Event-Driven XML Programming Guide

Data Management

2010-03-24

Contents

Figures, Tables, and Listings

Introduction to Event-Driven XML Programming Guide for Cocoa

> **Note:** This document was previously titled *Event-Driven XML Parsing*.

XML is a markup language that allows you to describe the structure of a document's data entirely in text, using tags that you can arbitrarily define. ("XML" stands for "Extensible Markup Language".) The rules governing the structure are specified in a language schema such as DTD (Document Type Definition). Cocoa provides a class, NSXMLParser, whose instances are event-driven parsers (sometimes called streaming parsers) that sequentially find the constructs of an XML document as well as any associated DTD declarations. They report what they find to a delegate, and it is up to the delegate to do something with this data. This document explains how to use NSXMLParser.

> **Note:** Namespace support was implemented in NSXMLParser for Mac OS X v10.4. Namespace-related methods of NSXMLParser prior to this version have no effect.

Organization of This Document

This programming topic includes the following articles:

See Also

There are many websites that are rich sources of information on XML, DTD, XML tools, open-source parsers, and related specifications and technologies. A few of them are listed below, but a search of the Internet (with, for example, "XML tutorial" in the search field) will find many excellent sources of information:

- World Wide Web Consortium — http://www.w3.org/
- The Annotated XML Specification — http://www.xml.com/axml/testaxml.htm
- W3 Schools — http://www.w3schools.com/default.asp
- O'Reilly XML.com — http://www.xml.com/
- The XML parser and toolkit of Gnome (libxml) — http://xmlsoft.org/

Parser Capabilities and Architecture

There are two general approaches to parsing and handling XML, each with its own style of API:

- **Tree-based API**: This approach maps an XML document into an internal tree structure that conforms to the logical structure described by a schema, then facilitates the navigation and manipulation of that tree. Many tree-based APIs are available, including the DOM (Document Object Model) proposed by the World Wide Web Consortium (W3C). The XML Path Language (XPath), XML Inclusions (XInclude), and XML Pointer Language (XPointer) are WC3 programmatic interfaces for querying and handling the XML in DOM-style tree structures.

- **Event-driven API**: In this approach the parser reports parsing events (such as the start and end of each element) to an application as it encounters them. In C-based APIs, this reporting is accomplished through callbacks implemented by the application to handle the types of events. SAX is the best-known example of this style of parsing API. This type of parser is sometimes referred to as a streaming parser.

The NSXMLParser class adopts the event-driven style of parsing. But instead of using callbacks, an NSXMLParser object (or simply, a parser) sends messages to its delegate; it sends a different message for each type of parsing event. As the parser sequentially encounters each item in an XML or DTD file—element, attribute, declaration, entity reference, and so on—it reports it to its delegate (if the delegate implements the associated method), along with any surrounding context. It does nothing with the item except report it.

For example, say you have a simple XML file such as the following:

```
<?xml version= "1.0" encoding="UTF8">
<article author="John Doe">
    <para>This is a very short article.</para>
</article>
```

The parser would report the following series of events to its delegate:

1. Started parsing document

2. Found start tag for element `article`

3. Found attribute `author` of element `article`, value "John Doe"

4. Found start tag for element `para`

5. Found characters `This is a very short article.`

6. Found end tag for element `para`

7. Found end tag for element `article`

8. Ended parsing document

Both the tree-based and event-based parsing approaches have their strengths and disadvantages. It can require considerable amounts of memory to construct an internal tree representing an XML document, especially if that document is large. This problem is compounded if it becomes necessary to map the tree structure of the parsed document to a more strongly typed, application-specific tree structure.

Event-driven parsing—because it deals with only one XML construct at a time and not all of them at once—consumes much less memory than tree-based parsing. It is ideal for situations where performance is a goal and modification of the parsed XML is not. One such application for event-driven parsing is searching a repository of XML documents (or even one XML document with multiple "records") for specific elements and doing something with the element content. For example, you could use NSXMLParser to search the property-list preferences files on all machines in a Bonjour network to gather network-configuration information.

Event-driven parsing is less suitable for tasks that require the XML to be subjected to extended user queries or to be modified and written back to a file. Event-driven parsers such as NSXMLParser also do not offer any help with validation (that is, it verifying whether XML conforms to the structuring rules as specified in a DTD or other schema). For these kinds of tasks, you need a DOM-style tree. However, you can construct your own internal tree structures using an event-driven parser such as NSXMLParser.

In addition to reporting parsing events, an NSXMLParser object verifies that the XML or DTD is well-formed. For example, it checks whether a start tag for an element has a matching end tag or whether an attribute has a value assigned. If it encounters any such syntactical error, it stops parsing and informs the delegate.

Although the parser "understands" only XML and DTD as markup languages, it can parse any XML-based language schema such as RELAX NG and XML Schema.

XML Parsing Basics

The essential steps for parsing an XML document using NSXMLParser are straightforward. It requires you complete the following general steps:

1. Locate the XML.

   Listing 1 shows code that lets the user select an XML file from a file-system browser (NSOpenPanel).

Listing 1   Opening an XML file

```
- (void)openXMLFile {

    NSArray *fileTypes = [NSArray arrayWithObject:@"xml"];
    NSOpenPanel *oPanel = [NSOpenPanel openPanel];
    NSString *startingDir = [[NSUserDefaults standardUserDefaults]
objectForKey:@"StartingDirectory"];
    if (!startingDir)
        startingDir = NSHomeDirectory();
    [oPanel setAllowsMultipleSelection:NO];
    [oPanel beginSheetForDirectory:startingDir file:nil types:fileTypes
      modalForWindow:[self window] modalDelegate:self
      didEndSelector:@selector(openPanelDidEnd:returnCode:contextInfo:)
      contextInfo:nil];
}

- (void)openPanelDidEnd:(NSOpenPanel *)sheet returnCode:(int)returnCode
contextInfo:(void *)contextInfo {
    NSString *pathToFile = nil;
    if (returnCode == NSOKButton) {
        pathToFile = [[[sheet filenames] objectAtIndex:0] copy];
    }
    if (pathToFile) {
        NSString *startingDir = [pathToFile stringByDeletingLastPathComponent];
        [[NSUserDefaults standardUserDefaults] setObject:startingDir
forKey:@"StartingDirectory"];
        [self parseXMLFile:pathToFile];
    }
}
```

Although an XML file is the common case, the source of the XML might not be a file. You could receive the XML from another object as a property-list object (such as an NSDictionary) or as a stream of bytes over a network. In cases like these, you must convert the form of the XML to an NSData object before initializing the NSXMLParser instance (see following step)

2. Create and initialize an instance of NSXMLParser., ensuring that you set a delegate.

   Listing 2 illustrates how you might do this.

