# BEA WebLogic Workshop™ Help

**Version 8.1 SP4**
**December 2004**

# Copyright



# Restricted Rights Legend



# Trademarks or Service Marks

# Table of Contents

# Handling XML with XMLBeans

XMLBeans provides a simple, easy−to−use means for accessing XML. With XMLBeans, you can compile schema to generate Java types that can be bound to XML based on the schema. You can also access XML with schema by using an XML cursor, which provides an access model that is an alternative to DOM−based models.

## Topics Included in This Section

Getting Started with XMLBeans

Introduces XMLBeans, a tool that makes it easier for you to work with XML data and documents in Java.

Java Types Generated from User−Derived Schema Types

Explains how to generate Java interfaces that help you work with XML based on its schema.

Methods for Types Generated From Schema

Lists the methods that you may see in the API generated by compiling schema.

XMLBeans Support for Built−In Schema Types

Provides an overview of the set of XMLBeans Java types that mirror built−in schema types.

Selecting XML with XQuery and XPath

Introduces XMLBeans methods for executing XQuery and XPath expressions agains the underlying XML.

Navigating XML with Cursors

Describes a way for you to navigate XML, including moving through XML in small increments, getting and setting values in XML, inserting annotations, and executing XQuery expressions.

Understanding XML Tokens

Provides an overview of XML tokens, which represent XML in cursor navigation.

Using Bookmarks to Annotate XML

Introduces bookmarks, through which you can associate arbitrary pieces of data with specific parts of XML.

Introduction to Schema Type Signatures

Introduces the "signatures" available from the schema type system.

Related Topics

Introduction to XML

XMLBeans Samples

# Getting Started with XMLBeans

XMLBeans provides intuitive ways to handle XML that make it easier for you to access and manipulate XML data and documents in Java.

Characteristics of XMLBeans approach to XML:

- It provides a familiar Java object−based view of XML data without losing access to the original, native XML structure.
- The XML's integrity as a document is not lost with XMLBeans. XML−oriented APIs commonly take the XML apart in order to bind to its parts. With XMLBeans, the entire XML instance document is handled as a whole. The XML data is stored in memory as XML. This means that the document order is preserved as well as the original element content with whitespace.
- With types generated from schema, access to XML instances is through JavaBean−like accessors, with get and set methods.
- It is designed with XML schema in mind from the beginning — XMLBeans supports all XML schema definitions.
- Access to XML is fast.

The starting point for XMLBeans is XML schema. A schema (contained in an XSD file) is an XML document that defines a set of rules to which other XML documents must conform. The XML Schema specification provides a rich data model that allows you to express sophisticated structure and constraints on your data. For example, an XML schema can enforce control over how data is ordered in a document, or constraints on particular values (for example, a birth date that must be later than 1900). Unfortunately, the ability to enforce rules like this is typically not available in Java without writing custom code. XMLBeans honors schema constraints.

***Note:*** Where an XML schema defines rules for an XML document, an XML *instance* is an XML document that conforms to the schema.

You compile a schema (XSD) file to generate a set of Java interfaces that mirror the schema. With these types, you process XML instance documents that conform to the schema. You bind an XML instance document to these types; changes made through the Java interface change the underlying XML representation.

Previous options for handling XML include using XML programming interfaces (such as DOM or SAX) or an XML marshalling/binding tool (such as JAXB). Because it lacks strong schema−oriented typing, navigation in a DOM−oriented model is more tedious and requires an understanding of the complete object model. JAXB provides support for the XML schema specification, but handles only a subset of it; XMLBeans supports all of it. Also, by storing the data in memory as XML, XMLBeans is able to reduce the overhead of marshalling and demarshalling.

# Accessing XML Using Its Schema

To get a glimpse of the kinds of things you can do with XMLBeans, take a look at an example using XML for a purchase order. The purchase order XML contains data exchanged by two parties, such as two companies. Both parties need to be able to rely on a consistent message shape, and a schema specifies the common ground.

