

# Modularization of XHTML

Skill Level: Introductory

[Nicholas Chase \(ibmquestions@nicholaschase.com\)](mailto:ibmquestions@nicholaschase.com)  
Freelance Writer

23 Oct 2001

In this tutorial, build your understanding of the modularization of XHTML. With this knowledge, you can use a subset of XHTML, or you can add your own tags to create your own markup language variants.

## Section 1. Before you start

### About this tutorial

This tutorial demonstrates the creation of Memory Markup Language, or MemoryML, which is designed to be viewed by a fictional browser embedded in a device such as a video camera, video tape player, or DVD player.

The modularization of XHTML allows you to choose which XHTML modules to support in an application. You can supplement those modules, and thus create new markup languages that fit seamlessly with XHTML.

### Prerequisites

You should have a thorough understanding of XML and at least a basic understanding of XHTML and how it is used. You should also understand XML validation and be familiar with Document Type Definitions (DTDs) and namespaces. The [Resources](#) section provides links to tutorials that can help you get up to speed in any of these areas.

You do not need any programming skills to understand this tutorial.

## System requirements

This tutorial demonstrates the building of XHTML modules. To actually build these modules, you need only a text editor. To take it one step further and test your new modules, you need a validating parser, such as the Java APIs for XML Processing (JAXP) or Xerces. The [Resources](#) section lists several validating parsers.

---

## Section 2. Modularization overview

### Why modularize?

The first official version of HTML, HTML 2.0, had less than 50 tags in it. In the intervening years, the number of tags has more than doubled, to the point where complete support of all of them is difficult even in the most popular and well-supported browsers. What chance does a tiny mobile phone or PDA have?

As independent devices begin to connect to the Web, developers are faced with a dilemma. If they can't support all of these tags, how should they decide which to support? And how would content providers know which tags were supported?

This tutorial addresses the issue by segmenting the language into standard modules that can be mixed and matched in a standard way. For example, XHTML Basic is a specific subset designed for devices such as mobile phones, pagers, and PDAs. A developer who wishes to create content for XHTML Basic knows exactly which tags are supported. Similarly, if a device supports specific XHTML modules, a developer can aim content toward them.

Modularization is more than that, however. It also allows for the addition of specialized tags in a standardized way, which makes it possible to create new derivative markup languages to serve specific purposes.

### The target device

This tutorial chronicles the creation of a new markup language that tracks data that's recorded on videotape, DVDs, and other media. This language can be used by devices that make, store, and play back these recordings.

These devices, which may have very little memory and processing power, may have

a built-in browser for displaying information such as menus that already appear in similar places (like the setup menu on a VCR). Such menus could easily be presented with XHTML Basic. However, if the information were presented as part of more descriptive XML, it could also be used in controlling the device.

Consider, for a moment, a device that acts as a controller between a video camera, VCR, and DVD. The controller might encounter a snippet of XML code such as:

```
<video tapeid="A323">
  <media mediaid="A323" type="VHS"/>
  <start>0:00:00</start>
  <end>32:12:45</end>
  <subdate>2001-05-23</subdate>
  <donor>John Baker</donor>
  <subject>Pitching a tent</subject>
  <location><description>Outside in the woods</description></location>
</video>
```

The controller might display the subject as part of a menu. In addition, if the user selected this clip, the controller would also know enough to signal the VHS machine to cue tape number A323, start at the beginning, and play for 32 minutes and 12 seconds.

Incorporating these tags into a new markup language integrated with XHTML allows content providers to author content according to a standard.

## The modular architecture

Creating a new markup language actually involves several related modules that fit together in a specific way.

- The abstract module.  
This optional file is not actually read by any application, but instead provides a semi-formalized human-readable description, including a prose description of any element groupings.
- The QName module.  
The QName module, or qualified name module, defines a series of parameter entities that represent the actual element names. These entities are referenced throughout the other modules.
- The declaration module.  
This module provides the actual declarations for elements and their attributes.
- The content model module.  
This module defines the content model not for the new elements, but for the overall module and the XHTML elements that will now have "new"

elements as children.

- The Qnames collection.  
This file links the QName modules together.
- The DTD driver.  
This is the file that is actually referenced when validating an XML file. It contains references to the other modules so they are all available for validation.

---

## Section 3. Conformance

### Conformance and the reasons for it

It wouldn't do any good to have a standard set of capabilities for an application to support if there were no standard way to support them. What if, for example, a browser "supported" the <h1> tag by rendering it as small text within the rest of the content, rather than as its own large block of text? Web authors would have no idea how to use these capabilities even though they were supposedly supported.

Conformance defines a set of specific requirements for both applications and types of languages. For example, an application that claims to be an XHTML user agent (such as a browser) must, among other things, process whitespace in a document a certain way and ignore unknown tags while continuing to process their contents.

In defining XHTML languages, there are two different sets of conformance requirements: XHTML host languages and XHTML integration sets.

### XHTML host languages

An XHTML host language is a new markup language, based on XHTML, that follows the same structural rules as a traditional XHTML page. For example, it must have <html> as its root element. XHTML host language conformance requires:

- Definition in a W3C recognized implementation.  
This currently means that modules must be defined via DTD, but the W3C XML Schema version of modularization is currently in the works, and other implementations are possible.
- The use of "XHTML" at the start of the public text description of the

PUBLIC identifier for the DTD.

For example, `-//COMPANY NAME//ELEMENTS XHTML Memory Language 1.0//EN`.

- Inclusion of the XHTML structure, hypertext, text, and list modules. Other modules may also be included, but any module used must be included in its entirety. A new DTD may extend an element by adding new attributes or content. However, any attributes or minimal content models that are required in XHTML must be required in the new language, and none may be removed.
- Use of a separate XML namespace. This is required for any additional elements or attributes that are added to the language.

It is not always necessary for a language to conform to all of these requirements.

## XHTML integration sets

Not all markup languages must be XHTML host languages. In some cases, the language only needs to describe content, but will never have to stand on its own -- or it may ultimately use a different structure than that of an XHTML page. These languages need only be XHTML integration-set conforming.

