


# <oo>→<dh> Digital humanities

Maintained by: David J. Birnbaum ([djbpitt@gmail.com](mailto:djbpitt@gmail.com)) 

Last modified: 2015-08-28T15:26:51+0000

## What is XML and why should humanists care? An even gentler introduction to XML

### Introduction

Most introductions to XML seek to explain when one should use XML instead of HTML (generic, standard web pages), an approach that is most easily understood by those who already have some experience with HTML authoring. Other resources intended to introduce XML to novices, like the Text Encoding Initiative (TEI) [Gentle introduction to XML](#), go beyond merely describing why humanities scholars might wish to use XML and what an XML document looks like, aiming also both to situate XML in a general context of markup languages and to introduce the syntax of the formal schema languages used to model them. Digital humanities scholars will need this information eventually, and perhaps even fairly early in their training, but including it in the very first introduction that complete novices will read entails the risk of inadvertently confusing or intimidating the new learner.

The present even gentler introduction aims to describe briefly some of the reasons humanities scholars use XML, and to do so in a way that will be accessible to absolute novice readers, who may have no idea what a web page looks like beneath the surface. It also describes the most basic features of XML documents, but it defers details about formal modeling and schema languages for another occasion. It ranges broadly, and novice readers will find new terms (italicized when first introduced) and concepts in almost every paragraph, but I have tried to provide definitions, descriptions, and examples that will make the unfamiliar accessible even to those with no background at all in informatics. It is not necessary on a first reading to take notes or study this introduction; it is intended to begin to make new terms and concepts familiar, and some parts may be clearer than others on first acquaintance.

The title of this essay was stolen from C. M. Sperberg-McQueen (<http://cmsmcq.com/1997/drh97.html>), a pioneer and leader in the world of digital humanities, and the subtitle from the TEI [Gentle introduction](#), where many digital humanists of my generation began their own learning.

### The context

*Digital humanities* (and, in particular, the subfield of DH traditionally called *humanities computing*) is the use of computer technology to conduct primary humanities research. This introduction has been written for an undergraduate course entitled “[Computational methods in the humanities](#),” a title chosen to emphasize the students’ creation of original programming for use in their own humanities research projects. This focus ignores other uses of computers in humanities scholarship, such as the creation of web pages, blogs, wikis, social networks, etc., not because those uses are unimportant, but because publishing something on the screen does not automatically move the researcher very far beyond what would be possible with paper publication, and because collaborative or social authoring on the Internet differs from other forms of collaboration more in ease of use than in the significance of the eventual research results. This course concentrates instead on three things that fundamentally distinguish digital humanities from non-digital humanities scholarship:

1. The creation of electronic texts that can be used in primary humanities research that would be impossible (or so impractical as to be essentially impossible) to conduct without computational assistance.
2. The use of existing computational tools to interrogate those texts and obtain original research results that would not be attainable without the tools.
3. Where existing computational tools are not able to meet the research goals, the development of original computer systems and programs to meet the researcher’s needs.

For reasons described below, most digital humanities projects nowadays (this document was first written in 2011) use XML as their document format. I describe below what an XML document looks like and why it has achieved the popularity it has as a way of modeling some of the inherent structural and semantic properties of the cultural documents used in humanities research. This is not the only use of XML (which is often employed to model other types of documents for other purposes, some of which have nothing to do with humanities scholarship), but it is the best place to start learning about why XML has come to play a uniquely important role in digital humanities scholarship.

## Document and their structures

We often think of the documents we read as consisting of linguistic text that conveys meaning, but much of that meaning is expressed by the *structure* of the text, and not just by its linguistic content. For example, a book may be divided into chapters (coordinated with a table of contents in the front). The chapters, in turn, may be divided into paragraphs and may be preceded by chapter titles, and the text within the paragraphs may have footnotes, bibliographic references (coordinated with a bibliographic list in the back), etc. If a book were merely a stream of words, with none of the layout and formatting that lets us recognize its constituent parts, the reader would have much more difficulty determining where logical sequences of thoughts (instantiated as chapters or paragraphs, described by titles, etc.) begin and end. For that matter, although we don't usually think of modern writing this way, the spaces between words are a fairly recent invention in human writing systems, they often do not correspond to pauses in spoken language, and they therefore serve as visual clues about the structure of the text as containing a stream of words, rather than just a stream of letters.

Chapters with titles and paragraphs are not the only type of organization found in documents. A poem might be divided not into chapters with paragraphs, but into cantos, stanzas, and lines. A play might be divided into acts, scenes, speeches (accompanied by the name of the speaker), and stage directions. A dictionary might be divided into entries, with headwords, pronunciations, definitions, examples (associated with specific definitions), etymologies, etc. If all of the words in any of these documents were simply run together in a continuous stream, the reader would have enormous, and perhaps even insurmountable, difficulty recognizing the structure—and the structure in these cases is part of the meaning. The name “Hamlet” means something different when spoken by a character in the play than it does when it serves as a label in an actor's script, where it is intended not to be pronounced during a performance, but to identify the speaker of a particular line. A word in a dictionary has a different function when it is a headword, part of a definition, or part of an example, and the specific function of each word must be recognized if the dictionary is to serve its intended purpose as a reference resource.

The structures described in the preceding paragraph can all be considered ordered and hierarchical. That is, a single book may contain multiple chapters, in a particular order, where each chapter may contain one title followed by multiple paragraphs, also in a particular order. The chapter title does not normally fall between, say, the third and fourth paragraph, and paragraphs do not normally contain chapters or books.

