XML is used to create languages - XML *applications*
- How are these languages defined?

# Defining XML documents

- XML is used to create languages - XML *applications*
- How are these languages defined?
- Use a set of rules about what elements and attributes are required where

# Defining XML documents

- XML is used to create languages - XML *applications*
- How are these languages defined?
- Use a set of rules about what elements and attributes are required where
- This set of rules is a *schema*

# Defining XML documents

- XML is used to create languages - XML *applications*
- How are these languages defined?
- Use a set of rules about what elements and attributes are required where
- This set of rules is a *schema*
- A document that abides by these rules is said to be *valid*

# Defining XML documents

- XML is used to create languages - XML *applications*
- How are these languages defined?
- Use a set of rules about what elements and attributes are required where
- This set of rules is a *schema*
- A document that abides by these rules is said to be *valid*
- There are various languages for expressing schemas

# Defining XML documents

- XML is used to create languages - XML *applications*
- How are these languages defined?
- Use a set of rules about what elements and attributes are required where
- This set of rules is a *schema*
- A document that abides by these rules is said to be *valid*
- There are various languages for expressing schemas
- We'll concentrate on Document Type Definition (DTD)



# Defining XML documents

- XML is used to create languages - XML *applications*
- How are these languages defined?
- Use a set of rules about what elements and attributes are required where
- This set of rules is a *schema*
- A document that abides by these rules is said to be *valid*
- There are various languages for expressing schemas
- We'll concentrate on Document Type Definition (DTD)
- Many XML tools can check a document against a DTD, including

# Defining XML documents

- XML is used to create languages - XML *applications*
- How are these languages defined?
- Use a set of rules about what elements and attributes are required where
- This set of rules is a *schema*
- A document that abides by these rules is said to be *valid*
- There are various languages for expressing schemas
- We'll concentrate on Document Type Definition (DTD)
- Many XML tools can check a document against a DTD, including
  - ◆ `xmllint` from Gnome libxml (common on Linux systems, even if they don't run Gnome)

# Defining XML documents

- XML is used to create languages - XML *applications*
- How are these languages defined?
- Use a set of rules about what elements and attributes are required where
- This set of rules is a *schema*
- A document that abides by these rules is said to be *valid*
- There are various languages for expressing schemas
- We'll concentrate on Document Type Definition (DTD)
- Many XML tools can check a document against a DTD, including
  - ◆ `xmllint` from Gnome libxml (common on Linux systems, even if they don't run Gnome)
  - ◆ James Clark's `onsgmls`

# Defining XML documents

- XML is used to create languages - XML *applications*
- How are these languages defined?
- Use a set of rules about what elements and attributes are required where
- This set of rules is a *schema*
- A document that abides by these rules is said to be *valid*
- There are various languages for expressing schemas
- We'll concentrate on Document Type Definition (DTD)
- Many XML tools can check a document against a DTD, including
  - ◆ `xmlLint` from Gnome libxml (common on Linux systems, even if they don't run Gnome)
  - ◆ James Clark's `onsgmls`
  - ◆ The website at

<http://www.stg.brown.edu/service/xmlvalid/>

# Document Type Definition

# Document Type Definition

- Old, quirky, and with a limited syntax

# Document Type Definition

- Old, quirky, and with a limited syntax
- Inherited from SGML

# Document Type Definition

- Old, quirky, and with a limited syntax
- Inherited from SGML
- DTDs are not themselves XML documents



# Document Type Definition

- Old, quirky, and with a limited syntax
- Inherited from SGML
- DTDs are not themselves XML documents
- They let you define:

# Document Type Definition

- Old, quirky, and with a limited syntax
- Inherited from SGML
- DTDs are not themselves XML documents
- They let you define:
  - ◆ Elements and their nesting

# Document Type Definition

- Old, quirky, and with a limited syntax
- Inherited from SGML
- DTDs are not themselves XML documents
- They let you define:
  - ◆ Elements and their nesting
  - ◆ The attributes of each element

# Document Type Definition

- Old, quirky, and with a limited syntax
- Inherited from SGML
- DTDs are not themselves XML documents
- They let you define:
  - ◆ Elements and their nesting
  - ◆ The attributes of each element
  - ◆ Short cuts (a.k.a. Entities)

# Document Type Definition

- Old, quirky, and with a limited syntax
- Inherited from SGML
- DTDs are not themselves XML documents
- They let you define:
  - ◆ Elements and their nesting
  - ◆ The attributes of each element
  - ◆ Short cuts (a.k.a. Entities)
- Even if you never write one of these, the ability to read them is invaluable.

# Defining Elements

# Defining Elements

- Write `<!ELEMENT tag content>`

# Defining Elements

- Write `<!ELEMENT tag content>`
- *tag* is the name of the element being defined



# Defining Elements

- Write `<!ELEMENT tag content>`
- *tag* is the name of the element being defined
- *content* is

# Defining Elements

- Write `<!ELEMENT tag content>`
- *tag* is the name of the element being defined
- *content* is
  - ◆ **EMPTY** if the element must be empty

# Defining Elements

- Write `<!ELEMENT tag content>`
- *tag* is the name of the element being defined
- *content* is
  - ◆ **EMPTY** if the element must be empty
  - ◆ **ANY** if the element can contain text or any other element (bad idea)

# Defining Elements

- Write `<!ELEMENT tag content>`
- *tag* is the name of the element being defined
- *content* is
  - ◆ **EMPTY** if the element must be empty
  - ◆ **ANY** if the element can contain text or any other element (bad idea)
  - ◆ (*content*), where *content* can be...

**What can appear as *content*?**

# What can appear as *content*?

- '#PCDATA' - character data:  
<!ELEMENT name (#PCDATA)>

# What can appear as *content*?

- '#PCDATA' - character data:

<!ELEMENT name (#PCDATA)>

- The name of a single other element:

<!ELEMENT founded (date)>

# What can appear as *content*?

- '#PCDATA' - character data:

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

- The name of a single other element:

```
<!ELEMENT founded (date)>
```

- A comma-separated sequence of other elements:

```
<!ELEMENT institution (name,address,website)>
```



# What can appear as *content*?

- '#PCDATA' - character data:

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

- The name of a single other element:

```
<!ELEMENT founded (date)>
```

- A comma-separated sequence of other elements:

```
<!ELEMENT institution (name,address,website)>
```

- A '|' -separated list of alternatives:

```
<!ELEMENT website (url|hostname)>
```

# What can appear as *content*?

- '#PCDATA' - character data:

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

- The name of a single other element:

```
<!ELEMENT founded (date)>
```

- A comma-separated sequence of other elements:

```
<!ELEMENT institution (name,address,website)>
```

- A '|' -separated list of alternatives:

```
<!ELEMENT website (url|hostname)>
```

- Anywhere an element name can appear, you can also have either sort of list in brackets

```
<!ELEMENT institution (seeother|(name,address))>
```

# Repeating elements

# Repeating elements

- Element names, and bracketed lists, can be followed by:

# Repeating elements

- Element names, and bracketed lists, can be followed by:
  - ◆ '?' if the element (or list) can occur zero or one times

# Repeating elements

- Element names, and bracketed lists, can be followed by:
  - ◆ '?' if the element (or list) can occur zero or one times
  - ◆ '\*' if the element (or list) can occur zero or more times

# Repeating elements

- Element names, and bracketed lists, can be followed by:
  - ◆ '?' if the element (or list) can occur zero or one times
  - ◆ '\*' if the element (or list) can occur zero or more times
  - ◆ '+' if the element (or list) can occur one or more times

# Repeating elements

- Element names, and bracketed lists, can be followed by:
  - ◆ '?' if the element (or list) can occur zero or one times
  - ◆ '\*' if the element (or list) can occur zero or more times
  - ◆ '+' if the element (or list) can occur one or more times
- '\*', applied to a list of choices implies any number of any of the choices, in any order



# Repeating elements

- Element names, and bracketed lists, can be followed by:
  - ◆ '?' if the element (or list) can occur zero or one times
  - ◆ '\*' if the element (or list) can occur zero or more times
  - ◆ '+' if the element (or list) can occur one or more times
- '\*', applied to a list of choices implies any number of any of the choices, in any order
- '#PCDATA' can only appear in a list of choices if there is a '\*' in force

# Repeating elements

- Element names, and bracketed lists, can be followed by:
  - ◆ '?' if the element (or list) can occur zero or one times
  - ◆ '\*' if the element (or list) can occur zero or more times
  - ◆ '+' if the element (or list) can occur one or more times
- '\*', applied to a list of choices implies any number of any of the choices, in any order
- '#PCDATA' can only appear in a list of choices if there is a '\*' in force

```
<!ELEMENT institution
 (name,note?,address+,contact*,seeother*)>
<!ELEMENT para (#PCDATA|inst|service|em|address)*>
```

# Defining Attributes

# Defining Attributes

- Write `<!ATTLIST tag attribute type default>`

# Defining Attributes

- Write `<!ATTLIST tag attribute type default>`
- *tag* is the element in which this attribute appears

# Defining Attributes

- Write `<!ATTLIST tag attribute type default>`
- *tag* is the element in which this attribute appears
- *attribute* is the name of the attribute

# Defining Attributes

- Write `<!ATTLIST tag attribute type default>`
- *tag* is the element in which this attribute appears
- *attribute* is the name of the attribute
- *type* is one of:

# Defining Attributes

- Write `<!ATTLIST tag attribute type default>`
- *tag* is the element in which this attribute appears
- *attribute* is the name of the attribute
- *type* is one of:
  - ◆ `CDATA` if the attribute's value consists of plain characters



# Defining Attributes

- Write `<!ATTLIST tag attribute type default>`
- *tag* is the element in which this attribute appears
- *attribute* is the name of the attribute
- *type* is one of:
  - ◆ `CDATA` if the attribute's value consists of plain characters
  - ◆ `(choice_1|choice_2|...)` where each *choice\_n* represents one possibility

# Defining Attributes

- Write `<!ATTLIST tag attribute type default>`
- *tag* is the element in which this attribute appears
- *attribute* is the name of the attribute
- *type* is one of:
  - ◆ `CDATA` if the attribute's value consists of plain characters
  - ◆ `(choice_1|choice_2|...)` where each *choice\_n* represents one possibility
- *default* is one of:

# Defining Attributes

- Write `<!ATTLIST tag attribute type default>`
- *tag* is the element in which this attribute appears
- *attribute* is the name of the attribute
- *type* is one of:
  - ◆ `CDATA` if the attribute's value consists of plain characters
  - ◆ `(choice_1|choice_2|...)` where each *choice\_n* represents one possibility
- *default* is one of:
  - ◆ `#REQUIRED` if the attribute must appear

# Defining Attributes

- Write `<!ATTLIST tag attribute type default>`
- *tag* is the element in which this attribute appears
- *attribute* is the name of the attribute
- *type* is one of:
  - ◆ `CDATA` if the attribute's value consists of plain characters
  - ◆ `(choice_1|choice_2|...)` where each *choice\_n* represents one possibility
- *default* is one of:
  - ◆ `#REQUIRED` if the attribute must appear
  - ◆ `#IMPLIED` if the attribute is optional and has no default

# Defining Attributes

- Write `<!ATTLIST tag attribute type default>`
- *tag* is the element in which this attribute appears
- *attribute* is the name of the attribute
- *type* is one of:
  - ◆ `CDATA` if the attribute's value consists of plain characters
  - ◆ `(choice_1|choice_2|...)` where each *choice\_n* represents one possibility
- *default* is one of:
  - ◆ `#REQUIRED` if the attribute must appear
  - ◆ `#IMPLIED` if the attribute is optional and has no default
- There are additional types: `ID`, `IDREF`, `IDREFS`, `NMTOKEN` and `NMTOKENS`

# Defining Attributes

- Write `<!ATTLIST tag attribute type default>`
- *tag* is the element in which this attribute appears
- *attribute* is the name of the attribute
- *type* is one of:
  - ◆ `CDATA` if the attribute's value consists of plain characters
  - ◆ `(choice_1|choice_2|...)` where each *choice\_n* represents one possibility
- *default* is one of:
  - ◆ `#REQUIRED` if the attribute must appear
  - ◆ `#IMPLIED` if the attribute is optional and has no default
- There are additional types: `ID`, `IDREF`, `IDREFS`, `NMTOKEN` and `NMTOKENS`
- ... and other defaults: `"value"` and `#FIXED "value"`.