XHTML integration set conformance is identical to XHTML host language conformance, with two exceptions:

1. While XHTML must appear in the public text section of the DTD's PUBLIC identifier, it must not be first. For example: `-//COMPANY NAME//ELEMENTS Memory Information XHTML 1.0//EN`.
2. The XHTML structure module is not required. However, the hypertext, text, and list modules are still required.

## Making the choice

In deciding whether to build an XHTML host language or an XHTML integration set language, the primary factor should be how the new markup language will be used. For example, an XHTML host language can stand on its own, while an XHTML integration set language is generally used in conjunction with a host language.

It might also be convenient to use a host language as the basis for extension, since an integration set language could easily lack the structure necessary to stand on its own in any environment, let alone one based on XHTML.

In choosing a path for the Memory Markup Language in this tutorial, consider how it will be used. The devices that will read it, such as the controller, have no other knowledge of XHTML, so they need some sort of structure built into the language.

Of course, there is nothing dictating that this must be the same structure as XHTML. The language could just as easily use `<video>` or `<player>` as its root element, so it could be built as an XHTML integration set language.

On the other hand, what if the controller were Web enabled, allowing access to it from a browser on the Web? In this case, it would certainly be better to use an XHTML host language, because the "pages" would at least be intelligible in a browser. (And with the addition of a style sheet to control the presentation of the new elements, it could even look nice.)

This tutorial demonstrates the building of Memory Markup Language as an XHTML host language.

---

## Section 4. The abstract module

### The purpose of this module

The definition of a markup language involves the creation of several modules (also called submodules), most of which are processed by the parser to validate a document. The abstract module is an exception to this rule.

The abstract module instead provides information necessary for a human reader to understand the structure of the language without having to thread through multiple documents. It lists the potential elements, the attributes they can carry, and any logical groupings that might exist.

The abstract module makes the overall structure of the language clear to the reader.

### Design goals

In creating Memory Markup Language, consider the following goals:

MemoryML should:

- Encode a description of the contents of a recording

- Support browsing by both specialized and non-specialized browser applications
- Be human readable
- Support processing by non-browser applications, such as software, to control a device

One way to begin to determine the structure is to create sample data.

## The source data

The actual design of a language is a field of study that can include fairly specialized skills, but a little common sense and forethought can go a long way in determining the basics of a language.

MemoryML is actually based on data for the Millennium Memory Project, which gathers recordings of everyday life at the beginning of the new millennium. Each listing includes information on what the recording contains and where it can be found. For example:

```
<?xml version="1.0"?>
<memories>
  <video tapeid="A323">
    <media mediaid="A323" type="VHS"/>
    <start>0:00:00</start>
    <end>32:12:45</end>
    <subdate>2001-05-23</subdate>
    <donor>John Baker</donor>
    <subject>Pitching a tent</subject>
    <location><description>Outside in the woods</description></location>
  </audio>
  <memory tapeid="C531">
    <media mediaid="C531" type="DVD"/>
    <start>12:09:23</start>
    <end>58:34:51</end>
    <subdate>2001-05-18</subdate>
    <donor>Elizabeth Davison</donor>
    <subject>Baseball Game</subject>
  </memory>
</memories>
```

(In a production environment, much more information would actually be needed. This tutorial uses this structure for simplicity's sake.)

This sample data shows only the extensions that will be added to the new language. Remember, the structure, hypertext, text, and list modules are also included.

There is no specific format required for describing these relationships, but certain conventions do exist.

## Conventions

Although there is no formal requirement for the formatting of an abstract module, certain conventions have been in use for some time and are expected by those reading the document. Many are similar or identical to conventions and modifiers used in DTDs themselves.

For example, `?`, `+`, and `*` have special meanings in a DTD -- specifically, zero or one instances, one or more instances, and zero or more instances, respectively. These conventions are also followed in the abstract module. Similarly, element names listed with commas are required in a particular order, and those separated by the `|` symbol are alternate choices. Grouping using parentheses also works the same way as it does in DTDs.

Some conventions used in abstract modules, however, are not included in DTDs. For example, to indicate that an element is being extended to include new attributes (as opposed to being created from scratch), the name is followed by an ampersand, as in `title&`.

Attributes may also have special needs. Required attributes should be followed by an asterisk, as in `tapeid*`. Permissible values may also be specified, with the default value specified with an asterisk, as in:

```
type("VHS" | "DAT" | "DTD" | "DVD" | "8mm"*)
```

In contrast to specific values, an attribute may require a particular data type, as in `tapeid(CDATA)`, or a fixed value, as in `license(="free")`.

In addition to these conventions, prose sections can be added to describe minimal content models.

### Groups and the minimal content model

In many cases, elements or attributes are grouped together to make it easier to indicate that a number of different attributes or elements are appropriate in a particular place. For example, it would be tremendously inconvenient to list even the core attributes that are available on every XHTML element; instead, it is common to just see a reference to `Common`.

Similarly, elements can be grouped together into an expression representing a particular minimal content model for convenience. For example:

```
Heading  
h1 | h2 | h3 | h4 | h5 | h6
```

In this way, it's convenient to simply refer to the `Heading` expression, rather than listing elements individually. The same thing can apply to custom markup. Consider the MemoryML example. It might be convenient to create a set of descriptors for a recording:

```
VideoInfo
media | start | end | subdate | donor | subject | location
```

This creates a second advantage. While it is now possible to say that a `video` element has as its children the `VideoInfo` content model, it is also possible to conveniently describe the children of an `audio` element, which consist of all of the `video` children except `location`. An abstract module can indicate this relationship using:

```
VideoInfo - location
```

Combining these conventions provides the tools for creating the abstract module.

## The abstract module

Combining these techniques into a single abstract module for MemoryML can create an intelligible description of the Memory extensions module.

Elements	Attributes	Minimal Content Model
memories		(audio   video)*
video	tapeid(IDREF)	VideoInfo
audio	tapeid(IDREF)	VideoInfo - location
start		PCDATA
end		PCDATA
subdate		PCDATA
donor		PCDATA
subject		PCDATA
location		place   description
description		PCDATA
place		PCDATA
media	mediaid(ID), type("VHS"   "DAT"   "DTD"   "DVD"   "8mm"*)	EMPTY

This module also defines the minimal content model `VideoInfo`:  
`media, start, end, subdate, donor, subject, location`

Now that the structure is set, it's time to start creating the actual DTD modules.