Here's what a purchase order XML instance might look like.

```
<po:purchase-order xmlns:po="http://openuri.org/easypo">
    <po:customer>
        <po:name>Gladys Kravitz</po:name>
        <po:address>Anytown, PA</po:address>
    </po:customer>
    <po:date>2003-01-07T14:16:00-05:00</po:date>
    <po:line-item>
        <po:description>Burnham's Celestial Handbook, Vol 1</po:description>
        <po:per-unit-ounces>5</po:per-unit-ounces>
        <po:price>21.79</po:price>
        <po:quantity>2</po:quantity>
    </po:line-item>
    <po:line-item>
        <po:description>Burnham's Celestial Handbook, Vol 2</po:description>
        <po:per-unit-ounces>5</po:per-unit-ounces>
        <po:price>19.89</po:price>
        <po:quantity>2</po:quantity>
    </po:line-item>
<po:shipper>
        <po:name>ZipShip</po:name>
        <po:per-ounce-rate>0.74</po:per-ounce-rate>
    </po:shipper>
</po:purchase-order>
```

This XML includes a root element, purchase−order, that has three kinds of child elements: customer, date, line−item, and shipper. An intuitive, object−based view of this XML would provide an object representing the purchase−order element, and it would have methods for getting the date and for getting subordinate objects for customer, line−item, and shipper elements. Each of the last three would have its own methods for getting the data inside them as well.

## Looking at the Schema

The following XML is the the schema for the preceding purchase order XML. It defines the XML's "shape" — what its elements are, what order they appear in, which are children of which, and so on.

```
<xs:schema targetNamespace="http://openuri.org/easypo"
    xmlns:po="http://openuri.org/easypo"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    elementFormDefault="qualified">

    <xs:element name="purchase-order">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="customer" type="po:customer"/>
                <xs:element name="date" type="xs:dateTime"/>
                <xs:element name="line-item" type="po:line-item" minOccurs="0" maxOccurs="unbou
                <xs:element name="shipper" type="po:shipper" minOccurs="0"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
    <xs:complexType name="customer">
        <xs:sequence>
            <xs:element name="name" type="xs:string"/>
            <xs:element name="address" type="xs:string"/>
        </xs:sequence>
    </xs:complexType>
```

```
    <xs:complexType name="line-item">
        <xs:sequence>
            <xs:element name="description" type="xs:string"/>
            <xs:element name="per-unit-ounces" type="xs:decimal"/>
            <xs:element name="price" type="xs:double"/>
            <xs:element name="quantity" type="xs:int"/>
        </xs:sequence>
    </xs:complexType>
    <xs:complexType name="shipper">
        <xs:sequence>
            <xs:element name="name" type="xs:string"/>
            <xs:element name="per-ounce-rate" type="xs:decimal"/>
        </xs:sequence>
    </xs:complexType>
</xs:schema>
```

This schema describes the purchase order XML instance by defining the following:

- Definitions for three complex types — customer, line−item, and shipper. These are the types used for the children of the purchase−order element. In schema, a complex type is one that defines an element that may have child elements and attributes. The sequence element nested in the complex type lists its child elements.
  These are also *global* types. They are global because they are at the top level of the schema (in other words, just beneath the schema root element). This means that they may be referenced from anywhere else in the schema.

- Use of simple types within the complex types. The name, address, and description elements (among others) are typed as simple types. As it happens, these are also *built−in* types. A built−in type (here, one with the "xs" prefix) is part of the schema specification. (The specification defines 46 built−in types.)
- A global element called purchase−order. This element definition includes a nested complex type definition that specifies the child elements for a purchase−order element. Notice that the complex type includes references to the other complex types defined in this schema.

In other words, the schema defines types for the child elements and describes their position as subordinate to the root element, purchase−order.

When you use the XMLBean compiler with an XSD file such as this one, you generate a JAR file containing the interfaces generated from the schema.