Listing 2   Creating and initializing a NSXMLParser instance

```
- (void)parseXMLFile:(NSString *)pathToFile {
    BOOL success;
    NSURL *xmlURL = [NSURL fileURLWithPath:pathToFile];
    if (addressParser) // addressParser is an NSXMLParser instance variable
        [addressParser release];
    addressParser = [[NSXMLParser alloc] initWithContentsOfURL:xmlURL];
    [addressParser setDelegate:self];
    [addressParser setShouldResolveExternalEntities:YES];
    success = [addressParser parse]; // return value not used
              // if not successful, delegate is informed of error
}
```

In this method, the client object converts the path to the XML file to an NSURL object and then uses that object to initialize the NSXMLParser instance with `initWithContentsOfURL:`. It also sets the delegate to be itself and lets the parser know it wants to resolve external entities (such as external DTD declarations). Other NSXMLParser methods let you set various namespace-related options. Finally, the client sends `parse` to the NSXMLParser instance to have it begin parsing the XML.

If the XML was in some form other than a file, you would convert it to an NSData object and then use the `initWithData:` initializer:

```
addressParser = [[NSXMLParser alloc] initWithData:xmlData];
```

3. Implement the delegation methods that are of interest to you.

    When the NSXMLParser object parses the XML, it sends a message to its delegate for each XML construct it encounters (but only if the delegate implements the associated method). Implementations of these methods vary by type of construct: DTD declarations, namespace prefixes, elements, and so on. Elements are the most common type of XML construct processed; see "Handling XML Elements and Attributes" (page 13) for details.

    All parsing operations begin with the delegate receiving `parserDidStartDocument:` and end with the delegate receiving `parserDidEndDocument:` (assuming, of course, the delegate implements the methods). The former method offers an opportunity for allocating and setting up resources needed for the parsing operation; the latter method is a good place to release those resources and properly dispose of any result.

4. Handle any parsing errors.

    If the parser encounters an error, it stops parsing and invokes the delegation method `parser:parseErrorOccurred:`. Implement this method to interpret the error and inform the user. (All parser errors are nonrecoverable.) See "Handling Parsing Errors" (page 17) for further information.

Memory management becomes a heightened concern when you are parsing XML. Processing the XML often requires you to create many objects; you should not allow these objects to accumulate in memory past their span of usefulness. One technique for dealing with these generated objects is for the delegate to create a local autorelease pools at the beginning of each implemented delegation method and release the autorelease pool just before returning. `NSXMLParser` manages the memory for each object it creates and sends to the delegate.

Handling XML Elements and Attributes

Generally, when you parse an XML document most of the processing involves elements and things related to elements, such as attributes and textual content. Elements hold most of the information in an XML document. When the NSXMLParser object traverses an element in an XML document, it sends at least three separate message to its delegate, in the following order:

```
parser:didStartElement:namespaceURI:qualifiedName:attributes:
parser:foundCharacters:
parser:didEndElement:namespaceURI:qualifiedName:
```

The parser might send the `parser:foundCharacters:` message multiple times for one element; however, if the characters consist of nothing but white-space characters (space, new line, tab, and similar characters) the parser sends `parser:foundIgnorableWhitespace:` instead.

When you are parsing XML elements, an advanced technique you can adopt is to switch processing responsibilities among multiple delegates, each of which knows how to handle a certain type of element. For more information see "Using Multiple Delegates" (page 19).

Design Considerations

In an object-oriented environment such as Cocoa, a common strategy for handling elements is to map them—at the higher nesting levels, at least—to objects. Root elements and other top-level elements are frequently equivalent to collections represented in Cocoa by NSDictionary and NSArray objects. Other elements might readily map to one or more of an application's custom model objects.

However, not all elements are best expressed as objects. Some lower level and particularly "leaf" elements are more logically viewed as properties of their parent element (if that element maps to an object). And, of course, you would probably make the actual attributes of any element a property (that is, an instance variable) of the corresponding object.

Notwithstanding these suggestions, there is no ready-made mapping formula, and indeed your application might not have to perform any element-to-object mapping to achieve its ends. These design decisions require some thought as well as some familiarity with the structure of the XML.

Handling an Element: An Example

The example code referred to in the following discussion processes an XML file containing personal-address information and converts that information into Address Book objects (ABPerson and ABMultipleValue) that can be added to a specified user's address database. A portion of the XML looks like the following:

Listing 1   Some of the sample XML

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE addresses SYSTEM "addresses.dtd">
<addresses owner="swilson">
    <person>
        <lastName>Doe</lastName>
        <firstName>John</firstName>
        <phone location="mobile">(201) 345-6789</phone>
        <email>jdoe@foo.com</email>
        <address>
            <street>100 Main Street</street>
            <city>Somewhere</city>
            <state>New Jersey</state>
            <zip>07670</zip>
        </address>
    </person>

    <!-- more person elements go here -->

</addresses>
```

Let's look at how the first three of these elements might be handled. When the parser first encounters these elements, it invokes the delegate's `parser:didStartElement:namespaceURI:qualifiedName:attributes:` method. For the first two elements, the delegate creates an equivalent object. For the third element (`lastName`), the delegate sets an appropriate property of the second object. Listing 2 shows the delegate's implementation for the start tags of the first three elements.

Listing 2   Implementing parser:didStartElement:namespaceURI:qualifiedName:attribute:

```
- (void)parser:(NSXMLParser *)parser didStartElement:(NSString *)elementName
namespaceURI:(NSString *)namespaceURI qualifiedName:(NSString *)qName
attributes:(NSDictionary *)attributeDict {

    if ( [elementName isEqualToString:@"addresses"]) {
        // addresses is an NSMutableArray instance variable
      if (!addresses)
            addresses = [[NSMutableArray alloc] init];
       return;
    }

    if ( [elementName isEqualToString:@"person"] ) {
        // currentPerson is an ABPerson instance variable
        currentPerson = [[ABPerson alloc] init];
        return;
    }

    if ( [elementName isEqualToString:@"lastName"] ) {
        [self setCurrentProperty:kABLastNameProperty];
        return;
    }
    // .... continued for remaining elements ....
}
```

The delegate identifies the element passed in (*elementName*), then processes it accordingly:

■ If it's an `addresses` element (the root element) it creates a mutable array to hold the ABPerson objects. This mutable array is held as an instance variable.

■ If it's a `person` element, it creates an ABPerson object. This object is held as an instance variable named `currentPerson`.

■ If it's a `lastName` element, it sets an instance variable holding the current Address Book property; this value is a `enum` constant declared in the Address Book framework.

The important action undertaken here is having a way (instance variables in this case) to track the current element throughout the parser's traversal of it. One reason for this importance is the semantics of `parser:foundCharacters:`, most likely the next delegation method invoked. This method can be invoked multiple times for the same element. In this method the delegate should append the characters passed in to the characters accumulated so far for the element. The NSMutableString method `appendString:` is useful for this purpose, as shown in Listing 3.

Listing 3   Implementing `parser:foundCharacters:`

```
- (void)parser:(NSXMLParser *)parser foundCharacters:(NSString *)string {
    if (!currentStringValue) {
        // currentStringValue is an NSMutableString instance variable
        currentStringValue = [[NSMutableString alloc] initWithCapacity:50];
    }
    [currentStringValue appendString:string];
}
```

Again the code uses an instance variable (`currentStringValue`) as a way to track and gather the content for the current element. If the parser encounters some white-space characters in the element content, it sends the message `parser:foundIgnorableWhitespace:` to give the delegate the opportunity to retain any white-space characters (such as tabs or new-lines).

Finally, when the parser encounters the end tag of an element, it invokes the delegation method `parser:didEndElement:namespaceURI:qualifiedName:`. Listing 4 presents the approach taken by the delegate in the example code.

Listing 4   Implementing `parser:didEndElement:namespaceURI:qualifiedName:`

```
- (void)parser:(NSXMLParser *)parser didEndElement:(NSString *)elementName
namespaceURI:(NSString *)namespaceURI qualifiedName:(NSString *)qName {
    // ignore root and empty elements
    if (( [elementName isEqualToString:@"addresses"]) ||
        ( [elementName isEqualToString:@"address"] )) return;

    if ( [elementName isEqualToString:@"person"] ) {
        // addresses and currentPerson are instance variables
        [addresses addObject:currentPerson];
        [currentPerson release];
        return;
    }
    NSString *prop = [self currentProperty];

    // ... here ABMultiValue objects are dealt with ...

    if (( [prop isEqualToString:kABLastNameProperty] ) ||
        ( [prop isEqualToString:kABFirstNameProperty] )) {
        [currentPerson setValue:(id)currentStringValue forProperty:prop];
```

```
    }
    // currentStringValue is an instance variable
    [currentStringValue release];
    currentStringValue = nil;
}
```

If the delegate determines that the end tag is for the `person` element, it adds the ABPerson object to the `addresses` array and releases the ABPerson object. If the end tag is for the `lastName` element (for example), the delegate uses the ABRecord method `setValue:forProperty:` to set the appropriate property in the ABPerson object (ABRecord is the superclass of ABPerson). Finally, the instance variable holding the accumulated content for the element (`currentStringValue`) is released.

Handling an Attribute

The `addresses` element shown in the example XML in Listing 1 (page 14) includes an attribute:

```
<addresses owner="swilson">
```

In this hypothetical case, the attribute allows the application parsing the XML to store the created Address Book information in a specific user directory on a multi-user system.

The NSXMLParser object presents attributes of an element to the delegate in a dictionary in the final parameter of `parser:didStartElement:namespaceURI:qualifiedName:attributes:`. Listing 5 shows how the delegate in the example handles the `owner` attribute.

Listing 5   Handling an attribute of an element

```
- (void)parser:(NSXMLParser *)parser didStartElement:(NSString *)elementName
namespaceURI:(NSString *)namespaceURI qualifiedName:(NSString *)qName
attributes:(NSDictionary *)attributeDict {

    if ( [elementName isEqualToString:@"addresses"]) {
        // addresses is an NSMutableArray instance variable
        if (!addresses)
            addresses = [[NSMutableArray alloc] init];
        NSString *thisOwner = [attributeDict objectForKey:@"owner"];
        if (thisOwner)
            [self setOwner:thisOwner forAddresses:addresses];
        return;
    // ... continued ...
}}
```

The delegate extracts the user name of the owner from the `attributeDict` dictionary using the attribute name (`owner`) as a key. It then invokes a private method that associates the owner with the imported Address Book data.

Handling Parsing Errors

When the parser encounters a syntactical error or any other problem in an XML document that prevents it from being well-formed, it stops parsing and sends a message to its delegate. The delegate, if it implements the `parser:parseErrorOccurred:` method, receives this message. In its implementation it should display a message informing users what the problem is. The parsing error is fatal (that is, unrecoverable) so informing the user is all that you can realistically accomplish. With this information, the user might be able to fix the XML so the document can be successfully parsed.

Listing 1 illustrates how you might implement `parser:parseErrorOccurred:.`

Listing 1 Handling parsing errors