XML is a formal model that is based on an ordered hierarchy, or, in technical informatic terms, a *tree*. It consists of a *root* (which contains everything else), the components under the root, which contain their own subcomponents, etc. These components and subcomponents are called *nodes*. In the case of our book example, the root node is the book, it might contain a title-page node, followed by a table-of-contents node (which contains nodes for each content item listed, in order, and each of those, in turn, might contain two nodes, one for the table-of-contents entry and the other for the page reference). The table-of-contents node might be followed by chapter nodes, in order, and each chapter node might contain a chapter-title node followed by paragraph nodes, once again in a particular order. Nodes may also contain text, and not just other nodes.

Humanities scholars represent their documents in XML for two reasons:

1. XML is a formal model designed to represent an ordered hierarchy, and to the extent that human documents are logically ordered and hierarchical, they can be formalized and represented easily as XML documents.
2. Computers can operate quickly and efficiently on trees (ordered hierarchies), much more quickly and efficiently than they can on non-hierarchical text. This means that if we can model the documents we need to study as trees, we can manage and manipulate large amounts of data efficiently.

## The beginning of every digital humanities project: Document analysis

The hierarchy we see in our document depends on our interests, and the same document might be amenable to more than one hierarchical analysis. A book in prose can be considered from an informational perspective as chapters containing paragraphs, but from a physical perspective it is pages that contain lines. These hierarchies may be entirely or partially independent of each other, so that a paragraph may begin on one page and end on the next, a page may contain an entire paragraph, and a paragraph may contain an entire page. Whether we consider the informational or the physical hierarchy in the present example our primary object of study may depend as much on our research agenda as it does on the text itself. Furthermore, once we have identified the hierarchy we wish to study, we do not need to examine every fine-grained detail with fractal precision, and we can concentrate instead on just the features that we actually care about for our specific purposes. For example, bibliographic entries can be richly structured: they tend to consist of authors, titles, publishers, years of publication, etc. If we need to access these subcomponents quickly, it makes sense to encode the hierarchical structure of an individual bibliographic entry. If all we intend to do with our bibliographic entries is read them, we can treat each of them like a little paragraph, ignoring the internal structure.

Every digital humanities project begins with *document analysis*, where we determine those aspects of the inherent hierarchical structure(s) of our documents that we wish to use for research. If we are able to identify a useful hierarchical structure, the next step is to *mark up* or *encode* our document to make that hierarchy accessible to a computer. If we simply pass the computer an indifferentiated stream of text, it will not be able to identify the beginning and end of the various structural subcomponents. Markup is the process of inserting information into our document (this is called *tagging* in the XML context) that will take the structure that humans recognize easily and make it accessible to a computer.

## Types of markup

The markup used in digital humanities projects is *descriptive*, which means that it describes what a textual subcomponent *is*. Descriptive markup differs from *presentational* markup, which describes what text *looks like*. For example, presentational markup might say that a sequence of words is rendered in italics, without any explanation of whether that's because they're a book title, a foreign phrase, something intended to be emphasized, etc. Descriptive markup also differs from *procedural* markup, which describes what to do with text (e.g., an instruction to a word processor to switch fonts in a particular place). The rationale for privileging descriptive markup over presentational and procedural markup in digital humanities scholarship is that descriptive markup can be transformed into presentational or procedural markup when needed for a particular purpose, but the reverse is not the case. For example, if foreign words and book titles are both marked up only as italic, it is not possible for a computer to tell whether a particular italic moment is foreign, a title, or both. If they are marked up differently, though, according to what they are rather than how they look, they can all be rendered the same way for reading purposes without losing an internal record of their difference that can be used by the computer for other purposes. The concern for encoding distinctions that may be needed for some purposes but not others is based on the idea of *multipurposing*: the user should be able to create and mark up a text only once and then use it for multiple purposes. Descriptive markup facilitates multipurposing in a way that is not the case with presentational or procedural markup.

## Is every document really a hierarchy?

Would that it were that simple! Many human documents can be conceptualized (and therefore marked up in XML, in a way that will be explained below) as hierarchies, but upon close examination many turn out to have multiple simultaneous hierarchies that are of interest to the humanities scholar. The example above, contrasting the informational structure of a book (chapters, paragraphs) with the physical structure (pages, lines) is one such case. XML does not cope as well with multiple simultaneous hierarchies as it does with single hierarchies. Other modeling languages are being explored as ways of representing multiple (or overlapping) hierarchies for use in digital humanities scholarship, but the discipline has not yet arrived at a consensus about which language or approach to use, and the ease with which computers can process XML trees has made scholars reluctant to surrender those practical advantages. As a result, most digital humanities projects today continue to rely on XML, utilizing various strategies for coping with the overlapping hierarchy problem. Curious readers might wish to look at the seminal article about this issue, "[Refining our notion of what text really is: the problem of overlapping hierarchies](#)" (1993). Some of the authors of that article were responsible for first propounding the idea that text was logically an ordered hierarchy of content objects (OHCO); in their 1993 report they return to the question and conclude that their earlier understanding was overly simplistic.

## Could I please see some XML already?

Here is a sample XML document:

```
<shopping_list>
  <item>bread</item>
  <item>milk</item>
</shopping_list>
```

XML models the hierarchy of a document by using *elements*, such as **shopping\_list** and **item**. An element consists of a *start tag*, an *end tag*, and *content*, which is whatever occurs between the tags. A start tag is delimited by angle brackets and an end tag looks like a start tag except that it has a forward slash after the opening angle bracket. The name of the element (e.g., **shopping\_list**, **item**) is technically called a *generic identifier* (or *gi*). Don't confuse elements with tags; an element consists of the start tag, the end tag, and everything in between, which may be other elements, plain text, or a mixture of the two (see below). The tags are actually written into the XML document during editing, and XML-aware software is then capable of using that information when needed and suppressing it when it is unwanted. For example, when XML documents are presented to end users, the end users typically don't see the angle brackets. What the users might see instead, though, is that the shopping list above has been numbered automatically because the system that renders the XML for human consumption knows to insert the numbers while hiding the angle brackets and the names of the elements.

