Tree-Based XML Programming Guide

Data Management

2009-02-04

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010

Simultaneously published in the United States and Canada.

Contents

Figures, Tables, and Listings

Introduction to Tree-Based XML Programming Guide for Cocoa

Note: This document was previously titled *Tree-Based XML Processing*.

XML is a ubiquitous and flexible markup standard for processing and exchanging data. You can find XML in property lists, as the file format of various applications, and as the format of various sources of information on the Internet, including web-based services. The NSXML classes of Foundation give you a way to process this information efficiently. NSXML logically represents an XML document as a hierarchical tree structure and allows you to query this structure and manipulate its nodes. It supports several XML-related technologies and standards, such as XQuery, XPath, XInclude, XSLT, DTD, and XHTML.

This document explains how you can use NSXML effectively. You might find this information valuable if you need to create, modify, and repeatedly query XML documents. If you simply need to parse XML and extract information from an existing source of XML, the NSXMLParser class is more suited to your needs. For information on using the NSXMLParser class, which is the Cocoa interface to a streaming XML parser, see *Event-Driven XML Programming Guide*.

Organization of This Document

This document includes the following articles:

- "NSXML and XML Processing" (page 9) describes the features and capabilities of NSXML and suggests how you can use it in your applications.
- "The Data Model of NSXML" (page 13) describes the XQuery-derived data model on which NSXML is based.
- "Creating an XML Document Object" (page 19) explains how to process input sources of XML, how to create an XML document programmatically, and how to transform an existing XML tree structure using XSLT.
- "Writing XML From NSXML Objects" (page 23) describes how to request an XML document, or a branch of a document, to emit its represented content as XML markup text.
- "Traversing an XML Tree" (page 27) explains how to use NSXML programmatic interfaces to move around within an XML tree structure.
- "Querying an XML Document" (page 29) explains how to use the methods that allow you to perform XQuery and XPath queries to locate particular nodes and objects in an XML tree structure.
- "Modifying an XML Document" (page 37) describes how to add, remove, and replace nodes in an XML tree and now to change the values of nodes.
- "Representing Object Values as Strings" (page 43) describes the string representations for the standard atomic types and explains how you can obtain string representations of custom objects used as node values.

- "Handling Attributes and Namespaces" (page 47) discusses how to use the methods that are specific to attribute nodes and namespace nodes.
- "Binding NSXML Objects to a User Interface" (page 55) examines a sample NSXML application to illustrate how you can establish bindings between NSXML objects and a user interface.
- "Using Tree Controllers With NSXML Objects" (page 63) shows how you can establish bindings between an NSXML document, an `NSTreeController` object, and an `NSOutlineView` object.

A XML Glossary (page 71) of XML terms is also included.

See Also

Many sources of information on XML and related standards and technologies are available on the Internet, including the following documents from the World Wide Web Consortium (W3C) website:

- "XQuery 1.0 and XPath 2.0 Data Model"—http://www.w3.org/TR/xpath-datamodel/
- "XMLSchema Part 1: Structures"—http://www.w3.org/TR/xmlschema-1/
- "XML Schema Part 2: Datatypes —http://www.w3.org/TR/xmlschema-2/
- "XML Query Use Cases"—http://www.w3.org/TR/xquery-use-cases/
- "XML Path Language (XPath) 2.0"—http://www.w3.org/TR/xpath20/
- "DOM Object Model Core"—http://www.w3.org/TR/2003/CR-DOM-Level-3-Core-20031107/core.html

The following are useful websites on XML:

- The Annotated XML Specification—http://www.xml.com/axml/testaxml.htm
- O'Reilly XML.com—http://www.xml.com/

Several XQuery tutorials are listed at the end of "Querying an XML Document" (page 29) under"Resources For Learning XQuery" (page 35).

NSXML and XML Processing

With the NSXML set of Foundation classes you can create, manipulate, query, and modify XML documents of various types, including webpages, configuration files, and XML-formatted data files. NSXML operates on abstract, logical tree structures that represent XML documents. You can have these tree representations write themselves out as XML documents. You can also convert them into other XML trees using XSLT. Input documents, output documents, or transformed documents can be HTML as well as XML. With NSXML you can also internalize a DTD (Document Type Definition) as a tree structure and validate an XML document against its DTD.

As a technology, NSXML also includes support for XQuery 1.0 and XPath 2.0, which enable you to perform queries on XML documents. For more information on XQuery and XPath, see "XQuery and Other XML Technologies" (page 11)

Capabilities of NSXML

XML is a ubiquitous and increasingly important document-markup format for structuring information that can be applied to virtually any computing purpose. The format is so flexible that XML applications can include technologies as diverse as publishing, electronic data interchange, network management, and vector graphics. The attraction of XML is apparent: It is a text-based, structured, cross-platform storage format for data of any sort.

You can use the NSXML API in your own applications when you need to process information in XML-formatted sources. With NSXML you can create, alter, query, and transform XML documents or website pages in just a few lines of code. The DOM-style tree-based data model of NSXML enables you to insert, delete, and modify nodes at any point in a tree. (For a discussion of this model, see "The Data Model of NSXML" (page 13).) You can either have NSXML read an existing XML document into an internal tree structure, or you can create a tree representation from scratch. NSXML lets you search for particular nodes and values in the tree by either walking the tree or using the XQuery or XPath query languages. When you have finished working with an XML document, you can ask the tree representation to print itself out as ordered and properly structured XML or XHTML code.

Architecturally, NSXML depends on an event-driven XML parser to parse input XML documents before it converts them into tree structures. The public Cocoa interface to this parser is the NSXMLParser class.

NSXML is not the best solution for all situations where XML must be processed. Internal tree representations can take up a lot of application memory, especially for operations such as validation and XSLT transformations. If you simply need to find certain values in an XML document, and don't need a persistent representation of the XML to modify, then the better Cocoa alternative is the event-driven parsing model offered by the NSXMLParser class.

In addition to the methods that allow you to create and manipulate the nodes and node values in DOM-style tree structures, NSXML has many other features, including the following:

- **Object values**. Nodes of all kinds, but particularly elements and attributes, can have values associated with them. These values are usually strings, but you can specify object values for nodes that represent non-string types, such as decimal, `float`, and date. Your application can, for example, interpret the format of a string value and convert it to an NSCalendarDate or NSNumber object. When you ask for the string value of an NSXML object, and its value is set as an object, NSXML returns the string formatted as a canonical type according to the W3C XML Schema: Datatypes specification (http://www.w3.org/TR/xmlschema-2/). You can also define the string representations of custom objects.
- **DTD nodes**. NSXML parses internal Document Type Definitions (DTDs) and composes trees representing the structure of the declarations. Among the nodes of such trees are objects representing entities, attribute-list declarations, and element declarations. You can programmatically modify such trees or even create them from scratch and then write out the new or modified DTD. When NSXML processes an XML document it can, if requested, validate it against an internal or external DTD. See "DTD and Other Schemas" (page 12) for further information.
- **Namespaces**. NSXML allows you to distinguish between elements and attributes in different namespaces in an XML document. With methods in the NSXMLNode and NSXMLElement classes you can create namespace nodes, associate them with an element and its attributes, query elements for their declared namespaces, and have the prefixes of namespaces resolve dynamically against the namespace URIs and local names associated with attributes and elements.
- **Tidy XML and HTML**. NSXML uses an open-source tool that checks and corrects HTML and XML documents to make them compliant with the Worldwide Web Consortium (W3C) standards. When HTML is "tidied," it is converted to XHTML. You request tidy XML or HTML as an option in initialization and output methods.
- **XQuery** and **XPath** . With XQuery or XPath, you can quickly locate particular nodes or values in an XML document and you can transform or create XML nodes or even documents. See "XQuery and Other XML Technologies" (page 11) for more information.
- **XSLT**. NSXML lets you apply XSLT (Extensible StyleSheet Language Transformations) to an XML document object (NSXMLDocument). XSLT transforms one XML document into another XML document or a document in another format, such as XHTML, HTML, RTF, or plain text. It does this by applying a set of patterns and template rules. "XQuery and Other XML Technologies" (page 11) discusses XSLT in greater detail.
- **XInclude**. NSXML supports XInclude, allowing you to merge XML from other sources into the current document.
- **Bindings**. The classes of NSXML conform to the key-value coding protocol and respond to observers conforming to the key-value observing protocol. Consequently you can bind the attributes of NSXML objects to properties of user-interface objects via the Cocoa controller classes.

The Classes of NSXML

The public interface of NSXML consists of the five Foundation classes listed in Table 1. NSXML fully supports the XML standard and can efficiently process the largest of XML documents. It relies on existing Foundation classes to avoid redundancy in its programmatic interface.

Table 1 NSXML Classes

NSXMLNode
NSXMLDocument

NSXMLElement
NSXMLDTD
NSXMLDTDNode

Note: Even though the NSXMLParser has the same prefix as the NSXML classes, it is not part of NSXML. It is based on a different model for processing XML (see *Event-Driven XML Programming Guide*).

The first three of these classes in Table 1 are for processing XML. As defined by the data model of XQuery (described in "The Data Model of NSXML" (page 13)) instances of these classes either represent the various kinds of nodes of an XML tree or, in the case of the document node, the entire tree itself. NSXML node object represent documents, elements, attributes, namespaces, comments, processing instructions, and text nodes.

The last two classes in Table 1 are for creating and modifying Document Type Definitions. For a discussion of the DTD-related nodes, see "DTD and Other Schemas" (page 12).

An obvious advantage of an object-oriented framework is that you can extend and specialize behavior through subclassing. This advantage applies to NSXML. For example, if processing instructions play a particularly important role in your application, you could create a subclass of NSXMLNode whose instances represent processing-instruction nodes capable of performing the required tasks. NSXML allows you to substitute your subclass for an NSXML class when the tree is built during the parsing phase. For more information on subclassing the NSXML classes, see the reference documentation.

Within the group of classes listed in Table 1, NSXMLNode is the base class—all of the other classes directly inherit from it. NSXMLNode defines an interface and a set of attributes common to NSXML node objects. Among these are the kind of node, the node name, the string or object value of the node, the location of the node relative to its sibling nodes, the level of the node in the tree hierarchy, and references to the node's parent and children. Through NSXMLNode, a node can find the nodes adjacent to it in the tree; it can print itself out as XML (or DTD) markup text; and it can be the context object for XQuery and XPath queries.

XQuery and Other XML Technologies

NSXML includes support for XQuery 1.0, a query language that you can use to retrieve and interpret information from different sources of XML. A functional and strongly typed language, XQuery operates on the abstract, logical structure of an XML document—its tree representation—rather than on its surface syntax. (This logical structure is informed by the data model discussed in "The Data Model of NSXML" (page 13).) The result of an XQuery query is a sequence of items, each of which is either a node (NSXML object) or an atomic value (string, integer, float, date, and so on).

Note: For more information on XQuery 1.0 and XPath 2.0, see the World Wide Web Consortium specifications on the language (http://www.w3.org/TR/2003/WD-xquery-20031112/) and on functions and operators (http://www.w3.org/TR/2003/WD-xpath-functions-20031112/). Also, see "Resources For Learning XQuery" (page 35), which lists recommended tutorials on XQuery.

The basic syntactical unit in XQuery is the expression, which is made up of symbols, keywords, and operands (which are always other expressions). Embedded XPath expressions locate nodes in the XML tree using specific criteria. FLWOR expressions (for the keywords `for`, `let`, `where`, `order`, and `return`) make richer

and more precise operations possible, including sorting, joins, and hierarchy inversions. XQuery also enables node construction (although you cannot use it to attach or otherwise manipulate such a node within an NSXML tree). You can use a number of built-in XQuery functions, such as `replace`, `distinct-values`, and `avg`, and you can create your own custom functions.

NSXML also supports XPath as a query language. You can use XPath to locate nodes in an XML tree based on position, relative position, node name, node kind, and several other criteria. Because XQuery 1.0 includes XPath 2.0, a syntactically valid path expression returns the same result in both languages.

NSXML exposes access to XQuery and XPath through two methods of the NSXMLNode class:

```
- (NSArray *)nodesForXPath:(NSString *)xpath error:(NSError **)error;
- (NSArray *)objectsForXQuery:(NSString *)xquery constants:(NSDictionary
*)constants error:(NSError **)error;
```

The node object receiving these messages is the context node for the query. Note that the sequence (NSArray object) that an XPath query returns always contains nodes, never atomic values.

NSXML also gives your code access to the Extensible Stylesheet Language Transformation (XSLT) technology. With XSLT, you can create a stylesheet that specifies the patterns and template rules for changing XML in an input tree to differently structured XML in an output tree, or to HTML, XHTML, plain text, or other forms of output. Then an XSLT processor carries out the transformation. A major use for XSLT is transforming an XML document into an XHTML or HTML document. NSXML gives you access to XSLT through the following NSXMLDocument methods:

```
- (id)objectByApplyingXSLT:(NSString *)xslt error:(NSError **)error;
- (id)objectByApplyingXSLTAtURL:(NSURL *)xsltURL arguments:(NSDictionary
*)argument error:(NSError **)error;
```

DTD and Other Schemas

NSXML provides some support for XML validation and for creating and modifying Document Type Definitions (DTDs).

Two NSXML classes, NSXMLDTD and NSXMLDTDNode, allow you create and modify DTDs as a shallow (two-level) tree structure. An instance of the NSXMLDTD class is analogous to an NSXMLDocument in that it represents the entire DTD. It functions as a root node to which instances of the NSXMLDTDNode class are added to as children (along with any comment nodes or processing-instruction nodes). NSXMLDTDNode objects represent element, attribute-list, and entity declarations of various kinds. When you read an XML document with an internal DTD, NSXML processes that DTD, creating a tree representation from it that is composed of NSXMLDTD and NSXMLDTDNode objects (as well as any comment nodes or processing-instruction nodes). Any tree you modify or create you can write out as a DTD document.

NSXML can validate documents when it initially processes them and later upon request. When you read and process an existing XML document and it has an associated schema (XML Schema or internal or external DTD), you can specify an initialization option requesting validation. If the document is successfully parsed and validated, the initialization method returns an NSXMLDocument object. If validation doesn't succeed, the method reports this as an error and does not create the document object. You can also validate a document as you modify it; if a change is invalid, NSXML reports the reasons for invalidity.

