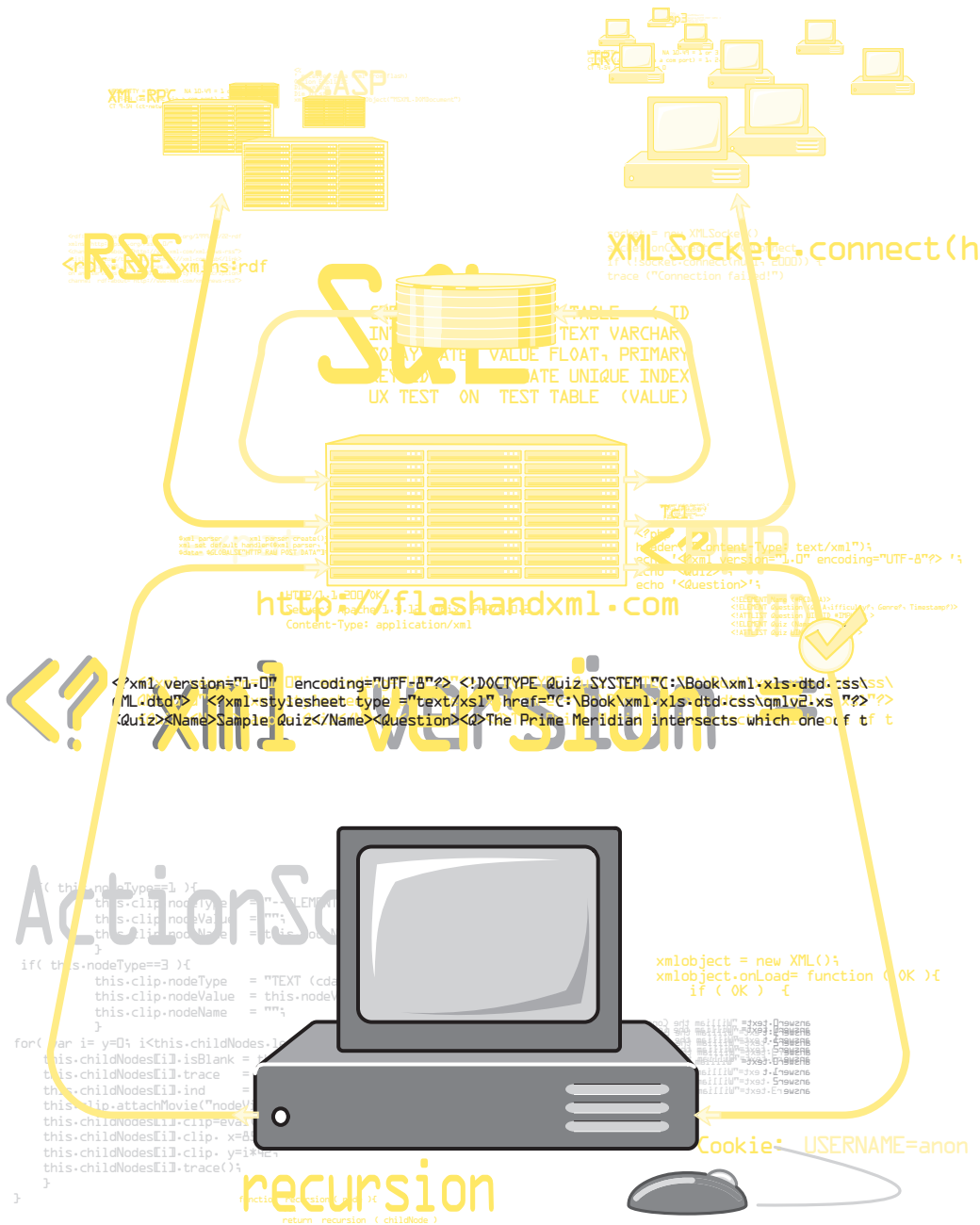
XML-RPC
ASP
RSS
XML Socket .connect(h
SQL
TEXT VARCHAR
VALUE FLOAT, PRIMARY
UNIQUE INDEX
UX TEST ON TEST TABLE (VALUE)
http://flashandxml.com
Content-Type: application/xml
<?xml version="1.0" encoding="UTF-8"?> <!DOCTYPE Quiz SYSTEM "C:\Book\xml.xls.dtd.css\
ML.dtd"> <?xml-stylesheet type ="text/xsl" href="C:\Book\xml.xls.dtd.css\qmlv2.xs "?>
<Quiz><Name>Sample Quiz</Name><Question><Q>The Prime Meridian intersects which one of t
<?xml version
xmlobject = new XML();
xmlobject.onLoad= function (OK){
if (OK) {
Cookie: USERNAME=anon
recursion