How the XML is transformed before being shown to the user is entirely under the control of the developer, and it is this control that enables the developer to encode a document solely according to what it means, without regard to how it should be styled when it is eventually presented to the reader. Many advantages of this separation of meaning and presentation are obvious. For example, if you write numbers into an ordered list and then insert, delete, or move an item while editing, you have to change the numbers, which is not only tedious and fragile, but also unnecessary. The numbering in an ordered list, after all, is actually just a rendering of the fact that the items occur in a logical order, and since that order can be determined by counting them, there is no need to enter the numbers yourself just to get them to appear in the output. The separation of content from presentation underlying XML means that by putting the items in order you've already made it possible for the computer to count them and insert the numbers during rendering. That the list is a sequence of items is the meaning, but the fact that that sequences may be numbered or lettered or bulleted or separated by commas in running text or otherwise formatted is a presentational decision that doesn't change the meaning: a list is still a list, no matter how it is presented.

In XML the person who creates the document decides what to call the elements and where to use them. In the example above, the document consists of a **shopping\_list** element, which contains two **item** elements. This models our understanding of what a shopping list really is: it's a list of items to buy. This isn't the only way to model a shopping list, though. We might, for example, organize our own lists according to sections of the market, in which case we would have another level of structure between the root **shopping\_list** element and the individual **item** elements. How we choose to model our shopping list is part of document analysis; we determine the hierarchy that interests us and introduce markup into our document to make that hierarchy accessible to a computer.

## Types of element content

XML elements may have four types of content:

1. *Element content*. An element may contain only other elements. For example, a **shopping\_list** element may contain nothing other than **item** elements.
2. *Text content*. An element may contain only plain text. For example, an **item** element might contain just the name of the item, with no other elements.
3. *Mixed content*. An element may contain a mixture of plain text and other elements. This is common in digital humanities projects. For example, a paragraph may contain mostly plain text, but the researcher may have tagged certain words or phrases because they are structurally or semantically important. For example, a place name may be tagged as a **place** element, along the lines of:

```
<paragraph>The American writer Jack London never
lived in <place>London</place>.</paragraph>
```

An author might wish to create an index of place names for the document, or cause a map to appear when the reader mouses over a place name while reading, or make it possible to search for the string “London” when it refers to the place, but not when it refers to the writer. Marking up place names differently from personal names facilitates all of these actions.

4. An element may have no content, in which case it is an *empty element*. Empty elements may be used to mark moments in a document that have no associated text. For example, a reader might want to insert a bookmark into a document to indicate a particular point, or *milestone*, that is not associated with any span of text.

## Attributes

Elements may also incorporate *attributes*, which are additional markup that provides supplementary information about an element. For example, instead of just tagging a foreign word as foreign, one might wish to distinguish French words in an English text from German words in the same text. One could create **French** and **German** elements, but then there would be no easy way to formalize the fact that both elements would be used to identify foreign words. Attributes make it possible to say something like “the textual content of this element is foreign, but it is also specifically French, which means it is similar to German text insofar as both are marked up as foreign, but different because it uses a different foreign language.”

Attributes are written inside the start tag of an element (but not inside the end tag), and they consist of an attribute *name* and an attribute *value*. The attribute value must be enclosed in either double straight quotation marks (") or single straight apostrophes ('). Double quotation marks and single apostrophes function identically, but the marks on either side of the attribute value must agree. For example, a French word might be marked up as **<foreign language="french">oui</foreign>** and a German word as **<foreign language="german">ja</foreign>**. This markup makes it possible to treat the two similarly (based on their shared generic identifier) or differently (based on their different values of the **language** attribute).

## The XML tree and its serialization

Marking up a document in XML consists of performing document analysis (determining the structural hierarchy and the semantics) and inserting tags (markup) into the text. XML may look like a sequence of characters, some of which are data content and some of which are markup, but structurally it is a tree, and the stream of characters, with markup and data content mixed together, is just the way it is presented to humans for reading. This representation of the XML tree (what it really is) as a stream of characters is called a *serialization*.

The distinction between the tree and the serialization is important because XML trees have certain properties that must be maintained in the serialization. These are

- The XML tree has a single root, that is, a single element that contains all other elements.
- All elements must be properly nested. This means that if you open element X and inside that you open element Y, you must close element Y before you close element X. That is, element Y must nest entirely within element X. It cannot start inside but close outside or start outside and close inside.

XML documents that meet the preceding two requirements are called *well formed*, and all XML documents must be well formed. (There are a few additional small and self-explanatory well-formedness requirements, such as wrapping attribute values in apostrophes or quotation marks, and these will be mentioned in the course as they arise.) The following is **not well formed** and therefore it is not an XML document, even though it consists of a mixture of text and markup:

```
<dairy>
  <item>milk</item>
  <item>cheese</item>
</dairy>
<snacks>
  <item>potato chips</item>
  <item>peanuts</item>
</snacks>
```

The preceding is not well formed because it doesn't have a single root element that contains everything else. To change it into well-formed XML, wrap the **dairy** and **snack** elements in a root element, such as



**shopping\_list**. The following also is **not well formed**:

```
<paragraph>He responded emphatically in French:
<emph><foreign language="french">oui</foreign>!</paragraph>
```

This example has a single root element (**paragraph**), but the **emph** and **foreign** elements inside the **paragraph** element are not properly nested. The word *oui* is both emphatic (marked with the **emph** element) and French (tagged as **foreign**, with the attribute **language** used to specify the exact foreign language), and it might appear as if the internal markup begins by identifying it as emphatic and foreign, then gives the word, and then indicates that whatever follows is no longer emphatic or foreign. That appearance is deceiving; the problem is that *tags are not toggle switches*, which is to say that if you open the **emph** element and then the **foreign** element, you must then close the **foreign** element before you close the **emph** element, so that the **foreign** element will be entirely nested (start tag, contents, and end tag) inside the **emph** element. In the erroneous example above, the elements are not nested because **emph** does not contain **foreign** and **foreign** does not contain **emph**; instead, the tags cross.