```
- (void)parser:(NSXMLParser *)parser parseErrorOccurred:(NSError *)parseError
{
    NSWindow *modWin = [self windowForSheet];
    if (!modWin) modWin = [NSApp mainWindow];
    NSAlert *parserAlert = [[NSAlert alloc] init];
    [parserAlert setMessageText:@"Parsing Error!"];
    [parserAlert setInformativeText:[NSString stringWithFormat:@"Error %i,
        Description: %@, Line: %i, Column: %i", [parseError code],
        [[parser parserError] localizedDescription], [parser lineNumber],
        [parser columnNumber]]];
    [parserAlert addButtonWithTitle:@"OK"];
    [parserAlert beginSheetModalForWindow:modWin modalDelegate:self
        didEndSelector:@selector(alertDidEnd:returnCode:contextInfo:)
        contextInfo:nil];
    [parserAlert release];
}

- (void)alertDidEnd:(NSAlert *)alert returnCode:(int)returnCode contextInfo:(void
 *)contextInfo { }
```

The key line in this example is the one that constructs the NSAlert object's informative text. This text includes the error code (an `NSXMLParserErrorenum` constant), a localized description of the error, and a line number and column (nesting level) number isolating the location of the error in the XML document. In the example, the delegate obtains this information from two different sources: from the parser object itself (provided in the first parameter of the method) or from the NSError object provided in the second parameter. From the parser object it can also get an NSError object, and from that it can get a localized description.

However, the default localized description of NSError is rudimentary. You might want to provide your own localized description instead of relying on the one obtained from the NSError object. Sometimes parsing errors may require an application-specific interpretation. To implement a function or method for this purpose, you can use the `NSXMLParserError` constant defining the error to determine which custom key to use in the `NSLocalizedString` macro. (Of course, you must also create a `strings` file and do whatever else is necessary to internationalize your application.)

Using Multiple Delegates

For some XML documents, particularly large and complex documents, having a single delegate for the NSXMLParser object might not be the best approach. The code for handling all of the different parsing events can easily become overly intricate and hard to manage. One technique for making things more manageable is to share the work of handling parsing events among multiple delegates.

Take as an example an application that constructs a DOM-style tree from elements as it encounters them. Starting from the root element, one element creates a child element and passes off control to it by setting it to be the delegate. That child element creates its children (and so on), each time resetting the delegate appropriately. If an element has no children, or if it's a mixed element, it accumulates the textual content for itself. Finally, when the parser encounters an element's end tag, the element sets the delegate to be its parent element. Listing 1 shows the pertinent code that accomplishes this processing.

Listing 1 Resetting the delegate for the next element

```
- (void)parser:(NSXMLParser *)parser
        didStartElement:(NSString *)elementName
        namespaceURI:(NSString *)namespaceURI
        qualifiedName:(NSString *)qualifiedName
        attributes:(NSDictionary *)attributeDict {
    // Element is a custom class for object representing element nodes
    // Creation of element sets child as delegate (see below)
    [self addChild:[Element elementWithName:elementName
        attributes:attributeDict parent:self children:nil parser:parser]];
}

- (void)parser:(NSXMLParser *)parser foundCharacters:(NSString *)string {
    [self appendString:string];
}

- (void)parser:(NSXMLParser *)parser didEndElement:(NSString *)elementName
namespaceURI:(NSString *)namespaceURI qualifiedName:(NSString *)qName {
    Element *parent = [self parent];
    [parser setDelegate:parent]; // RESET DELEGATE TO PARENT
}

+ (id)elementWithName:(NSString *)elementName attributes:(NSDictionary
*)attributes parent:(Element *)parent children:(NSArray *)children
parser:(NSXMLParser *)parser {
    return [[[[self class] alloc] initWithName:elementName
        attributes:attributes parent:parent children:children
        parser:parser] autorelease];
}

- (id)initWithName:(NSString *)elementName attributes:(NSDictionary *)attributes
 parent:(id)parent children:(NSArray *)children parser:(NSXMLParser *)parser {
    self = [super init];
    if (self) {
        [self setName:elementName];
        if (attributes) {
```

```
            [self addAttributes:attributes];
        }
        [self setParent:parent];
        if (children) {
            [self addChildren:children];
        }
        [parser setDelegate:self]; // CHILD SET AS DELEGATE
    }
    return self;
}
```

Another technique for managing multiple delegates is maintaining a number of delegate objects, each with its specialized role, in a collection such as an NSDictionary object. These objects would know who their child and parent elements are in any given context and so would be able to set the delegate for the next element (using the appropriate dictionary key) after their work with the current element has finished.

Constructing XML Tree Structures

Generally, if you wish to add or modify the content of an XML document, you must construct a static tree structure that completely represents the elements and other constructs in the document. Tree representations are also essential if you intend to validate an XML document against the DTD (or other language schema) that prescribes the logical structure of the document.

When most developers want to construct DOM-style tree representations of XML documents, they use a tree-based parser, not a streaming parser such as NSXMLParser. (Tree-based parsing engines, however, are typically built on top of streaming parsers.) Nonetheless, that does not mean that you cannot create tree structures using an NSXMLParser instance. Although this article does not go into great detail about techniques for constructing XML tree structures using NSXMLParser, it outlines a general approach that you could take.

> **Note:** DOM (for Document Object Model) is a model proposed by the World Wide Web Consortium for describing XML and HTML documents using a standard set of objects. It also defines an interface for accessing and manipulating those objects, which represent (among other things) the elements of a document and the attributes associated with each element. The procedure discussed below does not make specific use of DOM, although there are similarities.

You can represent any XML document as a hierarchical tree whose "nodes" are elements exhibiting relationships of parent and child with other elements. Each element can have one or more children and, with the exception of the root element, has exactly one parent element. The tree is anchored by a root element, which is the only element in the tree without a parent. The "leaf" nodes of the tree are typically those elements containing nothing but text, although they can also be mixed elements or empty elements.

For example, consider the following short XML document:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE addresses SYSTEM "addresses.dtd">
<addresses>
    <person idnum="0123">
        <lastName>Doe</lastName>
        <firstName>John</firstName>
        <phone location="mobile">(201) 345-6789</phone>
        <email>jdoe@foo.com</email>
        <address>
            <street>100 Main Street</street>
            <city>Somewhere</city>
            <state>New Jersey</state>
            <zip>07670</zip>
        </address>
    </person>
</addresses>
```