The Data Model of NSXML

An XML document is similar to an outline. Items in the outline are in a certain sequence and have certain hierarchical relationships with surrounding items. Similarly, order and hierarchy are the structural determinants of an XML document. Because of the hierarchical nature of XML markup, a tree structure is a natural abstraction for representing it. Yet even with the static tree representation, there is an order among the nodes in the tree that corresponds to their order in the markup text.

NSXML represents an XML document as an ordered, labeled tree in which each node has a unique identity and may have a value, attributes, and namespaces associated with it. Conceptually, NSXML is based on an enhanced data model of XQuery 1.0/XPath 2.0, which themself have affinities with the DOM Core standard. As does XQuery, NSXML operates on the abstract, logical structure of an XML document—the data model—rather than its surface syntax. In XQuery, each input or output to or from a query is an instance of the data model. This model consists of two general kinds of items: nodes and atomic values.

Nodes

The NSXML data model represents an XML document as a tree of nodes. The tree can have various kinds of nodes, each of which corresponds to a type of XML construct:

- Element
- Attribute
- Text
- Comment
- Processing instruction
- Namespace

An XML tree also has the notion of a document, an entity that represents the entire tree.

NSXML objects have an attribute that specifies their kind. A node's kind is permanently set at its creation; it cannot be changed into another kind of node.

One thing you might notice about the above list is that it does not contain all possible kinds of XML-markup constructs. The most notable omissions are CDATA sections and character and entity references. When NSXML processes an existing document, it resolves any character or entity references into standard text nodes—unless the appropriate fidelity options are set. Even when CDATA and character and entity references are preserved, NSXML treats them as no more than special-cased text nodes.

Node Name, Index, and Level

Each node in a tree has a unique identity. Most nodes have a name—document, comment, and text nodes do not—and many nodes have some string or other type of value associated with them. Thus a comment node, which doesn't have a name, almost always has a value. If there is another comment node in the tree with the same value, it is considered to be different because it occurs in a different part of the tree. Even if two nodes are of the same kind and have the same name and content, NSXML treats them as distinct nodes because they have different locations within the tree.

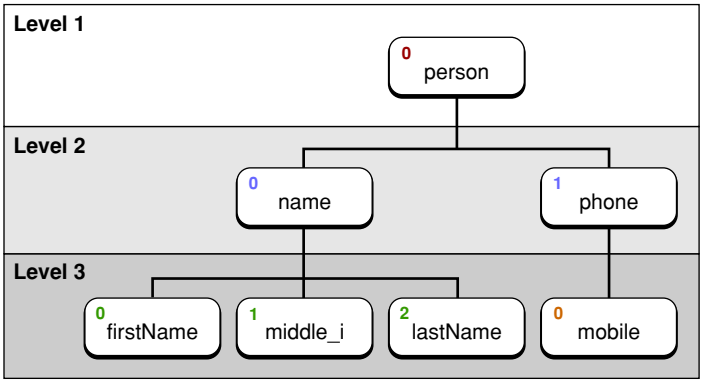
NSXML uses two node attributes to determine node location: index and level. The index is a zero-based number indicating a node's relative position to its sibling nodes (all children of the same parent node). The level is a number indicating a node's nesting level in the document hierarchy; the root-element node always has a level of 1 (and is the only node with this level number).

To see how these numbering schemes might play out, consider this simple XML document:

```
<person>
   <name>
      <firstName>John</firstName>
      <middle_i>J</middle_i>
      <lastName>Doe</lastName>
   </name>
   <phone>
      <mobile>(408) 362-4593</mobile>
   </phone>
</person>
```

After processing this document, NSXML represents it as a tree structure. Figure 1 shows the index and level of each node in this small tree (ignoring text nodes).

Figure 1 Node level and index



NSXML changes a node's level and index as the node changes location in a tree. For example, you could detach a node from one parent and attach it to another, and its level and index would change.

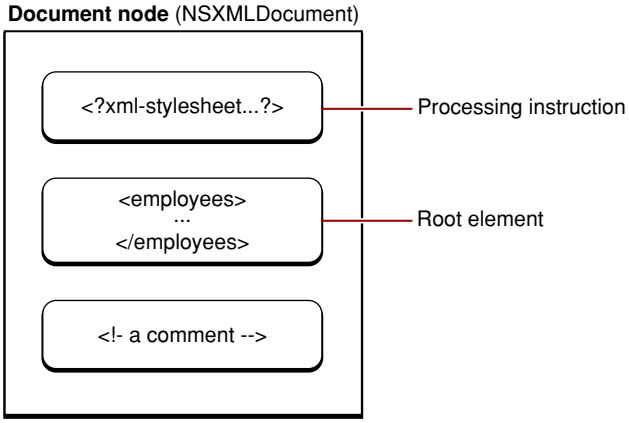
The Document Node

An NSXML tree representation of an XML document has one, and only one, document node. The document node is always first in any tree representation, and it is more than the first node in the tree. The document node encompasses the entire document. It represents the document itself.

The document node contains only one element, but that element is the root element, the element at the top of the tree. The root element is the only element in the document that has no parent element. All other elements "descend" from it. If you intend to process an internalized XML document by walking the tree, you would start from the root element.

However, a document node can contain nodes other than the root element. It can have child nodes representing processing instructions and comments (see Figure 2). A document node can also have document metadata associated with it, such as a URI or MIME type, a DTD, or the encoding or version specified in the XML declaration for the document. NSXML also allows you to specify the kind of output for a document, that is, whether a document writes out XML, XHTML, or HTML markup, or just plain text.

Figure 2 The document node



Element Nodes and Other Children

Element nodes are the most important nodes in a tree. Elements are the main structural ingredient for the information expressed by an XML document. Except for document nodes (see "The Document Node" (page 15)) and DTD nodes, only element nodes can have children. The kinds of nodes that may be children of an element node are text nodes, processing-instruction nodes, comment nodes, and other element nodes. Text nodes are a nameless, generic type of node that carry the text between the start and end tags of an element. For example, consider the following element:

```
<title>War and Peace</title>
```

The `title` element in this case has a single child, a text node with the content (string value) of "War and Peace." If an element has mixed content—that is, text intermixed with child nodes—and you request NSXML to normalize the content, it gathers the content of all child nodes into one text node. Take as an example the following XML:

```
<para>The novel <title>War and Peace</title> is huge.</para>
```

When the `para` element is normalized, the text in all child elements is integrated with the text node of the `para` element. The `title` element is discarded.

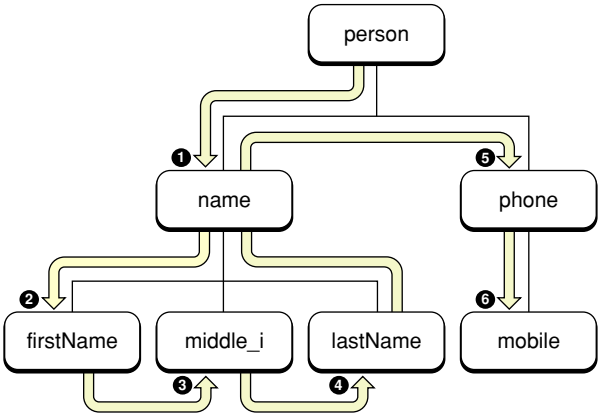
Although nodes representing attributes and namespaces may have an element as a parent in the tree, they are not considered true children of that element. They are integrally associated with an element and are not a separate item. When NSXML traverses an XML tree in document order, it passes over attribute nodes and namespace nodes. You must query an element (using NSXMLElement methods) to find out about an element's namespaces and attributes.

Document Order

Document order is, simply put, 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.

With a programmatic view of a tree, the order of traversal is determined by a "child first, sibling next" (or "depth first") logic. In other words, NSXML progresses through a tree by descending to the first child of the current parent (if any). If that node has its own children, the next node visited is the first child of that parent. Once NSXML reaches a leaf node (a node with no children), it proceeds to any sibling node—that is, a node with the same parent. If there are no (or no more) sibling nodes, NSXML returns one level and goes to the next child node of the previous parent (if any). Traversal of nodes continues in this way until the final leaf node the tree (or section of the tree) has been visited. Figure 3 depicts this traversal in document order.

Figure 3 Document order



Memory Management of NSXML Node Objects

The memory management of node objects in NSXML is similar to that performed by collection objects. A parent node manages the retained state of its "contained" objects—that is, its children. This management can be summarized as follows:

- When you add or insert a node, the parent retains it.
- When you remove a node, the parent releases it.
- When you detach a node, the node is retained and then autoreleased.

When NSXML parses a source of XML and builds a tree, the nodes in the tree are retained by their parents. Each of these nodes has a retain count of one.

The implications of this behavior for object ownership are clear. If you want to hold onto a node object that is removed or detached from a tree, you must ensure that it has the proper retain count. For the complete details of object ownership policy and management of object life spans, see *Memory Management Programming Guide*.

For memory-management and performance purposes, you should retain node objects rather than copy them. Nodes only need to be copied if they are part of a tree (that is, they have a parent) and you want to clone the node to a new location in the current tree or in a different tree.

Sequences and Atomic Values

In addition to nodes, the XQuery data model also allows atomic values.Atomic values are values having simple types as defined in the XML Schema standard: string, decimal, integer, float, double, Boolean, date, URI, array, and binary data. In NSXML, Foundation objects that are equivalent to these atomic values—for example, NSString, NSNumber, and NSCalendarDate objects—form the content, or object value, of nodes.

Atomic values can also be part of the input and output of XQuery queries. XQuery treats every value in a query as part of a sequence. A sequence is a collection of items, each of which can be a node or an atomic value. If a query in XQuery returns only one item, it will be returned in a sequence (or array) of one. The notion of sequences is why the XQuery and XPath methods of NSXMLNode—`objectsForXQuery:error:` and `nodesForXPath:error:`—return an NSArray object.

Creating an XML Document Object

You can create an object representing an XML document—that is, an instance of NSXMLDocument—in one of two general ways. Using NSXML, you can either parse an XML source and create a tree structure representative of the contents, or you can create an XML document programmatically.

Creating a Document Object From Existing XML

The NSXMLDocument class lets you create a document instance from XML markup text in one of several forms: as a string object, as an NSData object, and as a URL-designated file (remote or local). The following initializers of NSXMLDocument (corresponding to these forms) also include parameters for input-handling options and for obtaining information about initialization errors:

```
initWithXMLString:options:error:
initWithData:options:error:
initWithContentsOfURL:options:error:
```

Listing 1 shows how you might create an NSXMLDocument instance from an absolute path to a local XML file.

Listing 1 Creating a document from a file URL (local file)

```
- (void)createXMLDocumentFromFile:(NSString *)file {
    NSXMLDocument *xmlDoc;
    NSError *err=nil;
    NSURL *furl = [NSURL fileURLWithPath:file];
    if (!furl) {
        NSLog(@"Can't create an URL from file %@.", file);
        return;
    }
    xmlDoc = [[NSXMLDocument alloc] initWithContentsOfURL:furl
            options:(NSXMLNodePreserveWhitespace|NSXMLNodePreserveCDATA)
            error:&err];
    if (xmlDoc == nil) {
        xmlDoc = [[NSXMLDocument alloc] initWithContentsOfURL:furl
                    options:NSXMLDocumentTidyXML
                    error:&err];
    }
    if (xmlDoc == nil)  {
        if (err) {
            [self handleError:err];
        }
        return;
    }

    if (err) {
        [self handleError:err];
```

```
    }
}
```

If initialization fails, the NSXMLDocument `init` method directly returns `nil` and most likely returns by indirection an NSError reporting a parsing or connection error. If failure is due to a parsing error, you can try to recover (as in Listing 1) by initializing the document with the `NSXMLDocumentTidyXML` option; this option reformats the document prior to parsing to ensure that it is well-formed. However, it does have side effects, such as removing leading and trailing spaces; see the NSXMLDocument reference documentation for details.

If an NSXMLDocument `init` method indirectly returns an NSError object in its third parameter, you can display an alert dialog to inform the user about the error. However, NSError objects created for parsing errors or warnings might not contain much helpful information. Most NSError objects encapsulate errors or warnings emitted by the parser when documents are not well-formed; these can also be connection errors from attempts to access XML at a remote locations using `initWithContentsOfURL:options:error:`.

You set initialization options in the second parameter of the three NSXMLDocument initializers discussed above; if you want to specify multiple options, use a bitwise-OR expression. The initialization options cover three categories of input-handling:

- **Preservation**. The `enum` constants beginning with `NSXMLNodePreserve` support document fidelity. They allow you to ensure that aspects of the string XML input—for instance, quoting style of attribute values, CDATA sections, attribute and namespace order—remain the same when XML text is written out from the NSXMLDocument object. If you do not specify a preservation option for an aspect, NSXMLDocument handles it in a predetermined manner. For example, when NSXMLDocument reads in and parses an XML document, it normally strips white-space characters used in formatting (including tabs and carriage returns); however, if you specify `NSXMLNodePreserveWhitespace` these characters are kept hidden in the internal representation of the XML document. (“Hidden” means the white-space characters are retained for the output of the document, but are not visible within the tree representation.) Note that white space in text nodes is always preserved, and is not affected by the `NSXMLNodePreserveWhitespace` option.
- **Tidiness**. The `NSXMLDocumentTidyHTML` and `NSXMLDocumentTidyXML` options correct any malformed parts of a source HTML or XML document, respectively, before creating the internal representation of the document. Tidying maintains document content, but not necessarily maintain white space or the order of constructs in the document. Entities and character references are always resolved to their text or unicode character.If the source is HTML and `NSXMLDocumentTidyHTML` is specified, NSXMLDocument attempts to convert it to XHTML; corrections take into account the HTML context. If the source is XML and `NSXMLDocumentTidyXML` is specified, NSXMLDocument attempts to fix any defects to make it well-formed.
- **Validation**. The `NSXMLDocumentValidate` option validates the source XML document against its associated schema. That schema can be defined through a DTD (internal or external) or XML Schema. The validator reports reasons for document invalidity indirectly through the NSError parameter. To validate against an XML Schema, you must identify the schema document through a combination of namespace, `xmlns:xsi`, and `xsi:schemaLocation` attribute declarations in the root element. The URI value of the namespace must match one of the values in the location-namespace tuples in the `xsi:schemaLocation` declaration. Here is an example:

```
<MyRootElement xmlns="http://www.example.com/foo"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation=
        "http://www.example.com/foo http://www.example.com/foo.xsd
        http://www.w3.org/1999/xhtml http://www.w3.org/1999/xhtml.xsd">
```

For information on how you specify the location of schema definitions on the web, see the W3C XML Schema recommendation (http://www.w3.org/TR/xmlschema-1/). A schema should be physically installed on a Mac OS X system at `/System/Library/Schemas/`*elementName*`.xsd`.