# Defining entities

# Defining entities

- Entities are shortcuts to save typing



# Defining entities

- Entities are shortcuts to save typing
- You can define your own entities in a DTD

# Defining entities

- Entities are shortcuts to save typing
- You can define your own entities in a DTD
- Confusingly, the can stand for text in the DTD itself ...

# Defining entities

- Entities are shortcuts to save typing
- You can define your own entities in a DTD
- Confusingly, they can stand for text in the DTD itself ...
- ... *or* in the document the DTD describes.

# Shortcuts for the document being described

# Shortcuts for the document being described

- An 'Internal General Entity'

```
<!ENTITY uoc "the University of Cambridge">
```

# Shortcuts for the document being described

- An 'Internal General Entity'

```
<!ENTITY uoc "the University of Cambridge">
```

- With that in our DTD, our XML document can say

```
Here at &uoc; we all love our work
```

# Shortcuts for the document being described

- An 'Internal General Entity'

```
<!ENTITY uoc "the University of Cambridge">
```

- With that in our DTD, our XML document can say

```
Here at &uoc; we all love our work
```

- Entities are often used to stand for characters that are hard to type

```
<!ENTITY copy "©">
```

# Shortcuts for the document being described

- An 'Internal General Entity'

```
<!ENTITY uoc "the University of Cambridge">
```

- With that in our DTD, our XML document can say

```
Here at &uoc; we all love our work
```

- Entities are often used to stand for characters that are hard to type

```
<!ENTITY copy "©">
```

- Or we can define an 'External General Entity'

```
<!ENTITY footer SYSTEM "/boilerplate/footer.xml">
```



# Shortcuts for the document being described

- An 'Internal General Entity'

```
<!ENTITY uoc "the University of Cambridge">
```

- With that in our DTD, our XML document can say

```
Here at &uoc; we all love our work
```

- Entities are often used to stand for characters that are hard to type

```
<!ENTITY copy "©">
```

- Or we can define an 'External General Entity'

```
<!ENTITY footer SYSTEM "/boilerplate/footer.xml">
```

- Then we can include footer.xml by saying &footer;

# Shortcuts for the document being described

- An 'Internal General Entity'

```
<!ENTITY uoc "the University of Cambridge">
```

- With that in our DTD, our XML document can say

```
Here at &uoc; we all love our work
```

- Entities are often used to stand for characters that are hard to type

```
<!ENTITY copy "©">
```

- Or we can define an 'External General Entity'

```
<!ENTITY footer SYSTEM "/boilerplate/footer.xml">
```

- Then we can include footer.xml by saying `&footer;`

- External General Entities are useful if you want to maintain your XML document in multiple files

# Shortcuts for the document being described

- An 'Internal General Entity'

```
<!ENTITY uoc "the University of Cambridge">
```

- With that in our DTD, our XML document can say

```
Here at &uoc; we all love our work
```

- Entities are often used to stand for characters that are hard to type

```
<!ENTITY copy "©">
```

- Or we can define an 'External General Entity'

```
<!ENTITY footer SYSTEM "/boilerplate/footer.xml">
```

- Then we can include footer.xml by saying `&footer;`

- External General Entities are useful if you want to maintain your XML document in multiple files

- External General Entities don't need to have a single root element but otherwise must be well formed.

# Shortcuts for the DTD

# Shortcuts for the DTD

- An 'Internal Parameter Entity' acts as a 'macro' inside the DTD  
`<!ENTITY % contact_details "name,address,website">`

# Shortcuts for the DTD

- An 'Internal Parameter Entity' acts as a 'macro' inside the DTD

```
<!ENTITY % contact_details "name,address,website">
```

- Now, instead of saying

```
<!ELEMENT department (name,address,website)>
```

```
<!ELEMENT college (name,address,website)>
```

# Shortcuts for the DTD

- An 'Internal Parameter Entity' acts as a 'macro' inside the DTD

```
<!ENTITY % contact_details "name,address,website">
```

- Now, instead of saying

```
<!ELEMENT department (name,address,website)>
```

```
<!ELEMENT college (name,address,website)>
```

- we can say

```
<!ELEMENT department (%contact_details;)>
```

```
<!ELEMENT college (%contact_details;)>
```

# Shortcuts for the DTD

- An 'Internal Parameter Entity' acts as a 'macro' inside the DTD

```
<!ENTITY % contact_details "name,address,website">
```

- Now, instead of saying

```
<!ELEMENT department (name,address,website)>
```

```
<!ELEMENT college (name,address,website)>
```

- we can say

```
<!ELEMENT department (%contact_details;)>
```

```
<!ELEMENT college (%contact_details;)>
```

- An 'External Parameter Entity' lets us include sections of DTD just like external general entities do for XML documents

```
<!ENTITY % website_stuff SYSTEM "website.dtd">
```



# Shortcuts for the DTD

- An 'Internal Parameter Entity' acts as a 'macro' inside the DTD

```
<!ENTITY % contact_details "name,address,website">
```

- Now, instead of saying

```
<!ELEMENT department (name,address,website)>
```

```
<!ELEMENT college (name,address,website)>
```

- we can say

```
<!ELEMENT department (%contact_details;)>
```

```
<!ELEMENT college (%contact_details;)>
```

- An 'External Parameter Entity' lets us include sections of DTD just like external general entities do for XML documents

```
<!ENTITY % website_stuff SYSTEM "website.dtd">
```

- This can be useful for 'modulising' DTDs.

# Associating DTDs with XML documents

# Associating DTDs with XML documents

- To be valid, an XML document must include a reference to its DTD in a 'Document Type Declaration'

# Associating DTDs with XML documents

- To be valid, an XML document must include a reference to its DTD in a 'Document Type Declaration'
  - ◆ Note that 'Document Type Definition' and 'Document Type Declaration' have the same initials - DTD means 'Document Type Definition'

# Associating DTDs with XML documents

- To be valid, an XML document must include a reference to its DTD in a 'Document Type Declaration'
  - ◆ Note that 'Document Type Definition' and 'Document Type Declaration' have the same initials - DTD means 'Document Type Definition'
- The Document Type Declaration comes after the XML Declaration and before the start-tag of the root element

# Associating DTDs with XML documents

- To be valid, an XML document must include a reference to its DTD in a 'Document Type Declaration'
  - ◆ Note that 'Document Type Definition' and 'Document Type Declaration' have the same initials - DTD means 'Document Type Definition'
- The Document Type Declaration comes after the XML Declaration and before the start-tag of the root element
- The Document Type Declaration can either refer to a DTD in a separate document

# Associating DTDs with XML documents

- To be valid, an XML document must include a reference to its DTD in a 'Document Type Declaration'
  - ◆ Note that 'Document Type Definition' and 'Document Type Declaration' have the same initials - DTD means 'Document Type Definition'
- The Document Type Declaration comes after the XML Declaration and before the start-tag of the root element
- The Document Type Declaration can either refer to a DTD in a separate document
  - ◆ called an *External DTD Subset*

# Associating DTDs with XML documents

- To be valid, an XML document must include a reference to its DTD in a 'Document Type Declaration'
  - ◆ Note that 'Document Type Definition' and 'Document Type Declaration' have the same initials - DTD means 'Document Type Definition'
- The Document Type Declaration comes after the XML Declaration and before the start-tag of the root element
- The Document Type Declaration can either refer to a DTD in a separate document
  - ◆ called an *External DTD Subset*
- Or can contain it in-line



# Associating DTDs with XML documents

- To be valid, an XML document must include a reference to its DTD in a 'Document Type Declaration'
  - ◆ Note that 'Document Type Definition' and 'Document Type Declaration' have the same initials - DTD means 'Document Type Definition'
- The Document Type Declaration comes after the XML Declaration and before the start-tag of the root element
- The Document Type Declaration can either refer to a DTD in a separate document
  - ◆ called an *External DTD Subset*
- Or can contain it in-line
  - ◆ called an *Internal DTD Subset*.