The following tree of element nodes represents this document:

Figure 1   Tree representation of simple XML document
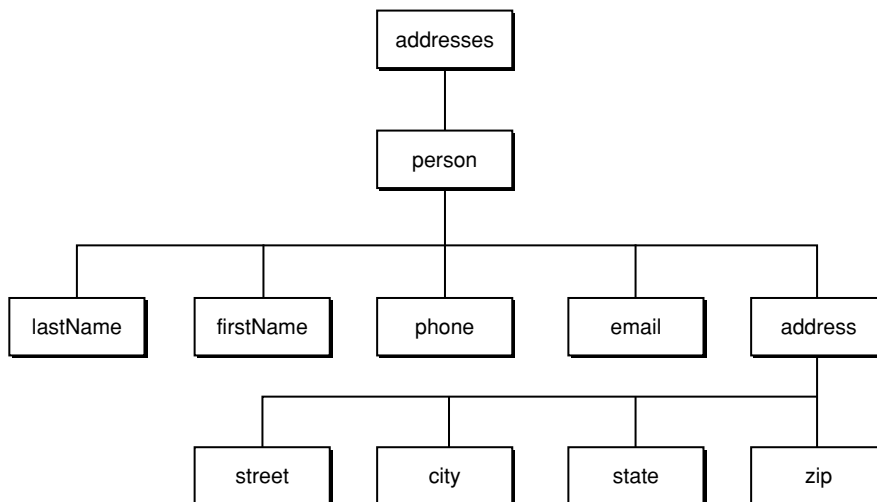
There are several possible ways to construct a tree representation of an XML document using NSXMLParser. This article, however, looks at a recursive, object-oriented approach that dynamically transfers delegation responsibilities among the objects representing the elements of a document. (This strategic shifting of the NSXMLParser delegate is discussed further in "Using Multiple Delegates" (page 19).) The programmatic result is doubly-linked lists of objects and arrays of objects; the abstract result is a tree representation of the document.

The procedure for constructing a tree using this approach entails the following steps:

1. Create a class whose instances represent the elements of an XML document. The class should define the name of the element and its parent (one-to-one) and children (one-to-many) relationships; it should also encapsulate the attributes associated with the element. As a shorthand notation for this procedure, we'll call this class MyElement.

2. From a top-level object in the application, load an XML document, create an NSXMLParser instance for it, assign the top-level object as delegate, and begin parsing the document (see "XML Parsing Basics" (page 11)).

3. The parser encounters the document's root element first and sends `parser:didStartElement:namespaceURI:qualifiedName:attributes:` to its delegate. The delegate creates a MyElement object to represent this root element and sets its parent to `nil`. The method that creates and initializes the object also sets it to be the new delegate of the NSXMLParser instance.

4. The parser encounters the next element of the document—the first child of the root element—and again sends the delegate `parser:didStartElement:namespaceURI:qualifiedName:attributes:`. The delegate is now the MyElement object recently created to represent the root element. It creates another MyElement object to represent the new element (in the process setting the new object to be the delegate and setting itself to be the parent) and adds the new object to its list of children.

5. The new delegate receives the next `parser:didStartElement:namespaceURI:qualifiedName:attributes:` message, identifying its first child element, and it creates it and adds it to its list of children.

6. This recursive descent through the first branch of the tree ends when the parser encounters "leaf" elements containing text, mixed content, or empty elements. If there is mixed content the descent is not truly over since `parser:didStartElement:namespaceURI:qualifiedName:attributes:` is sent to the delegate even after it receives `parser:foundCharacters:` for the current element. Processing depends on the kind of element:

   ■ If it's an empty element, processing skips ahead to the next step (end-element tag)

   ■ If there is only text associated with the current element node, the delegate responds to the `parser:foundCharacters:` message by accumulating text (in sequential `parser:foundCharacters:` invocations).

   ■ If there is mixed content, the delegate will process the text even after it receives messages notifying it of the start-element and end-element tags for the embedded elements. One way to handle this is to wrap the text in special text-element objects and insert these (in the proper order) in the element's child list.

7. Finally, the parser sends the `parser:didEndElement:namespaceURI:qualifiedName:` to the delegate, notifying it that the element is now complete. The delegate sets the new delegate to be its parent and returns.

8. If the parent has more children elements, the parser sends it the next `parser:didStartElement:namespaceURI:qualifiedName:attributes:` message; the parent MyElement object creates a MyElement instance to represent its next child (in the process setting it to be the new delegate and setting itself to be the parent of the new MyElement) and adds the newly created object to its list of children. However, if the parent has no more children to add to its list (that is, it receives the `parser:didEndElement:namespaceURI:qualifiedName:` message instead) it sets the new delegate to be its parent and returns.

9. The procedure continues in this fashion until the entire XML document is processed and all branches of the tree are constructed.

The objects that are the nodes of the tree (representing mostly elements) should be able to print themselves out as XML code. You application should also implement an algorithm that asks the objects to print themselves in the proper document sequence.

Validation Tips and Techniques

Validation is a procedure that ensures an XML document conforms to the rules governing its logical structure as specified in a language schema such as DTD (Document Type Definition). An XML document might be well-formed—that is, it obeys the syntactical rules of XML—and at the same time be invalid. For example, an element might include a child element when it is supposed to have only textual content, or a required attribute of an element might be missing.

To perform validation it helps to construct a tree of an XML document's schema that is parallel to a tree structure representing the document's actual content (see "Constructing XML Tree Structures" (page 21)). The schema tree presents a simple abstract view of how the document *should* be structured. Instead of nodes of objects representing the actual elements and text of the document, the schema tree contains nodes that express the rules by which the parts of the document can be combined. Validation tests the actual elements, attributes, and other parts of the document against the rules of the schema to see if the document conforms. If your application finds any violation of conformance, it can notify the user and perhaps require the user to fix the error. You can validate an XML document when it is first read and processed and later when users attempt to make any changes to it.