If XML were a stream of characters, the order in which one turns on and off the **emph** or **foreign** properties might not matter. The problem is that XML only looks like a stream of characters because that's how humans read text. As was noted above, that stream of characters is just a *serialization* of what is logically and structurally a hierarchy. The relative order of the end tags may not matter to the way humans understand the meaning, but they matter to an XML processor. If your elements are not properly nested, what you have isn't XML.

Every new XML user makes this mistake, and for many the concept is difficult to grasp, so it bears repeating:

1. **XML is a hierarchical tree. It is not a stream of characters.** It may look like a stream of characters, but that's just the way it is presented to the human user. Internally it is a hierarchy of elements (not tags), which means that each element (start tag, contents, and end tag) must nest fully inside other elements. The only exception is the root, which, as the outmost element that contains the entire rest of the document, logically cannot nest within any other element.
2. **Tags are not toggle switches that turn properties on and off.** Tags are the way elements are serialized by inserting angle brackets and other text into data content, but XML is really a hierarchy of elements and the tags are just a way of representing that hierarchy in a linear (serial) fashion. When you open an element, you need to close it, and when you open it in a particular context (for example, inside another element), you need to close it before you can leave that context.
3. **When you're writing XML, insert the whole element, with both the start and end tags, at the same time and only then back up and fill in the content.** This will ensure that you think in terms of nesting elements inside other elements, and not in terms of tags. When you are adding markup to existing plain-text documents to convert them to XML, don't just drop a start tag in one place and an end tag in another. If you use an XML editor (like the *<oxygen/>* editor we will use in this course), it lets you select the text you want to mark up and wrap it in start and end tags, inserting both simultaneously. This will help you think of the element hierarchy instead of thinking about the start and end tags separately.

## Valid and well-formed documents

As was noted above, all XML documents *must* be *well formed*, which means that they must have a single root element that contains all other elements, and all elements must be properly nested, with no overlapping tags. If it isn't well formed, it isn't XML.

Additionally, XML *may* be *valid*. This is a technical term that means that the document uses only certain elements, and that it uses them only in certain contexts. For example, a dictionary entry might contain a head word, a pronunciation, and a series of meanings, each with examples. If you want those component elements always to occur in the same order, and you want to ensure that, for example, a single head word can be associated with multiple definitions, each of which can have multiple illustrative examples, you can write a set of rules determining where the various elements can and cannot occur and which ones can or cannot be repeated. This set of rules, or *document grammar*, is called a *schema*, and XML provides several *schema languages* that are capable of formalizing a document grammar. An XML document that is well formed can be *validated* against a schema to determine whether it follows the rules, and this is commonly done during authoring and markup to ensure that one has used the elements and attributes in a consistent way. It is possible for a document to be well

formed but invalid. It is not possible for a document to be valid but not well formed, since if it isn't well formed, it isn't an XML document, and therefore cannot logically be a valid XML document.

All XML documents must be well formed, but whether they need to be valid (and need to be validated) depends on how you plan to use them. In practice, almost all digital humanities projects validate documents against a schema designed with the project goals in mind. In this course you will learn how to write a document grammar in a schema language and validate a document against it.

Some XML that is not well formed may nonetheless be *well balanced*. A well-balanced document fragment is one that may lack a single root node, but is otherwise well formed, that is, that does not have overlapping or missing tags. All well-formed documents are also well balanced (since not having overlapping or missing tags is one of the two requirements for well-formedness and the only requirement for being well balanced), but the reverse is not necessarily true. An XML *document* cannot be merely well balanced; if it isn't well formed, it isn't an XML document. An XML *document fragment*, though, may be merely well balanced, and there are stages in a digital humanities project where you may need to create well-balanced XML fragments that you will insert into an XML document, producing a well-formed (and perhaps also valid) XML document at the end. We'll discuss the use of well-balanced XML fragments later in the course.

## Entities and numerical character references

An XML document uses angle brackets to delimit markup. For this reason, XML cannot contain an angle bracket that is meant to represent a textual character, since XML software would be unable to distinguish this textual data from markup. XML reserves two characters that cannot be represented directly in text: the left angle bracket or less-than sign (" $<$ ") and the ampersand (" $&$ "). When these characters occur in text that is to be represented in XML, they must be replaced in the underlying marked-up document by *entities*. Entities begin with an ampersand and end with a semicolon; the part in between identifies the meaning of the entity. If you need to include these two characters as textual data characters in your XML, you can represent the left angle bracket as `&lt;` (the letters stand for "less than") and the ampersand as `&amp;`. For example, to write " $1 < 2$ " in an XML document you can write `1 &lt; 2`. If you write `1 < 2` literally, an XML application will be unable to parse your document, that is, unable to read it and understand its hierarchical structure.

These alternative representations are called *character entities* and they are essentially the only way to represent the characters in question in XML. XML also recognizes three other character entities: `&gt;` (for " $>$ "), `&quot;` (for a double straight quotation mark, " $"$ "), and `&apos;` (for a straight apostrophe, " $'$ "). These three (unlike the first two) can usually also be entered as plain text, but there are situations where their use is restricted, and the additional entities are designed to cope with those situations. For what it's worth, I usually use an entity for the right angle bracket and plain text for the quotation mark and apostrophe. When you undertake a digital humanities project using a text that was created outside of an XML environment, and that therefore may contain raw left angle bracket and ampersand characters, one of the first things to do is convert those to entities using a global search-and-replace operation.

