XML, XHTML, and HTML5

Credit to James Bowen, UCC, for the development of the material used here
HTML is not extensible

• One defect of HTML is that it is not extensible
• In HTML we cannot, for example, define notions like poem or verse or line
• The best we can do, if we want to markup a poem in HTML, is to “adapt” the existing tags to our own needs
• For example, we might
  – use a <h1> tag to mark the start of a poem
  – use <ol> and </ol> tags to mark the start and finish of a verse
  – use <li> and </li> tags to mark the start and end of lines in a verse
Trying to use HTML to markup a poem

• A poem might be marked up in HTML like this

```html
<h1>The Daffodils</h1>
<ol>
<li>I wandered lonely as a cloud</li>
<li>That floats on high o’er vales and hills</li>
<li>When all at once I saw a crowd</li>
<li>A host of golden daffodils</li>
</ol>
<ol>
<li>Something something something</li>
<li>Something something something else</li>
<li>...</li>
</ol>
```
This poem markup is not very satisfactory
For example, each line would be rendered, by default, with a bullet in front
Sure, we could use CSS to overcome this
But how would we markup the end of the poem?
OK, we might devise some way or other
But who else would know that we intend <h1> tags to markup the start of a poem, <ol> and </ol> tags to delimit verses, and so on?
What about search engines? How would they find our poems?
We need a way of defining new purpose-specific tags
Some History

- SGML
  - Standard Generalized Markup Language
- SGML has been in use for many years for marking up electronic documents
- SGML is actually a meta-language
  - it is used to define markup languages for specific applications, for example
    - poetry
    - legal documents
    - etc
- In fact, HTML was defined using SGML
Some History (contd.)

• If we want extensibility, why not use SGML?
  – SGML is very complex
  – many of SGML’s features are only rarely used

• XML is a version of SGML which omits the rarely-used features and simplifies the tasks of
  – defining document types
  – writing programs which process documents

• But XML is still SGML and software for processing SGML can still be used to process XML files

• Indeed, it has been said by SGML people that
  “XML is SGML-- not HTML++”
Some History (contd.)

• XML is a simpler version of SGML, adequate for web purposes

• Thus, XML is a meta-language.
  
  – that is, XML is a language that is used to define other languages.

• With XML, one can:
  
  • define new tags
  
  • use these to define platform-independent languages and data structures

• With XML one cannot specify rendering
  
  – other, related, technologies are needed for that purpose
What is XML good for?

• Defining new Markup Languages
  – eg, WML (Wireless Markup Language) was defined using XML

• Data-exchange
  – Companies have been interchanging electronic data for years
  – But XML makes it easier
  – XML enables us to exchange data across HTTP connections, instead of having to use dedicated channels
  – XML data structures are more flexible than earlier EDI formats -- as well as being computer-readable, they are also human-readable
Defining New Tags

- In XML, we define our own tags
- For example, to markup poetry, we might define the following new tags:

  <poem>     </poem>
  <title>     </title>
  <poet>      </poet>
  <verse>     </verse>
  <line>     </line>

- We could use these to markup the poem we saw earlier, as shown on the next slide
Using XML to mark up a poem

<poe>  
<title>The Daffodils</title>  
<poet>William Wordsworth</poet>  
<verse>  
<line>I wandered lonely as a cloud</line>  
<line>That floats on high o’er vales and hills</line>  
<line>When all at once I saw a crowd</line>  
<line>A host of golden daffodils</line>  
</verse>  
<verse>  
<line>Something something something</line>  
<line>Something something something else</line>  
<line>... </line>  
</verse>  
</poe>
We could even mark up the rhymes in a poem

- We might define a `rhyme` attribute for the `<line>` tag
- Using this, the first verse of the earlier poem might be marked up like this:

```xml
<verse>
  <line rhyme="owd">I wandered lonely as a cloud</line>
  <line rhyme="ills">That floats on high o’er vales and hills</line>
  <line rhyme="owd">When all at once I saw a crowd</line>
  <line rhyme="ills">A host of golden daffodils</line>
</verse>
```
Marking up the rhymes in a poem (contd.)