## Writing Java Code That Uses the Interfaces

With the XMLBeans interfaces in your application, you can write code that uses the new types to handle XML based on the schema. Here's an example that extracts information about each of the ordered items in the purchase order XML, counts the items, and calculates a total of their prices. In particular, look at the use of types generated from the schema and imported as part of the org.openuri.easypo package.

The printItems method receives a File object containing the purchase order XML file.

```
package docs.xmlbeans;

import java.io.File;
import com.bea.xml.*;
import org.openuri.easypo.PurchaseOrderDocument;
```

```
import org.openuri.easypo.PurchaseOrder;
import org.openuri.easypo.LineItem;

public class POHandler
{
    public static void printItems(File po) throws Exception
    {
        /*
         * All XMLBeans schema types provide a nested Factory class you can
         * use to bind XML to the type, or to create new instances of the type.
         * Note that a "Document" type such as this one is an XMLBeans
         * construct for representing a global element. It provides a way
         * for you to get and set the contents of the entire element.
         *
         * Also, note that the parse method will only succeed if the
         * XML you're parsing appears to conform to the schema.
         */
        PurchaseOrderDocument poDoc =
            PurchaseOrderDocument.Factory.parse(po);

        /*
         * The PurchaseOrder type represents the purchase-order element's
         * complex type.
         */
        PurchaseOrder po = poDoc.getPurchaseOrder();

        /*
         * When an element may occur more than once as a child element,
         * the schema compiler will generate methods that refer to an
         * array of that element. The line-item element is defined with
         * a maxOccurs attribute value of "unbounded", meaning that
         * it may occur as many times in an instance document as needed.
         * So there are methods such as getLineItemArray and setLineItemArray.
         */
        LineItem[] lineitems = po.getLineItemArray();
        System.out.println("Purchase order has " + lineitems.length + " line items.");

        double totalAmount = 0.0;
        int numberOfItems = 0;

        /*
         * Loop through the line-item elements, using generated accessors to
         * get values for child elements such a description, quantity, and
         * price.
         */
        for (int j = 0; j < lineitems.length; j++)
        {
            System.out.println(" Line item: " + j);
            System.out.println(
                "   Description: " + lineitems[j].getDescription());
            System.out.println("   Quantity: " + lineitems[j].getQuantity());
            System.out.println("   Price: " + lineitems[j].getPrice());
            numberOfItems += lineitems[j].getQuantity();
            totalAmount += lineitems[j].getPrice() * lineitems[j].getQuantity();
        }
        System.out.println("Total items: " + numberOfItems);
        System.out.println("Total amount: " + totalAmount);
    }
}
```

Notice that types generated from the schema reflect what's in the XML:

- A PurchaseOrderDocument represents the global root element.
- A getPurchaseOrder method returns a PurchaseOrderDocument.PurchaseOrder type that contains child elements, including line−item. A getLineItemArray method returns a LineItem array containing the line−item elements.
- Other methods, such as getQuantity, getPrice, and so on, follow naturally from what the schema describes, returning corresponding children of the line−item element.
- The name of the package containing these types is derived from the schema's target namespace.

Capitalization and punctuation for generated type names follow Java convention. Also, while this example parses the XML from a file, other parse methods support a Java InputStream object, a Reader object, and so on.

The preceding Java code prints the following to the console:

```
Purchase order has 3 line items.
 Line item 0
   Description: Burnham's Celestial Handbook, Vol 1
   Quantity: 2
   Price: 21.79
 Line item 1
   Description: Burnham's Celestial Handbook, Vol 2
   Quantity: 2
   Price: 19.89
Total items: 4
Total amount: 41.68
```

## Creating New XML Instances from Schema

As you've seen XMLBeans provides a "factory" class you can use to create new instances. The following example creates a new purchase−order element and adds a customer child element. It then inserts name and address child elements, creating the elements and setting their values with a single call to their set methods.

```
public PurchaseOrderDocument createPO()
{
    PurchaseOrderDocument newPODoc = PurchaseOrderDocument.Factory.newInstance();
    PurchaseOrder newPO = newPODoc.addNewPurchaseOrder();
    Customer newCustomer = newPO.addNewCustomer();
    newCustomer.setName("Doris Kravitz");
    newCustomer.setAddress("Bellflower, CA");
    return newPODoc;
}
```