## Section 5. The QName module

## The purpose of this module

A QName, or qualified name, is a name for an element or attribute that also includes any applicable namespace alias. For example, for the element

```
<mem:subject>Baseball game</mem:subject>
```

the qualified name of the element is `mem:subject`, where `mem` is the namespace alias.

However, the alias, or prefix, is not always displayed. In some cases, prefixing might be disabled altogether, or might be enabled only for the extensions to XHTML.

The modularization of XHTML makes this possible by *parameterizing* the definitions. In other words, instead of defining the name of the element as `subject` or `mem:subject`, the module defines a parameter entity as:

```
<!ENTITY % Memory.subject.qname "%Memory.pfx;subject" >
```

This way, if indexing is turned on, the value of `%Memory.pfx;` will be, say, `mem:`, making the value of `%Memory.pfx;subject` equal to `mem:subject`. If indexing is turned off, the value of `%Memory.pfx;` will be empty, leaving the value of `%Memory.pfx;subject` as simply `subject`.

The QName module provides adaptable definitions for all element names.

## Create the file

Each module (and thus the file containing it) that's involved in creating a new markup language should have, in addition to actual definitions, documentation that makes it easier to use. This information may include information on where the "official" version of the file is located, or the `PUBLIC` identifier that should be used to refer to it.

Let's take a sample file from the W3C as a starting point, since developers are familiar with this layout. The following example shows the start of the QName module file:

```
<!-- ..... -->
<!-- MemoryML QName module ..... -->
<!-- file: memory-qname-1.mod

PUBLIC "-//MY COMPANY//ELEMENTS XHTML MemoryML Qnames 1.0//EN"
SYSTEM "http://www.my.org/DTDs/memory-qname-1.mod"
```

```
xmlns:memory="http://www.my.org/namespaces/MemoryML"
..... -->
```

Note that all of this information is included in comments. It is intended only for developers who might be looking to it for implementation help.

The actual names are not important, but developers expect certain conventions. For example, the QName module is usually named `MODULENAME-qname-1.mod`, whereas the definition module itself would be `MODULENAME-1.mod`.

The `PUBLIC` identifier must follow the conformance rules mentioned in [XHTML host languages](#), but the `SYSTEM` information is up to you. Just be aware that the QName file should always be available at the URI listed as the `SYSTEM` identifier for those applications that do not recognize the `PUBLIC` identifier.

In order to conform to the XHTML host language requirements, all additional elements must be contained in their own namespace. This is the namespace noted in the example above, so it must be consistent across all files.

Whether the namespace information is seen as a prefix to every element is determined by *conditional sections* of the DTD.

## DTD conditional sections

Although a DTD is not a platform for actual programming, sections can be included or ignored, effectively changing a document type's definition. For example:

```
<![IGNORE[
  <!ELEMENT memory (video)*>
]]>
<![INCLUDE[
  <!ELEMENT memory (video | audio)*>
]]>
```

In this example, `memory` elements may include `video` or `audio`, because that section was included, and the previous section ignored. Swapping the following values

```
<![INCLUDE[
  <!ELEMENT memory (video)*>
]]>
<![IGNORE[
  <!ELEMENT memory (video | audio)*>
]]>
```

results in `memory` elements that can only contain `video`.

Entity definitions can also be included or excluded, but with an additional

complication: The first definition of an entity is the one used in any given situation. So the example

```
<![INCLUDE[
  <!ENTITY % prefixed "TRUE" >
]]>
<!ENTITY % prefixed "FALSE" >
```

results in %prefixed having a value of TRUE, because that definition was included in the document first.

On the other hand, if another DTD that was processed first provided a value for %prefixed, then that value would be used regardless of what is specified here. The value could also be provided in the internal subset of the DTD as part of the actual document.

Understanding this propagation of values is crucial when it comes to turning prefixing on and off, as well as other namespace handling issues.

## Namespace handling

A developer creating a document using MemoryML should be able to turn prefixing on and off, but shouldn't be *required* to do it. To that end, the QName module needs to define a parameter entity that can be set if desired, but that already has a value if it's not desired:

```
<!ENTITY % NS.prefixed "IGNORE" >
<!ENTITY % Memory.prefixed "%NS.prefixed;" >
```

In this way, the parameter is available, but if it is not set explicitly, it takes the value of %NS.prefixed, which is actually defined within the XHTML framework. A developer can choose to set %NS.prefixed to INCLUDE, effectively turning on all prefixing, or can instead set %Memory.prefixed to INCLUDE, turning prefixing on only for the memory module. If neither is set, both remain IGNORE.

Next, define the actual namespace and prefix:

```
<!ENTITY % Memory.xmlns "http://www.my.org/namespaces/MemoryML" >
<!ENTITY % Memory.prefix "memory" >
```

Once these values are determined, create the namespace declaration for the actual document:

```
<![%Memory.prefixed;[
  <!ENTITY % Memory.pfx "%Memory.prefix;" >
  <!ENTITY % Memory.xmlns.extra.attrib
    "xmlns:%Memory.prefix; %URI.datatype; #FIXED '%Memory.xmlns;' " >
```

```
]]>
<!ENTITY % Memory.pfx "" >
<!ENTITY % Memory.xmlns.extra.attrib "" >
```

In this way, if `%Memory.pfx` is turned on, the value of `%Memory.xmlns.extra.attrib` becomes:

```
xmlns:memory %URI.datatype; #FIXED 'http://www.my.org/namespaces/MemoryML'
```

(`%URL.datatype` is defined in the XHTML framework.)

Of course, it doesn't do any good to create this value if there's no way to get it back to the actual document. Fortunately, the XHTML framework defines a parameter entity that propagates the information back to the document:

```
<!ENTITY % XHTML.xmlns.extra.attrib "%Memory.xmlns.extra.attrib;" >
```

Once the namespace handling is taken care of, you're ready to add the actual elements.

## Elements and attributes

All of the juggling described above is intended to allow for easy creation (or exclusion) of namespace prefixes on elements. Like the prefixing information, these elements are parameterized in the QName module. For example:

```
<!ENTITY % Memory.subject.qname "%Memory.pfx;subject" >
```

The rest of the module-building process refers to this element as `%Memory.subject.qname`. This way, whether prefixing is on (making it `memory:subject`) or off (making it `subject`), it'll be correct.