# Using External DTD Subsets

# Using External DTD Subsets

- To refer to a DTD in a local file, you need something like

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE institutions SYSTEM "/dtd/inst.dtd">
<institutions>
 ...
</institutions>
```

# Using External DTD Subsets

- To refer to a DTD in a local file, you need something like

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE institutions SYSTEM "/dtd/inst.dtd">
<institutions>
 ...
</institutions>
```
- The thing after 'SYSTEM' is a URL

# Using External DTD Subsets

- To refer to a DTD in a local file, you need something like

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE institutions SYSTEM "/dtd/inst.dtd">
<institutions>
 ...
</institutions>
```
- The thing after 'SYSTEM' is a URL
- 'Official' DTDs can be named using a 'Formal Public Identifier' (FPI). FPIs are just names in a fixed format

# Using External DTD Subsets

- To refer to a DTD in a local file, you need something like

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE institutions SYSTEM "/dtd/inst.dtd">
<institutions>
 ...
</institutions>
```

- The thing after 'SYSTEM' is a URL
- 'Official' DTDs can be named using a 'Formal Public Identifier' (FPI). FPIs are just names in a fixed format
- To refer to a DTD by FPI you need something like

```
<!DOCTYPE book PUBLIC
 "-//OASIS//DTD DocBook XML V4.2//EN"
 "http://www.oasis-open.org/docbook/xml/4.0/docbookx.dtd"
```

# Using External DTD Subsets

- To refer to a DTD in a local file, you need something like

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE institutions SYSTEM "/dtd/inst.dtd">
<institutions>
 ...
</institutions>
```

- The thing after 'SYSTEM' is a URL
- 'Official' DTDs can be named using a 'Formal Public Identifier' (FPI). FPIs are just names in a fixed format
- To refer to a DTD by FPI you need something like

```
<!DOCTYPE book PUBLIC
 "-//OASIS//DTD DocBook XML V4.2//EN"
 "http://www.oasis-open.org/docbook/xml/4.0/docbookx.dtd"
```

- A 'catalogue' then maps the FPI to an appropriate copy of the corresponding DTD document

# Using External DTD Subsets

- To refer to a DTD in a local file, you need something like

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE institutions SYSTEM "/dtd/inst.dtd">
<institutions>
 ...
</institutions>
```

- The thing after 'SYSTEM' is a URL
- 'Official' DTDs can be named using a 'Formal Public Identifier' (FPI). FPIs are just names in a fixed format
- To refer to a DTD by FPI you need something like

```
<!DOCTYPE book PUBLIC
 "-//OASIS//DTD DocBook XML V4.2//EN"
 "http://www.oasis-open.org/docbook/xml/4.0/docbookx.dtd"
```

- A 'catalogue' then maps the FPI to an appropriate copy of the corresponding DTD document
- The URL is a backup in case the FPI can't be resolved.



# Using Internal DTD Subsets

# Using Internal DTD Subsets

- The DTD can be included in-line between square brackets

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<!DOCTYPE institutions [
 <!ELEMENT institution (name,address)>
 <!ELEMENT name (#PCDATA)>
 <!ELEMENT address (#PCDATA)>
]>
<institutions>
 ...
</institutions>
```

# Using Internal DTD Subsets

- The DTD can be included in-line between square brackets

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<!DOCTYPE institutions [
 <!ELEMENT institution (name,address)>
 <!ELEMENT name (#PCDATA)>
 <!ELEMENT address (#PCDATA)>
]>
<intuitions>
 ...
</institutions>
```

- You can have both at once, but note:

# Using Internal DTD Subsets

- The DTD can be included in-line between square brackets

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<!DOCTYPE institutions [
 <!ELEMENT institution (name,address)>
 <!ELEMENT name (#PCDATA)>
 <!ELEMENT address (#PCDATA)>
]>
<intuitions>
 ...
</institutions>
```

- You can have both at once, but note:
  - ◆ element declarations can't be overridden

# Using Internal DTD Subsets

- The DTD can be included in-line between square brackets

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<!DOCTYPE institutions [
 <!ELEMENT institution (name,address)>
 <!ELEMENT name (#PCDATA)>
 <!ELEMENT address (#PCDATA)>
]>
<intutions>
 ...
</institutions>
```

- You can have both at once, but note:
  - ◆ element declarations can't be overridden
  - ◆ the internal subset can override entities in the external subset.

# DTDs: Summary

# DTDs: Summary

- A *schema* is a set of rules defining an *XML Application*

# DTDs: Summary

- A *schema* is a set of rules defining an *XML Application*
- An XML document conforming to a schema is said to be *valid*



# DTDs: Summary

- A *schema* is a set of rules defining an *XML Application*
- An XML document conforming to a schema is said to be *valid*
- A *Document Type Definition* is one language for doing this

# DTDs: Summary

- A *schema* is a set of rules defining an *XML Application*
- An XML document conforming to a schema is said to be *valid*
- A *Document Type Definition* is one language for doing this
- Lets you define elements and their nesting, attribute, entities

# DTDs: Summary

- A *schema* is a set of rules defining an *XML Application*
- An XML document conforming to a schema is said to be *valid*
- A *Document Type Definition* is one language for doing this
- Lets you define elements and their nesting, attribute, entities
- A DTD can be associated with an XML document by including a *Document Type Declaration*.

# Namespaces

**What's the problem?**

# What's the problem?

- The need to include elements from one XML Application within documents belonging to a different one

# What's the problem?

- The need to include elements from one XML Application within documents belonging to a different one
- e.g. use a 'People' application to add contact details for people in Institutions

# What's the problem?

- The need to include elements from one XML Application within documents belonging to a different one
- e.g. use a 'People' application to add contact details for people in Institutions
- ... but People uses `<name>` for the names of people, and Institution uses `<name>` for the names of institutions.



**And the solution is...**

## **And the solution is...**

- Add a application-specific prefix to elements and attributes

## And the solution is...

- Add a application-specific prefix to elements and attributes
  - ◆ perhaps `<people:name>` and `<institution:name>`

## And the solution is...

- Add a application-specific prefix to elements and attributes
  - ◆ perhaps `<people:name>` and `<institution:name>`
- But we still need a way to create unique names

## And the solution is...

- Add a application-specific prefix to elements and attributes
  - ◆ perhaps `<people:name>` and `<institution:name>`
- But we still need a way to create unique names
- For that we use URIs

## And the solution is...

- Add a application-specific prefix to elements and attributes
  - ◆ perhaps `<people:name>` and `<institution:name>`
- But we still need a way to create unique names
- For that we use URIs
- These URIs are called 'Namespace Names'

## And the solution is...

- Add a application-specific prefix to elements and attributes
  - ◆ perhaps `<people:name>` and `<institution:name>`
- But we still need a way to create unique names
- For that we use URIs
- These URIs are called 'Namespace Names'
- Since URLs are URIs they are often used

## And the solution is...

- Add a application-specific prefix to elements and attributes
  - ◆ perhaps `<people:name>` and `<institution:name>`
- But we still need a way to create unique names
- For that we use URIs
- These URIs are called 'Namespace Names'
- Since URLs are URIs they are often used
  - ◆ ... but they don't have to point to anything!



## And the solution is...

- Add a application-specific prefix to elements and attributes
  - ◆ perhaps `<people:name>` and `<institution:name>`
- But we still need a way to create unique names
- For that we use URIs
- These URIs are called 'Namespace Names'
- Since URLs are URIs they are often used
  - ◆ ... but they don't have to point to anything!

`http://purl.org/dc/`

`http://www.w3c.org/TR/REC-rdf-syntax#`

`http://www.w3.org/1999/XSL/Transform`

## And the solution is...

- Add a application-specific prefix to elements and attributes
  - ◆ perhaps `<people:name>` and `<institution:name>`
- But we still need a way to create unique names
- For that we use URIs
- These URIs are called 'Namespace Names'
- Since URLs are URIs they are often used
  - ◆ ... but they don't have to point to anything!

`http://purl.org/dc/`

`http://www.w3c.org/TR/REC-rdf-syntax#`

`http://www.w3.org/1999/XSL/Transform`

- But we can't use URIs directly in tag names, so we either declare a default namespace, or we associate the name with a prefix and use the prefix.

# Associating names with elements - default namespace

# Associating names with elements - default namespace

- We can declare a default namespace with an xmlns attribute  
`<title xmlns="http://purl.org/dc/">...</title>`

# Associating names with elements - default namespace

- We can declare a default namespace with an xmlns attribute  
`<title xmlns="http://purl.org/dc/">...</title>`
- This namespace applies to the element it is declared in and to all its children

```
<institution type="acad"
 xmlns="http://www.example.org/inst">
 <name>Division of Anaesthesia</name>
 <contact method="tel">+44 1223 217889</contact>
 <website>
 <url xmlns="http://www.example.org/url">
 http://www.medschl.cam.ac.uk/anaesthetics/
 </url>
 </website>
</institution>
```

# Associating names with elements - by prefix

# Associating names with elements - by prefix

- We can declare a nickname or *prefix*

```
<dc:title xmlns:dc="http://purl.org/dc/">
```

```
...
```

```
</dc:title>
```

# Associating names with elements - by prefix

- We can declare a nickname or *prefix*

```
<dc:title xmlns:dc="http://purl.org/dc/">
```

```
 ...
</dc:title>
```

- Prefix and element name are written separated by ':'



# Associating names with elements - by prefix

- We can declare a nickname or *prefix*

```
<dc:title xmlns:dc="http://purl.org/dc/">
```

```
...
```

```
</dc:title>
```

- Prefix and element name are written separated by ':'

- Each namespaces often has a 'conventional' prefixes, like `dc` for `http://purl.org/dc/` above, but they can be anything

```
<snoopy:title xmlns:snoopy="http://purl.org/dc/">
```

```
...
```

```
</snoopy:title>
```

# Associating names with elements - by prefix

- We can declare a nickname or *prefix*

```
<dc:title xmlns:dc="http://purl.org/dc/">
 ...
</dc:title>
```

- Prefix and element name are written separated by ':'
- Each namespaces often has a 'conventional' prefixes, like `dc` for `http://purl.org/dc/` above, but they can be anything

```
<snoopy:title xmlns:snoopy="http://purl.org/dc/">
 ...
</snoopy:title>
```

- Prefixes are available to the element they are declared in and to all its children

```
<institution type="acad"
 xmlns:inst="http://www.example.org/inst">
 <inst:name>Division of Anaesthesia</name>
 <contact method="tel">+44 1223 217889</contact>
</institution>
```

# Attributes

# Attributes

- Attributes can be associated with a namespace

# Attributes

- Attributes can be associated with a namespace
  - ◆ but normally are not

# Attributes

- Attributes can be associated with a namespace
  - ◆ but normally are not
  - ◆ in which case they are in no namespace

# Attributes

- Attributes can be associated with a namespace
  - ◆ but normally are not
  - ◆ in which case they are in no namespace
- The default namespace *doesn't* apply to attributes.

# Namespaces: Summary



# Namespaces: Summary

- Namespaces allow XML schemas to be combined

# Namespaces: Summary

- Namespaces allow XML schemas to be combined
- Namespace names are URIs

# Namespaces: Summary

- Namespaces allow XML schemas to be combined
- Namespace names are URIs
- These URIs are often URLs, but don't have to point to anything

# Namespaces: Summary

- Namespaces allow XML schemas to be combined
- Namespace names are URIs
- These URIs are often URLs, but don't have to point to anything
- You can associate a default namespace with an element and its children with `xmlns="..."`

# Namespaces: Summary

- Namespaces allow XML schemas to be combined
- Namespace names are URIs
- These URIs are often URLs, but don't have to point to anything
- You can associate a default namespace with an element and its children with `xmlns="..."`
- You can define a prefix for use in an element and its children with `xmlns:prefix="..."`.