5 XML Structure

CHAPTER FIVE

In this chapter we closely examine the XML specification and learn how to create a well-formed XML document. The relevance of each topic to ActionScript is considered, often in the light of an evolving implementation.

Readers with moderate experience in XML might find this chapter useful. Each time the XML specification is restated in English, a new perspective is formed. XML workers might find a tiny highlight in this formulation that rounds out their mental image, if only by raising objections.

Even busy XML experts should read the sections entitled **Flash Context**. These sections point out the differences between the XML specification and the ActionScript implementation at the time of writing and as anticipated in the future.

Element

An XML document is simply a tree composed of a hierarchy of **elements**. In fact the document (exclusive of its optional prologue and extremely rare epilogue) is a single XML element. Since an element includes its child elements, a thousand-element tree, if properly formed, is also a single element—the one at the root.

Flash Context

An element appears in the Flash dataspace as a node. The root element appears as the only child element of the document. Each element has multiple pointers into the hierarchy: parent, siblings, and children. Each carries a list of the names and values of all its attributes. In subsequent chapters we become familiar with the precise arrangement of these data, and we become comfortable tracing and manipulating them.

Syntax

An element is bracketed by a pair of tags. The information between these tags is the content of the element. The content can be nothing (it often is) or it can be a hideously complex and bulky

data structure with hundreds of other complicated elements (it often is). Attribute information, if present, must be embedded in the start tag.The form of the element ID:

```
<start tag>optional content</end tag>
```

In the case of an **empty element** (which contains no content, though it has a name and may have attributes), the start tag and end tag are immediately consecutive and can be fused into a special symbol:

```
<empty element tag/>
```

Rules

Each element must have

- a start tag, introducing its Name
- an end tag (in some cases these two tags are fused)

Additionally, every element can contain any, all, or none of the following:

- More elements
- Text
- Attributes
- Entity references
- Processing instructions
- Comments

It may not have

- XML declaration
- document declaration
- unterminated elements, references, comments, processor instructions
- duplicate assignment of attribute

Examples of Elements

`<Address>125 Main St</Address>`	Address element with street number as content
`<Address>Fourscore... earth.</Address>`	Another address; it is up to the XML author
`<Address line="city">N.Y.C.</Address>`	Element with content and attribute assignment

`<Address>New York <b>City</b></Address>`	Element whose content includes another element
`<Address line="apartment"/>`	Empty element with an attribute
`<Address/>`	Empty element containing only a name token

Bad examples

`<Address line="optional" line="zip"/>`	Multiple assignments of the same attribute
`<Address>New York  City</Address>`	Incomplete inner element
`<Address>New York <b>City</Address></b>`	Incomplete element: improper nesting

Name

A name describes an element. It is not unique: lots of elements can have the same name. Therefore, a name cannot be used to identify an individual element. Instead, it tells us the kind of element, similar to declaring the class of an object or the type of a variable. There are no predefined element types and (with a single exception explained later) there are no reserved element names. It is up to the user to create and maintain meaning.