The name of the actual module, in this case `Memory`, is used at the beginning of all parameter entities for this module to distinguish them not just from XHTML entities, but also from additional extensions that might be added later.

Putting it all together completes the QName module.

## The QName module (code)

```
<!-- ..... -->
<!-- MemoryML QName module ..... -->
<!-- file: memory-qname-1.mod

PUBLIC "-//MY COMPANY//ELEMENTS XHTML MemoryML Qnames 1.0//EN"
```

```

SYSTEM "http://www.my.org/DTDs/memory-qname-1.mod"

xmlns:memory="http://www.my.org/namespaces/MemoryML"
..... -->

<!-- Declare the default value for prefixing of this module's elements -->
<!-- Note that the NS.prefixed will get overridden by the XHTML Framework or
    by a document instance. -->
<!ENTITY % NS.prefixed "IGNORE" >
<!ENTITY % Memory.prefixed "%NS.prefixed;" >

<!-- Declare the actual namespace of this module -->
<!ENTITY % Memory.xmlns "http://www.my.org/namespaces/MemoryML" >

<!-- Declare the default prefix for this module -->
<!ENTITY % Memory.prefix "memory" >

<!-- Declare the prefix and any prefixed namespaces that are required by
    this module -->
<![%Memory.prefixed;[
<!ENTITY % Memory.pfx "%Memory.prefix;" >
<!ENTITY % Memory.xmlns.extra.attrib
    "xmlns:%Memory.prefix; %URI.datatype; #FIXED '%Memory.xmlns;' >
]]>
<!ENTITY % Memory.pfx "" >
<!ENTITY % Memory.xmlns.extra.attrib "" >

<!ENTITY % XHTML.xmlns.extra.attrib "%Memory.xmlns.extra.attrib;" >

<!ENTITY % Memory.memories.qname "%Memory.pfx;memories" >
<!ENTITY % Memory.video.qname "%Memory.pfx;video" >
<!ENTITY % Memory.audio.qname "%Memory.pfx;audio" >
<!ENTITY % Memory.start.qname "%Memory.pfx;start" >
<!ENTITY % Memory.end.qname "%Memory.pfx;end" >
<!ENTITY % Memory.subdate.qname "%Memory.pfx;subdate" >
<!ENTITY % Memory.donor.qname "%Memory.pfx;donor" >
<!ENTITY % Memory.subject.qname "%Memory.pfx;subject" >
<!ENTITY % Memory.location.qname "%Memory.pfx;location" >
<!ENTITY % Memory.description.qname "%Memory.pfx;description" >
<!ENTITY % Memory.place.qname "%Memory.pfx;place" >
<!ENTITY % Memory.media.qname "%Memory.pfx;media" >

```

As long as there is only one module, this file takes care of all of the qualified names. But what if there is more than one module?

## The Qnames collection

There is no limit to the number of files that can be tied together, but adding more than one module introduces an additional complication. The Qnames module contains the definition:

```
<!ENTITY % XHTML.xmlns.extra.attrib "%Memory.xmlns.extra.attrib;" >
```

The final document uses `%XHTML.xmlns.extra.attrib` to add any extra namespaces for external modules. The difficulty lies in the fact that if more than one module is defined, each one redefines `%XHTML.xmlns.extra.attrib` so that only its information is included.

To solve the problem, you can create a Qnames collection module. This file simply combines all of the namespace information into a single entity. If, for example, there were an additional module called MemoryExtensions, the Qnames collection module would contain:

```
<!-- ..... -->
<!-- MemoryML/MemoryExtensions Qname Collection Module ..... -->
<!-- file: memory-qnames.mod ..... -->

<!-- Bring in both sets of Qnames in order to access their entities -->
<!ENTITY % Memory-qname.mod
    PUBLIC "-//MY COMPANY//ELEMENTS XHTML MemoryML Qname Collection 1.0//EN"
    SYSTEM "http://www.my.org/DTDs/memory-qnames.mod" >
%Memory-qname.mod;

<!ENTITY % MemoryExtension-qname.mod
    SYSTEM "memoryextension-qname-1.mod" >
%MemoryExtension-qname.mod;

<!-- Add both as an extension to XHTML -->
<!ENTITY % XHTML.xmlns.extra.attrib
    "%Memory.xmlns.extra.attrib;
    %MemoryExtension.xmlns.extra.attrib;" >
```

Any number of namespaces can be added to the overall language, as long as they're tied together this way.

This tutorial deals with only the memory module, so the Qname collection contains:

```
<!-- ..... -->
<!-- MemoryML Qname Collection Module ..... -->
<!-- file: xhtml-memory-qname-1.mod ..... -->

<!-- Bring in the MemoryML qualified names -->
<!ENTITY % Memory-qname.mod
    PUBLIC "-//MY COMPANY//ENTITIES XHTML MemoryML Qnames 1.0//EN"
    "memory-qname-1.mod" >
%Memory-qname.mod;

<!-- Define the xmlns extension attributes -->
<!ENTITY % XHTML.xmlns.extra.attrib
    "%Memory.xmlns.extra.attrib;" >
```

Now that you've got their names, it's time to define the actual elements.

## Section 6. The declaration module

### The purpose of this module

Now that all of the element names have been declared, it's time to declare the

elements themselves.

Declaring the elements is just like creating a DTD, except that instead of using direct references to element names, the structure references the parameterized names. In this way, the declaration module defines both the elements and their content models.

The declaration module also includes the namespace information for each element by [Adding the namespace attribute](#).

First, you need to declare the actual elements.

## Declare elements and attributes

The easiest way to declare elements and attributes is to first build the traditional DTD, then change the declarations over to parameters. So

```
<!ELEMENT memories ( audio | video )* >
```

becomes

```
<!ELEMENT %Memory.memories.qname;  
  ( %Memory.audio.qname; | %Memory.video.qname; )* >  
<!ATTLIST %Memory.memories.qname;  
  %Memory.xmlns.attrib;  
>
```

The extra attribute defines the namespace information added to the element. That namespace information is determined in much the same way as [Namespace handling](#).