# **Transforming XML - XSLT**

**XSLT**

# XSLT

- Specifies rules to transform one XML document into another



# XSLT

- Specifies rules to transform one XML document into another
- An XSLT stylesheet contains rules consisting of

# XSLT

- Specifies rules to transform one XML document into another
- An XSLT stylesheet contains rules consisting of
  - ◆ a pattern, and

# XSLT

- Specifies rules to transform one XML document into another
- An XSLT stylesheet contains rules consisting of
  - ◆ a pattern, and
  - ◆ a template

# XSLT

- Specifies rules to transform one XML document into another
- An XSLT stylesheet contains rules consisting of
  - ◆ a pattern, and
  - ◆ a template
- An XSLT processor tries to match parts of the input document to each patterns

# XSLT

- Specifies rules to transform one XML document into another
- An XSLT stylesheet contains rules consisting of
  - ◆ a pattern, and
  - ◆ a template
- An XSLT processor tries to match parts of the input document to each patterns
- If it can, it process the template and saves the results

# XSLT

- Specifies rules to transform one XML document into another
- An XSLT stylesheet contains rules consisting of
  - ◆ a pattern, and
  - ◆ a template
- An XSLT processor tries to match parts of the input document to each patterns
- If it can, it process the template and saves the results
- When processing is finished, these results are used to create an output document

# XSLT

- Specifies rules to transform one XML document into another
- An XSLT stylesheet contains rules consisting of
  - ◆ a pattern, and
  - ◆ a template
- An XSLT processor tries to match parts of the input document to each patterns
- If it can, it process the template and saves the results
- When processing is finished, these results are used to create an output document
- To apply an XML stylesheet you need a processor. Some examples include:

# XSLT

- Specifies rules to transform one XML document into another
- An XSLT stylesheet contains rules consisting of
  - ◆ a pattern, and
  - ◆ a template
- An XSLT processor tries to match parts of the input document to each patterns
- If it can, it process the template and saves the results
- When processing is finished, these results are used to create an output document
- To apply an XML stylesheet you need a processor. Some examples include:
  - ◆ `xs1tproc` from Gnome libxml (common on Unix systems, even if they don't run Gnome)



# XSLT

- Specifies rules to transform one XML document into another
- An XSLT stylesheet contains rules consisting of
  - ◆ a pattern, and
  - ◆ a template
- An XSLT processor tries to match parts of the input document to each patterns
- If it can, it process the template and saves the results
- When processing is finished, these results are used to create an output document
- To apply an XML stylesheet you need a processor. Some examples include:
  - ◆ `xs1tproc` from Gnome libxml (common on Unix systems, even if they don't run Gnome)
  - ◆ The Apache project's Xalan processor, available in Java and C++ versions

# XSLT

- Specifies rules to transform one XML document into another
- An XSLT stylesheet contains rules consisting of
  - ◆ a pattern, and
  - ◆ a template
- An XSLT processor tries to match parts of the input document to each patterns
- If it can, it process the template and saves the results
- When processing is finished, these results are used to create an output document
- To apply an XML stylesheet you need a processor. Some examples include:
  - ◆ `xmlproc` from Gnome libxml (common on Unix systems, even if they don't run Gnome)
  - ◆ The Apache project's Xalan processor, available in Java and C++ versions
  - ◆ Michael Kay's SAXON.

# An Example Document

# An Example Document

- We'll use *inst.xml* for the following examples:

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

```
<!DOCTYPE institutions SYSTEM "inst.dtd">
```

```
<institutions>
```

```
...
```

```
<institution type="acad">
```

```
 <name>Division of Anaesthesia</name>
```

```
 <contact type="tel">+44 1223 217889</contact>
```

```
 <website>
```

```
 <url>http://www.medschl.cam.ac.uk/anaesthetics/</url>
```

```
 </website>
```

```
</institution>
```

```
...
```

```
</institutions>
```

**XSLT Stylesheets are XML documents**

# XSLT Stylesheets are XML documents

- See *example1.xslt*.

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

```
<xsl:stylesheet
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
 version="1.0">
```

```
</xsl:stylesheet>
```

# XSLT Stylesheets are XML documents

- See *example1.xslt*.

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

```
<xsl:stylesheet
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
 version="1.0">
```

```
</xsl:stylesheet>
```

- The namespace name must be exactly as above

# XSLT Stylesheets are XML documents

- See *example1.xslt*.

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

```
<xsl:stylesheet
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
 version="1.0">
```

```
</xsl:stylesheet>
```

- The namespace name must be exactly as above
- the version attribute is required



# XSLT Stylesheets are XML documents

- See *example1.xslt*.

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

```
<xsl:stylesheet
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
 version="1.0">
```

```
</xsl:stylesheet>
```

- The namespace name must be exactly as above
- the version attribute is required
- This is a complete, though largely useless, stylesheet

# XSLT Stylesheets are XML documents

- See *example1.xslt*.

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

```
<xsl:stylesheet
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
 version="1.0">
```

```
</xsl:stylesheet>
```

- The namespace name must be exactly as above
- the version attribute is required
- This is a complete, though largely useless, stylesheet
- For reasons that we'll get to later, applying it to *inst.xml* returns all the text from within elements but nothing else!

# A simple template rule

# A simple template rule

- See *example2.xslt*.

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

```
<xsl:stylesheet
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
 version="1.0">
```

```
<xsl:template match="institution">
 An institution
</xsl:template>
```

```
</xsl:stylesheet>
```

# A simple template rule

- See *example2.xslt*.

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

```
<xsl:stylesheet
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
 version="1.0">
```

```
<xsl:template match="institution">
 An institution
</xsl:template>
```

```
</xsl:stylesheet>
```

- In effect this says

# A simple template rule

- See *example2.xslt*.

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

```
<xsl:stylesheet
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
 version="1.0">
```

```
<xsl:template match="institution">
 An institution
</xsl:template>
```

```
</xsl:stylesheet>
```

- In effect this says
  - ◆ for every `<institution>` element

# A simple template rule

- See *example2.xslt*.

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

```
<xsl:stylesheet
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
 version="1.0">
```

```
<xsl:template match="institution">
 An institution
</xsl:template>
```

```
</xsl:stylesheet>
```

- In effect this says
  - ◆ for every `<institution>` element
  - ◆ output "An institution"

# A simple template rule

- See *example2.xslt*.

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

```
<xsl:stylesheet
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
 version="1.0">
```

```
<xsl:template match="institution">
 An institution
</xsl:template>
```

```
</xsl:stylesheet>
```

- In effect this says
  - ◆ for every `<institution>` element
  - ◆ output "An institution"
  - ◆ and ignore the element's content



# A simple template rule

- See *example2.xslt*.

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

```
<xsl:stylesheet
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
 version="1.0">
```

```
<xsl:template match="institution">
 An institution
</xsl:template>
```

```
</xsl:stylesheet>
```

- In effect this says
  - ◆ for every `<institution>` element
  - ◆ output "An institution"
  - ◆ and ignore the element's content
- Anything other than XSLT tags is automatically added to the result of the transformation.

# Adding elements

# Adding elements

- See *example3.xslt*.

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

```
<xsl:stylesheet
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
 version="1.0">

 <xsl:template match="institution">
 <heading>An institution</heading>
 </xsl:template>

</xsl:stylesheet>
```

# Adding elements

- See *example3.xslt*.

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

```
<xsl:stylesheet
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
 version="1.0">
```

```
 <xsl:template match="institution">
 <heading>An institution</heading>
 </xsl:template>
```

```
</xsl:stylesheet>
```

- Tags not in the XSLT namespace are also added to the results

# Adding elements

- See *example3.xslt*.

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

```
<xsl:stylesheet
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
 version="1.0">
```

```
 <xsl:template match="institution">
 <heading>An institution</heading>
 </xsl:template>
```

```
</xsl:stylesheet>
```

- Tags not in the XSLT namespace are also added to the results
- The style sheet must remain well formed.

**Including information from the input document**

# Including information from the input document

- See *example4.xslt*.

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

```
<xsl:stylesheet
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
 version="1.0">
```

```
 <xsl:template match="institution">
 <heading>
 <xsl:value-of select="name"/>
 </heading>
 </xsl:template>
```

```
</xsl:stylesheet>
```

# Including information from the input document

- See *example4.xslt*.

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

```
<xsl:stylesheet
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
 version="1.0">
```

```
 <xsl:template match="institution">
 <heading>
 <xsl:value-of select="name"/>
 </heading>
 </xsl:template>
```

```
</xsl:stylesheet>
```

- `xsl:value-of` add a value to the results



# Including information from the input document

- See *example4.xslt*.

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

```
<xsl:stylesheet
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
 version="1.0">
```

```
 <xsl:template match="institution">
 <heading>
 <xsl:value-of select="name"/>
 </heading>
 </xsl:template>
```

```
</xsl:stylesheet>
```

- `xsl:value-of` add a value to the results
- What to add is identified by the "select" attribute

# Including information from the input document

- See *example4.xslt*.

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

```
<xsl:stylesheet
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
 version="1.0">
```

```
 <xsl:template match="institution">
 <heading>
 <xsl:value-of select="name"/>
 </heading>
 </xsl:template>
```

```
</xsl:stylesheet>
```

- `xsl:value-of` add a value to the results
- What to add is identified by the "select" attribute
- The value of an element is its text content after all the tags have been removed.

# Controlling processing order

# Controlling processing order

- See *example5.xslt*.

```
<xsl:template match="institutions">
 <heading>Here are a list of website URLs</heading>
 <xsl:apply-templates select="institution"/>
 <footing>Information provided by webmaster</footing>
</xsl:template>
```

```
<xsl:template match="institution">
 <xsl:apply-templates select="website"/>
</xsl:template>
```

```
<xsl:template match="website">
 <site>
 <xsl:value-of select="url"/>
 </site>
</xsl:template>
```

# Controlling processing order

- See *example5.xslt*.

```
<xsl:template match="institutions">
 <heading>Here are a list of website URLs</heading>
 <xsl:apply-templates select="institution"/>
 <footing>Information provided by webmaster</footing>
</xsl:template>
```

```
<xsl:template match="institution">
 <xsl:apply-templates select="website"/>
</xsl:template>
```

```
<xsl:template match="website">
 <site>
 <xsl:value-of select="url"/>
 </site>
</xsl:template>
```

- `xsl:apply-templates` lets you choose when particular elements will be processed.

# The rest of XSLT

# The rest of XSLT

- There is much more to XSLT than we've covered here, including

# The rest of XSLT

- There is much more to XSLT than we've covered here, including
  - ◆ Modes



# The rest of XSLT

- There is much more to XSLT than we've covered here, including
  - ◆ Modes
  - ◆ Named templates

# The rest of XSLT

- There is much more to XSLT than we've covered here, including
  - ◆ Modes
  - ◆ Named templates
  - ◆ Numbering and sorting output elements

# The rest of XSLT

- There is much more to XSLT than we've covered here, including
  - ◆ Modes
  - ◆ Named templates
  - ◆ Numbering and sorting output elements
  - ◆ Conditional processing

# The rest of XSLT

- There is much more to XSLT than we've covered here, including
  - ◆ Modes
  - ◆ Named templates
  - ◆ Numbering and sorting output elements
  - ◆ Conditional processing
  - ◆ Iteration

# The rest of XSLT

- There is much more to XSLT than we've covered here, including
  - ◆ Modes
  - ◆ Named templates
  - ◆ Numbering and sorting output elements
  - ◆ Conditional processing
  - ◆ Iteration
  - ◆ Extension elements.