- **XInclude**. If the `NSXMLDocumentXInclude` option is specified, the parser replaces `xsi:include` elements that refer to other XML documents (or parts of documents) with the referenced content. By default this option is turned on so your application can query and edit the nodes that define inclusions.

These `enum` constants are defined in `NSXMLNodeOptions.h`. For specific information about the initialization options, check the reference documentation for NSXMLDocument.

Creating a New XML Document

Sometimes you might want to create an XML document that is entirely new, without there being any original source file or website page. You may want to manipulate all of the nodes in a tree structure—adding, removing, and relocating them—and let NSXML take care of the formatting details when the document is written to a file.

The `initWithRootElement:` initializer of NSXMLDocument allows you to do this. An NSXMLDocument object has two defining kinds of data: a set of document-global attributes and a root element. Once you have the root element you can construct the rest of the document's tree representation by adding child nodes of various types to parent nodes at various levels.

Listing 2 shows an example of creating a brand-new XML document object.

Listing 2 Creating an NSXMLDocument instance representing a new document

```
NSXMLElement *root =
    (NSXMLElement *)[NSXMLNode elementWithName:@"addresses"];
NSXMLDocument *xmlDoc = [[NSXMLDocument alloc] initWithRootElement:root];
[xmlDoc setVersion:@"1.0"];
[xmlDoc setCharacterEncoding:@"UTF-8"];
[root addChild:[NSXMLNode commentWithStringValue:@"Hello world!"]];
```

This fragment of code uses methods of NSXMLDocument to set the `version` and `encoding` attributes of the XML declaration of the document. You could also set the standalone, URI, and MIME-type attributes using the corresponding NSXMLDocument "setter" methods. You may associate a Document Type Definition with the NSXMLDocument object through the `setDTD:` method. NSXMLDocument also allows you to specify the document's output format (using `setDocumentContentKind:`) as HTML, XHTML, text, or (the default) XML. For example, the following lists the different handlings of the `<br>` element:

- XML: `<br></br>`
- XHTML: `<br />`
- HTML: `<br>`
- Plain text: nothing

Once you create an XML document object and set its attributes, you can start adding content to it—that is, the elements, attributes, and other nodes that make up an XML document—through the appropriate methods of NSXMLElement and NSXMLNode.

If the XML document has a schema associated with it—defined either with a DTD or XML Schema —you might want to validate the document periodically. To validate a document, send the `validateAndReturnError:` message to the NSXMLDocument object. If the operation is not valid, the method returns `NO`; the reasons for invalidity are returned by indirection in the method's NSError parameter.

Writing XML From NSXML Objects

All NSXML objects can emit a textual representation of themselves as XML markup. In addition, NSXMLDocument objects can write themselves out as complete XML or XHTML documents. NSXML provides different sets of methods and options (as `enum` constants) for both writing out the contents of NSXMLNode and NSXMLDocument objects. It also offers methods for performing XSLT transformations on XML documents.

Writing Out an XML Document

Perhaps the most common pattern of an application that processes XML is something like the following:

1. Read an XML document from a file or website, converting it to an internal tree structure.
2. Modify, add, and delete nodes in the tree.
3. From the objects in the tree, write a new or updated XML document to a file or website.

Two methods of the NSXMLDocument class help you with the final step, `XMLData` and `XMLDataWithOptions:`. As the method names imply, both methods return an NSData object. The character data of the output is encoded based on the encoding specified as an attribute in the document's XML declaration. (If that doesn't exist, NSXML uses UTF-8 as a default encoding.)

`XMLData` simply invokes `XMLDataWithOptions:` with an argument of `NSXMLNodeOptionsNone`, requesting no output options. The latter method allows you to specify the following `enum`-constant options:

- `NSXMLNodePrettyPrint`—Makes the XML output more human-readable by, for example, inserting carriage returns and indenting nested elements.
- `NSXMLDocumentIncludeContentTypeDeclaration`—Includes a content type declaration for HTML or XHTML output.

If you want to specify multiple constants, use a bitwise-OR expression.

Listing 2 shows how you might invoke the `XMLDataWithOptions:` method in the `writeToFile:ofType:` method of an NSDocument subclass.

Listing 1 Writing an XML document to a file (NSDocument subclass)

```
- (BOOL)writeToFile:(NSString *)fileName ofType:(NSString *)type {

    NSData *xmlData = [xmlDoc XMLDataWithOptions:NSXMLNodePrettyPrint];
    if (![xmlData writeToFile:fileName atomically:YES]) {
        NSBeep();
        NSLog(@"Could not write document out...");
        return NO;
    }
```

```
    return YES;
}
```

Writing the XML Represented by Nodes

Occasionally you might want to have specific nodes or branches of an XML tree express themselves as XML markup rather than the entire document. For example, an application could walk through a large XML document looking for nodes matching specific criteria, and then construct a smaller XML document from the output of these nodes.

The primary NSXMLNode methods that you use for this purpose are `XMLString` and `XMLStringWithOptions:`. The former method simply invokes the latter specifying no output options (`NSXMLNodeOptionsNone`). Aside from being invokable on any NSXMLNode object (including NSXMLDocument objects), what distinguishes these methods from their NSXMLDocument counterparts is that they return a string object (NSString) rather than a data object (NSData).

When you send one of these messages to an NSXMLNode object that can (and does) have children—specifically, document, element, or document-type declaration nodes—the effect is recursive. All descendants of the node object print themselves out in document order.

`XMLStringWithOptions:` permits only one of the output options available to the NSXMLDocument method `XMLDataWithOptions:` — `NSXMLNodePrettyPrint` — for inserting whitespace to improve XML readability.

The method in Listing 2 gets the string representation of an XML document and uses that for the content of a text view and a web view. (WebView and WebFrame, which are referenced in the following two code listings, are classes of the Web Kit framework.)

Listing 2 Creating a second XML document using XMLStringWithOptions:

```
- (IBAction)reloadPreview:(id)sender {
    if (xmlDocument) {
        NSString *displayString = [xmlDocument
            XMLStringWithOptions:NSXMLNodePrettyPrint];
        [previewTextView setString:displayString];
        [[previewWebView mainFrame] loadHTMLString:displayString
            baseURL:[self baseDocURL]];
    }
}
```

You can obtain XML markup text that is in canonical form by sending any NSXMLNode object the message `canonicalXMLStringPreservingComments:`. By converting an XML document or node to canonical form you can determine if it is logically equivalent to another XML document (or node) in canonical form—which is perhaps the same document after minor changes are made to it. Putting XML in canonical form involves various conversions and "normalizations," such as resolving character and entity references, converting empty element tags to start-end pairs, and inserting default attribute values. See the reference documentation for the NSXMLNode class for a fuller description of `canonicalXMLStringPreservingComments:`.

Note: When you ask an NSXMLNode object to return its XML representation, and the node is one with content (such as an element), the XML representation includes the string value of that content. If an application had previously set the content of the node to an object value, the string value is derived from value transformers defined for the type of object value. For further information, see "Representing Object Values as Strings" (page 43).

Transforming a Document With XSLT

NSXML includes support for XSLT. With XSLT you can apply a set of template rules and patterns to a source XML document and thereby transform that document into another XML document or into an HTML, RTF, or plain-text document. Three NSXMLDocument methods are the interface to XSLT processing:

```
- (id)objectByApplyingXSLT:(NSData *)xslt arguments:(NSDictionary *)arguments
error:(NSError **)error
- (id)objectByApplyingXSLTString:(NSString *)xslt arguments:(NSDictionary
*)arguments error:(NSError **)error
- (id)objectByApplyingXSLTAtURL:(NSURL *)xsltURL arguments:(NSDictionary
*)arguments error:(NSError **)error
```

These methods return transformed XML or HTML documents in the form of an NSXMLDocument object; they return RTF or plain-text documents as NSData objects.

The first of these methods takes the XSLT code as a data object, the second as a string object, and the third takes a URL identifying a source of XSLT code. Listing 3 illustrates the usage of the third method. In this case, the file containing the XSLT code is stored as a nonlocalized resource of the application. The method gets the path to the XSLT file, converts the path to a URL, and then sends the `objectByApplyingXSLTAtURL:arguments:error:` method to an NSXMLDocument object. It gets the HTML contents of the resulting document object using `XMLString`; it then loads that string into a WebView object from which it is printed.

Listing 3 Transforming an XML document to HTML using XSLT

```
- (IBAction)printDocument:(id)sender {
    NSError *err=nil;
    // webView is an off-screen WebView (ivar)
    WebFrame *frame = [webView mainFrame];

    // find XSLT code
    NSString *xsltPath = [[NSBundle mainBundle]
        pathForResource:@"addresses_transform" ofType:@"xml"];
    if (!xsltPath)
        return;

    // transform through XSLT
    NSXMLDocument  *printDoc = (NSXMLDocument *)[xmlDoc
        objectByApplyingXSLTAtURL:[NSURL fileURLWithPath:xsltPath]
        arguments:nil  // no extra XSLT parameters needed
        error:&err];
    if (err) {
        [self handleError:err];
        if (!printDoc) return;
```

```
    }
    // put in WebFrame and print
    [frame loadHTMLString:[printDoc XMLString] baseURL:nil];
    [webView print:self];
}
```

Many online tutorials on XSLT are available, including one from W3Schools (http://www.w3schools.com/xsl/). You can search the web to find other sources of information on XSLT.

Traversing an XML Tree

NSXML gives you several ways to explore the tree structure representing an XML document and find nodes that are of interest. Each approach is based on a different conceptual model:

- Traversing nodes in document order
- Traversing the children of a node
- Accessing element nodes by name

Traversing a tree in document order is perhaps the simplest approach. You start at some node in the tree—specifically the root element if you want to go through the entire document—and iteratively invoke `nextNode` to get the next node in document order until you reach the end of the document. Along the way you can query nodes for attributes of interest, such as name, value, kind, or level. Listing 1 illustrates the use of `nextNode`, in this case extracting comments that are used as translation notes for the string value of the subsequent node.

Listing 1 Walking an XML tree with nextNode messages

```
NSXMLNode *aNode = [xmlDoc rootElement];
NSMutableString *translator_notes=nil;
while (aNode = [aNode nextNode]) {
    if ( [aNode kind] == NSXMLCommentKind ) {
        if (!translator_notes) {
            translator_notes = [[NSMutableString alloc] init];
        }
        [translator_notes appendString:[aNode stringValue]];
        [translator_notes appendString:@" ========> "];
        aNode = [aNode nextNode]; // element to be translated
        [translator_notes appendString:[aNode stringValue]];
        [translator_notes appendString:@"\n"];
    }
}
if (translator_notes) {
    [translator_notes writeToFile:[NSString
stringWithFormat:@"%@/translator_notes.txt", NSHomeDirectory()] atomically:YES];
    [translator_notes release];
}
```

Of course you can also go backward in document order by repeatedly sending `previousNode` to each returned node object.

While `nextNode` and `previousNode` take you sequentially through a represented XML document, many other NSXMLNode methods help you navigate hierarchically within an XML tree structure, between parent and children nodes and among the sibling nodes of a common parent. These methods include `children`, `childCount`, `childAtIndex:`, `nextSibling`, and `previousSibling`. The following code-fragment examples show how these methods might be used in combination to traverse sibling nodes and accomplish some task. In Listing 2, the `children` and `childCount` methods are used, along with the NSArray `objectAtIndex:` method.

Listing 2 Using the children and childCount methods to traverse sibling nodes

```
NSArray *children = [[xmlDoc rootElement] children];
int i, count = [children count];
for (i=0; i < count; i++) {
    NSXMLNode *child = [children objectAtIndex:i];
    [self doSomethingWithNode:child];
}
```

If you can, use the NSXMLNode `childCount` method to obtain the number of child nodes instead of sending `count` to the result of `children`. The former method offers better performance. The code example in Listing 3 is slightly simpler and bypasses the `children` method altogether.

Listing 3 Using the childAtIndex: and childCount methods to traverse sibling nodes

```
int i, count = [[xmlDoc rootElement] childCount];
for (i=0; i < count; i++) {
    NSXMLNode *child = [[xmlDoc rootElement] childAtIndex:i];
    [self doSomethingWithNode:child];
}
```

An even simpler approach to the same task is illustrated in Listing 4, which uses the NSXMLNode `childAtIndex:` and `nextSibling` methods.

Listing 4 Using the childAtIndex: and nextSibling methods to traverse sibling nodes

```
NSXMLNode *child = [[xmlDoc rootElement] childAtIndex:0];
do {
    [self doSomethingWithNode:child];
} while ( child = [child nextSibling] );
```

For going upward in the tree hierarchy, from child node to parent, you have the `parent` method. This is the only method needed for this direction because, except for the root element and standalone nodes, there is almost always a one-to-one relationship from a child to its parent in an XML tree. (Namespace and attribute nodes are also an exception to this relationship rule because they have an element as a parent but are not children of that element.)

If you want a more directed search, you can use the `elementsForName:` method. If you know the name of a child element, send `elementsForName:` to the parent element. This method returns the child NSXMLElement nodes with matching names in an array (an NSArray object is used in case more than one child has the specified name). If you have to deal with namespace-qualified elements, use the `elementsForLocalName:URI:` method instead.

Querying an XML Document

One way to find items of interest in an XML document is to use the NSXML methods that traverse the nodes in a document tree (see "Traversing an XML Tree" (page 27)). However, this can be a time-consuming approach, especially with large documents. A more efficient strategy is to perform XPath and XQuery queries on the document object or any of the nodes in the document. This article describes the basic steps for executing XQuery and XPath queries and suggests how you can integrate such queries with an application's user interface.

XPath and XQuery Basics

XQuery 1.0 and XPath 2.0 are query languages so well integrated that it is easy to think of them as one language. Their formal semantics, data model, and functions and operators are defined in the same W3C specifications (see "See Also" (page 8) in the introduction). However there are distinctions, the primary one being that XPath uses a POSIX-style path syntax to describe the locations of nodes within an XML tree. In addition, XPath deals with nodes, while XQuery deals with nodes *and* atomic values.