Flash Context

Lexically, XML names are similar to ActionScript names. XML is slightly more permissive. There is a potential for trouble from two characters, the colon (:) and the period (.). These have special meaning in ActionScript but are permitted in XML names. Yet even in XML they imply a sense of namespace hierarchy. The colon is explicitly discouraged (though not forbidden) in the XML protocol. And use of the period seems only to invite disaster. We avoid the use of either one due to their use as namespace delimiters.

Of course there is no requirement that an application map XML names directly to ActionScript names. Most applications that tried to do so would come across an obstacle larger than a dot or two.

ActionScript names are unique within their scope (you cannot have two variables called “foo” on the same level). XML element names are very frequently identical to the names of their siblings. While ActionScript names serve to identify an individual datum, XML element names do not.

Syntax

The restrictions on names are simple. Names can use any letter in the alphabet. The alphabet does not have to be ASCII, Latin-1, or even 8 bit. XML supports Unicode and allows names in the current alphabet. Our examples, however, will all be within the Latin alphabet.

Case matters, but it is your choice. There is no standard practice for use of case in XML. Often element names are all lower case with dashes standing in for spaces:

```
first-name
```

Names can use any digits, but they may not start with a digit:

```
first-name2
```

XML defines no element names, but it does reserve all names beginning with the three letters "xml" (case insensitive), so *xml-test* is not a permitted name.

Rules

CHARACTER	INITIAL	INTERIOR	SAMPLE
letters	yes	yes	A B C a b c Å ß ç
digits		yes	0 1 2 3
underscore	yes	yes	_
colon	yes, but*	yes, but*	:
dot and dash		yes	.—
xml prefix		yes	XML xml xML

*Colons should be avoided due to the namespace connotation.

Examples of Names

ADDRESS	Street, city, etc.
player	Contains personal data, such as an address
team	Probably contains several player elements
_team	Different kind of team element
team.mascot	Just a simple name. The dot has no meaning.
team_mascot	Alternative name
team-mascot	Traditional XML name formation
Formula1	Digits can be included
test-xml	Noninitial XML triplet is fine

Bad examples

xmlTEST	Cannot begin with xml
9-iron	Cannot begin with digit
team:player	Legal, but very bad form unless *team* is a namespace

Start Tag

XML data is a serial stream. The start tag informs the receiving software that all the incoming data will belong to this element until the matching end tag is encountered. Very often, more start tags are encountered before the end tag. Each start tag must be matched with its own end tag before the previous start tag can be matched.

Syntax

A start tag is simply an element name framed by angle brackets:

XML
```
<TAG>
```

If the element has attributes assigned to it, they belong within the start tag. An element can have multiple attributes, but they must be unique:

XML
```
<TAG attribute="value">
```

Space is forbidden between the opening bracket and the name. It is required before each attribute assignment. It is optional before the end bracket.

Start tags should look familiar to anyone who has worked with HTML.

Rules

Each start tag must have

- the < character
- a name
- the > character

Additionally, every start tag may contain:

- Attribute assignments
- Whitespace after the tag's name

Examples of Start Tags

```
<ADDRESS>
```
Street, city, etc.

```
<player   >
```
Contains legal but probably useless whitespace

```
<team   mascot= "wildcat">
```
Contains valid attribute assignment

```
<team   mascot="wildcat" color="purple">
```
Multiple attribute assignment is permitted

Bad Examples

```
<  player  >
```
Illegal whitespace before name

```
<team   color="purple" color="white">
```
Duplicate attribute assignment is forbidden.

End Tag

The end tag functions exactly like a right parenthesis or a closing quotation mark or a right curly brace. It contains no data of its own; it simply ends the most recent (innermost) tag with the same name.

XML requires a matching end tag for each start tag. Not only must they have matching names, but they must also nest properly. The most recently encountered start tag must be paired with the next end tag. If they do not match, it is a fatal error.

Syntax

The end tag contains exactly the same name as the start tag it closes. Simple punctuation distinguishes it from the start tag:

XML

```
</TAG>
```

It cannot include attribute assignments or any other content.