# **XSLT: Summary so far**

# XSLT: Summary so far

- XSLT transforms one XML document into another

# XSLT: Summary so far

- XSLT transforms one XML document into another
- It does so using templates that are triggered by patterns in the input document



# XSLT: Summary so far

- XSLT transforms one XML document into another
- It does so using templates that are triggered by patterns in the input document
- Within templates, text and non-xslt elements are copied to the output document

# XSLT: Summary so far

- XSLT transforms one XML document into another
- It does so using templates that are triggered by patterns in the input document
- Within templates, text and non-xslt elements are copied to the output document
- `<xsl:value-of>` can insert the string value of an element into the output

# XSLT: Summary so far

- XSLT transforms one XML document into another
- It does so using templates that are triggered by patterns in the input document
- Within templates, text and non-xslt elements are copied to the output document
- `<xsl:value-of>` can insert the string value of an element into the output
- `<xsl:apply-templates>` controls the processing order.

**XPath**

# XPath

# XPath

- XSLT needs a general way to identify parts of the input document

# XPath

- XSLT needs a general way to identify parts of the input document
- Enter XPath, a non-XML language to identify parts of an XML document

# XPath

- XSLT needs a general way to identify parts of the input document
- Enter XPath, a non-XML language to identify parts of an XML document
- Used in XSLT `match=` and `select=` attributes



# XPath

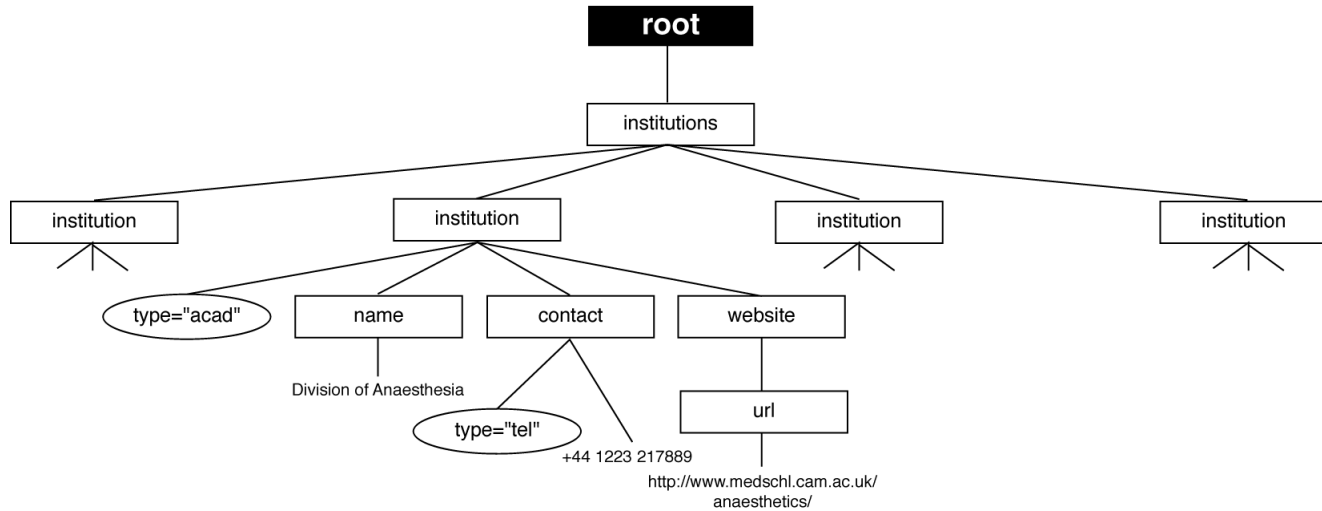
- XSLT needs a general way to identify parts of the input document
- Enter XPath, a non-XML language to identify parts of an XML document
- Used in XSLT `match=` and `select=` attributes
- In `<xsl:template match="institution">`, "institution" is an XPath expression, referring to elements of type "institution"

# XPath

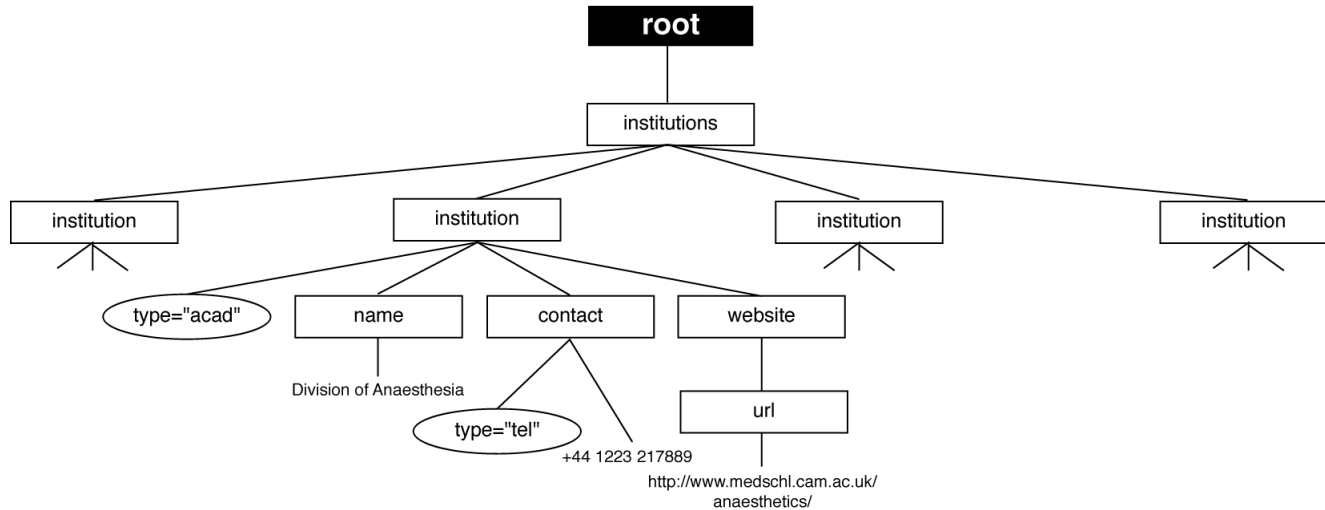
- XSLT needs a general way to identify parts of the input document
- Enter XPath, a non-XML language to identify parts of an XML document
- Used in XSLT `match=` and `select=` attributes
- In `<xsl:template match="institution">`, "institution" is an XPath expression, referring to elements of type "institution"
- XPATH is also used in XPointer, XML Schema, XForms, etc.

# XPath's view of the world

# XPath's view of the world

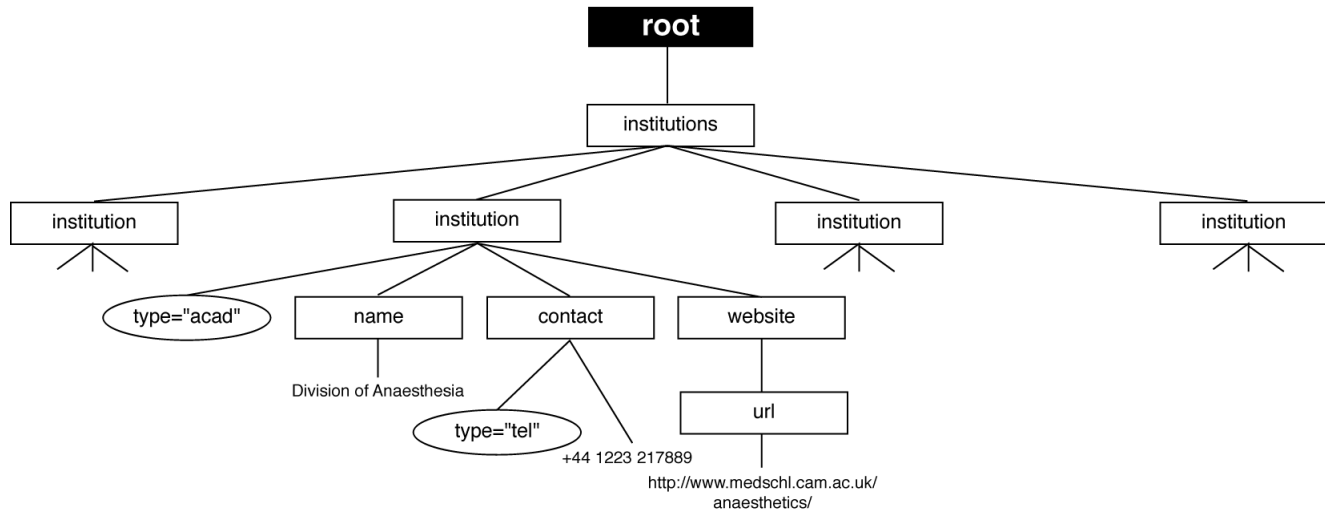


# XPath's view of the world



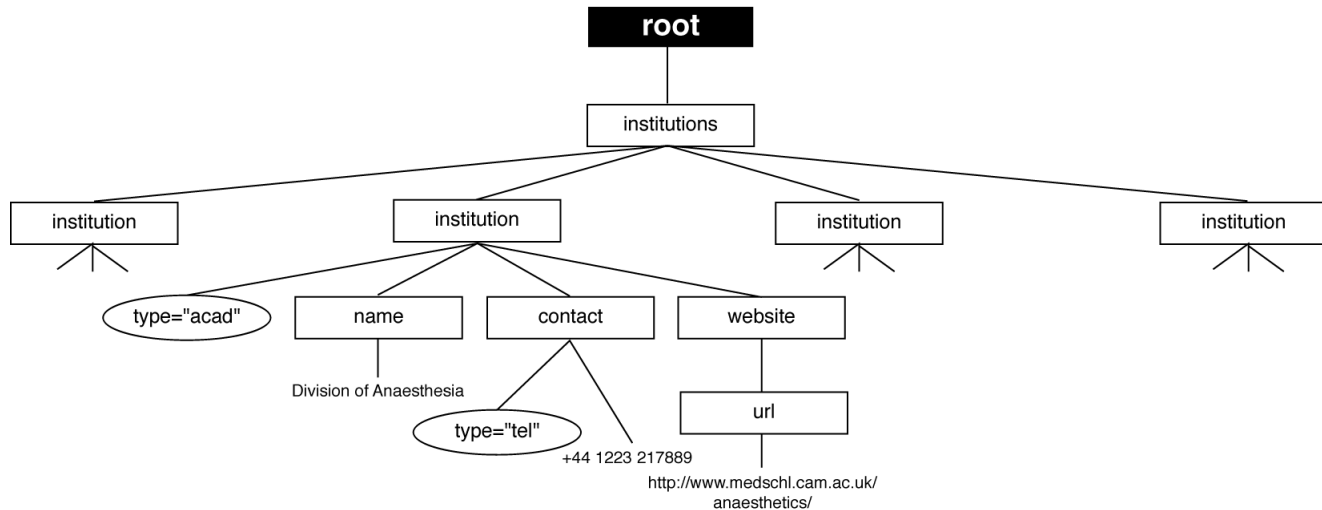
- The tree contains root, element, text and attribute nodes

# XPath's view of the world



- The tree contains root, element, text and attribute nodes
- .. also comment, processing-instruction, and namespace nodes (but that's not important right now)

# XPath's view of the world



- The tree contains root, element, text and attribute nodes
- .. also comment, processing-instruction, and namespace nodes (but that's not important right now)
- Root node is not the same as the root element.