• Alternatively, we might define a pair of new tags:
  
  <rhyme>   </rhyme>

• Using these tags, the first verse of the earlier poem might be marked up like this:

  <verse>
  <line>I wandered lonely as a cloud <rhyme>owd</rhyme></line>
  <line>That floats on high o’er vales and hills
     <rhyme>ills</rhyme>  </line>
  <line>When all at once I saw a crowd <rhyme>owd</rhyme></line>
  <line>A host of golden daffodils <rhyme>ills</rhyme>  </line>
  </verse>

• Remember: we have to use other technology to control rendering, so, regarding the content of rhyme tags, we could decide not to print it in certain renderings or to render it in a special way in other cases
Marking up the rhymes in a poem (contd.)

- As a third option, we might define the tags
  
  `<rhyme>          </rhyme>`

  but give the `<rhyme>` tag a `sound` attribute

- Using this approach, the verse might be marked up like this:

  `<verse>`
  `<line>`I wandered lonely as a cloud</line>`
  `<line>`That floats on high o'er vales and hills</line>`
  `<line>`When all at once I saw a crowd</line>`
  `<line>`A host of golden daffodils</line>`
  `</verse>`
Marking up the rhymes in a poem (contd.)

- As a fourth option, we might define the tags
  
  `<sound>          </sound>`

  for tagging a child element of the `<rhyme>` tag

- Using this approach, the verse might be marked up like this:

  `<verse>`
  `<line>`I wandered lonely as a
  `  <rhyme><sound>owd</sound>cloud</rhyme>`</line>
  `<line>`That floats on high o'er vales and
  `  <rhyme><sound>ills</sound>hills</rhyme>`</line>
  `<line>`When all at once I saw a
  `  <rhyme><sound>owd</sound>crowd</rhyme>`</line>
  `<line>`A host of golden
  `  <rhyme><sound>ills</sound>daffodils</rhyme>`</line>
  `</verse>`
Document Type Definitions and Schemata

• We have seen several ways of using XML to markup a poem, but we could invent a host of other possible approaches.

• The important point is that, once we have invented a new set of tags and attributes,
  – XML provides us with a way of documenting the relationship between these tags/attributes so that other people can adopt/understand our approach.

• In fact, XML provides two different ways:
  – the original way is to write a Document Type Definition (DTD)
  – a newer way it to write an XML Schema

If we publish our DTD or our schema, we are, in fact, publishing a new XML-based language that others can use.
Tags in XML

• XML elements are tagged like HTML elements
• However, there are some differences
• XML tags are case sensitive:
  – the tag `<Verse>` is different from the tags `<VERSE>` and `<verse>`
• Every start tag **must have** a corresponding closing tag
  – a tag like `<verse>` must have a corresponding tag `</verse>`
• This implies that XML tags must be nested properly
• Besides starting tags and closing tags, XML allows empty tags (an empty tag is a start tag which does not have a corresponding closing tag)
  – Empty tags must be written in a special way, with a terminal slash character, as in

  `<someEmptyTag/>`
Content of XML elements

• The content of an element is what lies between its start and closing tags

• Elements can have different types of content

• The following `<trustworthy/>` element has “empty content”
  
  `<trustworthy/>
  
• The following `<name>` and `<age>` elements have “character content”
  
  `<name>Fred</name>
  `<age>25</age>`
Content of XML elements (contd.)

- The following `<person>` element has “element content”

```xml
  <person>
    <age>25</age>
    <name>Fred</name>
    <trustworthy/>
  </person>
```

- The following `<person>` element has “mixed content”

```xml
  <person>
    Fred
    <age>25</age>
    <trustworthy/>
  </person>
```
Attributes in XML

• As in HTML, start tags in XML can have attributes.