Rules

Each end tag must have

- the </ character pair
- the name
- the > character

Alternatively, in the special case of an empty element, the end tag may be fused with the start tag. This results in an empty tag containing

- the < character
- the name
- optional attribute assignments
- the /> character pair

Examples of End Tags

`<Team> . . . </Team>`	Street, city, etc.
`<Team> . . . <Player> . . . </Player> . . . </Team>-`	Proper nesting
`<Play> </Play>`	Empty *Play* element
`<Play/>`	Exactly equivalent to the preceding, using an empty tag
`<Play speed= "normal"/>`	Empty *Play* element with *speed* attribute

`<A><B> . . . <C/> <C/><B> . . .` `<C></C></B></B></A>`	Complex nesting: A contains only B, which contains data, two empty Cs, and another B. This inner B also has data and a single empty C. All three Cs are equivalent.

Bad Examples

`<Team>…</team>`	Not a match (case violation)
`<Team/>. . . </Team>`	End tag paired with self-ending empty start tag
`<Team> . . . <Player> . . .` `</Team> </Player>`	Bad nesting

Attributes

Every element can have an unlimited number of attributes. Attributes are a special form of content. They are meant to represent unique properties that are bound very tightly to an individual element. Unlike the element's text content or subelement structure, the attributes tend to be very constrained.

An element can only have a single value for any attribute. A tag that contains two attributes with the same name produces a fatal error.

The XML document definition mechanisms allow us to define attributes unambiguously. We state which attributes each element may or must have. We establish the type of each attribute and we can limit its valid values. Default values can be specified by the data designer.

Validating XML parsers will insure that a document's attributes fit the constraints we've established. No other content can be validated this rigorously, so attributes have an important role in an XML document in transmitting critical data.

Flash Context

Attributes are organized in ActionScript as an associative array, which is a member of the element node object. The name of the attribute becomes the index into the array and the value becomes the string content of the associated array element. Parsing the value into numeric form (for example) is the responsibility of the programmer, although in practice it is often performed automatically by ActionScript's implicit casting operations.

Syntax

Attributes are presented as names and values joined with an equal sign (=).

XML
```
attrib = "value"
```

Attribute names must comply with the name constraints we studied earlier. Spaces are permitted but not required. Most important, the value must always be a quoted string.

Rules

Each attribute must have

- the name
- an equals sign
- an open quote
- a value
- a close quote

It may include:

- space.

Attribute values must not make references outside the document they appear in. (We will see later how these references are made in other parts of the XML document.)

Attribute values cannot contain a < character, except as `<`.

Examples of Attribute

`name= "Anna"`	Simple value
`name= "Anna Leah"`	Value can contain whitespace.
`name= "&daughter;"`	Value can be an internal reference.
`url= "http://bigfun.net/index.xlm"`	Value can be a complex string.
`Speed= "4.5"`	Numeric values represented as strings

Bad Examples

`url= http://bigfun.net/index.xlm`	Forgot the quotes
`tag = "<IMG>"`	Forbidden character
`name = "&//bigfun.net:daughter;"`	Illegal external reference
`Speed = 4.5`	Forgot the quotes again

Text (Character Data)

Text—or, more properly, **character data**—is all the document content other than markup. (Markup is the XML codes: tags, comments, declarations, and so on.) The intent of the XML standard is that the significant content of the document is contained in the character data while the structure of the document is encoded in the markup. This intent is generally followed.

All character data is found within an XML element and forms the content of that element:

XML
```
<ELEMENT>Some text.</ELEMENT>
```

Elements are often found within text blocks. In this case, the outer element contains both character data and a child element:

XML
```
<ELEMENT>Some text.<CHILD/>More text</ELEMENT>
```

The embedded element breaks the character data into two distinct objects that applications can treat separately. When we study the Document Object Model, we will find these two blocks of text in separate nodes. There would then be three children of this element—two text nodes and one child element.