# Location paths



# Location paths

- Identify a set of 'nodes' in a document

# Location paths

- Identify a set of 'nodes' in a document
- Operate relative to a 'context node' (c.f current directory)

# Location paths

- Identify a set of 'nodes' in a document
- Operate relative to a 'context node' (c.f current directory)
- Simplest is a single element name

```
<xsl:apply-templates select="contact">
```

# Location paths

- Identify a set of 'nodes' in a document
- Operate relative to a 'context node' (c.f current directory)
- Simplest is a single element name

```
<xsl:apply-templates select="contact">
```

- "/" matches the root node

```
<xsl:apply-templates select="/">
```

# Location paths

- Identify a set of 'nodes' in a document
- Operate relative to a 'context node' (c.f current directory)

- Simplest is a single element name

```
<xsl:apply-templates select="contact">
```

- "/" matches the root node

```
<xsl:apply-templates select="/">
```

- Attribute nodes can be selected using a "@" and the attribute name

```
<xsl:value-of select="@type">
```

# Location paths

- Identify a set of 'nodes' in a document
- Operate relative to a 'context node' (c.f current directory)

- Simplest is a single element name

```
<xsl:apply-templates select="contact">
```

- "/" matches the root node

```
<xsl:apply-templates select="/">
```

- Attribute nodes can be selected using a "@" and the attribute name

```
<xsl:value-of select="@type">
```

- Text nodes can be selected using `text()`

```
<xsl:value-of select="text()">
```

# Location paths

- Identify a set of 'nodes' in a document
- Operate relative to a 'context node' (c.f current directory)

- Simplest is a single element name

```
<xsl:apply-templates select="contact">
```

- "/" matches the root node

```
<xsl:apply-templates select="/">
```

- Attribute nodes can be selected using a "@" and the attribute name

```
<xsl:value-of select="@type">
```

- Text nodes can be selected using `text()`

```
<xsl:value-of select="text()">
```

- All of these can be chained together

```
<xsl:value-of select="website/url">
```

# More paths



# More paths

- Wildcards:

# More paths

- Wildcards:
  - ◆ \* - all element node

# More paths

- Wildcards:
  - ◆ \* - all element node
  - ◆ @\* - all attribute nodes

# More paths

- Wildcards:
  - ◆ \* - all element node
  - ◆ @\* - all attribute nodes
  - ◆ node() - all nodes

# More paths

- Wildcards:
  - ◆ \* - all element node
  - ◆ @\* - all attribute nodes
  - ◆ node() - all nodes

`institution/*/website`

# More paths

- Wildcards:
  - ◆ \* - all element node
  - ◆ @\* - all attribute nodes
  - ◆ node() - all nodes

`institution/*/website`

- Paths can specify alternatives with '|'

# More paths

- Wildcards:
  - ◆ \* - all element node
  - ◆ @\* - all attribute nodes
  - ◆ node() - all nodes

`institution/*/website`

- Paths can specify alternatives with '|'  
`contact | website`

# More paths

- Wildcards:
  - ◆ \* - all element node
  - ◆ @\* - all attribute nodes
  - ◆ node() - all nodes

`institution/*/website`

- Paths can specify alternatives with '|'  
`contact | website`
- '.' represents current node



# More paths

- Wildcards:

- ◆ \* - all element node
- ◆ @\* - all attribute nodes
- ◆ node() - all nodes

`institution/*/website`

- Paths can specify alternatives with '|'

`contact | website`

- '.' represents current node
- '..' represents the current node's parent

# More paths

- Wildcards:

- ◆ \* - all element node
- ◆ @\* - all attribute nodes
- ◆ node() - all nodes

`institution/*/website`

- Paths can specify alternatives with '|'

`contact | website`

- '.' represents current node

- '..' represents the current node's parent

`../../name`

# More paths

- Wildcards:

- ◆ \* - all element node
- ◆ @\* - all attribute nodes
- ◆ node() - all nodes

`institution/*/website`

- Paths can specify alternatives with '|'

`contact | website`

- '.' represents current node

- '..' represents the current node's parent

`../../name`

- A leading '/' makes a path absolute

# More paths

- Wildcards:

- ◆ \* - all element node
- ◆ @\* - all attribute nodes
- ◆ node() - all nodes

`institution/*/website`

- Paths can specify alternatives with '|'

`contact | website`

- '.' represents current node

- '..' represents the current node's parent

`../../name`

- A leading '/' makes a path absolute

- '// selects from all descendants

# More paths

- Wildcards:

- ◆ \* - all element node
- ◆ @\* - all attribute nodes
- ◆ node() - all nodes

`institution/*/website`

- Paths can specify alternatives with '|'

`contact | website`

- '.' represents current node

- '..' represents the current node's parent

`../../name`

- A leading '/' makes a path absolute

- '// selects from all descendants

`/institutions//url`

# Predicates

# Predicates

- An XPath expression commonly selects more than one node

# Predicates

- An XPath expression commonly selects more than one node
- Sometimes you don't want all of them



# Predicates

- An XPath expression commonly selects more than one node
- Sometimes you don't want all of them
- Each step in a location path can have a condition attached

# Predicates

- An XPath expression commonly selects more than one node
- Sometimes you don't want all of them
- Each step in a location path can have a condition attached
- This is called a *predicate*

# Predicates

- An XPath expression commonly selects more than one node
- Sometimes you don't want all of them
- Each step in a location path can have a condition attached
- This is called a *predicate*
- The predicate contains a boolean expression

# Predicates

- An XPath expression commonly selects more than one node
- Sometimes you don't want all of them
- Each step in a location path can have a condition attached
- This is called a *predicate*
- The predicate contains a boolean expression

```
//contact[@method="tel"]
```

```
//institution[@type="acad"]/contact[@method="tel"]
```

# Unabbreviated Location Paths

# Unabbreviated Location Paths

- So far we've been using abbreviated location paths

# Unabbreviated Location Paths

- So far we've been using abbreviated location paths
- There is an unabbreviated form that's even more powerful

# Unabbreviated Location Paths

- So far we've been using abbreviated location paths
- There is an unabbreviated form that's even more powerful
- For example `child::institution/attribute::type` is the same as `institution/@type`



# Unabbreviated Location Paths

- So far we've been using abbreviated location paths
- There is an unabbreviated form that's even more powerful
- For example `child::institution/attribute::type` is the same as `institution/@type`
- Abbreviated paths allow you to navigate along the following

# Unabbreviated Location Paths

- So far we've been using abbreviated location paths
- There is an unabbreviated form that's even more powerful
- For example `child::institution/attribute::type` is the same as `institution/@type`
- Abbreviated paths allow you to navigate along the following
  - ◆ child and parent

# Unabbreviated Location Paths

- So far we've been using abbreviated location paths
- There is an unabbreviated form that's even more powerful
- For example `child::institution/attribute::type` is the same as `institution/@type`
- Abbreviated paths allow you to navigate along the following
  - ◆ child and parent
  - ◆ self

# Unabbreviated Location Paths

- So far we've been using abbreviated location paths
- There is an unabbreviated form that's even more powerful
- For example `child::institution/attribute::type` is the same as `institution/@type`
- Abbreviated paths allow you to navigate along the following
  - ◆ child and parent
  - ◆ self
  - ◆ attribute

# Unabbreviated Location Paths

- So far we've been using abbreviated location paths
- There is an unabbreviated form that's even more powerful
- For example `child::institution/attribute::type` is the same as `institution/@type`
- Abbreviated paths allow you to navigate along the following
  - ◆ child and parent
  - ◆ self
  - ◆ attribute
  - ◆ descendant-or-self

# Unabbreviated Location Paths

- So far we've been using abbreviated location paths
- There is an unabbreviated form that's even more powerful
- For example `child::institution/attribute::type` is the same as `institution/@type`
- Abbreviated paths allow you to navigate along the following
  - ◆ child and parent
  - ◆ self
  - ◆ attribute
  - ◆ descendant-or-self
- The unabbreviated form additionally lets you navigate

# Unabbreviated Location Paths

- So far we've been using abbreviated location paths
- There is an unabbreviated form that's even more powerful
- For example `child::institution/attribute::type` is the same as `institution/@type`
- Abbreviated paths allow you to navigate along the following
  - ◆ child and parent
  - ◆ self
  - ◆ attribute
  - ◆ descendant-or-self
- The unabbreviated form additionally lets you navigate
  - ◆ ancestor

# Unabbreviated Location Paths

- So far we've been using abbreviated location paths
- There is an unabbreviated form that's even more powerful
- For example `child::institution/attribute::type` is the same as `institution/@type`
- Abbreviated paths allow you to navigate along the following
  - ◆ child and parent
  - ◆ self
  - ◆ attribute
  - ◆ descendant-or-self
- The unabbreviated form additionally lets you navigate
  - ◆ ancestor
  - ◆ following and preceding



# Unabbreviated Location Paths

- So far we've been using abbreviated location paths
- There is an unabbreviated form that's even more powerful
- For example `child::institution/attribute::type` is the same as `institution/@type`
- Abbreviated paths allow you to navigate along the following
  - ◆ child and parent
  - ◆ self
  - ◆ attribute
  - ◆ descendant-or-self
- The unabbreviated form additionally lets you navigate
  - ◆ ancestor
  - ◆ following and preceding
  - ◆ following-sibling and preceding-sibling

# Unabbreviated Location Paths

- So far we've been using abbreviated location paths
- There is an unabbreviated form that's even more powerful
- For example `child::institution/attribute::type` is the same as `institution/@type`
- Abbreviated paths allow you to navigate along the following
  - ◆ child and parent
  - ◆ self
  - ◆ attribute
  - ◆ descendant-or-self
- The unabbreviated form additionally lets you navigate
  - ◆ ancestor
  - ◆ following and preceding
  - ◆ following-sibling and preceding-sibling
  - ◆ namespace

# Unabbreviated Location Paths

- So far we've been using abbreviated location paths
- There is an unabbreviated form that's even more powerful
- For example `child::institution/attribute::type` is the same as `institution/@type`
- Abbreviated paths allow you to navigate along the following
  - ◆ child and parent
  - ◆ self
  - ◆ attribute
  - ◆ descendant-or-self
- The unabbreviated form additionally lets you navigate
  - ◆ ancestor
  - ◆ following and preceding
  - ◆ following-sibling and preceding-sibling
  - ◆ namespace
  - ◆ descendant

# Unabbreviated Location Paths

- So far we've been using abbreviated location paths
- There is an unabbreviated form that's even more powerful
- For example `child::institution/attribute::type` is the same as `institution/@type`
- Abbreviated paths allow you to navigate along the following
  - ◆ child and parent
  - ◆ self
  - ◆ attribute
  - ◆ descendant-or-self
- The unabbreviated form additionally lets you navigate
  - ◆ ancestor
  - ◆ following and preceding
  - ◆ following-sibling and preceding-sibling
  - ◆ namespace
  - ◆ descendant
  - ◆ ancestor-or-self.