The following is the XML that results. Note that XMLBeans assigns the correct namespace based on the schema, using an "ns1" (or, "namespace 1") prefix. For practical purposes, the prefix itself doesn't really matter — it's the namespace URI (http://openuri.org/easypo) that defines the namespace. The prefix is merely a marker that represents it.

```
<ns1:purchase−order xmlns:ns1="http://openuri.org/easypo">
    <ns1:customer>
        <ns1:name>Doris Kravitz</ns1:name>
        <ns1:address>Bellflower, CA</ns1:address>
    </ns1:customer>
</ns1:purchase−order>
```

Note that all types (including those generated from schema) inherit from XmlObject, and so provide a Factory class. For an overview of the type system in which XmlObject fits, see XMLBeans Support for Built−In Schema Types. For reference information, see XmlObject Interface.

# XMLBeans Hierarchy

The generated types you saw used in the preceding example are actually part of a hierarchy of XMLBeans types. This hierarchy is one of the ways in which XMLBeans presents an intuitive view of schema. At the top of the hierarchy is XmlObject, the base interface for XMLBeans types. Beneath this level, there are two main type categories: generated types that represent user−derived schema types, and included types that represent built−in schema types.

This topic has already introduced generated types. For more information, see Java Types Generated from User−Derived Schema Types.

## Built−In Type Support

In addition to types generated from a given schema, XMLBeans provides 46 Java types that mirror the 46 built−in types defined by the XML schema specification. Where schema defines xs:string, xs:decimal, and xs:int, for example, XMLBeans provides XmlString, XmlDecimal, and XmlInt. Each of these also inherits from XmlObject, which corresponds to the built−in schema type xs:anyType.

XMLBeans provides a way for you to handle XML data as these built−in types. Where your schema includes an element whose type is, for example, xs:int, XMLBeans will provide a generated method designed to return an XmlInt. In addition, as you saw in the preceding example, for most types there will also be a method that returns a natural Java type such as int. The following two lines of code return the quantity element's value, but return it as different types.

```
// Methods that return simple types begin with an "x".
XmlInt xmlQuantity = lineitems[j].xgetQuantity();
// Methods that return a natural Java type are unadorned.
int javaQuantity = lineitems[j].getQuantity();
```

In a sense both get methods navigate to the quantity element; the getQuantity method goes a step further and converts the elements value to the most appropriate natural Java type before returning it. (XMLBeans also provides a means for validating the XML as you work with it.)

If you know a bit about XML schema, XMLBeans types should seem fairly intuitive. If you don't, you'll learn a lot by experimenting with XMLBeans using your own schemas and XML instances based on them.

For more information on the methods of types generated from schema, see Methods for Types Generated From Schema. For more about the how XMLBeans represents built−in schema types, see XMLBeans Support for Built−In Schema Types.

# Using XQuery Expressions

With XMLBeans you can use XQuery to query XML for specific pieces of data. XQuery is sometimes referred to as "SQL for XML" because it provides a mechanism to access data directly from XML documents, much as SQL provides a mechanism for accessing data in traditional databases.

XQuery borrows some of its syntax from XPath, a syntax for specifying nested data in XML. The following example returns all of the line−item elements whose price child elements have values less than or equal to 20.00:

```
PurchaseOrderDocument doc = PurchaseOrderDocument.Factory.parse(po);

/*
 * The XQuery expression is the following two strings combined. They're
 * declared separately here for convenience. The first string declares
 * the namespace prefix that's used in the query expression; the second
 * declares the expression itself.
 */
String nsText = "declare namespace po = 'http://openuri.org/easypo'";
String pathText = "$this/po:purchase−order/po:line−item[po:price <= 20.00]";
String queryText = nsText + pathText;

XmlCursor itemCursor = doc.newCursor().execQuery(queryText);
System.out.println(itemCursor.xmlText());
```

This code creates a new cursor at the start of the document. From there, it uses the XmlCursor interface's execQuery method to execute the query expression. In this example, the method's parameter is an XQuery expression that simply says, "From my current location, navigate through the purchase−order element and retrieve those line−item elements whose value is less than or equal to 20.00." The $this variable means "the current position."