## Adding the namespace attribute

The information that is actually contained in `%Memory.xmlns.attrib;` depends on the namespace definition and prefixing decisions made earlier:

```
<![%Memory.prefixed;[  
<!ENTITY % Memory.xmlns.attrib  
  "%NS.decl.attrib;"  
>  
]]>  
<!ENTITY % Memory.xmlns.attrib  
  "%NS.decl.attrib;  
  xmlns %URI.datatype; #FIXED '%Memory.xmlns;'"  
>
```

The value of `%NS.decl.attrib;` represents the global namespace attribute

information, which is determined by conditionals in the XHTML framework DTD. If prefixing is on, then only this information needs to appear on every element, as the prefix refers back to the specifics for this element. However, if prefixing is off, then the element also needs the specific namespace information for the module.

All elements must reference `%Memory.xmlns.attrib;` as an attribute. If an element already has an attribute, namespace information is added on, as in:

```
<!ATTLIST %Memory.media.qname;
      mediaid ID #REQUIRED
      type CDATA #IMPLIED
      %Memory.xmlns.attrib;
>
```

## The declaration module (code)

```
<!-- ..... -->
<!-- Memory Elements Module ..... -->
<!-- file: memory-1.mod

PUBLIC "-//MY COMPANY//ELEMENTS XHTML MemoryML Declaration 1.0//EN"
SYSTEM "http://www.my.org/DTDs/memory-1.mod"

xmlns:memory="http://www.my.org/namespaces/MemoryML"
..... -->

<!-- Memory Module
memories
  audio
    media
      start
      end
      subdate
      donor
      subject
  video
    media
      start
      end
      subdate
      donor
      subject
      location
      description | place
-->

<!-- Define the global namespace attributes -->
<![%Memory.prefixed;[
<!ENTITY % Memory.xmlns.attrib
      "%NS.decl.attrib;"
>
]]>
<!ENTITY % Memory.xmlns.attrib
      "%NS.decl.attrib;
      xmlns %URI.datatype; #FIXED '%Memory.xmlns;'"
>

<!ELEMENT %Memory.memories.qname;
      ( %Memory.audio.qname; | %Memory.video.qname; )* >
<!ATTLIST %Memory.memories.qname;
```

```

    %Memory.xmlns.attrib;
>
<!ELEMENT %Memory.video.qname;
  ( %Memory.media.qname; , %Memory.start.qname; ,
    %Memory.end.qname; , %Memory.subdate.qname; ,
    %Memory.donor.qname; , %Memory.subject.qname; ,
    %Memory.location.qname;) >
<!ATTLIST %Memory.video.qname;
  tapeid IDREF #REQUIRED
  %Memory.xmlns.attrib;
>

<!ELEMENT %Memory.audio.qname;
  ( %Memory.media.qname; , %Memory.start.qname; ,
    %Memory.end.qname; , %Memory.subdate.qname; ,
    %Memory.donor.qname; , %Memory.subject.qname;) >
<!ATTLIST %Memory.audio.qname;
  tapeid IDREF #REQUIRED
  %Memory.xmlns.attrib;
>

<!ELEMENT %Memory.media.qname; EMPTY >
<!ATTLIST %Memory.media.qname;
  mediaid ID #REQUIRED
  type CDATA #IMPLIED
  %Memory.xmlns.attrib;
>

<!ELEMENT %Memory.start.qname; (#PCDATA) >
<!ATTLIST %Memory.start.qname;
  %Memory.xmlns.attrib;
>

<!ELEMENT %Memory.end.qname; (#PCDATA) >
<!ATTLIST %Memory.end.qname;
  %Memory.xmlns.attrib;
>

<!ELEMENT %Memory.subdate.qname; (#PCDATA) >
<!ATTLIST %Memory.subdate.qname;
  %Memory.xmlns.attrib;
>

<!ELEMENT %Memory.donor.qname; (#PCDATA) >
<!ATTLIST %Memory.donor.qname;
  %Memory.xmlns.attrib;
>

<!ELEMENT %Memory.subject.qname; (#PCDATA) >
<!ATTLIST %Memory.subject.qname;
  %Memory.xmlns.attrib;
>

<!ELEMENT %Memory.description.qname; (#PCDATA) >
<!ATTLIST %Memory.description.qname;
  %Memory.xmlns.attrib;
>

<!ELEMENT %Memory.place.qname; (#PCDATA) >
<!ATTLIST %Memory.place.qname;
  %Memory.xmlns.attrib;
>

<!ELEMENT %Memory.location.qname;
  ( %Memory.place.qname; | %Memory.description.qname;) >
<!ATTLIST %Memory.location.qname;
  %Memory.xmlns.attrib;
>

```

## Section 7. The content model module

### The purpose of this module

Just as a DTD defines the structure for the new module, the overall markup language has a structure. For example, the `memory` element can contain `video` and `audio` elements, but at the moment, there are no XHTML elements that can contain a `memory` element, so there is no way to add one to a page!

The content model module defines these structural elements. Because XHTML has been parameterized, definitions for each of its elements can be altered. Just remember: Conformance requires that all mandatory elements remain mandatory, and that optional elements remain available.

For example, adding `subject` as a potential child for the `li` element involves redefining the content model for `li`, as defined in the XHTML core modules:

```
<!ENTITY % li.content
"( #PCDATA | %Flow.mix; | %Memory.subject.qname; )*"
>
```

At first glance, it would seem that only elements that are being extended need to be redefined -- after all, the original definitions still exist. Unfortunately, it's not that simple.

In the final DTD driver file, the content model module will be listed first, so its definitions will take precedence. As a consequence, parameters such as `%Flow.mix;` have not yet been defined when this file is processed.

This means that in order to add `subject` to `li`, the `%Flow.mix;` parameter entity and all of the entities that make up `%Flow.mix;` need to be defined.

### Define the structure

The first decisions that have to be made determine where the new elements will appear in relation to the old ones. Much of this depends on how the data will be used. Should the `subject` appear as part of a hypertext link (`<a>`), or should it simply appear as part of a list item (`<li>`) and act like a hypertext link when it is selected? How will the data appear on a page under normal circumstances?

In some situations, it make sense to consider both uses. For example, the browser built into the video controller might know that a `subject` should act like a hypertext

link, but a browser used to access the controller over the Web might not, making it necessary to add it to a normal link tag.