• Unlike in HTML, the value of every attribute in XML must be quoted, as we saw in

  <rhyme sound="owd">cloud</rhyme>

• Every attribute must be quoted, even if it a number, as in

  <person age="25">Fred</person>
Comments in XML

• Comments in XML are the same as in HTML
• Thus

<!-- This is a comment -->
XML version declarations

• The first line of an XML document may declare the version of XML that the document is based on.

• This line uses a special tag and is of the following form:

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

   where the 1.0 will change when a newer version of XML is used.

• Although an XML version declaration is optional, it will become critical when newer versions of XML are defined, since it will enable programs processing a document to know which syntactic rules to apply.
Syntactic acceptability of an XML document

• XML documents can have different levels of syntactic acceptability
  – at the lowest level of acceptability, a document is not a proper XML document at all -- it is ill-formed syntactically
  – at the next level of acceptability, a document obeys the basic rules of XML syntax -- it is well-formed syntactically
  – at the highest level of acceptability, a document obeys the syntax rules of a special language based on XML -- it is said to be syntactically valid according to the rules for this special XML-based language
Well-formed XML documents

• To be well-formed an XML document must satisfy the following basic rules of XML syntax:
  – the document contains exactly one root element
  – all other elements are descendents of the root element
  – start and closing tags for an element have the same spelling
  – the start tag for an empty element has a final /
  – attribute names are used only once within the same start tag
  – elements are properly nested
Example well-formed XML document

<?xml version="1.0"?>
<poem>
  <title>The Daffodils</title>
  <poet>William Wordsworth</poet>
  <verse>
    <line>I wandered lonely as a cloud</line>
    <line>That floats on high o' er vales and hills</line>
    <line>When all at once I saw a crowd</line>
    <line>A host of golden daffodils</line>
  </verse>
  <verse>
    <line>Something something something</line>
    <line>Something something something else</line>
    <line>... </line>
  </verse>
</poem>
Another well-formed XML document

<?xml version="1.0"?>
<people>
  <man>
    <name>
      Ray Burke
    </name>
    <numberOfWives>
      1
    </numberOfWives>
  </man>
  <woman>
    <name>
      Margaret Thatcher
    </name>
    <age>
      75
    </age>
  </woman>
</people>
Valid XML documents

- To be valid, an XML document
  - must be well-formed according to the general XML syntax rules
  - and, in addition, it must satisfy the syntax rules for an application-specific language based on XML

- Well-formed XML documents are useful but, to be really useful, a document must be valid

- For example, if two companies wish to exchange information using XML,
  - the companies must design a special XML-based language which will be capable of expressing the data they wish to exchange and
  - all documents that are exchanged must be valid according to the syntax rules for this special language
Valid XML documents (contd.)

• A valid XML document must declare the special XML-based language according to whose rules the XML document is claimed to be valid

• This done using a document type declaration, which must appear before the start tag of the root element in a valid XML document

• It has the following general format

  `<!DOCTYPE nameOfRootElement someDTDspec>`

where `someDTDspec` would specify the rules for a valid element whose name is `nameOfRootElement`

• `someDTDspec` would either
  – directly contain an internal document type definition (a DTD)
  – consist of a reference to an external DTD
Valid XML documents (contd.)

• For example, if we wanted to turn the example well-formed XML document we have just seen into a valid document, we would insert a line which would look like

    <!DOCTYPE people someDTDspec >

where someDTDspec would specify what rules must be followed by a valid people element

• Although a doctype statement may directly contain an internal DTD, we will consider only the cases where a doctype statement refers to an external DTD, because these are the most useful
DOCTYPEs referring to external DTDs

• A doctype statement which refers to an external DTD has one of two formats:

  `<!DOCTYPE nameOfRootElement SYSTEM "fileNameAndPath" >`

  `<!DOCTYPE nameOfRootElement PUBLIC "fpi" "url" >`

• In the first format,

  `<!DOCTYPE nameOfRootElement SYSTEM "fileNameAndPath" >`

  the keyword `SYSTEM` says that the external DTD is on the local machine and `fileNameAndPath` says where to find it