Because the programmatic interface of NSXMLParser is designed to report only XML constructs and DTD declarations, this article focuses on that language schema. However, if you use an XML-based language schema, such as RELAX NG, then NSXMLParser can process the schema just it would as any XML file, reporting what it finds to its delegate. You can use the data you thereby acquire for validation.

The sections on constructing rules focus primarily on element and attribute declarations because these are by far the most common and most important type of declaration. "Handling Other Declarations" (page 29) briefly discusses what to do with other kinds of declarations, such as those for entities and notations.

Using NSXMLParser to Handle DTD Declarations

The NSXMLParser class reports to its delegate DTD declarations it encounters in a document (assuming the delegate implements the necessary methods). If the language schema you use is DTD, NSXMLParser helps you acquire the data you need either for validation or for other purposes, such as enforcing correctness when dynamically constructing objects (for example, a menu template).

The DTD Delegation Methods

The NSXMLParser class defines a half dozen delegation methods that the parser invokes when the parser encounters a DTD declaration in a internal or external source. These methods are of the form:

```
parser:foundTypeDeclarationWithName:…
```

The third parameter and any subsequent parameters depend on the type of declaration. The following list briefly describes the NSXMLParser delegation methods related to DTD declarations.

- `parser:foundElementDeclarationWithName:model:`
    Example: `<!ELEMENT dictionary (documentation?, suite+)>`
- `parser:foundAttributeDeclarationWithName:forElement:type:defaultValue:`
    Example: `<!ATTLIST dictionary title CDATA #IMPLIED >`
- `parser:foundInternalEntityDeclarationWithName:value:`
    Example: `<!ENTITY % OSType "CDATA">`
- `parser:foundExternalEntityDeclarationWithName:publicID:systemID:`
    Example: `<!ENTITY name SYSTEM "name.xml">`
- `parser:foundNotationDeclarationWithName:publicID:systemID:`
    Example: `<!NOTATION img PUBLIC "urn:mime:image/jpeg">`
- `parser:foundUnparsedEntityDeclarationWithName:publicID:systemID: notationName:`
    Example: `<!ENTITY corplogo SYSTEM "logo.jpg" NDATA img>`

Resolving External DTD Entities

An XML document, in the `DOCTYPE` declaration that occurs near its beginning, often identifies an external DTD file whose declarations prescribe its logical structure. For example, the following `DOCTYPE` declaration says that the DTD related to the root element "addresses" can be located by the system identifier "addresses.dtd".

`<!DOCTYPE addresses SYSTEM "addresses.dtd">`

Often the system identifier assumes a standard file-system location for DTDs—for example, `/System/Library/DTDs`. At the start of processing, the NSXMLParser delegate is given an opportunity to resolve this external entity and give the parser a list of DTD declarations to parse.

1. When you prepare the NSXMLParser instance, send it the `setShouldResolveExternalEntities:` with an argument of `YES`.

2. Implement the delegation method `parser:resolveExternalEntityName:systemID:` to return the declarations in the external DTD file as an NSData object.

If the DTD declarations are internal to an XML document, then the delegate will receive the DTD-declaration messages automatically (assuming, of course, that it implements the related methods).

Constructing Rules for Elements

Just as elements are typically the most common kind of construct in an XML document, element declarations are the most common kind of declaration in a DTD. They express rules for the composition of elements from child elements, text, and other constituents.

An element declaration has three parts: the `!ELEMENT` keyword, the element name, and a content model. The content model is everything after the name up to the terminating angle bracket. Consider the following examples:

```
<!ELEMENT cocoa EMPTY>
<!ELEMENT keyboard (layouts+, modifierMap+, keyMapSet+, actions*, terminators*)>
```

```
<!ELEMENT dict (key, %plistObject;)*>
<!ELEMENT string (#PCDATA)>
```

The content model can specify no content (`EMPTY`), any content (`ANY`, which is rare), textual content (`#PCDATA`), and child elements. It may identify child elements by name or by an entity reference (such as `%plistObject;` in the third example above). The model can also specify mixed content—that is, the element can contain text and child elements in any order. Through occurrence modifiers (`*`, `+`, `?`) and other syntactical conventions, the content model can also specify the order of child elements, whether an element is required or optional, how many times an element may occur, and acceptable choices between elements. Occurrence modifiers can be applied to groups of elements (in parentheses) as well as individual elements.

The job required for validation is to examine the content model of an element declaration and derive rules for the composition of that element. As one approach, you might design classes for each type of rule as well as for the scope of a rule (individual element or group of elements). You could then associate instances of that rule class with an element through the name of the element. During validation the instances are queried with regard to a current or potential member of an element.

Table 1 lists the most important rules derivable from an element declaration's content model.

Table 1   Possible rules for element validation

Rule	Sample content model	Comments
Textual content only	`(#PCDATA)`	
Mixed content	`(#PCDATA | bold | italic)`	Vertical bars in this case have a meaning different from choice; when `#PCDATA` is present, they mean that text and child elements can be intermixed.
No content	`EMPTY`	For flag-type values.
Required sequence	`(name, address, phone)`	Commas indicate prescribed sequence.
Choice	`(read | write | readwrite)`	Without `#PCDATA` being a member (see Mixed content), the vertical bars mean that one of the listed elements must be used.
Occurs exactly once	`(name, address, phone)`	No modifier punctuation mark. Can apply to individual element or group.
Occurs zero or more times	`(%plistObject;)*`	Occurrence modifier is asterisk ("*"). Can apply to individual element or group.
Occurs one or more times	`(property+)`	Occurrence modifier is plus sign ("+"). Can apply to individual element or group.
Occurs zero or one time	`(%implementation;?)`	Occurrence modifier is question mark ("?"). Can apply to individual element or group.