The NSXMLNode class has two query-related methods, one for making XPath queries and the other for making XQuery queries:

```
- (NSArray *)nodesForXPath:(NSString *)xpath error:(NSError **)error
- (NSArray *)objectsForXQuery:(NSString *)xquery constants:(NSDictionary
*)constants error:(NSError **)error
```

Both methods return an array of found items—corresponding to a sequence in the data model—but the nature of the items differs, and this difference is reflected in the method names. The `nodesForXPath:error:` method returns an array of NSXMLNode objects while the `objectsForXQuery:constants:error:` method returns an array potentially containing both NSXMLNode objects and Foundation objects corresponding to the atomic types (NSNumber, NSString, NSCalendarDate, and so on). Even if these methods find only one item, they return it in an array.

Both methods also start a query with reference to an initial *context node*. The context node is the node against which the evaluation of a location path or other expression is applied. In both methods the initial context node is the receiver of the message. In a query expression, you can also refer to the context node with a period (for example, `".//products"`).

You often begin an XPath expression with a double slash (`"//"`) followed by an element name. This gives you all the descendent elements of the context node with that name, regardless of their level in the hierarchy. (You can also use a double slash elsewhere in a path.) Single slashes, followed by an element name, indicate a singular path down the tree hierarchy. Predicates are Boolean tests within square brackets that select a subset of the nodes as evaluated to that point. Numbers within square brackets after elements identify particular child nodes by their index (which in this case is 1-based). To see this in practice, consider the following path expression:

```
.//part/chapter[1]/section[title="Path Expressions"]
```

The evaluation of a path expression works from left to right. This expression first gets all the elements named `part` and from that selects the first element named `chapter`; from that it gets all child elements named `section`, and from that sequence it returns the section element whose title is "Path Expressions". XPath also lets you find attributes by name by using an at-sign (@) prefix. For example, the following path expression gets the modification date (an attribute) of the first chapter:

```
.//part/chapter[1]/@modDate
```

XPath also has function-like type specifiers that let you refer to child nodes other than elements and attributes, including text nodes (`text()`), processing instructions (`processing-instruction()`), and comments (`comment()`). If a parent has more than one node of a given type, all are returned.

Listing 1 is a fragment of code that illustrates how to make an XPath query using `nodesForXPath:error:`. It references a `city` element, which is a child of an `address` element, which is the third child element named `person` of the context node. If there are multiple elements named `city` at the end of this path, all of them are returned.

Listing 1 Executing an XPath query

```
NSError *err=nil;
NSXMLElement *thisCity;
NSArray *nodes = [theDocument nodesForXPath:@"./person[3]/address/city"
        error:&err];
if ([nodes count] > 0 ) {
    thisCity = [nodes objectAtIndex:0];
    // do something with element
}
if (err != nil) {
    [self handleError:err];
}
```

The initial context node for this query, an NSXMLDocument object (`theDocument`), is the receiver of the message. XPath returns the node or nodes that it finds in an array (a sequence). The code in Listing 1 is interested only in extracting the first node in the array, which it knows to be an NSXMLElement object. (Other kinds of child nodes can also be returned.) If XPath has trouble processing the query string—for example, the query has a syntax error—it directly returns `nil` and indirectly returns an NSError object. This code merely passes that object to another method to handle.

The NSXMLNode class defines an `XPath` method that can be quite useful when making XPath queries. As the name suggests, you can send an `XPath` message to any node object to get an XPath string describing that node's location in a tree. You can send or cache this string so that the node can easily be retrieved later via an XPath query. One possible scenario is that when a node changes its location within a tree, you can broadcast a notification that contains the new location as an XPath string in the `userInfo` dictionary.

XQuery is a flexible and powerful query language that encompasses XPath. XQuery lets you compose logically complex queries using operators, quantifiers, functions and FLOWR expressions (referring to the keywords `for`, `let`, `order by`, `where`, and `return`). With XQuery you can sort returned values, construct nodes, perform joins, invert hierarchies, and dynamically create new XML documents.

As an example, consider the simple query shown in Listing 2.

Listing 2 A simple XQuery query

```
for $p in .//person
where $p/address/zip_code > 90000
order by $p/last_name
```

```
return $p
```

This query cycles through every descendent element of the context node named `person` and evaluates whether the child element at path `/address/zip_code` has a value greater than 90000. It sorts the elements that satisfy this test by the value of their `last_name` child element and returns the resulting sequence.

Executing an XQuery query in NSXML is largely a matter of passing in the query string when invoking the `objectsForXQuery:constants:error:` method. The example method in Listing 3 gets the string from a text view in the user interface.

Listing 3 Executing an XQuery query

```
- (IBAction)applyXQuery:(id)sender {
    if (document) {
        NSError *error;
        NSArray *result = [document objectsForXQuery:
            [xquerySourceTextView string] constants:nil error:&error];
        if (result) {
            unsigned count = [result count];
            unsigned i;
            NSMutableString *stringResult = [[NSMutableString alloc] init];
            for (i = 0; i < count; i++) {
                [stringResult appendString:
                    [NSString stringWithFormat:@"%d: {\r", i]];
                [stringResult appendString:[[result objectAtIndex:i]
                    description]];
                [stringResult appendString:@"\r}\r"];
            }
            [xqueryResultTextView setString:stringResult];
            [stringResult release];
        } else if (error) {
            [self handleError:error];
        }
    }
}
```

This method applies the user-supplied query and formats the results before displaying it in another text view. If there is an error processing the request, it displays information about the error in an alert panel.

The second parameter of `objectsForXQuery:constants:error:` takes an NSDictionary object whose keys are the name of variables defined as external. The value of such a key is assigned as the value of the variable when it is used in a query. Through this mechanism, the constants dictionary lets you reuse a query string containing a variable whose value can change for each separate execution of a query. For more about the constants dictionary and its potential uses for a user interface, see the following section.

Integrating XQuery Into the User Interface

Although you can make use of XPath and XQuery in your application, you can't expect users to know enough about these query languages to construct their own queries. To use these languages effectively in your application, you need to find a way to incorporate the user's intent into each query. There are various ways you could go about this. Two of them are discussed here, the constants dictionary and formatted strings.

The Constants Dictionary

The constants dictionary, introduced in "XPath and XQuery Basics" (page 29), allows you to assign values to external variables in an XQuery query. You declare the variables in the prolog of the query and reference the variables where required in the query. (A prolog is a series of declarations and imports that creates the environment for query processing; it includes such things as variable definitions, module declarations, and schema imports.) The query can be reused as many times as you want. Then you create a dictionary containing key-value pairs where the key is the name of the variable and the value is whatever you want it to be. The value can be derived directly from a user-interface object.

With the constants dictionary you can, for instance, have a query that looks up user-supplied terms from an online dictionary. Figure 1 gives a simple example of the user interface for this query.

Figure 1 Simple user interface for a terminology query

With the constants dictionary, you can assign the value of the term field to an external variable in the query string.

Let's say a controller object connects the text field to an outlet named `termField` and connects the button to an action method. The following steps show how you might (in the action method) integrate the term variable into the query string:

1. Put the following line in the prolog of a query:

   ```
   declare variable $term as xs:string external;
   ```

2. Insert the variable (`$term`) in the query where you want the value to be used or evaluated.

3. Create a dictionary with a key of the same name as the variable and a value obtained from the user interface:

   ```
   NSDictionary *dict = [NSDictionary dictionaryWithObject:[termField stringValue]
    forKey:@"term"];
   ```

4. Send the `objectsForXQuery:constants:error:` method to the context node, passing in the dictionary:

   ```
   NSArray *result = [document objectsForXQuery:queryString constants:dict
   error:&err];
   ```

Keep in mind that external variables have assigned values, similar to the following expression:

```
let $term := "a value"
```

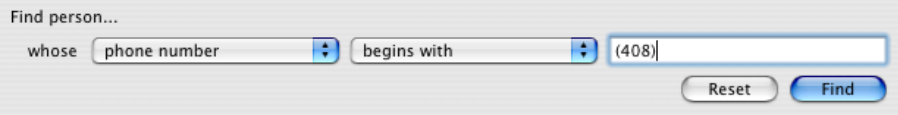
External variables are not like macros where the value is substituted for a placeholder. Consequently, you cannot use the constants-dictionary mechanism to do things such as dynamically changing the XQuery functions used in a query string.

Formatted Strings

The `stringWithFormat:` class method of NSString is a powerful tool. With it you can compose strings whose components can vary because of external factors, such as input from users. You can use format XQuery query strings in the same way. And unlike external variables, whose values are assigned from the constants dictionary, the "var-args" values in a formatted string are substituted for their place holders. Thus, formatted strings enable you to change language-related parts of the query string dynamically, including operators, function names, and path expressions.

To see how this might work, it helps to follow a simple example. Figure 2 shows a possible user interface for performing an XQuery query on an XML document.

Figure 2 User interface for a more complex query



The choices and entries a user makes with the pop-up lists and the text field become part of the query executed when the user clicks the Find button. For the specific choices and entry in the example above, the query string becomes the following:

```
for $p in //person
where starts-with($p/phone/text(), "(408)")
order by $p/lastName
return $p
```

For the pop-up lists, the incorporated values are "represented objects" associated with the items (NSMenuItem objects) of the lists. The represented objects of the first list's items are XPath path expressions relative to a context node; the represented objects of the second list's items are either operators or names of XQuery functions. Listing 4 is a method that dynamically creates the items of the second pop-up list and sets their represented objects.

Listing 4 Setting the represented objects for pop-up list items

```
- (IBAction)changeOperationsList:(id)sender {
    [operationPopUp removeAllItems];

    if ([elementPopUp indexOfSelectedItem] > 4) { // operators
        NSMenuItem *anItem;
        [operationPopUp addItemsWithTitles:[NSArray arrayWithObjects:
            @"equals", @"greater than", @"less than", nil]];

        anItem = (NSMenuItem *)[operationPopUp itemAtIndex:0];
        if (anItem) {
            [anItem setRepresentedObject:@"="];
        }
        anItem = (NSMenuItem *)[operationPopUp itemAtIndex:1];
        if (anItem) {
            [anItem setRepresentedObject:@">"];
        }
        // continued ...
```

```
    } else { // functions
        NSMenuItem *anItem;
        [operationPopUp addItemsWithTitles:[NSArray arrayWithObjects:
            @"is", @"contains", @"begins with", @"ends with", nil]];
        anItem = (NSMenuItem *)[operationPopUp itemAtIndex:0];
        if (anItem) {
            [anItem setRepresentedObject:@"matches"];
            [anItem setTag:XQFunction];
        }
        anItem = (NSMenuItem *)[operationPopUp itemAtIndex:1];
        if (anItem) {
            [anItem setRepresentedObject:@"contains"];
            [anItem setTag:XQFunction];
        }
        // continued ...
    }
}
```

When the user chooses popup-list items, enters a value for comparison in the text field, and clicks the Find button, the action method shown in Listing 5 is invoked. This method composes the query string—differently, according to whether a function or operator is required—from the represented objects and the value of the text field. It then sends the `objectsForXQuery:constants:error:` message to the context node (the document object) to execute the query.

Listing 5 Composing and executing the query

```
- (IBAction)findByXQuery:(id)sender {

    NSString *queryString;
    NSArray *results;
    NSError *err=nil;
    [queryStatus setStringValue:@""];
    if ([[operationPopUp selectedItem] tag] == XQFunction ) {
        queryString = [NSString stringWithFormat:@" \
            for $p in //person \
            where %@($p/%@/text(), \"%@\") \
            order by $p/lastName \
            return $p",
            [[operationPopUp selectedItem] representedObject],
            [[elementPopUp selectedItem] representedObject],
            [queryValue stringValue]];

    } else {
        queryString = [NSString stringWithFormat:@" \
            for $p in //person \
            where $p/%@/text() %@ \"%@\" \
            order by $p/lastName \
            return $p",
            [[elementPopUp selectedItem] representedObject],
            [[operationPopUp selectedItem] representedObject],
            [queryValue stringValue]];

    }
    results = [xmlDoc objectsForXQuery:queryString constants:nil
              error:&err];
    if (results && [results count] > 0) {
        [self doSomethingWithQueryResults:results];
    } else {
```

```
        [queryStatus setStringValue:[NSString stringWithFormat:
            @"No records found, query errors: %@",
            (err ? [err localizedDescription] : @"None")]];
    }
}
```

Instead of fetching the string value of the `queryValue` field this way, you could declare an external variable in the query prolog, put the value of the `queryValue` field in the constants dictionary, and reference the variable in the query. See "The Constants Dictionary" (page 32) for details on this approach.

Resources For Learning XQuery

XQuery is a fairly complex language, and although this document summarizes the salient aspects of syntax and behavior, it does not attempt a thorough description of XQuery. Fortunately a number of excellent online and downloadable tutorials are available, as a search of the Internet reveals. Here are some notable XQuery tutorials:

- "XQuery: A Guided Tour," DataDirect Technologies, downloadable (PDF) from http://www.datadirect.com/news/whatsnew/xquerybook/index.ssp

  This tutorial is excerpted from *XQuery from the Experts: A Guide to the W3C XML Query Language* by Howard Katz, Don Chamberlin, Denise Draper, Mary Fernandez, Michael Kay, Jonathan Robie, Michael Rys, Jerome Simeon, Jim Tivy, and Philip Wadler, (ISBN 0-321-18060-7), Addison-Wesley, 2004.
- "XQuery Tutorial," W3Schools, http://www.w3schools.com/xquery/default.asp
- "XQuery Tutorial," Ipedo, Inc., http://www.ipedo.com/html/xquery/xquery_tutorial/index.html

Because resources on the web might occasionally become unavailable or outdated, it's a good idea to perform your own web searches now and then for XQuery information.

Modifying an XML Document

You can modify an XML document with NSXML methods in two general ways:

- By changing the values of node objects
- By adding, removing, or replacing nodes, or by changing their locations within the tree

This article discusses both aspects of XML modification.

Note: Although some of the information contained in this article applies to attribute and namespace nodes, "Handling Attributes and Namespaces" (page 47) discusses how to create and modify these types of nodes in more detail.