For the sake of this tutorial, assume the following:

- A `subject` might appear as part of a list item (`li`) or a link (`a`).
- A `memory` might appear on the body of the page itself.

A production environment is more likely to require the addition of elements to, say, the set of all block elements. For simplicity's sake this example adds them directly to these elements.

These decisions help to form the content model module.

## The content model module (code)

```
<!-- ..... -->
<!-- XHTML Memory Model Module ..... -->
<!-- file: xhtml-memory-model-1.mod

SYSTEM "xhtml-memory-model-1.mod"
..... -->

<!-- Define the content model for Misc.extra -->
<!ENTITY % Misc.class
    "| %script.qname; | %noscript.qname; ">

<!-- ..... Inline Elements ..... -->

<!ENTITY % HeadOpts.mix
    "( %meta.qname; )" * >

<!ENTITY % I18n.class "" >

<!ENTITY % InlStruct.class "%br.qname; | %span.qname;" >

<!ENTITY % InlPhras.class
    "| %em.qname; | %strong.qname; | %dfn.qname; | %code.qname;
    | %samp.qname; | %kbd.qname; | %var.qname; | %cite.qname;
    | %abbr.qname; | %acronym.qname; | %q.qname;" >

<!ENTITY % InlPres.class
    "| %tt.qname; | %i.qname; | %b.qname; | %big.qname;
    | %small.qname; | %sub.qname; | %sup.qname;" >

<!ENTITY % Anchor.class "| %a.qname;" >

<!ENTITY % InlSpecial.class "| %img.qname;" >

<!ENTITY % Inline.extra "" >

<!-- %Inline.class; includes all inline elements,
    used as a component in mixes
-->
<!ENTITY % Inline.class
    "%InlStruct.class;
    %InlPhras.class;
    %InlPres.class;
```

```

    %Anchor.class;
    %InlSpecial.class;"
>

<!-- %InlNoAnchor.class; includes all non-anchor inlines,
used as a component in mixes
-->
<!ENTITY % InlNoAnchor.class
    "%InlStruct.class;
    %InlPhras.class;
    %InlPres.class;
    %InlSpecial.class;
    | %Memory.subject.qname; "
>

<!-- %InlNoAnchor.mix; includes all non-anchor inlines
-->
<!ENTITY % InlNoAnchor.mix
    "%InlNoAnchor.class;
    %Misc.class;"
>

<!-- %Inline.mix; includes all inline elements, including %Misc.class;
-->
<!ENTITY % Inline.mix
    "%Inline.class;
    %Misc.class;"
>

<!-- ..... Block Elements ..... -->

<!ENTITY % Heading.class
    "%h1.qname; | %h2.qname; | %h3.qname;
    | %h4.qname; | %h5.qname; | %h6.qname;" >

<!ENTITY % List.class "%ul.qname; | %ol.qname; | %dl.qname;" >

<!ENTITY % Blkstruct.class "%p.qname; | %div.qname;" >

<!ENTITY % Blkphras.class
    "| %pre.qname; | %blockquote.qname; | %address.qname;" >

<!ENTITY % Blkpres.class "| %hr.qname;" >

<!ENTITY % Block.extra " | %Memory.video.qname; | %Memory.audio.qname; " >

<!-- %Block.class; includes all block elements,
used as an component in mixes
-->
<!ENTITY % Block.class
    "%Blkstruct.class;
    %Blkphras.class;
    %Blkpres.class;
    %Block.extra;"
>

<!-- %Block.mix; includes all block elements plus %Misc.class;
-->
<!ENTITY % Block.mix
    "%Heading.class;
    | %List.class;
    | %Block.class;
    %Misc.class;"
>

<!-- ..... All Content Elements ..... -->

<!-- %Flow.mix; includes all text content, block and inline
-->

```

```
<!ENTITY % Flow.mix
    "%Heading.class;
    | %List.class;
    | %Block.class;
    | %Inline.class;
    | %Misc.class;
    | %Memory.subject.qname; "
>
```

---

## Section 8. The DTD driver

### The purpose of this module

Now that all of the pieces exist, it's time to put them together into a single DTD that can be called by an XML document.

This file incorporates not just the files created in the previous examples, but also the support files, such as the XHTML framework itself, and the individual modules required for conformance, such as the structure, hypertext, text, and list modules. The driver can include any other modules as well.

### Linking to XHTML

To begin building the DTD driver, you need to add a version parameter to identify the markup language:

```
<!ENTITY % XHTML.version
    "-//MY COMPANY//DTD XHTML MemoryML 1.0//EN" >
```

Next, start linking the files together, bringing in the files that combine information from both namespaces. Start with the Qnames collection file, which links the two namespaces:

```
<!ENTITY % xhtml-qname-extra.mod
    SYSTEM "xhtml-memory-qname-1.mod" >
```

Also bring in the content model that actually ties the two structures together:

```
<!ENTITY % xhtml-model.mod
    SYSTEM "xhtml-memory-model-1.mod" >
```

Next, you'll add the XHTML framework itself.

## The XHTML framework

A single file contains the XHTML framework, but there are several parameters involved in creating settings for it.

First, add the `%XHTML.profile` entity, which is reserved for use with XHTML profiles when they are available. Though it's not actually used for anything yet, it is referenced within other files and could conceivably be referenced in the content model, so it must be defined before the content model.

```
<!ENTITY % XHTML.profile  "" >
```

Next, disable bidirectional text support, since it won't be in use. (This parameter actually includes a section that removes bidirectional support, hence the odd syntax.)

```
<!ENTITY % XHTML.bidi  "INCLUDE" >
```

Finally, bring in the XHTML framework itself:

```
<!ENTITY % xhtml-framework.mod
    PUBLIC "-//W3C//ENTITIES XHTML Modular Framework 1.0//EN"
           "http://www.w3.org/TR/xhtml-modularization/DTD/xhtml-framework-1.mod" >
%xhtml-framework.mod;
```

Next, you'll include all of the modules.