*Numerical character references* look somewhat like entities in that they begin with an ampersand and end with a semicolon, but the opening ampersand must be followed by a hash mark (" $\#$ ") and then a number (e.g., `&#xa0;`). The hash mark in the preceding example indicates that we're dealing with a numerical character reference, and not a character entity. The letter **x** tells us that the number is hexadecimal (base sixteen; if you don't know what this means, don't worry about it for now). A regular decimal (base ten) number could have been used instead; hexadecimal `&#xa0;` and decimal `&#160;` mean the same thing, i.e., represent the same character value. The meaning of specific numbers (that is, the characters they represent) is based on Unicode, about which see below.

Numerical character references are particularly useful for representing characters that are difficult to display. For example, a web browser is able to distinguish a plain space from a non-breaking space, and it knows not to wrap to a new line between two words that are separated by only a non-breaking space. This is handy when you need to ensure that certain words will be kept together even when the user resizes the browser window. The regular and the non-breaking space differ in their behavior, but not their appearance, which means that it would be impossible to distinguish between them visually if all you could do was type the raw character. On the other hand, it is easy to distinguish a plain space character (regular space) from the string `&#xa0;`, which is the numerical character reference corresponding to the non-breaking space character. Any character may be represented by a numerical character reference, but normal practice is to write raw characters (normal letters, numbers, etc.) unless there is a good reason not to (as with different types of spaces, which look alike to the human eye).

Since whether a character is represented as a raw character or a numerical character reference has no effect on the meaning (the representations are exactly equivalent informationally), that choice is part of the serialization that the human reads, rather than the internal representation that the XML processor sees. This means that when you write a program to read your XML and convert it to different XML, it may or may not replace raw characters with numerical character references or vice versa. There are limited ways to control this aspect of the serialization while processing XML, but normally you shouldn't worry about it, since it has no effect on meaning, and is not considered informational in an XML context.

## The life cycle of an XML document

When you undertake a digital humanities project, you'll work with one or more documents. Those may already exist as XML documents (e.g., you might use XML that someone else created for a different purpose), or they may already exist as documents in other forms. (In that latter case, you'll need to convert them to XML, which typically involves a mixture of *auto-tagging*, where you run some global search-and-replace operations to insert markup, and manual tagging.) Or you may create new documents entirely from scratch, where you are creating not just the markup, but also the data content. Whatever the source of your data:

1. The first stage of a digital humanities project is document analysis, where you determine the hierarchical structure of the documents that you wish to model in XML.
2. You then typically begin to design a schema, a formal grammar describing where the various elements and attributes you want to use can and cannot be employed.
3. Once you have a draft schema, you begin to mark up your document according to the schema and validate it, that is, verify that you have used markup only in a way that conforms to the schema.

The preceding three steps constitute an iterative process. It is common during the markup phase to notice that the document analysis or the schema needs to be refined and modified, and once you've changed the schema, you often need to adjust the markup accordingly. In an ideal world, you would complete each of the three steps before beginning the next, so that you wouldn't have to redo or undo previous work, but in practice, no matter how scrupulous the data analysis, there are almost inevitably complications that become evident only during the markup phase. Nonetheless, because a lot of structural and semantic information inherent in your documents will easily be discernible during document analysis, and getting your schema started before beginning the markup leads to better markup, it is never a good idea to skip the document-analysis and schema-development stages completely and begin with marking up text.

Once your documents have been encoded, you'll write programs in XML-oriented languages to transform them, publish them (often in very different views, with different selections and arrangements of the same underlying data), analyze them, and generate reports. Because what you choose to model in your schema and mark up in your documents is dictated by your research agenda, it is important to conduct your document analysis and develop your schema with your goals in mind. For example, if you know that you want to study issues of gender in Shakespeare's plays, you need to think at the beginning of how your programs will identify gender-related features, and those considerations typically inform your schema design and markup. Don't try to mark up a text until you've given some thought to what you want to do with it, that is, to why you want to mark it up in the first place.

## The XML family of standards

XML is part of a family of standards that cooperate to support the study and processing of texts. We'll learn about all of these standards in this course, and we'll use some of them in our own work. Don't worry about memorizing them now because they'll become familiar over the semester, but so that you'll have all of the names in one place, they include:

### Schema languages

Languages used for the formal modeling of document structure. The schema languages in use in the XML world are Document Type Definitions (DTDs), Relax NG, and W3C Schema. In this course we use Relax NG.

### XSLT

eXtensible Style Sheet Language Transformations, a programming language used to transform XML to other forms (other XML, HTML [web pages], plain text, etc.). The term *style sheet* is somewhat misleading; XSLT can be used to style an XML document, but it can also be used to create entirely new documents by transforming existing ones in almost unlimited ways.



## XQuery

A language used to query XML databases.

## XPath

A formal method of navigating the XML hierarchy, used by XSLT and XQuery. For example, if you want to generate a table of contents based on chapter titles, you use XPath to find (navigate to) all the chapter titles in your document and XSLT to generate an output document that contains the newly constructed table of contents.

## XSL-FO

eXtensible Stylesheet Language Formatting Objects, used with XSLT to transform XML into Portable Document Format (PDF, the type of document read in Adobe Reader).

## SVG

Scalable Vector Graphics. An XML vocabulary (schema) for describing graphics. Useful in digital humanities projects for creating graphic representations of textual data. For example, one could use XSLT to transform a Shakespearean play into a bar graph illustrating how much each character speaks.

## Namespaces

A technology used to manage different XML vocabularies in the same project. For example, if your project includes a schema with elements for bibliography, which might include a **title** element for book titles, together with a schema for representing persons, which might also include a **title** element, this time for royalty, namespaces allow you to call them both **title** in your document while retaining the ability to distinguish them.

## Schematron

A constraint modeling language used to restrict what is permitted in a document in ways that schema languages alone cannot address.

## XProc

A pipelining language, intended to allow the user to transform one XML document into another in stages, feeding (“piping”) the output of one transformation into the next as input.