# Other sorts of XPath expression

# Other sorts of XPath expression

- So far we've looked only at location paths

# Other sorts of XPath expression

- So far we've looked only at location paths
- These return *node-sets* which identify a set of nodes in a document

# Other sorts of XPath expression

- So far we've looked only at location paths
- These return *node-sets* which identify a set of nodes in a document
- XPath expressions can also represent numbers, strings, and booleans



# Other sorts of XPath expression

- So far we've looked only at location paths
- These return *node-sets* which identify a set of nodes in a document
- XPath expressions can also represent numbers, strings, and booleans
- Most types convert as you might expect, for example an empty node-set is 'false' when used as a boolean

# Other sorts of XPath expression

- So far we've looked only at location paths
- These return *node-sets* which identify a set of nodes in a document
- XPath expressions can also represent numbers, strings, and booleans
- Most types convert as you might expect, for example an empty node-set is 'false' when used as a boolean
- XPATH also provides useful built-in functions, for example

# Other sorts of XPath expression

- So far we've looked only at location paths
- These return *node-sets* which identify a set of nodes in a document
- XPath expressions can also represent numbers, strings, and booleans
- Most types convert as you might expect, for example an empty node-set is 'false' when used as a boolean
- XPATH also provides useful built-in functions, for example
  - ◆ `position()` returns the position of the current node in the node-set being processed

# Other sorts of XPath expression

- So far we've looked only at location paths
- These return *node-sets* which identify a set of nodes in a document
- XPath expressions can also represent numbers, strings, and booleans
- Most types convert as you might expect, for example an empty node-set is 'false' when used as a boolean
- XPATH also provides useful built-in functions, for example
  - ◆ `position()` returns the position of the current node in the node-set being processed
  - ◆ `round()` rounds a number to the nearest integer

# Other sorts of XPath expression

- So far we've looked only at location paths
- These return *node-sets* which identify a set of nodes in a document
- XPath expressions can also represent numbers, strings, and booleans
- Most types convert as you might expect, for example an empty node-set is 'false' when used as a boolean
- XPATH also provides useful built-in functions, for example
  - ◆ `position()` returns the position of the current node in the node-set being processed
  - ◆ `round()` rounds a number to the nearest integer
  - ◆ `concat()` joins strings

# Other sorts of XPath expression

- So far we've looked only at location paths
- These return *node-sets* which identify a set of nodes in a document
- XPath expressions can also represent numbers, strings, and booleans
- Most types convert as you might expect, for example an empty node-set is 'false' when used as a boolean
- XPATH also provides useful built-in functions, for example
  - ◆ `position()` returns the position of the current node in the node-set being processed
  - ◆ `round()` rounds a number to the nearest integer
  - ◆ `concat()` joins strings
  - ◆ ..etc.

# **XSLT reprise - default rules**

# XSLT reprise - default rules

- XSLT processors start by trying to process the root node



# XSLT reprise - default rules

- XSLT processors start by trying to process the root node
- If nothing else matches they apply some default template rules

# XSLT reprise - default rules

- XSLT processors start by trying to process the root node
- If nothing else matches they apply some default template rules
- For element and root nodes:

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

# XSLT reprise - default rules

- XSLT processors start by trying to process the root node
- If nothing else matches they apply some default template rules

- For element and root nodes:

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

- For text and attribute nodes:

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

# XSLT reprise - default rules

- XSLT processors start by trying to process the root node
- If nothing else matches they apply some default template rules
- For element and root nodes:

```
<xsl:template match="*|/">
 <xsl:apply-templates/>
</xsl:template>
```
- For text and attribute nodes:

```
<xsl:template match="text()|@">
 <xsl:value-of select=".">
</xsl:template>
```
- Taken together, this means that all element nodes will be visited and the text from each added to the results

# XSLT reprise - default rules

- XSLT processors start by trying to process the root node
- If nothing else matches they apply some default template rules
- For element and root nodes:

```
<xsl:template match="*|/">
 <xsl:apply-templates/>
</xsl:template>
```
- For text and attribute nodes:

```
<xsl:template match="text()|@">
 <xsl:value-of select=".">
</xsl:template>
```
- Taken together, this means that all element nodes will be visited and the text from each added to the results
- While there is a default rule for attribute nodes, none of the default rules cause attributes to be processed.

# XPath: Summary

# XPath: Summary

- A language to identify parts of an XML document

# XPath: Summary

- A language to identify parts of an XML document
- Used by XPATH and other XML technologies



# XPath: Summary

- A language to identify parts of an XML document
- Used by XPATH and other XML technologies
- Needs it's own tree view of the data

# XPath: Summary

- A language to identify parts of an XML document
- Used by XPATH and other XML technologies
- Needs it's own tree view of the data
- *Location paths* select nodes from the tree

# XPath: Summary

- A language to identify parts of an XML document
- Used by XPATH and other XML technologies
- Needs it's own tree view of the data
- *Location paths* select nodes from the tree
- There are abbreviated and unabbreviated forms of location paths

# XPath: Summary

- A language to identify parts of an XML document
- Used by XPATH and other XML technologies
- Needs it's own tree view of the data
- *Location paths* select nodes from the tree
- There are abbreviated and unabbreviated forms of location paths
- *Predicates* can filter node sets

# XPath: Summary

- A language to identify parts of an XML document
- Used by XPATH and other XML technologies
- Needs it's own tree view of the data
- *Location paths* select nodes from the tree
- There are abbreviated and unabbreviated forms of location paths
- *Predicates* can filter node sets
- XPath expressions can also return numbers, strings, and booleans

# XPath: Summary

- A language to identify parts of an XML document
- Used by XPATH and other XML technologies
- Needs it's own tree view of the data
- *Location paths* select nodes from the tree
- There are abbreviated and unabbreviated forms of location paths
- *Predicates* can filter node sets
- XPath expressions can also return numbers, strings, and booleans
- XPath includes a number of useful functions.

# **Programing with XML**

# Programing with XML



# Programing with XML

- While XML may be human readable, humans shouldn't *have* to read it

# Programing with XML

- While XML may be human readable, humans shouldn't *have* to read it
- So we want programs to do so

# Programing with XML

- While XML may be human readable, humans shouldn't *have* to read it
- So we want programs to do so
- Two approaches, exemplified by two standardised APIs

# Programing with XML

- While XML may be human readable, humans shouldn't *have* to read it
- So we want programs to do so
- Two approaches, exemplified by two standardised APIs
  - ◆ DOM (the Document Object Model)

# Programing with XML

- While XML may be human readable, humans shouldn't *have* to read it
- So we want programs to do so
- Two approaches, exemplified by two standardised APIs
  - ◆ DOM (the Document Object Model)
  - ◆ SAX (the Simple API for XML)

# Programing with XML

- While XML may be human readable, humans shouldn't *have* to read it
- So we want programs to do so
- Two approaches, exemplified by two standardised APIs
  - ◆ DOM (the Document Object Model)
  - ◆ SAX (the Simple API for XML)
- Implementations of both are available for Java, Perl, Python, C, etc., etc.

**DOM**

# DOM

- Originally developed for working on HTML and XML in a browser context



# DOM

- Originally developed for working on HTML and XML in a browser context
- Involves parsing an entire document into an interlinked set of objects and traversing the resulting tree

# DOM

- Originally developed for working on HTML and XML in a browser context
- Involves parsing an entire document into an interlinked set of objects and traversing the resulting tree
- Successive versions defined as 'levels'

# DOM

- Originally developed for working on HTML and XML in a browser context
- Involves parsing an entire document into an interlinked set of objects and traversing the resulting tree
- Successive versions defined as 'levels'
- Most recent is Level 3

# DOM

- Originally developed for working on HTML and XML in a browser context
- Involves parsing an entire document into an interlinked set of objects and traversing the resulting tree
- Successive versions defined as 'levels'
- Most recent is Level 3
- Defined in OS- and language-independent form, translated to concrete implementation in the various languages

# DOM

- Originally developed for working on HTML and XML in a browser context
- Involves parsing an entire document into an interlinked set of objects and traversing the resulting tree
- Successive versions defined as 'levels'
- Most recent is Level 3
- Defined in OS- and language-independent form, translated to concrete implementation in the various languages
- Upside being the ability to easily traverse the tree, add and delete parts, etc.

# DOM

- Originally developed for working on HTML and XML in a browser context
- Involves parsing an entire document into an interlinked set of objects and traversing the resulting tree
- Successive versions defined as 'levels'
- Most recent is Level 3
- Defined in OS- and language-independent form, translated to concrete implementation in the various languages
- Upside being the ability to easily traverse the tree, add and delete parts, etc.
- Downside is the need to parse and store the entire document in memory

# DOM

- Originally developed for working on HTML and XML in a browser context
- Involves parsing an entire document into an interlinked set of objects and traversing the resulting tree
- Successive versions defined as 'levels'
- Most recent is Level 3
- Defined in OS- and language-independent form, translated to concrete implementation in the various languages
- Upside being the ability to easily traverse the tree, add and delete parts, etc.
- Downside is the need to parse and store the entire document in memory
- See *dom.pl*.