## Include the new modules

With the framework in place, it's time to begin adding modules. First, add the XHTML modules included in MemoryML:

```
<!-- Text Module (Required) ..... -->
<!ENTITY % xhtml-text.mod
    PUBLIC "-//W3C//ELEMENTS XHTML Text 1.0//EN"
           "http://www.w3.org/TR/xhtml-modularization/DTD/xhtml-text-1.mod" >
%xhtml-text.mod;

<!-- Hypertext Module (required) ..... -->
<!ENTITY % xhtml-hypertext.mod
    PUBLIC "-//W3C//ELEMENTS XHTML Hypertext 1.0//EN"
           "http://www.w3.org/TR/xhtml-modularization/DTD/xhtml-hypertext-1.mod" >
%xhtml-hypertext.mod;

<!-- Lists Module (required) ..... -->
<!ENTITY % xhtml-list.mod
    PUBLIC "-//W3C//ELEMENTS XHTML Lists 1.0//EN"
           "http://www.w3.org/TR/xhtml-modularization/DTD/xhtml-list-1.mod" >
%xhtml-list.mod;
```

```
<!-- Document Structure Module (required) ..... -->
<!ENTITY % xhtml-struct.mod
    PUBLIC "-//W3C//ELEMENTS XHTML Document Structure 1.0//EN"
        "http://www.w3.org/TR/xhtml-modularization/DTD/xhtml-struct-1.mod" >
%xhtml-struct.mod;
```

Last, but certainly not least, don't forget to add the MemoryML module!

```
<!-- Memory Module ..... -->
<!ENTITY % Memory-elements.mod
    SYSTEM "memory-1.mod" >
%Memory-elements.mod;
```

With these additions, the driver file is complete and ready to use.

## The DTD driver (code)

```
<!-- ..... -->
<!-- Memory Extension DTD ..... -->
<!-- file: xhtml-memory-1.dtd -->

<!-- This is the DTD driver for Memory extension 1.0.

    Please use this formal public identifier to identify it:

        "-//MY COMPANY//DTD XHTML MemoryML 1.0//EN"

    And this namespace for extension-unique elements:

        xmlns:Memory="http://www.my.org/namespaces/MemoryML"

-->
<!ENTITY % XHTML.version "-//MY COMPANY//DTD XHTML MemoryML 1.0//EN" >

<!-- Define the xhtml qualified names module to be ours -->
<!ENTITY % xhtml-qname-extra.mod
    SYSTEM "xhtml-memory-qname-1.mod" >

<!-- reserved for use with document profiles -->
<!ENTITY % XHTML.profile "" >

<!-- Define the Content Model for the framework to use -->
<!ENTITY % xhtml-model.mod
    SYSTEM "xhtml-memory-model-1.mod" >

<!-- Disable bidirectional text support -->
<!ENTITY % XHTML.bidi "INCLUDE" >

<!-- Bring in the XHTML Framework -->
<!ENTITY % xhtml-framework.mod
    PUBLIC "-//W3C//ENTITIES XHTML Modular Framework 1.0//EN"
        "http://www.w3.org/TR/xhtml-modularization/DTD/xhtml-framework-1.mod" >
%xhtml-framework.mod;

<!-- Text Module (Required) ..... -->
<!ENTITY % xhtml-text.mod
    PUBLIC "-//W3C//ELEMENTS XHTML Text 1.0//EN"
        "http://www.w3.org/TR/xhtml-modularization/DTD/xhtml-text-1.mod" >
%xhtml-text.mod;

<!-- Hypertext Module (required) ..... -->
```

```

<!ENTITY % xhtml-hypertext.mod
    PUBLIC "-//W3C//ELEMENTS XHTML Hypertext 1.0//EN"
    "http://www.w3.org/TR/xhtml-modularization/DTD/xhtml-hypertext-1.mod" >
%xhtml-hypertext.mod;

<!-- Lists Module (required) ..... -->
<!ENTITY % xhtml-list.mod
    PUBLIC "-//W3C//ELEMENTS XHTML Lists 1.0//EN"
    "http://www.w3.org/TR/xhtml-modularization/DTD/xhtml-list-1.mod" >
%xhtml-list.mod;

<!-- Document Structure Module (required) ..... -->
<!ENTITY % xhtml-struct.mod
    PUBLIC "-//W3C//ELEMENTS XHTML Document Structure 1.0//EN"
    "http://www.w3.org/TR/xhtml-modularization/DTD/xhtml-struct-1.mod" >
%xhtml-struct.mod;

<!-- Memory Module ..... -->
<!ENTITY % Memory-elements.mod
    SYSTEM "memory-1.mod" >
%Memory-elements.mod;

```

---

## Section 9. Building a MemoryML document

### The basic document

Based on the structure we've defined, a simple file that uses the new elements and incorporates them into an XHTML page might look like this:

```

<?xml version="1.0"?>
<!DOCTYPE html SYSTEM "xhtml-memory-1.dtd">
<html>
<head><title>Sample Page</title></head>
<body>
<p><a href="newpage.mml"><subject>Pitching a
tent</subject></a></p>

<ul>
<li><subject>Pitching a tent</subject></li>
<li><subject>Baseball game</subject></li>
</ul>

    <video tapeid="A323">
      <media mediaid="A323" type="VHS"/>
      <start>0:00:00</start>
      <end>32:12:45</end>
      <subdate>2001-05-23</subdate>
      <donor>John Baker</donor>
      <subject>Pitching a tent</subject>
      <location><description>Outside in the
woods</description></location>
    </video>
    <audio tapeid="C531">
      <media mediaid="C531" type="DAT"/>
      <start>12:09:23</start>
      <end>58:34:51</end>

```

```

    <subdate>2001-05-18</subdate>
    <donor>Elizabeth Davison</donor>
    <subject>Baseball Game</subject>
  </audio>
</body>
</html>

```

Notice that the page uses the new DTD. In this case, no namespace information is provided because it's not necessary.