## Regular expressions

A symbolic system for representing text with wildcards or in other flexible ways. Regular expression processing might be a part of a routine to convert a date in “4/1/2011” (month/date/year) format to something like “April 1, 2011.” A regular expression would be used to break the string of digits and slashes into the three subcomponents of the date, and they could then be stitched back together differently, looking up the name of the month in a table and inserting punctuation and spaces where needed.

## XForms

An XML-oriented replacement for the forms used on the Web on commercial sites and elsewhere.

## Other web standards

XML can be used without reference to the Web or the Internet. For example, one could write XSLT to generate a table of information from an XML document and copy and insert it into a Microsoft Word document for printing on paper—all without ever being connected to the Internet. Nonetheless, XML is most commonly used in a Web-aware environment, where it interacts with the following standards:

### HTML

Hypertext Markup Language is a set of XML schemas used to describe the structure of web pages. There are several versions and variants of HTML; in this course we will most commonly be transforming our XML into HTML5 with XML syntax for publication on the Web.

### CSS

Cascading Style Sheets are a strategy for describing how XML (and HTML) should be rendered. Descriptive markup describes what the elements in a document mean, but not how they look, and CSS is intended to let the

designer specify the rendering separately from the XML, so that meaning and appearance do not become conflated or confused.

## JavaScript

JavaScript is a *client-side* programming language for manipulating, among other things, the appearance of web pages in the browser. Client-side means that JavaScript runs in the user's browser, so that, for example, the user can change what is rendered in the browser window without having to fetch new information from the server (the place from which the original page was downloaded). JavaScript is frequently used to modify CSS in response to user activities or *events*, such as mouse clicks or movements or keyboard presses, so that the document remains the same inside the browser, but the way it is rendered on the screen is updated in response to events. The use of JavaScript to modify CSS on the client side is called *dynamic HTML* (DHTML). One common use of JavaScript involves the collapsing menus that proliferate on the web, where different topics expand to show subtopics and then contract to hide them in response to mouse clicks. The different views of the menu are not downloaded individually from the server. The entire menu is always present on the user's machine, but certain parts are shown or hidden by a JavaScript program that runs in the background and listens for mouse clicks. Don't confuse JavaScript with Java; despite the similarity in the names, they are completely different programming languages.

## PHP

PHP is a *server-side* scripting language, which means that it can be used to construct the web page that it sends the user after taking into consideration user-supplied or other information. PHP is commonly used for login validation; the user supplies a username and password and the system decides what to return after authenticating the user in a database. For example, the server might return a page content customized for different people according to the way their roles in an organization are stored in the database. Client-side JavaScript wouldn't work for this purpose because it would be a security violation to download the entire password database to the user's machine and authenticate the user there! On the other hand, because the transfer of information between a server and client can take a long time (in computer terms), tasks normally should be performed on the client unless there is some reason they require the involvement of the server.

## Character sets and Unicode

What the user sees on the screen as letters of a particular alphabet or numbers or punctuation marks or other symbols represent *characters*, the abstract units of information used to represent, among other things, written language. While humans regard letters and numbers and punctuation as fundamentally different from one another, all characters are represented inside the computer as numerical values, and those values are associated not only with how a character looks, but also with what it means. For example, the Latin-alphabet small letter "a" has the internal numerical representation of decimal 97. The Cyrillic small letter "а" looks exactly the same, but it has the internal numerical representation of decimal 1072. If all we did with text was read it and print it the difference wouldn't matter, since we can't see a difference just by looking, but when we search for English "a" we don't want also to retrieve Russian "а".

Unicode is the system all modern computers and computer software use to represent characters. You don't usually need to think about it, and when you do, you can look up the values conveniently on the [Unicode Consortium web site](http://www.unicode.org/).

## Practicing what we preach

This document was originally authored in XHTML 1.1, which is an XML schema that determines which elements may be used and how they may be used to create pages that web browsers will be able to render in conformity with user expectations. Because this page is an XML document, it is possible to use XSLT to transform it into a different XML document. As an exercise to demonstrate the use of XSLT to transform XML to different XML, before I inserted the table below into this page, I ran the page through an XSLT program, which created the table. That is, I didn't type anything myself to write the table; instead, I wrote an XSLT program to generate it automatically from this page of prose.

To do that, in the document I had already marked up new terms with the XHTML **dfn** element, which is intended to identify terms that are being defined. How did I know that this was the best element to use for definition terms? Because XHTML is someone else's schema, the elements and their use have been predefined, and I had to read the XHTML documentation (at, for example, [SitePoint](http://www.sitepoint.com/), an excellent general resource for web-related

information). When you create your own schema, you determine the elements and the rules for their use, but when you use someone else's, you need to read their documentation.

By default, text that is tagged as **dfn** is typically rendered in the browser in an oblique (slanted) or italic font, and the latter styling is also the default for text tagged as emphatic (**em**) or a citation (**cite**). In practice, of the five browsers I tested (Firefox, Internet Explorer, Chrome, Opera, and Safari, all on Windows 7), the first three used italic (not oblique) styling for **dfn** elements and the other two did not style the text tagged as **dfn** any differently from the surrounding plain text. Because XHTML is primarily about structure and semantics, rather than about presentation, browsers are free to style elements as they please, although they normally employ a common set of conventions, which users have come to expect. For example, in all major browsers clickable links are colored blue and underlined, and the color changes to purple once the user has clicked on a link and visited the site. Page authors may overwrite the default styles using CSS; I do that on this page by leaving the default coloring for links in place but removing the underlining, which I find distracting. On this page I also use CSS to ensure that **dfn** elements will be italicized in all browsers, including Chrome and Safari, where by default no styling at all is applied to this element.