Changing the Values of Nodes

Most kinds of NSXMLNode objects have values associated with them. The NSXMLNode class lets you set a node's value either to a string object or to some other type of object. The two major value-setting NSXMLNode methods reflect this choice:

```
- (void)setStringValue:(NSString *)string;
- (void)setObjectValue:(id)value;
```

Note: NSXMLNode also declares the `setStringValue:resolvingEntities:` method which, if the `resolvingEntities:` parameter is `YES`, resolves character references and references to predefined and user-defined entities in the string. (References to user-defined entities require an associated DTD in which the entity is declared). The `setStringValue:` method of NSXMLNode invokes this primitive method with a "resolvingEntities" flag of `NO`.

When NSXML processes an input file or other source of XML, it automatically sets the values of nodes as string objects. (It has no other indication of the types of values other than the form it finds them in: strings.) Keeping the value of nodes as strings is acceptable for most applications. Changing the string value is simply a matter of finding a particular node and sending `setStringValue:` or `setStringValue:resolvingEntities:` to it. The value itself often comes from the user interface, as shown in Listing 1.

Listing 1 Setting the string value of a node

```
int index = [form indexOfSelectedItem];
NSXMLNode *node;
switch (index) {
    case 0: // idField
        [[person attributeForName:@"idnum"] setStringValue:
            [idField stringValue]];
```

```
        break;
    case 1: // lastName
        node = [[person elementsForName:@"lastName"] objectAtIndex:0];
        if (!node) {
            node = [NSXMLElement elementWithName:@"lastName"];
            [person addChild:node];
        }
        [node setStringValue:[lastName stringValue]];
        break;
// ...
```

Setting the value of a node to an object other than an NSString object can give you certain advantages. As an example, suppose your application has element nodes for such financial values as item price, quantity, tax, and total. By converting the values of these nodes from strings (`"45.99"`) to numbers (`45.99`), your application can perform calculations directly with the object values of the nodes.

You can set any arbitrary object as the object value of the node. When an NSXMLNode object is asked to emit its object value as an XML string, it produces a string representation of the value. For the standard atomic types, this value is in a canonical form for the type (as defined by the W3C XML Schema Data Types specification). To take advantage of this default behavior, the object you set should be an instance of a Foundation class that corresponds to one of the atomic types in the XQuery data model. Table 1 lists these classes and their corresponding types.

Table 1 Equivalent classes and atomic types

Foundation class	Atomic type
NSString	string (use `setObjectValue:`)
NSNumber	integer, decimal, float, double, Boolean
NSCalendarDate	date
NSData	base64 and hexadecimal binary
NSURL	URI
NSArray	`NMTOKENS`, `IDREFS`, `ENTITIES`

You can also create value transformers that render string values representative of custom objects. For more on this subject, and for further information on the standard string representations, see "Representing Object Values as Strings" (page 43).

You must set the object value of each node in a document yourself—NSXML does not do it for you automatically. Typically you would first set object values globally after NSXML processes a source of XML and produces a tree representation of the document. You can walk the tree or otherwise search for the nodes of the document whose values you want to set to objects. If thereafter you create new nodes, you can set their object values at that point.

The code fragment in Listing 2 uses the NSTableView data-source method `tableView:objectValueForTableColumn:row:` as the point in which to set the object values of attribute nodes representing dates. The string values of these attributes must conform to a pattern of "mm/dd/yy" where "mm" is the month number, "dd" is the day of the month, and "yy" is the last two digits of the year.

(You can attach an NSDateFormatter object to an input field to ensure that newly entered dates conform to this pattern too.) The code fragment uses the NSCalendarDate method `dateWithString:calendarFormat:` to effect the conversion from string value to object value.

Listing 2 Setting an object value based on a string pattern

```
// in tableView:objectValueForTableColumn:row:...
 if ([columnID isEqualToString:@"Hire Date"]) {
        NSXMLNode *attrNode = [[employeeArray objectAtIndex:rowIndex]
            attributeForName:@"hireDate"];
        if (attrNode) {
            if (![[attrNode objectValue] isKindOfClass:
                [NSCalendarDate class]]) {
                NSCalendarDate *date = [NSCalendarDate dateWithString:
                    [attrNode stringValue] calendarFormat:@"%m/%d/%y"];
                [attrNode setObjectValue:date];
            }
            return [attrNode objectValue];
        }
    }
// ...
```

Once the object values of the `hireDate` attributes have been set, the application can go about comparing dates, computing periods between dates, and anything else that can be done with the methods of the NSCalendarDate and NSDate classes. Listing 3 is a method that uses the NSCalendarDate object values to identify the employee that has worked the longest for a hypothetical company.

Listing 3 Using the object value of an attribute

```
- (NSXMLElement *)earliestHire {

    NSXMLElement *earliest;
    int i;
    NSError *err;
    NSArray *nodes = [xmlDoc nodesForXPath:@".//employee" error:&err];
    if (!nodes || [nodes count] == 0)
        return nil;
    earliest = [nodes objectAtIndex:0];
    for (i = 1; i < [nodes count]; i++) {
        NSXMLElement *current = [nodes objectAtIndex:i];
        id date1 = [[earliest attributeForName:@"hireDate"] objectValue];
        id date2 = [[current attributeForName:@"hireDate"] objectValue];
        if ([date1 isKindOfClass:[NSDate class]] &&
            [date2 isKindOfClass:[NSDate class]] ) {
            if ([(NSDate *)date2 compare:(NSDate *)date1]
                        == NSOrderedAscending) {
                earliest = current;
            }
        }
    }
    return earliest;
}
```

Important: Setting the string or object value of an NSXMLNode object removes all existing children, including processing instructions and comments. Setting the string or object value of an NSXMLElement object creates a text node as the sole child.

Adding, Removing, and Relocating Nodes

In addition to modifying an XML document's content, you can modify a document's structure using NSXML methods. Structure modification entails adding, deleting, and replacing nodes, as well as moving existing nodes to new locations in the XML tree. Table 2 lists the methods that enable you to perform these operations.

Table 2 NSXML methods for manipulating nodes

Method	Declared in classes
`detach`	NSXMLNode
`addChild:`	NSXMLElement, NSXMLDocument, and NSXMLDTD
`setChildren:`	NSXMLElement, NSXMLDocument, and NSXMLDTD
`insertChild:atIndex:`	NSXMLElement, NSXMLDocument, and NSXMLDTD
`insertChildren: atIndex:`	NSXMLElement, NSXMLDocument, and NSXMLDTD
`removeChildAtIndex:`	NSXMLElement, NSXMLDocument, and NSXMLDTD
`replaceChildAtIndex: withNode:`	NSXMLElement, NSXMLDocument, and NSXMLDTD

The `detach` method is different from the others in this list in that you invoke it on the child node itself and not the parent of the node. You can send `detach` to any NSXMLNode object (not just the ones allowed to have children) to break the connection between that node and its parent. Once a node is detached from its parent, you are free to reattach it to another parent elsewhere in the tree using one of the methods for inserting, adding, or replacing child nodes.

All the other methods in Table 2 are sent to the parent of the node being added, removed, or replaced. The parent must be an object that is an instance of the NSXMLElement, NSXMLDocument, or NSXMLDTD class. Some of these classes place restrictions on what kinds of children can be added, depending on the class of the parent:

- NSXMLDocument objects can take only comment nodes, processing-instruction nodes, and a single element node (the root element) as children.
- NSXMLDTD objects can take only children that are comment nodes, processing-instruction nodes, and NSXMLDTDNode objects with kinds as specified by `NSXMLDTDNodeKind` constants.

The node-manipulation methods of these classes are identical. And they are straightforward, with their intended uses made unambiguous by their names. They may require you to do one or more things before invoking them:

- Find the parent node, the receiver of the message.

If you already have a reference to an existing child node, you can get its parent by sending the child a `parent` message.

- Find the index of the child to replace, remove, or insert the new child before.

  You can get the index by sending the node object an `index` message.

- Create the node object to be added to the parent.

  The methods that add or replace children usually (but not always) require you to create the child node first. Create the required node object using the `initWithKind:` method of the NSXMLNode class, one of the class factory methods of the same class, one of the initializers of the NSXMLElement class, or the `initWithXMLString:` method of the NSXMLDTDNode class.

The most common receiver of node-manipulation messages is an instance of the NSXMLElement class. The following code examples illustrate how the NSXMLElement node-manipulation methods might be used in your application. Listing 4 includes code that creates nested elements (the inner one containing an attribute) with the following XML string form:

```
<Figure><Graphic href="../Art/cc_finalui.gif"></Graphic>
```

Listing 4 Adding nested elements

```
// Note: relativePathToFilename is a custom method
NSString *relativePathToArt = [[self fileName]
    relativePathToFilename:selectedFile];

NSXMLElement *newFigureElement = [NSXMLElement elementWithName:@"Figure"];
NSXMLElement *newGraphicElement = [NSXMLElement elementWithName:@"Graphic"];
NSXMLNode *pathToHrefAttribute = [NSXMLNode attributeWithName:@"href"
    stringValue:relativePathToArt];

[newGraphicElement addAttribute:pathToHrefAttribute];
[newFigureElement addChild:newGraphicElement];

NSXMLElement *selectedParent = [treeView selectedItem];
[selectedParent addChild:newFigureElement];
// ...
```

Listing 5 is a method that replaces a child node with a text node containing the child's string value.

Listing 5 Replacing an element

```
- (void)unwrapElement:(NSXMLElement *)childToUnwrap {

    id parent = [childToUnwrap parent];
    int indexOfChildToUnwrap = [childToUnwrap index];
    NSXMLNode *textNode = [NSXMLNode textWithStringValue:[childToUnwrap
stringValue]];

    // Unwrapping is defined as replacing the selected element
    // with a text representation of itself.
    [parent replaceChildAtIndex:indexOfChildToUnwrap withNode:textNode];
}
```

As you add and relocate nodes, you may occasionally want to validate the document to verify that the changes conform to the governing schema. To validate a document, send the `validateAndReturnError:` message to the NSXMLDocument object. If the operation is not valid, the method returns `NO`; the reasons for invalidity are returned by indirection in the method's NSError parameter.

Representing Object Values as Strings

"Changing the Values of Nodes" (page 37) discusses how, through the `setObjectValue:` method of NSXMLNode, you can set the value of an NSXML node to be an Objective-C object other than an NSString. For example, you can set the value of a "date" node to be an NSCalendarDate object derived from a string value such as `"10/31/04"`. With the value as an object, you can easily perform computations, comparisons, and other operations that the object's class makes possible.

When you request a node to emit its value by sending it a `stringValue` or an `XMLString` message, it must, of course, represent that value as a string. The following sections describe the string representations provided for standard atomic data types and explain how to obtain string representations for custom objects.

Standard String Representations of Object Values

The NSXML classes provide canonical string representations for a standard set of atomic data types, each of which is represented by a Foundation object as an object value. For example, if you store an NSCalendarDate as a node's object value and then send `stringValue` to that node, you get back something like this:

```
2004-06-23T14:31:04
```

The canonical string representations for simple and derived types are defined in the W3C XML Schema Data Types specification (http://www.w3.org/TR/xmlschema-2/). Table 1 lists the more important types, along with their associated Foundation class and canonical string representation.

Table 1 Canonical string representations of W3C types

Class	W3C type	Canonical string representation
NSNumber	boolean	
NSNumber	decimal	Example: `0.3` Not allowed: other leading or trailing zeroes, "+" sign
NSNumber	integer	Examples: `-1`, `0`, `1267896`
NSNumber	float	Examples: `-1E4`, `1267.43233E12`, `12`, `INF`, `NaN`
NSNumber	double	Examples: `-1E4`, `1267.43233E12`, `12`, `INF`, `NaN`
NSCalendarDate	dateTime	Format: *CCYY*`-`*MM*`-`*DD*`T`*hh*`:`*mm*`:`*ss*`[Z]` Example: `2004-10-31T20:15:34` Notes: "T" is date-time separator; optional "Z" indicates Coordinated Universal Time.
NSData	base64Binary	Encoded using Base64 Content Transfer Encoding (RFC 2045)
NSURL	URI (URL)	A URI as defined by RFC 2396 and amended by RFC 2732

These string representations follow the XML Schema canonical form as a result of a design choice, not because the NSXML classes look at the `xsi:type` attribute or pick an output format based on an element's associated type in the validating schema.

Custom String Representations of Object Values

The NSXML classes implement their string representations for the standard W3C types through value transformers—that is, instances of custom subclasses of NSValueTransformer. You can easily create and register value transformers for your own custom objects. You can even override the implementations of the standard value transformers implemented by the NSXML classes.

When requested to render an object value as a string, the NSXML classes scan the global registry of value transformers looking for transformer names in the following format:

`NSXML` + *Class Name* + `TransformerName`

For example, if your class was named "MyCustomClass", the proper transformer name would be "NSXMLMyCustomClassTransformerName". The NSXML classes extract the class name from the transformer name and use it to identify the object value to transform.

For complete details on creating a custom value transformer, see *Value Transformer Programming Guide*. In summary, you must override the NSValueTransformer class methods `transformedValueClass` and `allowsReverseTransformation` and the instance method `transformedValue`. Listing 1 shows an example of such a transformation for a custom object with a name and a date instance variable.

Listing 1 Implementing a value transformer for an object value

```
@implementation NSXMLNameDateTransformer

+ (Class)transformedValueClass {
    return [NameDate class];
}

// transformation is oneway
+ (BOOL)allowsReverseTransformation {
    return NO;
}

- (id)transformedValue:(id)value {
    if ( [value isKindOfClass:[NameDate class]] ) {
        NSString *dateStr = [[value date]
            descriptionWithCalendarFormat:@"%Y-%m-%dT%H:%M:%S"];
        NSString *retVal = [NSString stringWithFormat:@"Name: %@ - Date: %@",
            [value name], dateStr];
        return retVal;
    }
    return @"";
}
@end
```

Your application should register the custom value transformer with the NSValueTransformer class as soon as possible. A good place for registration is in an override of the `initialize` class method. To register the value transformer, send a `setValueTransformer:forName:` message to the NSValueTransformer class. Make sure that the transformer name conforms to the requirement for NSXML value transformers. See Listing 2 for an example.

Listing 2 Registering the value transformer

```
#import "NSXMLNameDateTransformer.h"
+ (void)initialize {
    NSValueTransformer *myTransformer = [[[NSXMLNameDateTransformer alloc]
        init] autorelease];
    [NSValueTransformer setValueTransformer:myTransformer
        forName:@"NSXMLNameDateTransformerName"];
}
```

Note: With any value transformer created for an NSXML object value, the transformation is oneway only, from a nonstring object value to a string object. Any transformation in the other direction, from string to object value, requires you application to interpret the string and convert it into the appropriate object.