## Prefixing the new elements

Some cases might require that namespace information be provided. Because the internal DTD subset always takes precedence, it's easy to use it to turn on prefixing. Remember `%Memory.prefixed;?` Turn it on here and it overrides any other settings.

```

<?xml version="1.0"?>
<!DOCTYPE html SYSTEM "xhtml-memory-1.dtd" [
  <!ENTITY % Memory.prefixed "INCLUDE">
  <!ENTITY % Memory.prefix "mem">
]>
<html xmlns:mem="http://www.my.org/namespaces/MemoryML">
<head><title>Sample Page</title></head>
<body>
<p><a href="newpage.mml"><mem:subject>Pitching a tent</mem:subject></a></p>

<ul>
<li><mem:subject>Pitching a tent</mem:subject></li>
<li><mem:subject>Baseball game</mem:subject></li>
</ul>

  <mem:video tapeid="A323">
    <mem:media mediaid="A323" type="VHS"/>
    <mem:start>0:00:00</mem:start>
    <mem:end>32:12:45</mem:end>
    <mem:subdate>2001-05-23</mem:subdate>
    <mem:donor>John Baker</mem:donor>
    <mem:subject>Pitching a tent</mem:subject>
    <mem:location><mem:description>Outside in the woods</mem:description></mem:location>
  </mem:video>
  <mem:audio tapeid="C531">
    <mem:media mediaid="C531" type="DAT"/>
    <mem:start>12:09:23</mem:start>
    <mem:end>58:34:51</mem:end>
    <mem:subdate>2001-05-18</mem:subdate>
    <mem:donor>Elizabeth Davison</mem:donor>
    <mem:subject>Baseball Game</mem:subject>
  </mem:audio>
</body>
</html>

```

Notice also that the prefix in this file is different from the one specified in the original Qnames module. This is not a problem. Like the prefix switch, this value takes precedence over any others.

A similar method turns on all prefixing.

## Prefixing all elements

Turning on all prefixing is as simple as changing the default prefixing setting:

```
<?xml version="1.0"?>
<!DOCTYPE xhtml:html SYSTEM "xhtml-memory-1.dtd" [
  <!ENTITY % NS.prefixed "INCLUDE">
  <!ENTITY % XHTML.prefix "xhtml" >
  <!ENTITY % Memory.prefix "mem">
]>
<xhtml:html xmlns:xhtml="http://www.w3.org/1999/xhtml"
  xmlns:mem="http://www.my.org/namespaces/MemoryML">
<xhtml:head><xhtml:title>Sample Page</xhtml:title></xhtml:head>
<xhtml:body>
<xhtml:p>
<xhtml:a href="newpage.mml"><mem:subject>Pitching a tent</mem:subject></xhtml:a></xhtml:p>

<xhtml:ul>
<xhtml:li><mem:subject>Pitching a tent</mem:subject></xhtml:li>
<xhtml:li><mem:subject>Baseball game</mem:subject></xhtml:li>
</xhtml:ul>

  <mem:video tapeid="A323">
    <mem:media mediaid="A323" type="VHS"/>
    <mem:start>0:00:00</mem:start>
    <mem:end>32:12:45</mem:end>
    <mem:subdate>2001-05-23</mem:subdate>
    <mem:donor>John Baker</mem:donor>
    <mem:subject>Pitching a tent</mem:subject>
    <mem:location><mem:description>Outside in the woods</mem:description></mem:location>
  </mem:video>
  <mem:audio tapeid="C531">
    <mem:media mediaid="C531" type="DAT"/>
    <mem:start>12:09:23</mem:start>
    <mem:end>58:34:51</mem:end>
    <mem:subdate>2001-05-18</mem:subdate>
    <mem:donor>Elizabeth Davison</mem:donor>
    <mem:subject>Baseball Game</mem:subject>
  </mem:audio>
</xhtml:body>
</xhtml:html>
```

Notice also that because %Memory.prefixed; inherits the value of %NS.prefixed; it does not need to be set separately.

## Section 10. Summary

This tutorial has demonstrated the methodology and procedure needed to build a new XHTML-based markup language using the modularization of XHTML.

The procedure involves creating definition files that are parameterized, allowing for customization even as far down the line as the instance document itself. These files

are then knitted together by a single DTD driver file that acts as the DTD for the instance document.

Required and optional XHTML modules are joined to the extensions through several files, which create new content models and namespace settings.

# Resources

## Learn

- Read the "[Introduction to XML](#)" tutorial (developerWorks, August 2002) for a basic grounding in XML.
- Read the "[Understanding DOM](#)" tutorial (developerWorks, July 2003) for information on the Document Object Model and basic information on XML namespaces.
- Read the "[Validating XML](#)" tutorial for information on creating Document Type Definitions (DTDs) (developerWorks, August 2003).
- Read the W3C's [XHTML Modularization Recommendation](#).
- Read [XHTML Modules and Markup Languages -- How to create XHTML Family modules and markup languages for fun and profit](#), by Shane McCarron, for a look at creating languages with multiple extension modules.
- Read the three-part "[XML programming in Java technology](#)" tutorial series (developerWorks, January--July, 2004) for an intro to programming XML.
- Order [XML and Java from Scratch](#) , by Nicholas Chase. It covers the general use of XML and Java, and it also covers other data-centric views of XML, such as XML Query, and other uses for XML, such as SOAP.

## Get products and technologies

- Download a [zip archive of the sample code](#) presented in this tutorial.
- The IBM Centre for Java Technology Development provides [developer kits](#) for creating and testing Java applets and applications on a range of platforms.
- Download [JAXP 1.1](#), the Java APIs for XML Processing.
- Download the [Xerces parser for Java](#) from the Apache XML project.
- Download the [Xerces parser for C++](#) from the Apache XML project.
- Download the [Xerces parser for Perl](#) from the Apache XML project.
- Download the [Eclipse Modeling Framework \(EMF\)](#), which provides tools and runtime support to produce a set of Java classes for the model, a set of adapter classes that enable viewing and command-based editing of the model, and a basic editor. Models can be specified using annotated Java, XML documents, or modeling tools like Rational Rose, then imported into EMF.

## About the author

## Nicholas Chase

Nicholas Chase has been involved in Web site development for companies such as Lucent Technologies, Sun Microsystems, Oracle, and the Tampa Bay Buccaneers. Nick has been a high school physics teacher, a low-level radioactive waste facility manager, an online science fiction magazine editor, a multimedia engineer, an Oracle instructor, and the Chief Technology Officer of an interactive communications company. He is the author of several books, including *XML Primer Plus* (Sams). He is also a partner in InterSection Unlimited, which specializes in creating Second Life content and applications. You can find him in-world as Chase Marellan.