For more information about XQuery, see XQuery 1.0: An XML Query Language at the W3C web site.

## Using XML Cursors

In the preceding example you may have noticed the XmlCursor interface. In addition to providing a way to execute XQuery expression, an XML cursors offers a fine−grained model for manipulating data. The XML cursor API, analogous to the DOM's object API, is simply a way to point at a particular piece of data. So, just like a cursor helps navigate through a word processing document, the XML cursor defines a location in XML where you can perform actions on the selected XML.

Cursors are ideal for moving through an XML document when there's no schema available. Once you've got the cursor at the location you're interested in, you can perform a variety of operations with it. For example, you can set and get values, insert and remove fragments of XML, copy fragments of XML to other parts of the document, and make other fine−grained changes to the XML document.

The following example uses an XML cursor to navigate to the customer element's name child element.

```
PurchaseOrderDocument doc =
    PurchaseOrderDocument.Factory.parse(po);

XmlCursor cursor = doc.newCursor();
cursor.toFirstContentToken();
cursor.toFirstChildElement();
cursor.toFirstChildElement();
System.out.println(cursor.getText());

cursor.dispose();
```

What's happening here? As with the earlier example, the code loads the XML from a File object. After loading the document, the code creates a cursor at its beginning. Moving the cursor a few times takes it to the

nested name element. Once there, the getText method retrieves the element's value.

This is just an introduction to XML cursors. For more information about using cursors, see Navigating XML with Cursors.

### Where to Go Next

- XMLBeans provides intuitive ways to handle XML, particularly if you're starting with schema. If you're accessing XML that's based on a schema, you'll probably find it most efficient to access the XML through generated types specific to the schema. To do this, you begin by compiling the schema to generate interfaces. For more information on using XMLBeans types generated by compiling schema, see Java Types Generated From User−Derived Schema Types and Methods for Types Generated From Schema.
- You might be interested in reading more about the type system on which XMLBeans is based, particularly if you're using types generated from schema. XMLBeans provides a hierarchical system of types that mirror what you find in the XML schema specification itself. If you're working with schema, you might find it helps to understand how these types work. For more information, see XMLBeans Support for Built−In Schema Types and Introduction to Schema Type Signatures.
- XMLBeans provides access to XML through XQuery, which borrows path syntax from XPath. With XQuery, you can specify specific fragments of XML data with or without schema. To learn more about using XQuery and XPath in XMLBeans, see Selecting XML with XQuery and XPath.
- You can use the XmlCursor interface for fine−grained navigation and manipulation of XML. For more information, see Navigating XML with Cursors.

***Note:*** The xbean.jar file that contains the XMLBeans library is fully functional as a standalone library.

## Related Topics

XMLBeans Samples

## Java Types Generated from User−Derived Schema Types

When you compile XML schema, the resulting API is made up of two categories of types: built−in types that mirror those in the schema specification and others that are generated from user−derived schema types. This topic provides an overview of the Java types generated for user−derived types, describing the methods the Java types provide. For more information about built−in types, see XMLBeans Support for Built−In Schema Types. For specific information about the methods exposed by generated types, see Methods for Generated Java Types.

In general, an API generated from schema is an intuitive means to access XML instances based on the schema. You'll probably find that for most uses it's unnecessary to know the rules for generating it in order to use it. However, for those cases when it's unclear what's going on behind the scenes (or if you're just curious), this topic describes the rules.

***Note:*** The XMLBeans API also provides a way for you to get information *about* the type system itself — in other words, about the API and the underlying schema. For more information, see Introduction to Schema Type Signatures.