**SAX**



# SAX

- Originally defined for Java API, but subsequently ported

# SAX

- Originally defined for Java API, but subsequently ported
- An event-based API for reading XML

# SAX

- Originally defined for Java API, but subsequently ported
- An event-based API for reading XML
- Normal implementation involves a parser that invokes a user-supplied function for each event

# SAX

- Originally defined for Java API, but subsequently ported
- An event-based API for reading XML
- Normal implementation involves a parser that invokes a user-supplied function for each event
- Upside:

# SAX

- Originally defined for Java API, but subsequently ported
- An event-based API for reading XML
- Normal implementation involves a parser that invokes a user-supplied function for each event
- Upside:
  - ◆ Don't need to hold the entire tree in memory

# SAX

- Originally defined for Java API, but subsequently ported
- An event-based API for reading XML
- Normal implementation involves a parser that invokes a user-supplied function for each event
- Upside:
  - ◆ Don't need to hold the entire tree in memory
  - ◆ Incremental processing possible

# SAX

- Originally defined for Java API, but subsequently ported
- An event-based API for reading XML
- Normal implementation involves a parser that invokes a user-supplied function for each event
- Upside:
  - ◆ Don't need to hold the entire tree in memory
  - ◆ Incremental processing possible
- Downside:

# SAX

- Originally defined for Java API, but subsequently ported
- An event-based API for reading XML
- Normal implementation involves a parser that invokes a user-supplied function for each event
- Upside:
  - ◆ Don't need to hold the entire tree in memory
  - ◆ Incremental processing possible
- Downside:
  - ◆ Can be harder to program



# SAX

- Originally defined for Java API, but subsequently ported
- An event-based API for reading XML
- Normal implementation involves a parser that invokes a user-supplied function for each event
- Upside:
  - ◆ Don't need to hold the entire tree in memory
  - ◆ Incremental processing possible
- Downside:
  - ◆ Can be harder to program
  - ◆ Need to maintain your own data structures to keep track of tree position, result data, etc.

# SAX

- Originally defined for Java API, but subsequently ported
- An event-based API for reading XML
- Normal implementation involves a parser that invokes a user-supplied function for each event
- Upside:
  - ◆ Don't need to hold the entire tree in memory
  - ◆ Incremental processing possible
- Downside:
  - ◆ Can be harder to program
  - ◆ Need to maintain your own data structures to keep track of tree position, result data, etc.
- Note that many DOM implementations use a SAX parser to build the DOM tree

# SAX

- Originally defined for Java API, but subsequently ported
- An event-based API for reading XML
- Normal implementation involves a parser that invokes a user-supplied function for each event
- Upside:
  - ◆ Don't need to hold the entire tree in memory
  - ◆ Incremental processing possible
- Downside:
  - ◆ Can be harder to program
  - ◆ Need to maintain your own data structures to keep track of tree position, result data, etc.
- Note that many DOM implementations use a SAX parser to build the DOM tree
- See *sax.pl*.

# **XML Programming: Summary**

# XML Programming: Summary

- While human readable, XML is really for programs

# XML Programming: Summary

- While human readable, XML is really for programs
- There are two main approaches

# XML Programming: Summary

- While human readable, XML is really for programs
- There are two main approaches
  - ◆ Tree-based, as exemplified by the DOM

# XML Programming: Summary

- While human readable, XML is really for programs
- There are two main approaches
  - ◆ Tree-based, as exemplified by the DOM
  - ◆ Event based, as exemplified by SAX.



**Some other core XML technologies**

# **XSL Formatting Objects (XSL-FO)**

# **XSL Formatting Objects (XSL-FO)**

- An XML application for describing the layout of text on a page

# **XSL Formatting Objects (XSL-FO)**

- An XML application for describing the layout of text on a page
- Normally created as the target of an XSLT transformation

# XSL Formatting Objects (XSL-FO)

- An XML application for describing the layout of text on a page
- Normally created as the target of an XSLT transformation
- See *example7.xslt*

# XSL Formatting Objects (XSL-FO)

- An XML application for describing the layout of text on a page
- Normally created as the target of an XSLT transformation
- See *example7.xslt*
- This can be hard, but that's because laying out pages is hard

# XSL Formatting Objects (XSL-FO)

- An XML application for describing the layout of text on a page
- Normally created as the target of an XSLT transformation
- See *example7.xslt*
- This can be hard, but that's because laying out pages is hard
- Needs a processor to convert XSL-FO to print

# XSL Formatting Objects (XSL-FO)

- An XML application for describing the layout of text on a page
- Normally created as the target of an XSLT transformation
- See *example7.xslt*
- This can be hard, but that's because laying out pages is hard
- Needs a processor to convert XSL-FO to print
- Most free ones seem to be poor



# XSL Formatting Objects (XSL-FO)

- An XML application for describing the layout of text on a page
- Normally created as the target of an XSLT transformation
- See *example7.xslt*
- This can be hard, but that's because laying out pages is hard
- Needs a processor to convert XSL-FO to print
- Most free ones seem to be poor
- Examples include

# XSL Formatting Objects (XSL-FO)

- An XML application for describing the layout of text on a page
- Normally created as the target of an XSLT transformation
- See *example7.xslt*
- This can be hard, but that's because laying out pages is hard
- Needs a processor to convert XSL-FO to print
- Most free ones seem to be poor
- Examples include
  - ◆ The Apache project's FOP

# XSL Formatting Objects (XSL-FO)

- An XML application for describing the layout of text on a page
- Normally created as the target of an XSLT transformation
- See *example7.xslt*
- This can be hard, but that's because laying out pages is hard
- Needs a processor to convert XSL-FO to print
- Most free ones seem to be poor
- Examples include
  - ◆ The Apache project's FOP
  - ◆ Sebastian Rahtz's PassiveTeX.

# XML Schema

# XML Schema

- DTD's are traditionally used when defining XML schemas

# XML Schema

- DTD's are traditionally used when defining XML schemas
- But they are limited in what they can do

# XML Schema

- DTD's are traditionally used when defining XML schemas
- But they are limited in what they can do
- and are not themselves expressed in XML

# XML Schema

- DTD's are traditionally used when defining XML schemas
- But they are limited in what they can do
- and are not themselves expressed in XML
- *XML Schema*, a W3C recommendation, attempts to address this



# XML Schema

- DTD's are traditionally used when defining XML schemas
- But they are limited in what they can do
- and are not themselves expressed in XML
- *XML Schema*, a W3C recommendation, attempts to address this
- Can describe complex restrictions on elements and attributes

# XML Schema

- DTD's are traditionally used when defining XML schemas
- But they are limited in what they can do
- and are not themselves expressed in XML
- *XML Schema*, a W3C recommendation, attempts to address this
- Can describe complex restrictions on elements and attributes
- Understands namespaces

# XML Schema

- DTD's are traditionally used when defining XML schemas
- But they are limited in what they can do
- and are not themselves expressed in XML
- *XML Schema*, a W3C recommendation, attempts to address this
- Can describe complex restrictions on elements and attributes
- Understands namespaces
- Multiple XML Schemas can be combined

# XML Schema

- DTD's are traditionally used when defining XML schemas
- But they are limited in what they can do
- and are not themselves expressed in XML
- *XML Schema*, a W3C recommendation, attempts to address this
- Can describe complex restrictions on elements and attributes
- Understands namespaces
- Multiple XML Schemas can be combined
- There are yet more schema languages, such as RELAX NG and Schematron.

# XLinks

# XLinks

- An attribute-based syntax for attaching links to XML documents

# XLinks

- An attribute-based syntax for attaching links to XML documents
- Like HTML's `<a>` tag on steroids

# XLinks

- An attribute-based syntax for attaching links to XML documents
- Like HTML's `<a>` tag on steroids
  - ◆ Unidirectional



# XLinks

- An attribute-based syntax for attaching links to XML documents
- Like HTML's `<a>` tag on steroids
  - ◆ Unidirectional
  - ◆ Bidirectional

# XLinks

- An attribute-based syntax for attaching links to XML documents
- Like HTML's <a> tag on steroids
  - ◆ Unidirectional
  - ◆ Bidirectional
  - ◆ Multidirectional

```
<book xmlns:xlink="http://www.w3.org/1999/xlink"
 xlink:type="simple"
 xlink:href=
 "http://ftp.archive.org/etext/etext93/wizoz10.txt">
```

# XPointer

# XPointer

- A non-XML syntax for identifying locations inside XML documents

# XPointer

- A non-XML syntax for identifying locations inside XML documents
- Intended to be used as a fragment identifier in a URL

# XPointer

- A non-XML syntax for identifying locations inside XML documents
- Intended to be used as a fragment identifier in a URL
- Leverages XPath

```
http://www.example.org/
 inst.dtd#xpointer(//institution[1])
```

# Other stuff

# Other stuff

- XInclude - technology for combining XML documents



## Other stuff

- XInclude - technology for combining XML documents
- XForms - reformulation and extension of HTML forms

## Other stuff

- XInclude - technology for combining XML documents
- XForms - reformulation and extension of HTML forms
- ... and many more.

# **Example XML applications**

# Narrative

# Narrative

- Text Encoding Initiate (TEI)

# Narrative

- Text Encoding Initiative (TEI)
- DocBook

# Narrative

- Text Encoding Initiative (TEI)
- DocBook
- OpenOffice

# Narrative

- Text Encoding Initiative (TEI)
- DocBook
- OpenOffice
- XHTML



# Narrative

- Text Encoding Initiative (TEI)
- DocBook
- OpenOffice
- XHTML
  - ◆ Can be created as the output from an XSLT transformation - for example see *example6.xslt*

# Narrative

- Text Encoding Initiative (TEI)
- DocBook
- OpenOffice
- XHTML
  - ◆ Can be created as the output from an XSLT transformation - for example see *example6.xslt*
  - ◆ Even on-the-fly by modern browsers

# Narrative

- Text Encoding Initiative (TEI)
- DocBook
- OpenOffice
- XHTML
  - ◆ Can be created as the output from an XSLT transformation - for example see *example6.xslt*
  - ◆ Even on-the-fly by modern browsers
  - ◆ Can also be styled using CSS - see *example1.css*.

# Data-oriented

# Data-oriented

- SVG - scalable vector graphics

# Data-oriented

- SVG - scalable vector graphics
- RSS and Atom - content summary

# Data-oriented

- SVG - scalable vector graphics
- RSS and Atom - content summary
- Jabber - Instant Messaging carried by XML

# Data-oriented

- SVG - scalable vector graphics
- RSS and Atom - content summary
- Jabber - Instant Messaging carried by XML
- Web services - XML-RPC, SOAP carrying information over XML.



**Where to go from here?**

# Where to go from here?

- Choose what interests you

# Where to go from here?

- Choose what interests you
- Remember it's a *huge* field

# Where to go from here?

- Choose what interests you
- Remember it's a *huge* field
- Explore the available resources

# Where to go from here?

- Choose what interests you
- Remember it's a *huge* field
- Explore the available resources
  - ◆ in print

# Where to go from here?

- Choose what interests you
- Remember it's a *huge* field
- Explore the available resources
  - ◆ in print
  - ◆ in the standards and recommendations

# Where to go from here?

- Choose what interests you
- Remember it's a *huge* field
- Explore the available resources
  - ◆ in print
  - ◆ in the standards and recommendations
  - ◆ elsewhere on the web.

**That's All Folks**

**If you have been, thanks for listening**