• In the second format,

  `<!DOCTYPE nameOfRootElement PUBLIC "fpi" "url" >`

  the keyword `PUBLIC` says that the external DTD is publicly available on the Internet, the `fpi` (Formal Public Identifier) gives its formal name and the `url` says where it can be found
Example DOCTYPEs referring to external DTDs

- The doctype statement
  
  ```xml
  <!DOCTYPE people SYSTEM "myFirstDTD.dtd">
  ```

  says that the external DTD is on the local machine, in a file called `myFirstDTD.dtd`

- The doctype statement
  
  ```xml
  <!DOCTYPE people PUBLIC "-//UCC//DTD PEOPLE v1.0//EN"
  "student.cs.ucc.ie/path/to/myFirstDTD.dtd">
  ```

  says that the external DTD is publicly available on the Internet, gives it a formal name which should be used by everybody who wants to refer to it, and says where it can be found
XML document which claims to be valid

<?xml version="1.0"?>
<!DOCTYPE poem SYSTEM "poem.dtd">
<poem>
  <title>The Daffodils</title>
  <poet>William Wordsworth</poet>
  <verse>
    <line>I wandered lonely as a cloud</line>
    <line>That floats on high o'er vales and hills</line>
    <line>When all at once I saw a crowd</line>
    <line>A host of golden daffodils</line>
  </verse>
  <verse>
    <line>Something something something</line>
    <line>... </line>
  </verse>
</poem>
Another XML document claiming to be valid

```xml
<?xml version="1.0"?>
<!DOCTYPE people SYSTEM "personnel.dtd">
<people>
  <man>
    <name>
      Ray Burke
    </name>
    <numberOfWives>
      1
    </numberOfWives>
  </man>
  <woman>
    <name>
      Margaret Thatcher
    </name>
    <age>
      75
    </age>
  </woman>
</people>
```
Document Type Definitions

• A DTD defines the syntax rules for an application-specific type of XML-documents.

• A DTD describes the root element that may appear in a valid document and, in doing so, also describes the kinds of children elements that the root element may possess.

• A DTD must contain **element type declarations**, statements which have the following general format:

  ```
  <!ELEMENT elementName contentModel >
  ```

• In addition, a DTD may contain a range of other types of statement, only one of which we will cover here, **attribute declarations**, statements which are used to define the attributes which elements may have.
Element Type Declarations

• An element type declaration is a statement which has the following general format:

```
<!ELEMENT elementName ( contentModel )>
```

where contentModel describes the content (if any) that may or must exist between the start tag and the closing tag (if any) of the element whose name is elementName
Empty Elements

- Empty elements are declared using the keyword **EMPTY** inside the parentheses

  ```xml
  <!ELEMENT element-name (EMPTY)>
  ```

- Example:

  ```xml
  <!ELEMENT trustworthy (EMPTY)>
  which declares an element which only has a start tag and which, unless it has some attribute(s), must be written as
  ```xml
  <trustworthy/>
  ```
Elements with Character content

- Elements that contain character data are declared as follows:

  ```xml
  <!ELEMENT element-name (#PCDATA)>
  ```

  where

  
  #PCDATA stands for “parsed character data”

- Example:

  ```xml
  <!ELEMENT month (#PCDATA)>
  ```

  This element declaration would be satisfied by the following text in an XML file

  ```xml
  <month>January</month>
  ```
Elements with element content

- Elements that contain only children elements are declared as follows:

  ```xml
  <!ELEMENT element-name (child-element-name)>
  ```

  or

  ```xml
  <!ELEMENT element-name
      (child-name1, child-name2, ... )>
  ```

- Example

  ```xml
  <!ELEMENT country (position)>
  ```

  which specifies that a `country` element contains exactly one child element, a `position` element:

  ```xml
  <country>
    <position> ..... </position>
  </country>
  ```
Elements with element content (contd.)

- Example

```xml
<!ELEMENT memorandum
    (sender, recipient, message)>
```

which specifies that a `memorandum` element contains a `sender` element, followed by a `recipient` element, followed by a `message` element

- When a sequence of children elements is declared, as above, the children must appear in exactly the same sequence in conforming XML documents:

```xml
<memorandum>
    <sender> ..... </sender>
    <recipient> ...... </recipient>
    <message> ...... </message>
</memorandum>
```
Full declaration

• When an element is declared to have element content, the children element types must also be declared

• Example:

```xml
<!ELEMENT memorandum (sender,recipient,message)>
<!ELEMENT sender     (#PCDATA)>
<!ELEMENT recipient  (#PCDATA)>
<!ELEMENT message    (#PCDATA)>
```

to which the following XML fragment would conform:

```xml
<memorandum>
  <sender>Bertie Ahern</sender>
  <recipient>Ned O’Keefe</recipient>
  <message>Please resign immediately</message>
</memorandum>
```
Elements with element content (contd.)

- When an element has children, the children may be optional, may occur only once, or may be repeated.
- Declaring exactly one occurrence of a child:
  $$\text{<!ELEMENT element-name (child-name)>}$$
  For example
  $$\text{<!ELEMENT poem (verse)>}$$
  declares that a poem element must contain exactly one verse child-element.
- Declaring one or more occurrences of a child:
  $$\text{<!ELEMENT element-name (child-name+)}>$$
  For example
  $$\text{<!ELEMENT poem (verse+)>}$$
  declares that a poem element must contain one or more verse child-elements.
Elements with element content (contd.)

• Declaring zero or more occurrences of a child:
  ```xml
  <!ELEMENT element-name (child-name*)>
  ```
  For example
  ```xml
  <!ELEMENT poem (author*, verse+)>
  ```
declares that a poem element contains zero or more author elements, followed by one or more verse elements.

• Declaring zero or one occurrence of a child:
  ```xml
  <!ELEMENT element-name (child-name?)>
  ```
  For example
  ```xml
  <!ELEMENT poem (author?, verse+)>
  ```
declares that a poem element contains an optional author element, followed by one or more verse elements.
Elements with element content (contd.)

• Parentheses can be used to group a sequence of child-elements and subject them to + * ? quantification.

• For example

  ```
  <!ELEMENT song
  (author?, verse, (chorus, verse)*) >
  ```

deployes that a song element contains an optional author element, followed by a verse element, followed by zero or more instances of a chorus–verse sequence.

• In other words, a song element contains an optional author element, followed by one or more verse elements, the verse elements being separated by chorus elements.
Elements with element content (contd.)

• An element can have alternative children

```xml
<!ELEMENT element-name (option1|option2|option3)>
```

• For example

```xml
<!ELEMENT person (male|female)>
```

declares that a `person` element contains either a `male` child-element or a `female` child-element

• It would be satisfied by either

```xml
<person>
  <male>Bertie Ahern</male>
</person>
```

or

```xml
<person>
  <female>Celia Larkin</female>
</person>
```
Elements with element content (contd.)

- Alternatives can also be subjected to quantification
- For example

```xml
<!ELEMENT family
  (father? mother? (male|female)*) >
```

declares that a `family` element contains an optional `father`, followed by an optional `mother`, followed by zero or more `male` or `female` children
Elements with mixed content

- Example

```xml
<!ELEMENT person (#PCDATA | name) * >
```

declares that a `person` element contains either `#PCDATA` or `name` elements

- It would be satisfied by

  ```xml
  <person>Ned O’Keefe</person>
  or <person><name>Bertie Ahern</name></person>
  or <person><name>Bertie</name>
       <name>Boss</name></person>
  or <person>Bertie<name>Boss</name>
       Taoiseach<name>PM</name></person>
  or <person></person>
```

- Note that `*` is required in mixed content. Why? Dunno.
Elements with arbitrary content

- In
  ```xml
  <!ELEMENT element-name (ANY)>
  ```
  **ANY** means that the element may contain arbitrary content

- Typically, this **ANY** content model is only used at the start of developing a DTD and is replaced as the DTD is fleshed out.
Attribute Declarations

- The attributes for an element are declared in a statement which has the following general format:

  ```xml
  <!ATTLIST elementName attributeDefinition,
    attributeDefinition,
    ...
    attributeDefinition  >
  
  where an attributeDefinition has the following general format

  attributeName  attributeType  attributeDefault
  ```

- An attribute default specifies whether the attribute is required and, if not, how a program processing an XML document should behave if the attribute is absent.
Attribute Declarations (contd.)

• Example:

```xml
<!ATTLIST person name CDATA #IMPLIED
     id    ID       #REQUIRED
     sex   (male|female) "male">
```

which would be satisfied by the following XML start tag

```xml
<person name="Fred" id="p001" sex="male">
```

or by

```xml
<person id="p001">
```
Attribute Declarations (contd.)

• The statement

```xml
<!ATTLIST person name  CDATA         #IMPLIED
     id    ID            #REQUIRED
     sex   (male|female) “male”>
```

used three attribute types:

- **CDATA** denoting a string-valued attribute, which can contain any character apart from `< > & ‘ “`
- **ID** one of a set of types with predefined validity constraints,
- *(male|female)* an enumerated type

• The defaults in the statement specified that
  - the **id** attribute must be provided in a `<person>` tag
  - the **name** attribute is optional
  - if no **sex** attribute is given the value **male** should be assumed
Attribute Declarations (contd.)

• The **ID** type token imposes the following requirements (called **validity constraints**) on an attribute with this type:
  
  – an element can have only one attribute of type **ID**;
  
  – the value of an **ID** attribute must start with a letter, underscore or colon which may be followed by a sequence containing any of these or digits
  
  – the value of an **ID** attribute must uniquely identify the element which bears it -- no other element may have the same value for an **ID** attribute;
  
  – an **ID** attribute must have a default of `#REQUIRED` or `#IMPLIED`

• Apart from **ID**, other types with predefined validity constraints are **IDREF** **IDREFS** **ENTITY** **ENTITIES** **NMTOKEN** and **NMTOKENS** -- their details are beyond our scope here
Attribute Declarations (contd.)

- Attribute defaults are
  - the keyword **#REQUIRED**, which means that an explicit value must be given for the attribute
  - the keyword **#IMPLIED**, which means that the attribute is optional
  - one value from an enumerated attribute type, which means that this value can be assumed if no explicit value is given
  - the keyword **#FIXED** followed by a default value, which means that instances of the attribute must match the default value
Other types of Declarations

• Apart from Element and Attribute declarations, DTDs can contain other types of declarations, but they are beyond our scope here
Example DTD

• The following is a very simple example of a DTD

```xml
<!ELEMENT people (person)+>
<!ELEMENT person (female|male)>
<!ELEMENT male (#PCDATA)>
<!ELEMENT female (#PCDATA)>
```

• It would be satisfied by the following XML document:

```xml
<?xml version="1.0" ?>
<!DOCTYPE people SYSTEM "personnel2.dtd">
<people>
  <person>  <female>Celia Larkin</female> </person>
  <person>  <male>Bertie Ahern</male> </person>
</people>
```
Another Example DTD

• The following is a very simple example of a DTD

```xml
<!ELEMENT people (person)+>
<!ELEMENT person (female|male)>
<!ELEMENT male (#PCDATA)>
<!ATTLIST male age CDATA #REQUIRED>
<!ELEMENT female (#PCDATA)>
```

• It would be satisfied by the following XML document:

```xml
<?xml version="1.0" ?>
<!DOCTYPE people SYSTEM "personnel3.dtd">
<people>
  <person>  <female>Celia Larkin</female> </person>
  <person>  <male age="52">Bertie Ahern</male> </person>
</people>
```
XHTML

• XHTML is one of the many XML-based languages that have been defined

• XHTML is, essentially, a “cleaned-up” version of HTML 4, reformulated using XML DTD technology
  – there are three XHTML DTDs, corresponding to the three versions of HTML 4 (strict, transitional and frameset)

• XHTML is designed to be compatible with XML-oriented user-agents

• XHTML is also acceptable to HTML 4-oriented user agents

• Therefore, Web developers who write their HTML documents to conform to XHTML will give a longer working-life to these documents
XHTML versus HTML

- An XHTML document must be a well-formed XML document and must be valid according to one of the DTDs which define the three varieties of XHTML:
  - the Strict DTD, which should be used when rendering is controlled by CSS
    ```xml
    <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
    ```
  - the Transitional DTD, to be used for browsers that cannot handle CSS
    ```xml
    <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
    ```
  - the Frameset DTD, to be used when frames are used to divide up the browser window:
    ```xml
    <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Frameset//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-frameset.dtd">
    ```
XHTML versus HTML

- Since an XHTML document must be a well-formed XML document and must be valid according to one of the DTDs,
  - an XHTML document must contain one root element
    - (an XML well-formedness requirement)
  - the root element must be delimited by `<html>` and `</html>` tags
    - (a validity requirement, since `html` is defined as the root element in the XHTML DTDs)
  - all XHTML tags and attributes must be in lower-case
    - (a validity requirement, since the XHTML DTDs define the tags and attributes as lower-case and XML is case-sensitive)
XHTML versus HTML (contd.)

- a non-empty element must have start and closing tags, for example, every `<p>` tag must have a corresponding `</p>` tag and every `<li>` tag must have a corresponding `</li>` tag
  
  • (a well-formedness requirement)

- the start tag for an empty element must have a final `/`, for example `<img src="some.jpg" />`
  
  • (a well-formedness requirement)

- elements must be properly nested
  
  • (a well-formedness requirement)

- attribute values must be quoted
  
  • (a well-formedness requirement)
XHTML versus HTML (contd.)

– attributes must have values
  
    • (a well-formedness requirement)
  
    • Ill-formed example:

        `<input type="checkbox" name="day" value="1" checked />`

  
    • Well-formed example:

        `<input type="checkbox" name="day" value="1" checked="checked" />`
XHTML versus HTML (contd.)

• Since style-sheets and scripts are not XML, they must be escaped by placing them inside the special CDATA tags which XML provides for escaping non-XML text

• Example style element

```html
<style>
<![CDATA[
    body {background-color: white; color: red}
    h1 {background-color: orange; color: blue}
]]>
</style>
```

• Example script element

```html
<script language="JavaScript" type="text/javascript">
<![CDATA[
    alert("Check-out the specimen exam paper");
]]>
</script>
```
XHTML versus HTML (contd.)

- Use the `id` attribute instead of the `name` attribute
  - although the `name` attribute is still supported in XHTML 1.0, it is expected to be eliminated in future DTDs

- One advantage of adopting XHTML is that you can validate your documents, instead hoping that users who find them on the web will be able to view them. So use one of the following document type declarations

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

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

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

- Meta Language
- Language
- Usage of the Language

SGML

defines

XML

simplifies

HTML

Web pages

XHTML

XML Definitions

XML Documents

Meta Data

Data
HTML5

• So what is HTML5, really? Basically, it’s about extending HTML/XHTML with new more semantically rich elements, deprecating attributes, introducing new attributes and altering how some element and attributes are allowed to be used.
HTML5 APIs

- 2D drawing API with the canvas element
- API for playing of video and audio
- API that enables offline Web applications.
- API that allows a Web application to register itself for certain protocols or media types.
- Editing API in combination with a new global contenteditable attribute.
- Drag & drop API in combination with a draggable attribute.
- API that exposes the history and allows pages to add to it to prevent breaking the back button.
- Cross-document messaging with postMessage.
How?

One of the main reasons behind the adoption of HTML5 is that it sets out to be backwards compatible and work with the web browsers there already are in the market. This is done through new elements which, generally, have no particular look or behavior attached to them, but rather offering more semantic richness and then up to you style them via CSS according to your liking.
New Input Element Types

- tel
- search
- url
- email
- datetime
- date
- month
- week
- time
- datetime-local
- number
- range
- color