Of course, the child element might also have its own character data, which is not directly part of the outer element:

XML
```
<ELEMENT>Some text.<CHILD>Child's private text</CHILD>More text
  </ELEMENT>
```

Flash Context

Within ActionScript, each block of character data appears as an individual node. In the object model, these text nodes are as significant as element nodes. They are differentiated by NodeType and they must always be leaf nodes. Text that is interrupted by an element (as in the previous line of code) appears in memory as two separate text nodes plus the interrupting element (e.g., `<CHILD>`). This is especially problematic when the content contains even simple HTML markup. For example, it might include the common `<u>underline</u>` tag.

Syntax

Character data is any sequence of characters in the document's alphabet. The only disallowed characters are the two that introduce XML markup: the left angle bracket (`<`), which initiates all XML tags, and the ampersand (`&`), which initiates an entity. These two characters are inserted into character data with escape sequences.

Rules

Character data is composed of:

- any character (including whitespace) except the `<` or `&`.

It may also include

- escape sequences
- other entity references
- CDATA blocks

Examples of Character Data

`<Team>Duluth Wildcats</Team>`	"Duluth Wildcats" is the content.
`<EQ>2*2<5</EQ>`	Content is "2*2 <5".
`<weather>El Ni&#0241;o</weather>`	Content is "El Niño".

Bad Examples

`<longdistance>AT&T</longdistance>`	Forbidden character
`<caption>The "<IMG>" tag</caption>`	Forbidden even within quotes

Entity References

It is frequently convenient and often necessary to perform string substitution during the parsing of an XML file. The XML **entity** concept allows such substitution. We will encounter other entity types later, but the only references found in an XML file proper are **general entity references**. These are references that replace short symbols with legitimate XML character strings. They serve many purposes.

The simplest (and perhaps most common) case is the need to insert into a text stream characters that are meant literally as character data but that the parser would consider as markup. The two characters susceptible to such misinterpretation are the `<` character, which introduces a tag, and the `&` character, which introduces, in fact, an entity reference. These two characters can appear in an XML document only when they are serving their specific functions. XML defines entities that substitute for these two. In addition, it provides for a substitute for the `>` character to satisfy an obscure SGML requirement and for both single and double quote marks, for reasons of convenience that should be obvious.

Additionally a similar escape sequence can exist for any character. The character code (e.g., ASCII or Unicode value) can be expressed in decimal or hexadecimal form. This lets us embed characters that are difficult to produce on the keyboard or unreliable to display.

If the DTD (a preprocessor file we will encounter later) defines it, we can use our own notation, like `&yen;` to represent the ¥ symbol more clearly than `&#165;` does.

Of course we are not limited to single characters. The entity can be invoke a string of arbitrary length. An entity is a useful way to represent frequently used and lengthy text. It is even more valuable for representing volatile data. For example, an entity called &webmaster; might list the name and contact information for the person responsible for supporting an XML document. Data encoders enjoy the ability to represent this lengthy, frequently repeated data with a short clear symbol. And if the webmaster position has high turnover, everyone will appreciate the entity's ability to manage volatility.

Flash Context

Few of the abilities of the entity reference are actually realized in the version of Flash available at the time of this writing. The ActionScript parser is declared to be nonvalidating. A nonvalidating parser does not check the XML file against the declarations in the reference DTD file.

The complex capabilities of entity processing—the surface of which we have merely scratched—depend on use of DTD. So it is not surprising to find that user-defined entities such as `&yen;` or `&webmaster;` do not function as of the time of this writing.

The five predefined escape sequence entities work as expected, and so does the decimal encoding of single characters. The hexadecimal equivalent is not functional at the time of writing but might be fixed soon, as it seems to be the product of oversight, not architecture.

Syntax

The entity reference generally presumes a matching entity declaration in a referenced document: the DTD.