Handling Attributes and Namespaces

XML namespaces and attributes are similar in some respects but different in others. Attributes are name-value pairs that are typically used to hold metadata related to the element. Namespaces are considerably more complex in purpose. They serve to distinguish elements and attributes from different sources that have identical names but different meanings. They are also used to group related attributes and elements so that a processing application can easily recognize them. Namespaces are declared in a manner similar to the way attributes are, but with an `xmlns:prefix` name assigned a value that is a URI, for example:

```
<employee xmlns:emp="http://www.acme.com/defs/empl.html"/>
```

This construction maps the prefix to a unique URI identifier; thereafter the prefix can be used to identify the namespace of an attribute or element (for example, "<emp:title></emp:title>"). Default, prefix-less namespaces can also be declared that affect all current and descendent elements and attributes past the point of declaration, unless they are overridden by another namespace.

Although the purposes of attributes and namespaces are different, they are conceptually similar. They cannot be children of an element node, but they are always closely associated with one. Indeed the associated element *is* their parent. Even though they are NSXML nodes—NSXMLNode objects of kinds `NSXMLAttributeKind` and `NSXMLNamespaceKind`—they cannot have children and cannot be children of any node. (Namespaces, however, can qualify the attributes of an element as well as the element itself.) Namespace and attribute nodes are not encountered during document traversal.

The programmatic interface of NSXMLElement reflects this architectural affinity. It offers similar sets of methods for namespace nodes and attribute nodes. This articles explains how to use these methods and then discusses a unique feature of the namespace API: resolving namespaces.

Methods for Manipulating Attributes and Namespaces

The NSXMLElement class defines methods for manipulating and accessing an element's attributes that are nearly identical in form to another set of methods for manipulating and accessing namespace nodes. Table 1 lists these complementary sets of methods.

Table 1 Attribute and namespace node manipulation methods

Attribute methods	Namespace methods
`addAttribute:`	`addNamespace:`
`setAttributes:setAttributesAsDictionary:`	`setNamespaces:`
`removeAttributeForName:`	`removeNamespaceForPrefix:`
`attributes`	`namespaces`
`attributeForName:attributeForLocalName: URI:`	`namespaceForPrefix:`

The names of these methods clearly indicate what you use them for, but some comments about each category of method are warranted:

- The `add...` and `set...` methods usually require you to create NSXMLNode objects of the appropriate kind (`NSXMLAttributeKind`or `NSXMXLNamespaceKind`) before adding or setting the object. To create these node objects, you can use the NSXMLNode class factory methods `namespaceWithName:stringValue:`, `attributeWithName:stringValue:`, and `attributeWithLocalName:URI:stringValue:`. The last of these methods creates an attribute that is bound to a namespace identified by the URI parameter.

  The `setAttributesAsDictionary:` method lets you set an element's attributes without having to create NSXMLNode objects first. The keys in the dictionary are the names of the attributes and the values are the string values of the attributes.

  All of the methods that set attributes or namespaces of an element remove all existing attributes or namespaces.

- The methods that remove an attribute or namespace from an element, or that access a particular attribute or namespace, require you to know the name of the attribute or the prefix of the namespace. For an attribute node, you can simply ask the node for its name using the NSXMLNode `name` method. For a namespace node, however, the `name` method returns a qualified name (that is, the prefix plus the local name, separated by a colon). You can obtain the prefix by invoking the NSXMLNode class method `prefixForName:`, passing in the qualified name.

- The `attributeForLocalName:URI:` requires you to supply the local (nonqualified) name of an attribute as well as the namespace URI it's bound to. If you can access the associated namespace node, you can obtain the URI by sending the node a `stringValue` message. You can get the local name from the qualified name by using the NSXMLNode class method `localNameForName:`.

- If you want to access or remove an existing namespace or attribute node, you can obtain a reference to its element by sending the namespace or attribute node a `parent` message.

Once you have accessed a specific namespace or attribute node, you can get or set its string or object value (see "Changing the Values of Nodes" (page 37) for details). Bear in mind that the value of a namespace node is the URI of the namespace; if you want to set the URI as an object value, make it an NSURL object.

Resolving Namespaces

If your application knows (or suspects) that it might be dealing with XML from different sources or authored in different XML vocabularies, such as XSLT and RDF, it has to deal with namespaces. At any point of processing it might have to know the namespace to which an element or attribute is bound in order to handle that element or attribute appropriately. A case in point is the `set` element, which is defined by both SVG and MathML for different purposes. Before you can determine the meaning of a `set` element in a document containing both SVG and MathGL markup, you have to find out which namespace it belongs to. To find out the namespace affiliation of an element you must resolve it.

For namespace resolution, NSXMLElement declares two methods beyond the ones discussed in "Methods for Manipulating Attributes and Namespaces" (page 47).

```
resolveNamespaceForName:
resolvePrefixForNamespaceURI:
```

The first method takes the qualified or local name of an element and returns the namespace node representing the namespace to which the element belongs. Your application can get the URI from the node (its string value) and compare it to a list of known or expected URIs to determine namespace affiliation. If there is no associated namespace, the `resolveNamespaceForName:` method returns `nil`.

The second namespace-resolution method, `resolvePrefixForNamespaceURI:`, works in the opposite direction. You pass in a URI and get back the prefix bound to that URI. You can use this method, for example, when you are adding elements to a document and need to know the prefixes of their qualified names.

Creating and Modifying Document Type Definitions

With the DTD classes of NSXML you can create a Document Type Definition (DTD) or process an existing one, converting its string representation into a shallow tree structure. You can add nodes representing DTD declarations to the tree, remove nodes from it, or change the values of those declarations. Finally, you can write out the created or modified DTD "inline" with its XML string representation. The DTD classes are NSXMLDTD and NSXMLDTDNode.

Usually DTDs in NSXML are internal and are set as a property of an NSXMLDocument object. When it processes an XML document, NSXML creates a DTD tree structure and attaches it to a document but only if the DTD is internal. However, you can create a DTD programmatically and write it out to its own file (that is, an external DTD) using NSXML.

When NSXML processes an existing source of XML and the `NSXMLDocumentValidate` input option is specified, it validates the XML against an internal or external DTD (see "Creating a Document Object From Existing XML" (page 19)). However, validation is a separate process from the creation of the tree structure representing a DTD. (You can also dynamically validate an NSXMLDocument against its DTD by sending the `validateAndReturnError:` message to the document object.)

Creating a DTD

If you read and process an existing source of XML using the NSXML interfaces, and that XML includes an internal DTD, a shallow (two-level) tree is created to represent the DTD. At the root of this tree is an instance of the NSXMLDTD class; this object has children consisting of NSXMLDTDNodeDTD nodes objects, each representing the declaration for an element, entity, attribute-list, or notation in the DTD. (The children may also include NSXMLNode objects representing comments and processing instructions found in the source DTD.) As it parses declarations and creates DTD nodes, NSXML sets the kind (`NSXMLNodeKind`) and DTD subkind (`NSXMLDTDNodeKind`) of each created node.

You can also create a standalone NSXMLDTD object from an external DTD with the `initWithContentsOfURL:options:error:` or `initWithData:options:error:` methods. Then you can associate this DTD object with an XML document object through the `setDTD:` method of the NSXMLDocument class.

The created DTD tree is associated with the XML document as a property. In programmatic terms, this means that you can access the NSXMLDTD object (and its NSXMLDTDNode children) by sending the NSXMLDocument object a `DTD` message. Once you have accessed the NSXMLDTD object you can add children, remove children, change declarations, and so on (see "Modifying a DTD" (page 52)).

You can also create an entirely new DTD tree programmatically. To do so, complete the following steps:

1. Create an instance of the NSXMLDTD class.
2. If the DTD is to be external, set its system ID (`setSystemID:`) and, if necessary, its public ID (`setPublicID:`).

3. Create instances of the NSXMLDTDNode class for each declaration you want in the DTD. To create these objects you can use the `initWithXMLString:` method of the NSXMLDTDNode class, the NSXMLNode class method `DTDNodeWithXMLString:` , or the NSXMLNode `initWithKind:` method. The first two of these methods take a string equivalent to a DTD declaration, for example

   ```
   <!ELEMENT name (#PCDATA)>
   ```

   (If you use the `initWithKind:` method, you must set the string or object value of the DTD node appropriately—see "Modifying a DTD" (page 52).) Each created DTD node is automatically assigned a node kind (if one is not explicitly assigned) and a DTD subkind, based on the declaration parsed. You may reassign the subkind with the `setDTDKind:` method.

4. Add each created NSXMLDTDNode object to the NSXMLDTD object as a child (using `addChild:` or `insertChild:atIndex:`).

   For more on creating, adding, and otherwise manipulating children of an NSXMLDTD object, see "Modifying a DTD" (page 52).

Once you have created a DTD, you must decide whether it is to be an internal or external DTD. If an internal DTD, attach it to the NSXMLDocument object using the `setDTD:` method. When you write the XML document out, the DTD string representation appears inline in the XML. If the DTD is external, don't attach it to an NSXMLDocument but write it out separately, as illustrated in Listing 1.

Listing 1 Creating an external DTD

```
- (void)createExternalDTD {
    NSXMLDTD *theDTD = [[[NSXMLDTD alloc] init] autorelease];
    [theDTD setSystemID:@"http://www.boggle.com/DTDs/index.html"];
    NSXMLDTDNode *attrNode = [[[NSXMLDTDNode alloc] initWithXMLString:
        @"<!ATTLIST phone location (home | office | mobile) 'home'>"]
        autorelease];
    NSXMLDTDNode *elemNode = [[[NSXMLDTDNode alloc] initWithXMLString:
        @"<!ELEMENT address (street1, street2, city, state, zip)>"]
        autorelease];
    NSXMLDTDNode *entNode = [[[NSXMLDTDNode alloc] initWithXMLString:
        @"<!ENTITY % children 'first, middle, last'>" ]
         autorelease];
    NSString *thePath = [self savePath]; // custom method
    [theDTD addChild:attrNode];
    [theDTD addChild:elemNode];
    [theDTD addChild:entNode];
    [[theDTD XMLString] writeToFile:thePath atomically:YES];
}
```

Modifying a DTD

As with modifying an XML document, you can modify a DTD in two general ways. You can alter the structure of the tree representing the DTD by adding, removing, and replacing nodes. And you can change the value of DTD nodes, essentially modifying the declaration.

When you change the value of NSXMLDTDNode objects using `setStringValue:` or `setObjectValue:` what changes depends on the type of declaration:

- If the type is mixed or an element declaration, the validation string is changed. For example, in the declarations

  ```
  <!ELEMENT title (#PCDATA)*>
  <!ELEMENT item (head, (p | list | note))>
  ```

  `(#PCDATA)*` and `(head, (p | list | note))` are validation strings.

- If the type is an entity, the entity value changes. In the declaration

  ```
  <!ENTITY foo "bar">
  ```

  `"bar"` is the entity value.

- If the type is an attribute-list declaration, the default value changes. In the declaration

  ```
  <!ATTLIST bar foo (moo | boo) "moo">
  ```

  `"moo"` is the default value.

- Notations have no changeable value.

The NSXMLDTD methods for manipulating the structure of a DTD are the following:

```
- (void)addChild:(NSXMLNode *)child
- (void)setChildren:(NSArray *)children
- (void)insertChild:(NSXMLNode *)child atIndex:(unsigned)index
- (void)insertChildren:(NSArray *)children atIndex:(unsigned)index
- (void)removeChildAtIndex:(unsigned)index
- (void)replaceChildAtIndex:(unsigned)index withNode:(NSXMLNode *)node
```

These methods, which are identical in signature and purpose to methods of NSXMLDocument and NSXMLElement, have unambiguous usages. All of them except `removeChildAtIndex:` require you to create an NSXMLDTDNode object first (or detach an existing NSXMLDTDNode object first). As briefly discussed in "Creating a DTD" (page 51), you typically create an NSXMLDTDNode object with the `initWithXMLString:` method or the `DTDNodeWithXMLString:` or `initWithKind:` methods of the NSXMLNode class. The argument of the first two methods is an NSString object representing a DTD declaration, as shown in this example:

```
NSXMLDTDNode *attrNode = [[[NSXMLDTDNode alloc] initWithXMLString:
    @"<!ATTLIST phone location (home | office | mobile) 'home'>"]
    autorelease];
```

Note that you can also create and add children to an NSXMLDTD object that are NSXMLNode objects representing comments and processing instructions.

Finding the Declaration for an XML Construct

If you are implementing your own scheme for dynamic validation of XML, you need to know which DTD declaration governs the placement, form, and allowable values of a particular XML construct. The following convenience methods of NSXMLDTD provide you with that information:

```
- (NSXMLDTDNode *)entityDeclarationForName:(NSString *)name
```

```
- (NSXMLDTDNode *)elementDeclarationForName:(NSString *)name
- (NSXMLDTDNode *)attributeDeclarationForName:(NSString *)name
elementName:(NSString *)elementName
- (NSXMLDTDNode *)notationDeclarationForName:(NSString *)name
```

You provide these methods with the name of an entity, element, notation, and attribute (plus, for attributes, the element name), and they return the NSXMLDTDNode that governs the XML construct.

Alternatively, you could get the children of an NSXMLDTD object and analyze them.

Binding NSXML Objects to a User Interface

Instances of all NSXML classes are model objects that conform to the key-value coding protocol and that notify observers conforming to the key-value observing protocol of any change in their attributes. Consequently, you can establish bindings between NSXML objects and objects on a user interface using the Cocoa bindings technology. With bindings, a change in an attribute value of a NSXML object is propagated automatically to a user-interface property (such as the string value of a text field), and vice versa. Instead of a custom controller containing reams of "glue code" for mediating between the model and view objects of your application, you can, in Interface Builder, bind model attributes to user-interface properties via ready-made controllers.