Because I marked up my terms according to what they are, instead of according to how they appear, my document rigorously distinguishes terms from other elements that might also be rendered in italics. For example, XHTML includes the **cite** element, intended for cited references (such as book titles), the **em** element, intended for emphasized text, and the **i** element, intended for italic text of other types. (The **i** element is a left-over from an earlier era, when HTML markup was more presentational. Although it is still valid in XHTML, many modern developers avoid it precisely because it conveys no structural or semantic information, and merely describes how to render the text visually.) Had I simply encoded all information that I wanted to render in italics with the same markup, I would not have been able to distinguish among definition terms, cited references, emphasis, and generic italic text programmatically. But because definition terms and only definition terms are marked up as **dfn** elements in my document, my XSLT program can find them easily.

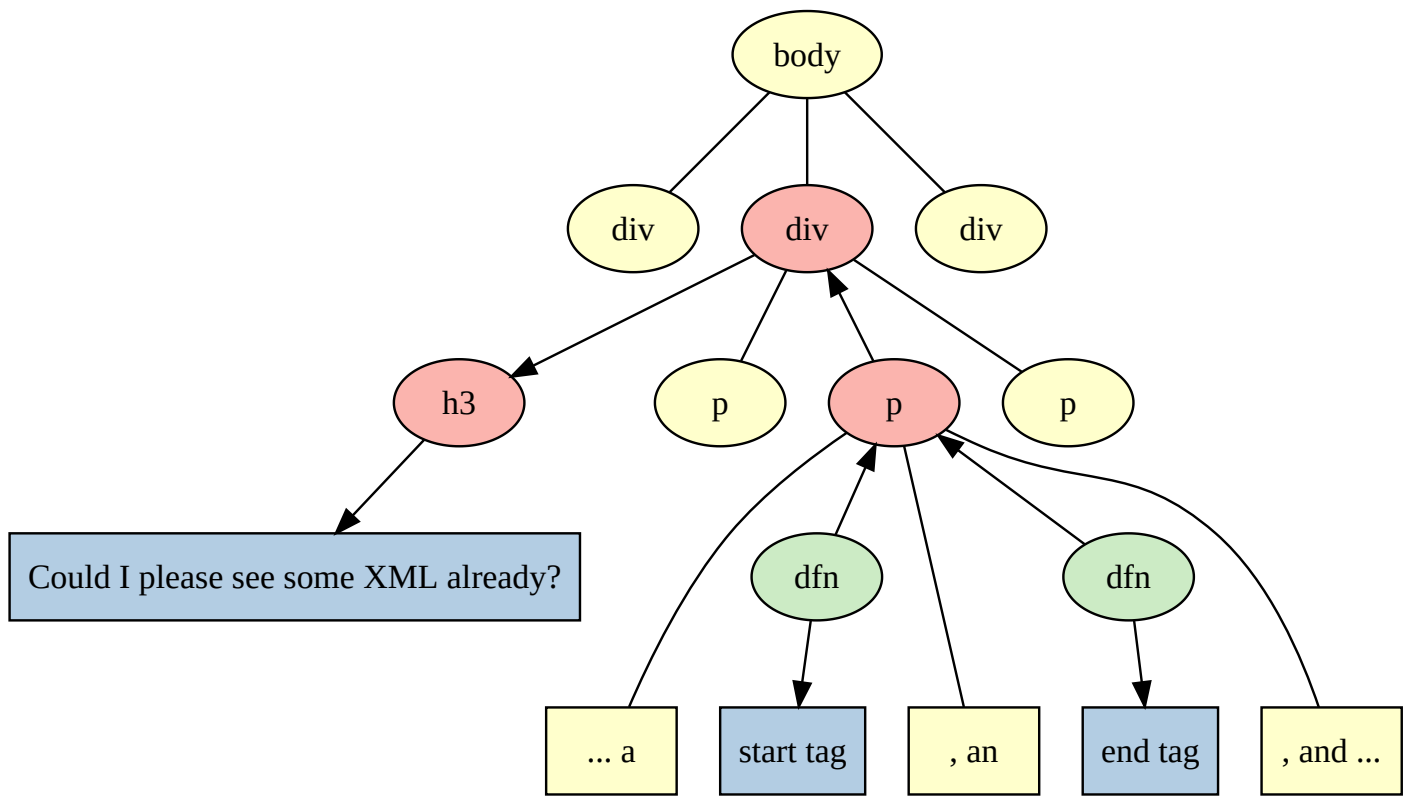
To demonstrate the use of XSLT to transform XML, I wrote a *script* (another term for a program) that finds each word or phrase that has been tagged as a **dfn** element and outputs it in a table next to the header of the section in which it appears. The resulting table looks like this:

Term ( <b>dfn</b> )	Section heading ( <b>h2</b> )
attributes	Attributes
auto-tagging	The life cycle of an XML document
character entities	Entities and numerical character references
characters	Character sets and Unicode
client-side	Other web standards
content	Could I please see some XML already?
descriptive	Types of markup
digital humanities	The context
document analysis	The beginning of every digital humanities project: Document analysis
document grammar	Valid and well-formed documents
dynamic html	Other web standards
element content	Types of element content
elements	Could I please see some XML already?
empty element	Types of element content
encode	The beginning of every digital humanities project: Document analysis
end tag	Could I please see some XML already?
entities	Entities and numerical character references

events	Other web standards
generic identifier	Could I please see some XML already?
gi	Could I please see some XML already?
humanities computing	The context
mark up	The beginning of every digital humanities project: Document analysis
milestone	Types of element content
mixed content	Types of element content
multipurposing	Types of markup
name	Attributes
nodes	Document and their structures
numerical character references	Entities and numerical character references
presentational	Types of markup
procedural	Types of markup
root	Document and their structures
schema	Valid and well-formed documents
schema languages	Valid and well-formed documents
script	Practicing what we preach
serialization	The XML tree and its serialization
server-side	Other web standards
start tag	Could I please see some XML already?
style sheet	The XML family of standards
tagging	The beginning of every digital humanities project: Document analysis
text content	Types of element content
tree	Document and their structures
valid	Valid and well-formed documents
validated	Valid and well-formed documents
value	Attributes
well balanced	Valid and well-formed documents
well formed	The XML tree and its serialization