Alternatively there are five symbolic predefined entities and a mechanism for specifying them with decimal or hexadecimal numbers. All these invoke only a single character:

XML

```
&#ddd;          decimal character code ddd
&#xhh;          hex character code hh
&amp;           &
&lt;            <
&gt;            >
&quot;          "
&apo;           '
```

Rules

Each entity reference must have

- an opening ampersand `(&)`
- a valid entity reference
- a closing semicolon `(;)`

It may have (in the entity reference position)

- the `#` followed by a decimal number representing a character code
- the pair `#x` followed by a hexadecimal character code
- one of five standard character entities: `lt`, `gt`, `apos`, `amp`, `quot`
- any token defined in the DTD

It may not have

- any characters or sequence not valid in an XML name.

Examples of General Entity References

`if (x&gt;min &amp;&amp; x&lt; <max)` `print(&quot;it&apos;s ok&quot;);`	if(x >min && x <max) print("it's ok");
`&#xA9;2000 Jacobson &amp; Jacobson`	©2000 Jacobson & Jacobson
`&copyrt;2000 Jacobson &amp; Jacobson`	Same as above if DTD defines copyrt as #xA9

Bad Examples

`&copyrt;2000 Jacobson &amp; Jacobson`	If no definition of copyrt exists, this resolves to &copyrt;2000 Jacobson & Jacobson
`<ELEMENT attribute=&quot;value&quote;>`	Entity references cannot be terminators.

Comments

A comment is text, inserted into the document tree, that is not part of the XML content itself. Comments are meaningful only to humans (when they are meaningful at all). They are typically used by the publisher of an XML document to help other developers extract the most meaning from the document.

Comments are most valuable during the software development phase. During normal XML data exchange they are disregarded.

While most editors preserve comments, not all software will. Browsers, such as IE5, usually preserve both the comment's text and its relationship to the hierarchy of the XML object, allowing comments to be carefully placed where the context is significant.

Flash Context

The Flash XML parser discards incoming comments. This is perfectly legitimate—comments are meant to be read not by applications but only by humans. They are not available to the ActionScript application except by using heroic measures (i.e., writing our own mini-parser).

In fact, older releases of Flash (5.0) cannot properly parse comments. When Flash reads in XML with comments, the entire object model is corrupted.

You are strongly urged to avoid comments in the XML that will feed into your Flash application until it has been tested thoroughly.

ActionScript cannot generate comments in the XML object it outputs. It would be possible to insert comments only "manually" by serializing the XML (generating an XML string from the XML object) and then using string operations to insert or concatenate the comment. This extreme effort should be warranted by some extreme requirement.

If it were necessary to preserve comments in and out of our ActionScript, we would be better off inventing a `<COMMENT>` element. This would be a normal XML element containing normal text data. It would pass through parsers and code generators in place and intact. It would be incumbent on us, of course, to code our applications to treat it appropriately, and indeed it could be treated quite intelligently.

Syntax

The comment is introduced by the string `"<!--"` and terminated by `"-->"`. Between these can go any text except for two consecutive dashes (this restriction was introduced for compatibilty with legacy of SGML):

```
<!--This is commentary.-->
```

Rules

Each comment must have

- the four-character string `<!--`
- any text
- the character triplet `-->`

It may have

- the `&` and `<` characters, forbidden in character data.

It may not have

- a pair of consecutive hyphens `(--)`

Examples of Comments

`<!-- this is a message from me.-->`	Normal usage
`<phone>555-0308 <!--use dash--></phone>`	Comments can be inside an element.
`<town>Kolumbis<!--sic-->, Ohio</town>`	Comments can be inside a text string. (Note: This will cause Flash to incorrectly break the text node into two nodes.)
`<!DOCTYPE MyXML [<!--by Jesse-->]>`	Comments can even be inside a document declaration.

Bad Examples

`<town <!--or city--> >New York</town>`	Comments cannot be inside a tag.
`<!-- It's a--brief--message from me.-->`	Double dashes are disallowed.
`<!-- If grade is B+ or A--->`	A triple dash terminator is disallowed.