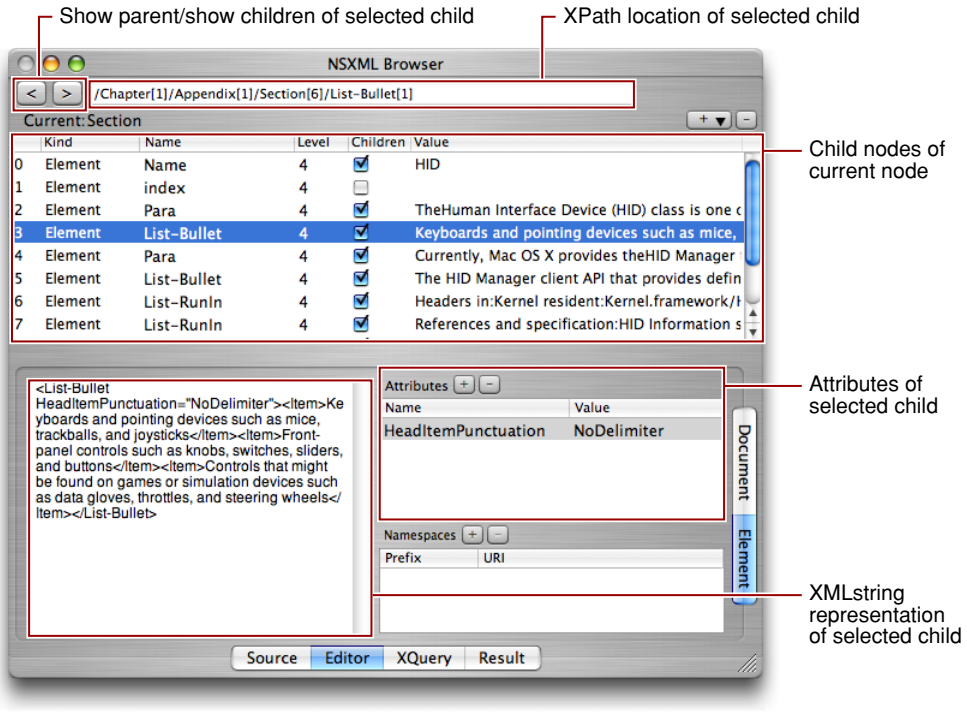
This article explores how you can effectively establish bindings between NSXML objects and a user interface by looking at the *XMLBrowser* sample project. It is not a tutorial on bindings; instead, it is a case study of how you can factor in the hierarchical data model of NSXML when designing the bindings connecting the objects of your application.

Note: For complete information on what bindings are and how to use them, see *Cocoa Bindings Programming Topics*.

The Interface

To simplify matters, let's look at just one part of the user interface for the XMLBrowser application. Figure 1 shows the pane made visible when the Editor tab is clicked; it also shows the Elements subpane of that pane.

Figure 1 The Editor pane of the XMLBrowser application (Element subpane)



The current node, whose name is shown just above the largest table view, is the ultimate reference point for everything else on the window. At first, the current node is the NSXMLDocument object representing the entire document. Any children of the current node are displayed in the large table view; initially, this is the root element. When you select a child node in the table view, the text field at the top of the window displays its XPath-notated location in the tree and the text view in the lower-left quadrant of the window displays its XML string representation. If the selected child is an element node with attributes or namespaces, the application displays them in the Attributes and Namespaces table views.

When you click the ">" button while a child node is selected, the selected node becomes the current node and the rest of the window is updated to reflect this. The children of the new current node are displayed in the large table view, with the first child automatically selected. The XPath text field and the XML-string text view change contents accordingly, and any attributes or namespaces of the new child appear in their respective table views. To go back up the hierarchy, thereby resetting the current node to its parent, you click the "<" button.

Connecting Controllers and NSXMLNode Objects

The object model for the bindings used in the window shown in Figure 1 (page 56)) is fairly simple:

- There is an NSXMLNode object designated the "current" node. When the application resets the instance variable holding this object, it uses the NSXMLNode methods `parent` and `childAtIndex:` to navigate the tree hierarchy (where the index is supplied by the master table view's current selection).
- There is a to-many relationship between the current node and its children (accessed via its `children` property).

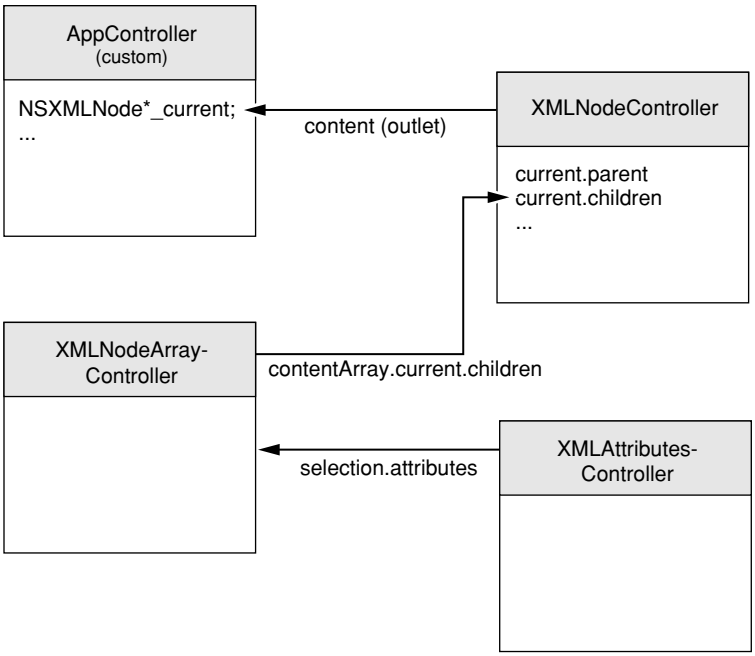
- In the array holding the children of the current node, one of the children is selected (the NSArrayController `selection` property).
- There is a to-many relationship between the selected child node and its attributes and namespaces.

The configuration of the NSXML objects and the controller objects in the application reflect this object model. Four off-the-shelf controller objects are added to the nib file of NSXMLBrowser for this window:

- A NSObjectController instance for managing bindings with the current node (named "XMLNodeController")
- A NSArrayController instance for managing bindings with the children of the current node (named "XMLNodeArrayController")
- Two NSArrayController instances for managing bindings with, respectively, the attributes and the namespaces of any selected child element (named "XMLAttributesController" and "XMLNamespacesController")

Figure 2 depicts the connections among these controller instances and the custom application controller, which is named AppController. (The XMLNamespacesController is omitted because its configuration is nearly identical to that of XMLAttributesController.)

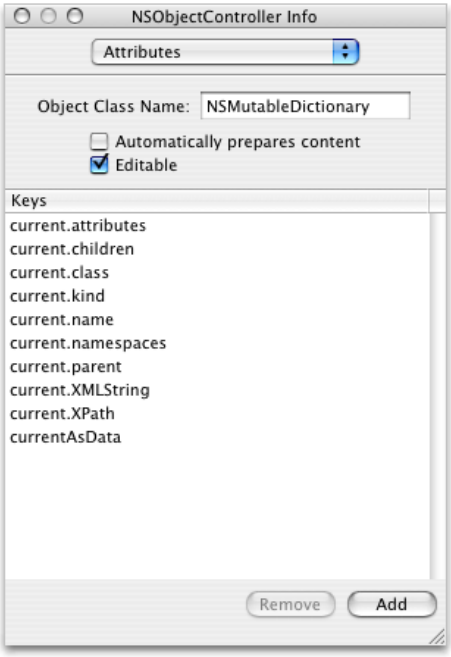
Figure 2 How the controller objects are configured



Specifically, configuration of the bindings between the controllers and the NSXML model objects consist of the following steps:

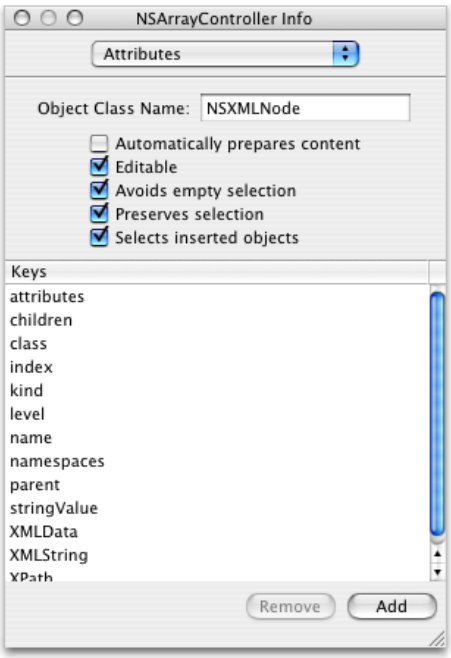
1. Connect the content outlet of the XMLNodeController instance to the application controller (AppController).

2. Set the keys of the XMLNodeController as shown:



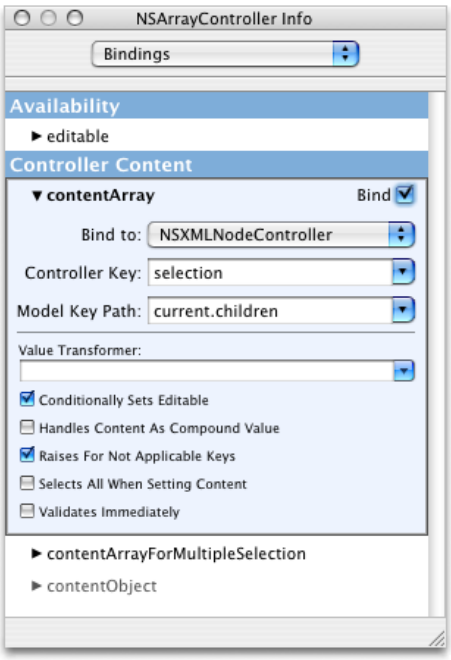
The first part of each key path identifies the `current` attribute of the AppController and the second part identifies an attribute of the current NSXMLNode object.

3. Set the keys of the XMLNodeArrayController to be the attributes of the NSXMLNode class:

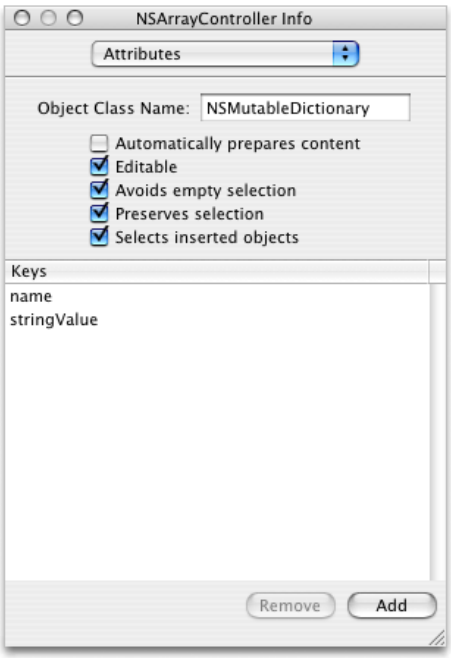


Note that the object class in this case is specified as NSXMLNode.

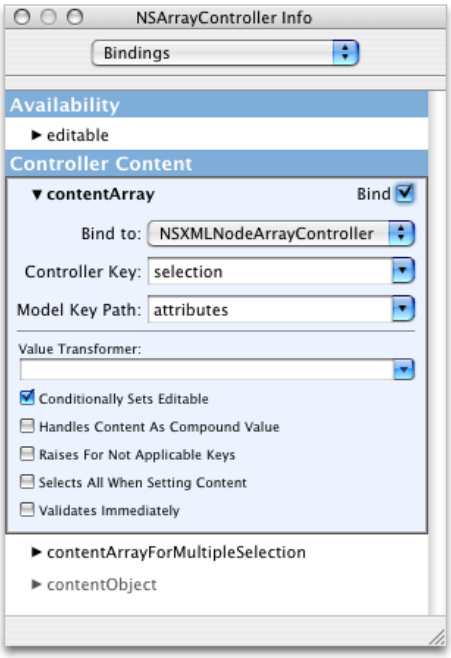
4. Establish a binding between the `contentArray` property of the XMLNodeArrayController and the `current.children` attribute accessed by the XMLNodeController through its `selection` key.



5. In the Attributes pane for XMLAttributesController, set the `name` and `stringValue` keys for the XML attributes.



6. In the Bindings pane for XMLAttributesController, establish a binding between the `contentArray` property of the controller and the `attributes` attribute accessed by the XMLNodeArrayController through its `selection` key.



7. Repeat the last two steps for the XMLNamespacesController, substituting `namespaces` for `attributes` for the model key path.

Binding the User Interface

Once you have established the associations between the model NSXMLNode objects and the controllers of the XMLBrowser application, you can establish bindings between objects on the user interface and their controllers, consequently extending the bindings to the associated model objects. To do this, select a user interface object and find the required binding property in the Bindings pane of the Info window. For user-interface objects showing single values, such as text fields and table columns, this property is usually named `value` or `data`. For objects showing arrays of data, such as table views, the properties are named (in this case) `contentArray` and `selectionIndexes`. For user-interface objects, the binding keys appear in the "Value" category of the bindings while for controllers their data appears in the "Controller Content" category.

Expand the view for the content-related binding property of the user-interface object. Then specify the controller to establish a binding to, the key of the controller to use for the binding, and the key path to the model attribute. For example, the bindings for the XPath field in the Editor pane of XMLBrowser appear as in Figure 3.

Figure 3 Bindings for the text field displaying the selected child's XPath location

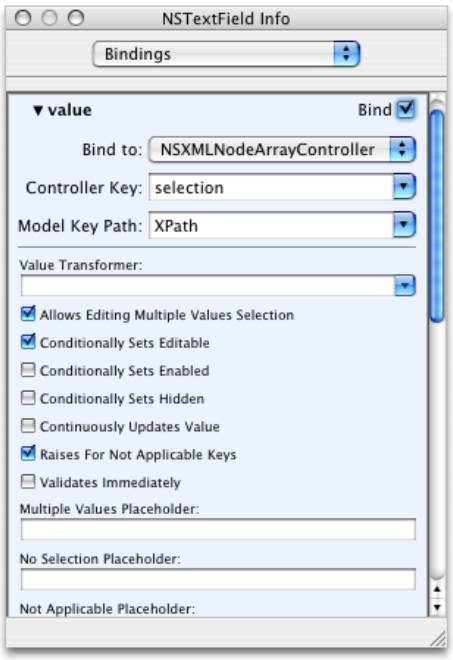


Table 1 lists the bindings for most user-interface objects in this part of the XMLBrowser application.