Each of the types generated when you compile a schema is designed specifically for access to XML instances conforming to that part of the schema. Start by taking a look at a simple XML and schema example. The

following schema describes an XML document to contain a stock price quote.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:element name="price-quote">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="stock-symbol" type="xs:string"/>
                <xs:element name="stock-price" type="xs:float"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
</xs:schema>
```

The following is an example of XML that conforms to this schema.

```
<price-quote>
    <stock-symbol>BEAS</stock-symbol>
    <stock-price>59.21</stock-price>
</price-quote>
```

When you compile this schema, you get two generated XMLBeans interfaces: PriceQuoteDocument and PriceQuoteDocument.PriceQuote.

From the schema point of view, the generated PriceQuote interface represents the *complex type* you see inside the schema's price−quote element declaration. Looking at the XML instance, you can see that this complex type translates into a sequence of two elements, stock−symbol and stock−price. So it's not surprising that the PriceQuote interface exposes methods such as getStockPrice and setStockPrice to set the value stock−price element.

The PriceQuoteDocument interface, on the other hand, represents the price−quote *document* that contains the root price−quote element. XMLBeans creates a special "document" type for global element types. A document type provides a way for you to get and set the value of the underlying type, here represented by PriceQuote. The price−quote element is considered a *global* element because it can be referenced from anywhere else in the schema. For global elements, the XMLBeans schema compiler generates an interface whose name ends with "Document." This is because an XML schema has no way of defining a "root" element; any global element can be the root.

The following bit of Java code illustrates how you might use these interfaces to get the stock price contained in the XML.

```
public static float getStockPrice(java.io.File orderXML) throws Exception
{
    PriceQuoteDocument docXML = PriceQuoteDocument.Factory.parse(orderXML);
    PriceQuote quoteXML = docXML.getPriceQuote();
    float stockPrice = quoteXML.getStockPrice();
    return stockPrice;
}
```

This code loads the XML from a File object, converting the parse method's return value to a PriceQuoteDocument instance. It then uses this instance to get an instance of PriceQuote. With PriceQuote, the code extracts the stock price.

The XML schema specification provides a rich set of rules through which you can derive new types. When you generate interfaces from your schema, XMLBeans uses the schema's rules to determine how to generate

interfaces. The following describes some of the guidelines by which this is done.

## Names for Interfaces

Interfaces are generated for schema types (both simple and complex). Anonymous schema types result in inner interfaces inside the type interface in which they are defined. Their name comes from the element or attribute in which they is defined.

Names for schema types become Java−friendly names when the schema is compiled. In other words, names such as "price−quote" become "PriceQuote." In addition, a schema's XML namespace URIs become package names for XMLBean types generated from the schema. The way this name translation is done is described by section C of the Java API for XML Binding (JAXB) specification at http://java.sun.com/xml/jaxb.html.

Here are a few examples:

| ***Schema Target Namespace*** | ***XML Localname*** | ***Fully−Qualified XMLBean Type Name*** |
|---|---|---|
| http://www.mycompany.com/2002/buyer | purchase−order−4 | com.mycompany.x2002.buyer.PurchaseOrder4 |
| http://myco.com/sample.html | SampleDocument | com.myco.sample.SampleDocument |
| http://openuri.org/test_case_1 | test_type | org.openuri.testCase1.TestType |

When there are name collisions, the generated types will have names with numerals appended — for example, "TestType2".

***Note:*** Due to a Windows operating system limitation on file path lengths (256 characters), you might have trouble compiling schemas in which the generated types are nested in a very deep package hierarchy. This may result in an error message stating that a particular class couldn't be found. To work around this limiitation, try using an xsdconfig file to guide the naming of generated packages so that the hierarchy is less deep, package names are shorter, etc. For more information, see How Do I: Guide XMLBeans Type Naming During Schema Compilation?

## Global Elements and Attributes

In schema, global element and attribute definitions are those that are declared at the top level of the schema (that is, immediately within the schema root element). Because they are global, they may be referenced from inside the schema by name. The creditReport (not the creditReportType complex type) element defined in the following schema is an example.