The XSLT script begins by finding the **dfn** elements in the page. To the XSLT processor, XML is a tree, so what the processor sees is something like the following diagram (if you don't see a tree diagram, you're using an old browser; upgrade to the latest version of Firefox, Internet Explorer, Chrome, Safari, or Opera):





The human developer looking at the preceding XML snippet in an editor would see something like:

```
<body>
  <div> ... </div>
  <div>
    <h3>Could I please see some XML already?</h3>
    <p> ... </p>
    <p> ... a <dfn>start tag</dfn>, an <dfn>end tag</dfn>, and ... </p>
    <p> ... </p>
  </div>
  <div> ... </div>
</body>
```

The textual representation with the angle brackets is a serialization, but it isn't the real XML. The real XML is the tree, and that, rather than the serialization, is all that's accessible to an XML tool (such as the XSLT processor that created the table above). The XSLT script finds all of the **dfn** elements in the tree; the diagram above is a snippet of the full tree and contains two such elements (colored green). The script sorts the text of all of the terms it finds (in the blue text boxes under the green **dfn** nodes) and creates a new row in the table for each term, writing it (converted to lower case, if necessary) into the left column of the row. The script must then find the title of the section in which the term occurs and put that title into the corresponding right column of the row, and to do that it must walk the tree, starting at each **dfn** node and ending at the appropriate section header.

To find the appropriate section head, for each **dfn** node the script walks up the tree looking for the ancestral section (**div**) node. That route is colored pink and marked with arrowheads in the diagram. The script first walks up from the **dfn** to the **p** that contains it, and then from the **p** to the **div** that contains it. When the process reaches a **div**, it changes direction and looks under the **div** for the section title (**h3**) node, and it retrieves the section title text from inside the **h3** node. The text that gets copied into the table is colored blue; the text under the **dfn** node goes in the left column and the text under the **h3** goes in the right. The snippet represented by the tree creates two rows in the table, one for each **dfn** node, and since they are located inside the same **div** they are matched with the same **h3** text. The element nodes that are not used to create the table and the text that is not copied into the table are colored yellow.

All of this takes just a few lines of XSLT code. It does require me to know the structure of my input document (this page), that is, to know that I used **h3** elements for section titles and not something else. As you develop your sites, you'll want to develop your schemas with an eye toward the processing you want to do with your data, and

you'll want to design your XSLT or XQuery programs to take advantage of the features you build into your schema.

Remember multipurposing, the programmatic generation of multiple output documents from a single XML source document? Suppose we also need to produce a completely different table, one that lists each of the section headings in the order in which it occurs in the document alongside a count of the terms that are introduced in that section and a list of the terms themselves. Oh, and let's sort the terms alphabetically, instead of listing them in the order in which they occur, and render them all in lower case, even if they might originally have had initial capitalization. A different small XSLT program produces the following table from the same input document, this page, here too without my having to construct any part of the table manually:

Introduction	0	(none)
The context	2	digital humanities, humanities computing
Document and their structures	3	nodes, root, tree
The beginning of every digital humanities project: Document analysis	4	document analysis, encode, mark up, tagging
Types of markup	4	descriptive, multipurposing, presentational, procedural
Is every document really a hierarchy?	0	(none)
Could I please see some XML already?	6	content, elements, end tag, generic identifier, gi, start tag
Types of element content	5	element content, empty element, milestone, mixed content, text content
Attributes	3	attributes, name, value
The XML tree and its serialization	3	serialization, serialization, well formed
Valid and well-formed documents	6	document grammar, schema, schema languages, valid, validated, well balanced
Entities and numerical character references	3	character entities, entities, numerical character references
The life cycle of an XML document	1	auto-tagging
The XML family of standards	1	style sheet
Other web standards	4	client-side, dynamic html, events, server-side
Character sets and Unicode	1	characters
Practicing what we preach	1	script
Conclusion	0	(none)

This is a modest example of the use of XSLT to transform XML, constrained by my desire to work with the present document. In this course you'll identify more interesting processing tasks, of the sort that can be used to facilitate real humanities research, and you'll design systems and write programs to undertake that research.

## Conclusion

The most important things to remember from this introduction are that:

- XML is a way of modeling a textual document as an ordered hierarchy, or tree, so that it can be explored with computational tools. Humanities scholars use XML to represent their documents because the tree model is convenient both as a logical representation (some aspects of the inherent structure of documents are tree-like) and for programming purposes (computers can process tree representations efficiently).
- Digital humanities researchers typically mark up the documents they wish to study in XML and then transform, explore, and manipulate those documents using XSLT (and XQuery).
- XML in digital humanities research should be used to model the structure and semantics of a document. Presentation is important, but it should be handled separately, and not confused or conflated with structure.

and semantics.

- The life cycle of a digital humanities project starts with document analysis and also includes schema design, markup, and processing.
- XML markup consists of elements and attributes. Sometimes unusual written symbols (characters) may be represented more easily by character entities or numerical character references than as raw text.
- XML is a tree of elements. It is not a stream of tags and text. It looks like a stream because we usually read character by character, but the internal representation of an XML document, which is what the computer sees, is an ordered hierarchy of objects, and not a string of characters.
- XML *must* be well formed. It *may* also be valid against a particular schema. In digital humanities projects XML is usually validated against a schema.

It isn't necessary to memorize—or even understand—all of the details of this introduction on first reading. You probably understood some parts well, while others may have seemed complex or confusing. That's not a problem; you've been exposed to the terms, which will sound familiar and make more sense when you hear them again in context, and you can return to this introduction at any time to as you gain more hands-on experience working with XML yourself.