Constructing Rules for Attributes

Elements frequently have attributes associated with them, and consequently attribute-list declarations are frequently encountered in DTDs. Attribute-list declarations specify the rules for attributes using a syntax that is different from element declarations. They specify, in order, the associated element, the name of the attribute, the type of the attribute, and a default value. For example, the declaration

```
<!ATTLIST modifierMap defaultIndex NMTOKEN #REQUIRED >
```

states that the `defaultIndex` attribute, which is associated with the `modifierMap` element, is of type `NMTOKEN` (meaning that it must be a valid XML name); the `#REQUIRED` keyword given as the default value means that a value for the attribute must be supplied.

When a NSXMLParser instance encounters an attribute-list declaration, it sends `parser:foundAttributeDeclarationWithName:forElement:type:defaultValue:` to its delegate. Passed in as parameters are attribute name, the associated element, the the attribute type, and its default value. The rules for attributes derive from combinations of the last two parameter (type and default value). Table 2 lists some the possible rules that you can construct from attribute-list declarations.

Table 2   Possible rules for attribute validation

Rule	Keywords or example	Type or default	Comments
Unique value	`ID`	type	The attribute value must be unique in the XML document.
Required value	`#REQUIRED`	default	The value of the attribute must be specified in the document.
Refers to unique attribute value	`IDREFIDREFS`	type	Value must refer to valid `ID`-type value elsewhere in document. `IDREFS` specifies a list of `ID` references (in parentheses).
Valid XML name	`NMTOKENNMTOKENS`	type	Value must be valid XML name (including entity references). `NMTOKENS` specifies a list of XML names (in parentheses).
Value is fixed	`#FIXED "value"`	default	Value must be "value".
Valid XML name in list	`(name |address | phone)`	type	Attribute enumeration: value must be one of the XML names in parentheses.
Valid defined type in list	`NOTATION (tiff | gif | jpg)`	type	Attribute enumeration: value must be one of the defined types in parentheses.

Handling Other Declarations

Other DTD declarations such as those for entities and notations are less common than element and attribute-list declarations. You can easily derive rule constructions for these other declarations after reviewing some DTD documentation. However, there are a couple of things to keep in mind:

■ You need to record entity declarations in case they are used as part of the content model for an element declaration.

■ Because notations can be made an attribute type, you should also keep track of them.

XML Glossary

This glossary defines some of the terms specific to XML, DTD, and related specifications and technologies. It focuses primarily on terms that are part of the names of methods and constants declared by the NSXMLParser, NSXMLNode, NSXMLDocument, NSXMLElement, NSXMLDTD, and NSXMLDTDNode classes.

atomic value
> A value with a simple type as defined by the XML Schema standard. The types include string, decimal, integer, float, double, Boolean, date, URI, array, and binary data. An XQuery query returns a **sequence** of items that can contain one or more nodes or atomic values.

attribute
> A property of an **element** expressed as a name-value pair. Attributes are used to encode data or provide metadata that is associated with an element. In the following example, "version" is the name of an attribute of element `plist` and its value is `"1.0"`:
>
> `<plist version="1.0">`

attribute list declaration
> Identifies in a DTD an element that has attributes, the names of those attributes, what values the attributes may have, and default values. Example:
>
> `<!ATTLIST phone location (home | office | mobile) "home">`
>
> In this example, `phone` is the element name, `location` is the attribute name, `(home | office | mobile)` is the allowable values, and `home` is the default value.

canonical
> A form of an XML document in which it can be compared against another document for equivalence. If two documents with differing physical representations have the same canonical form, they are considered logically equivalent within the given application context. The canonical form of an XML document is defined by the World Wide Web Consortium at `http://www.w3.org/TR/xml-c14n`.

CDATA block
> A section of text that the parser should pass uninterpreted to the client application. It appears as element content. CDATA blocks are often used for code or data that contains "prohibited" characters, that is characters of special syntactical significance to the parser (for example, "<" and "&"). You can also use an **entity reference** to express any of these prohibited characters (for example, `<`) is a built-in entity reference for specifying the "escaped" `<` character.

content model
> The part of an **element declaration** that defines what the element may contain. A content model consists of the names of child elements, `#PCDATA` (indicating text), entity references, or `EMPTY` (indicating an empty element such as `<true/>`). Child elements and `#PCDATA` are enclosed within parentheses. Commas between child elements specify that the elements must occur in the given sequence. The vertical-bar character ("|") instead of a comma indicates a logical OR relationship and can be used with `#PCDATA`. Occurrence modifiers can be applied to individual elements or groups of elements:
>
> ■ "+" indicates the element or group can be repeated more than once but must occur at lease once.
>
> ■ "?" indicates the element or group is optional and may occur only once.

31

- "*" indicates the element or group is optional and can occur more than once.

- No modifier indicates that the element or group must occur only once.

Examples of content models.

```
(#PCDATA)
(%plistObject)*
(lastName, middleInitial?, firstName, phone*)*
```

document order

The order of XML mark-up constructs as they appear in a document. When you send the NSXMLNode messages `nextNode` (or `previousNode`) to each successive node object encountered in an NSXML tree, you are traversing the tree forward (or backward) in document order.

DOM (Document Object Model)

An API for accessing and manipulating XML documents as tree structures. DOM derives from a World Wide Web Consortium recommendation for a general object model for storing hierarchically structured documents in memory.

DTD (Document Type Definition)

A way to define the legal elements and other building blocks of an XML document.

element

Markup tags that identify the nature of the content they surround. Elements have names and may contain textual data, child elements, **processing instructions**, comments, and **CDATA blocks**. An element has a single parent element, except for a document's root element, which has no parent. An element may also have **attributes** and **namespace prefixes** associated with it. Elements can also be empty (that is, without content) and the developer can use them as flags.