Processor Instructions

XML anticipates the need to embed code within a data document. The PI is a mechanism to contain such code. Each PI node identifies the **target**, the application that will interpret and execute the code. It follows the target identification with the code itself. The XML standard

establishes few protocols for engaging this process. It does, however, require parsers to maintain the PI nodes and make them available to the application.

Flash Context

Flash does not like processor instructions. The PI data is not preserved. Even if it were, it could not contain real ActionScript code and make it executable by the application. That would require the Flash player to include an ActionScript interpreter. For reasons of size, it does not. Anyone who has used the `eval()` operator in Flash and elsewhere will recognize that the Flash implementation is extremely limited. This limitation is to avoid burdening the player with a full interpreter.

But it is worse. At presstime, a bug in the Flash parser misinterprets the PI and recovers poorly. All subsequent data is incorrectly positioned in the hierarchy. Since this bug corrupts the entire XML object—sometimes trivially, often fatally—it would be unwise to include PI material in any XML meant to integrate with Flash. If it is unavoidable (i.e., it is someone else's data), you should check. The PI bug may be corrected by the time you read this. If not, you might need to use an XLS preprocessor to filter the XML (or use a server-based proxy) or you might sign up for some heavy sessions inside the Flash debugger, determining the specific behavior you need to code around.

Syntax

PI begins with `<?` and ends with `?>`. Between these is any text. The syntax is defined by the foreign processing language, not by XML. The XML parser will not interpret this text, nor will it expand any embedded entity references.

Rules

Each PI must have

- the opening character pair `<?`
- the target application (or device) to which the PI is directed
- whitespace
- the actual processing instruction stream
- the closing character pair `?>`

It may have

- any characters whatsoever up to the closing `?>`.

It may not have

- an embedded `?>` pair
- entity references that are meant to be expanded

Examples of PI

`<?php counter++; ?>`	Embedded line of PHP script
`<?xml version='1.0'?>`	Standard XML declaration is a PI to the XML processor.
`<?xm-replace_text {Click here} ?>`	Proprietary instruction of the XMeTal editor

Bad Examples

`<?mathpro 3.1?>x**2=>n! ?>`	Forbidden characters in string
`<?actionscript this.nextFrame(); ?>`	Not yet, not yet

Note

XML FOR THE HTML EXPERT

XML will feel familiar to the HTML coder. It has the same notions of start and end tags enclosing very loosely defined text data. They both interject well-specified attribute data within the body of the opening tag.

In practice, XML is far stricter than HTML. Coders used to the forgiving nature of web browsers and the lax definition of many HTML tags should prepare for a more stringent environment. Respect the following differences:

Case sensitivity. HTML is not sensitive to case; XML is.

`<Price>` is not the same as `<PRICE>`.

`<name>` will not be terminated by `</Name>`.

Explicit end tags. HTML tolerates unclosed elements. XML does not.

Stand-alone `<P>` and `<br>` will not work.

`<P></P>` and `<br></br>` will work.

Stand-alone `<P/>` and `<br/>` will work.

Value delimiters. HTML permits unquoted attribute values. XML does not.

`<OBJECT width=100>` does not work.

`<OBJECT width="100">` works.

Proper nesting. HTML permits element overlap. XML is strict.

`"I just <B><I>know</B></I> it!"` will not process.

`"I just <B><I>know</I></B> it!"` will.

XHTML

Since XML is more demanding than HTML but similar in syntax, it is possible to write proper HTML that is legitimate XML as well. This more formal HTML is called XHTML. It is HTML 4.0 that follows the disciplines required to be well-formed and a few others required to be valid.

Conclusion

The structure of basic XML is relatively simple and not unfamiliar to those exposed to HTML. However, it is stricter than HTML.

Many XML features are not implemented in Flash. The features that are missing would be useful in a very complex architecture, but their omission does not prevent XML from being powerful and perfectly useful within Flash.