Table 1 User-interface bindings in Editor pane (Element subpane)

UI object	Binding	Controller	Controller Key	Model Key Path
XPath text field	value	XMLNodeArrayController	selection	XPath
Current node ("Current:")	value	XMLNodeController	selection	current.name
Child nodes table view	content	XMLNodeArrayController	arrangedObjects	
Child nodes table view	selectionIndexes	XMLNodeArrayController	selectionIndexs	
Index column	value	XMLNodeArrayController	arrangedObjects	index
Kind column	value	XMLNodeArrayController	arrangedObjects (see note)	kind
Name column	value	XMLNodeArrayController	arrangedObjects	name
Level column	value	XMLNodeArrayController	arrangedObjects	level
Children column	value	XMLNodeArrayController	arrangedObjects (see note)	children

UI object	Binding	Controller	Controller Key	Model Key Path
Value column	value	XMLNodeArrayController	arrangedObjects	stringValue
XML text view	data	XMLNodeArrayController	selection	XMLData
Attributes table view	content	XMLAttributesController		
Attributes table view	selectionIndexes	NSXMLAttributesController		
Attribute name column	value	NSXMLAttributesController	arrangedObjects	name
Attribute name value	value	XMLAttributesController	arrangedObjects	stringValue

The “content” binding on table views is optional and is created automatically when you bind a table column.

Note: Two of the user-interface bindings in this table use value transformers (objects that transform the value accessed via the binding). The first binding is from the Kind column (children table view); it uses a custom NSValueTransfomer object to convert the integer indicating node kind (a `NSXMLNodeKind` constant) to a string. The second binding is from the Children column of the same table view; it uses an Apple-provided value transformer (NSIsNotNil) to set the enabled state of items in the column depending upon the existence of child nodes.

Using Tree Controllers With NSXML Objects

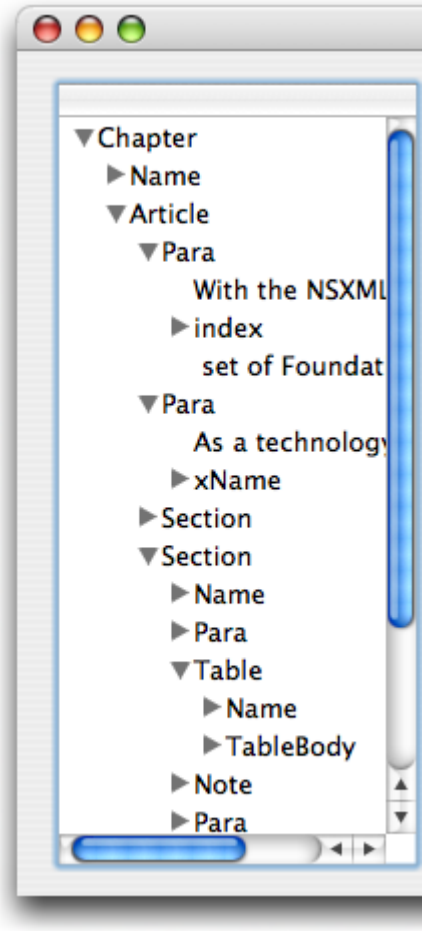
An `NSXMLDocument` object represents a hierarchical tree structure composed of `NSXMLNode` objects. In the Model-View-Controller design pattern, an `NSXMLDocument` is a model object because it represents application data in XML format. It makes sense then that an `NSXMLDocument` object would be ideally matched with a view object and a controller object that are designed to handle data arranged in tree structures. For this purpose, Cocoa provides the `NSTreeController` class for the controller layer; for hierarchy-displaying views, there are the `NSOutlineView` and `NSBrowser` classes.

The following sections walk you through the steps for using an `NSTreeController` object paired with an `NSOutlineView` object in an NSXML application. For communication of data between view, controller, and model objects, the example uses bindings, and does not rely on any data-source, delegation, or action methods.

Creating the Document-Based Application Project

The project example is an application that displays the structure of an XML file in an outline view; when the user selects a node in the outline view, the application shows the textual content of that node (and its children) in a text view. Figure 1 shows the outline-view part of the application at runtime.

Figure 1 Outline view displaying XML data



For this example, the application is document-based—a likely scenario for an application whose document data is XML markup. An important requirement for Cocoa bindings is that each instance of the `NSDocument` subclass holds a reference (as an instance variable) to the `NSXMLDocument` object representing an XML file or other source. (In the object modeling terminology of bindings, the `NSXMLDocument` object is a *property* of the document instance.) The instance of the `NSDocument` subclass is the File's Owner for the document's nib file, which contains the `NSOutlineView` object and the `NSTreeController` used for establishing bindings.

Listing 1 shows the germane part of the `NSDocument` subclass implementation. In `readFromData:ofType:error:`, the application creates an `NSXMLDocument` instance from the XML file selected by the user and calls the accessor method `setTheDocument:` to assign and retain the instance variable. This "setter" method and its complementary "getter" method not only ensure proper memory management of the instance variable, but help to make the document subclass compliant with the requirements of key-value coding.

Listing 1 Setting the NSXMLDocument object as a property of File's Owner

```
- (BOOL)readFromData:(NSData *)data ofType:(NSString *)typeName error:(NSError
 **)outError
{
```

```
    NSXMLDocument *newDocument = [[NSXMLDocument alloc] initWithData:data
        options:NSXMLNodeOptionsNone error:nil];
    [self setTheDocument:newDocument];
    [newDocument release];
    return YES;
}

- (NSXMLDocument *)theDocument
{
    return [[theDocument retain] autorelease];
}

- (void)setTheDocument:(NSXMLDocument *)newTheDocument
{
    if (theDocument != newTheDocument)
    {
        [newTheDocument retain];
        [theDocument release];
        theDocument = newTheDocument;
    }
}
```

Note: See *Document-Based Applications Overview* for information on creating document-based applications. To learn more about key-value coding, read *Key-Value Coding Programming Guide*.

Adding Methods to NSXMLNode

Categories are a powerful feature of Objective-C. The let you add methods to a class without having to make a subclass. To make the NSXML objects in our sample application work together with the `NSTreeController` object, it is necessary to add a couple methods to the `NSXMLNode` class through a category. Listing 2 shows the implementation of the methods in the category.

In its configuration, the `NSTreeController` object requires some way to know whether any given node in a tree structure is a leaf node—that is, a node with no children. The category's `isLeaf` method serves this purpose by returning `YES` if the node is a text node. In addition, each node in the outline view must have some text to represent it. For elements (`NSXMLElement` objects), that text could be the element name. However, other XML nodes objects, such as text nodes, don't have names but do have string values. Thus a method is needed to unify the textual representation returned for a given node; the category method `displayName` does this.

Listing 2 Adding a category to NSXMLNode

```
@implementation NSXMLNode (NSXMLNodeAdditions)

- (NSString *)displayName {

    NSString *displayName = [self name];
    if (!displayName) {
        displayName = [self stringValue];
    }
    return displayName;
}
```

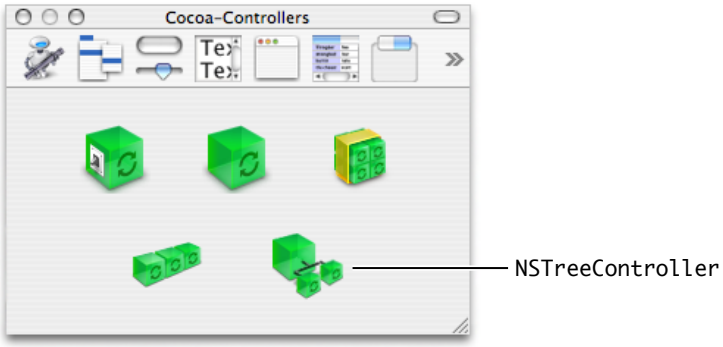
```
- (BOOL)isLeaf {

    return [self kind] == NSXMLTextKind ? YES : NO;
}

@end
```

Note: For the sake of clarity, the `isLeaf` method in this example deals with text nodes only. In a real application the method should test for all types of nodes that cannot have children, including processing instructions and comments,

Establishing the Bindings

To complete this application in Interface Builder, you must create the user interface and establish bindings between the tree controller and the `NSXMLDocument` object and the outline view and the tree controller. Drag the outline view object from the Cocoa-Data palette and place it in the document window; make it a single column, size it, and set any other attributes needed to configure it. Next, drag the tree-controller object on the Cocoa-Controllers palette into the nib file window. Figure 2 identifies this controller object.

Figure 2 The Cocoa-Controllers palette



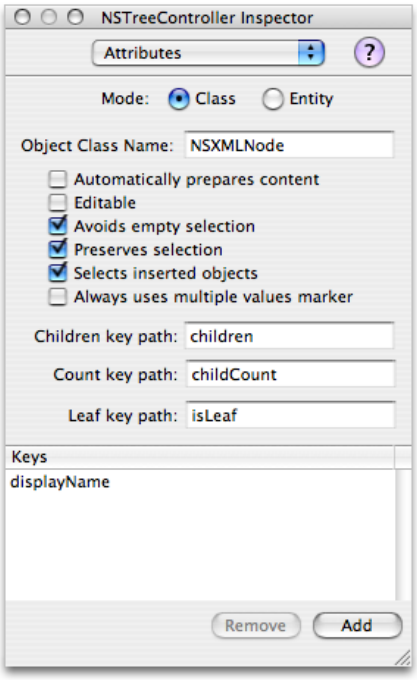
With the tree controller selected, open the Attributes pane of the inspector window (Command-1). First, set the value of the Object Class Name field to `NSXMLNode`. This is the underlying class type of the objects that compose the tree structure.

The Attributes pane has several "key path" attributes that are specific to the `NSTreeController` object. Enter key paths in the first and third of these text fields:

- Children key path: Enter the name of the `NSXMLNode` method that returns the children of a node (`children`).
- Count key path: Enter the name of the `NSXMLNode` method that returns the number of children for a given node (`childCount`). This method is more efficient than calling `count` on the children array.
- Leaf key path: Enter `isLeaf`, the name of the category method you implemented in "Adding Methods to NSXMLNode" (page 65).

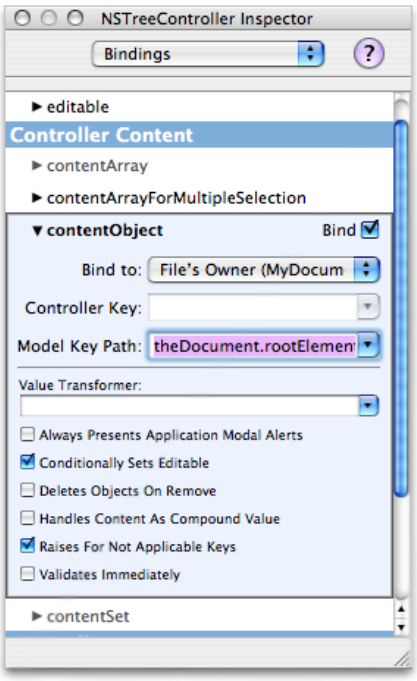
Finally, add to this pane's Keys table the name of the other method in the category on `NSXMLNode`: `displayName`. When you are finished, the Attributes pane for the tree controller should look like the one in Figure 3.

Figure 3 Setting the model keys



Once the tree controller is configured with the relevant key paths of the model object, you can establish a binding between the tree controller and the `NSXMLDocument` instance held as a property of the document object. With the `NSTreeController` object still selected, open the Bindings pane of the inspector (Command-4). Expose the **contentObject** property and set the values of the Bind To pop-up menu and Model Key Path text field as shown in Figure 4. This step makes the `NSXMLDocument` object the source of content for the tree controller.

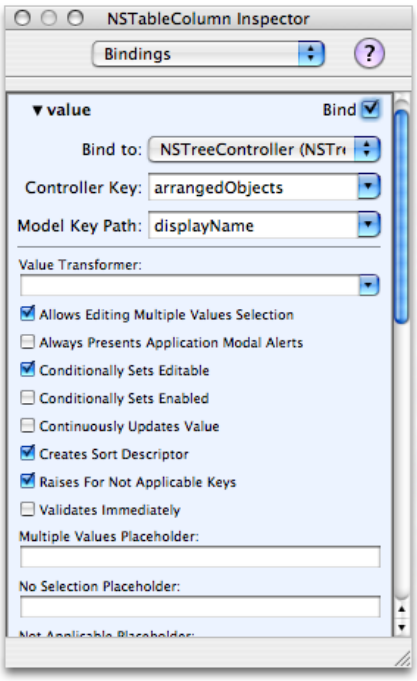
Figure 4 Setting the tree controller's content object



Here the Bind To value is set to File's Owner, which is the instance of the `NSDocument` subclass that holds the `NSXMLDocument` object as a property. Also note that the key path entered in the Model Key Path field includes not only the `theDocument` property of the document but the `rootElement` property, which corresponds to the `rootElement` method of `NSXMLDocument`. The content object of the tree controller must be the object at the root of the tree hierarchy, which for an XML document is its root element.

The final part of the procedure requires you to establish bindings between the outline view and the tree controller. Open the Bindings pane of the inspector and repeatedly click the top of the outline view until the inspector shows Table Column. Then expose the value property in the Bindings pane of the inspector and establish a binding through the tree controller's `arrangedObjects` property to the `displayName` attribute of each node in the model object (the `NSXMLDocument` instance). The bindings setting should look like the example in Figure 5.

Figure 5 Setting the displayed value for items in the outline view



Compile the project, run it, and observe how the outline view now reflects the structure of the XML document.

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.

- "*" 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 `&apos;`, `>`, `&#231;` ; 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 specificed 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.

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 *Tree-Based XML Programming Guide*.

Date	Notes
2009-02-04	Replaced use of deprecated method loadDataRepresentation:ofType: with readFromData:ofType:error:.
2008-10-15	Made various small corrections.
2006-11-07	Made several minor corrections.
2006-04-04	Added new article on using NSTreeController objects with NSXMLDocument objects. Also updated code listing showing use of objectByApplyingXSLTAtURL:arguments:error:.
2005-07-07	Fixed typos and made other small changes.
2004-11-03	Added "Representing Object Values as Strings" (page 43) and updated to discuss newest methods and options.
2004-07-20	Added "Creating and Modifying Document Type Definitions" (page 51) and an updated XML glossary (shared with the *Event-Driven XML Programming Guide* topic). Also indexed topic.
2004-06-28	New document that explains how to use the NSXML classes of Foundation to process, modify, and query XML data. Seed distribution only.

Index

Symbols

A

B

C

D

E

F

I

L

M

N

O

P

Q

R

S

T

U

V

X