The following is an example of an element with an attribute and mixed content (in this case, text, a child element, and a CDATA block):

```
<para ref_num="80458">
    The following C++ code gives an example of how
    <code>cout</code> is used:
    <![CDATA[std::cout << "Hello, World!\n";
    >
</para>
```

element declaration

Specifies in a DTD the name of an element and what is permitted as content of the element. The declaration may specify child elements, text, and entity references as content. It prescribes the order of child elements and (for single elements or for the entire group) whether it is required and whether it can appear multiple times. Examples:

```
<!ELEMENT addresses (person)*>
<!ELEMENT person (lastName, firstName, phone*, email*, address*)>
<!ELEMENT lastName (#PCDATA)>
```

See also **content model**.

entity declaration

Associates in a DTD a name with some piece of XML content that is identified by an **entity reference**. That content can be a literal value (such as identified by a character reference), a variable value specified elsewhere in the DTD, or some textual or binary value referenced in an external file. The last type of entity is called an external entity. Examples:

```
<!ENTITY % plistObject "(array | date | dict | real | integer | string | true
| false )" >
<!ENTITY CompanyLogo SYSTEM "/Library/Images/logo.gif" NDATA GIF87A>
```

entity and character reference

> A reference in text to an externally or internally declared **entity declaration**. It must begin with an ampersand and end with a semicolon. You can refer to entities that you declare elsewhere. There are five predefined entities: "<", ">", "&", single-quote character, and double-quote character. Character references start with "&#" and are followed by numerical code points. Examples of references are `',>,ç`; the first two are built-in entity references and the last is a character reference. See also **unparsed entity**.

model

> See **content model**.

namespace

> A URI (Universal Resource Identifier) that qualifies an element or attribute name so as to avoid name conflicts when a document contains XML from different sources. You declare a namespace in the start tag of an element by appending a prefix to the predefined `xmlns` attribute (separated by a colon), and then associating this with the value of the URI; for example:

```
<h:table xmlns:h="http://www.w3.org/TR/html4/">
```

> Thereafter, you need only use a **namespace prefix** ("h" in the above example) with an element (separated by a colon) to identify the element unambiguously. All child elements of the element with the namespace declaration are associated with the same namespace through the prefix. The prefix-element name combination (`h:table` from the example above) is called a **qualified name**. A namespace declaration with no prefix after `xmlns` defines a default namespace, unless the value is an empty string, which means "no namespace." The URI in a namespace declaration doesn't have to point to anything; it is just a convenient way to get a unique name.

namespace prefix

> A prefix defined in a namespace declaration to identify the namespace a particular element is associated with. The namespace's qualified name (`xmlns`:*localname*) appears only during output. All other operations, such as those that get or set a namespace node's value, use the local name only. See also **namespace**.

normalize

> To coalesce all adjacent child text nodes into a single text node while removing empty text nodes. Normalization is highly recommended before performing XPath and XQuery queries.

notation

> Identifies by name the format either of an **unparsed entity** or an element bearing a specific notation attribute; it can also identify the target of a processing instruction. A notation declaration gives a name to the notation and an external identifier that enables a parser or its client to locate a helper application that can process the data specified by the notation. Notations occur in attribute values, attribute-list declarations, and entity declarations.

processing instruction

> A construct that provides information to the application processing the XML document. The instructions could instruct the application how, for example, to interpret the XML or display the results. Processing instructions can occur within elements or at the top level of a document. The first word of the processing instruction is called the target (its name) and every thing else is its object value. Example:

```
<?sort alpha-ascending?>
```

qualified name

> An element's full name, consisting of prefix, colon, and local name. See also **namespace**.

sequence

> A collection of items, each of which can be a node or an **atomic value**. XQuery queries return a sequence (an NSArray in Cocoa), which may contain only a single item.

33

validation

A procedure that checks an XML document against the logical structure described by declarations in the associated DTD (or other schema) to see if the XML conforms to it. Some of the constraints involved in validation are proper element sequence and nesting, specification of required attributes, and correct attribute type. For example, if an element is supposed to have one or more child elements but doesn't, the document containing the element is invalid. Before an XML document can be validated, it must first be **well-formed**.

unparsed entity

An external resource referred to by **entity reference** whose contents may be binary data or text (including non-XML text). Each unparsed entity has a **notation** associated with it.

well-formed

Refers to an XML document that obeys the syntax of XML. A parser cannot parse a document if its XML is not well-formed. Some of the checks for well-formedness are:

- Element start tags must have end tags (except for empty elements).

- Attribute values must be quoted.

- Parameter entities must be declared before they are used.

- Markup constructs appear only where permitted.

XHTML

A more strictly prescribed version of HTML that makes it well-formed XML. XHTML is an official World Wide Web Consortium recommendation.

XPath

An XML query language for locating nodes with an XML tree structure. It allows location paths, predicates, and general expressions in queries. The Cocoa implementation uses XPath 2.0, which is a World Wide Web Consortium recommendation. The NSXMLNode class enables XPath queries through its `nodesForXPath:error:` method. (Note that the NSXML classes do not support deprecated XPath 1.0 features such as namespace axis.)

XQuery

A flexible and powerful XML query language that lets you compose logically complex queries using operators, quantifiers, functions and FLOWR expressions (referring to the keywords `for`, `let`, `order by`, `where`, and `return`). The NSXMLNode class enables XQuery 1.0 queries through its `objectsForXQuery:error:` method

XSLT (Extensible Stylesheet Language Transformations)

An XML application for transforming an XML document into another XML document or into an HTML, RTF, or plain-text document. The stylesheet used in a transformation has template rules, each consisting of a pattern and a template. The NSXMLDocument class permits access to XSLT through its `objectByApplyingXSLT:error:` and `objectByApplyingXSLTAtURL:error:` methods.

Document Revision History

This table describes the changes to *Event-Driven XML Programming Guide*.

Date	Notes
2010-03-24	Update example code to new initializer pattern.
2008-09-09	Added note about introduction of namespace support in v10.4.
2006-12-05	Added memory management guideline and corrected code examples.
2005-04-29	Updated the glossary of XML terms. Changed title from "Event-Driven XML Parsing." Changed "Rendezvous" to "Bonjour."
	Updated the XML glossary to define additional terms primarily related to the NSXML set of classes. This glossary is shared with *Tree-Based XML Programming Guide*.
2004-07-27	Minor bug fix.
2004-01-21	First version of *Event-Driven XML Parsing*.