```
<xs:schema targetNamespace="http://openuri.org/samples/creditReport"
    xmlns:cr="http://openuri.org/samples/creditReport"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    elementFormDefault="qualified"
    attributeFormDefault="unqualified">
    <xs:complexType name="creditReportType">
        <xs:sequence>
            <xs:element name="bankReport" type="xs:string"/>
            <xs:element name="taxReport" type="xs:string"/>
        </xs:sequence>
    </xs:complexType>
    <xs:element name="creditReport" type="cr:creditReportType"/>
</xs:schema>
```

XMLBeans generates a separate interface for each of these. Also, global element and attribute types are somewhat unique in that the schema compiler will create special types to hold the globally defined element or attribute. The names of these types will be appended with "Document" (for elements) or "Attribute" (for attributes). You can retrieve the element or attribute itself (or create a new one) by calling the accessor methods that the special types provide. The following example would create a new instance of the creditReport element.

```
// Create an instance of the special document type.
CreditReportDocument reportDoc = CreditReportDocument.Factory.newInstance();
/*
 * Use the document type to add a new creditReport element to the XML instance.
 * Note that the type returned by the addNewCreditReport method is the
 * creditReportType complex type defined for it in schema.
 */
CreditReportType report = reportDoc.addNewCreditReport();
```

## Global User−Derived Types

A *user−derived* type is one defined with a complexType or simpleType element in schema. User−derived types at the top level of a schema are global. XMLBeans generates an interface for each of these, as it does with global elements and attributes. These interfaces include methods through which you can get and set the type's values, including any nested derived types it may contain. The following schema snippet defines a user−derived complex type called itemType, along with a priceType that uses it as the type for an item child element.

```
<xs:complexType name="itemType">
    <xs:sequence>
        <xs:element name="name" type="xs:string"/>
        <xs:element name="amount" type="xs:int"/>
        <xs:element name="price" type="xs:double"/>
    </xs:sequence>
</xs:complexType>
<xs:complexType name="priceType">
    <xs:sequence>
        <xs:element name="item" type="ps:itemType" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
</xs:complexType>
<xs:element name="price" type="ps:priceType"/>
```

By default, the generated Java type for itemType would be an interface called ItemType. With this type, you would be able to get and set the values of its name, amount, and price child elements. However, a user−derived type (as opposed to an element or attribute) is always intended for use as the type to which an element or attribute is bound. In other words, it's contained by an element or attribute. While you can create a new instance of a user−derived type, the new instance's underlying XML is a fragment. As the generated API would make clear, the itemType becomes the return type of a get method, or the parameter of a set method.

```
// Create a new price document.
PriceDocument priceDoc = PriceDocument.Factory.newInstance();
PriceType price = priceDoc.getPrice();

/*
 * Create a new instance of ItemType and set the values of its
 * child elements.
 */
ItemType item = ItemType.Factory.newInstance();
item.setName("bicycle");
```

```
item.setAmount(12);
item.setPrice(560.00);

/*
 * Use the new ItemType instance to set the value of the
 * price element's first item child element. Notice that the set method
 * here is an "Array" method. This is because the item element
 * is defined with a maxOccurs="unbounded" attribute. It can occur
 * many times as a child of price.
 */
price.setItemArray(0, item);
```

## Nested Elements and Derived Types

When your schema includes named types that are declared locally—within the declaration of another element or type—the schema type's generated Java interface will be an inner interface within the type it's nested in.

For example, the following schema snippet defines name and gender elements nested within a person complex type. In particular, note that the gender element is defined as deriving from the xs:NMTOKEN built−in type.

```
<xs:complexType name="person">
    <xs:sequence>
        <xs:element name="name" type="xs:string"/>
        <xs:element name="gender">
            <xs:simpleType>
                <xs:restriction base="xs:NMTOKEN">
                    <xs:enumeration value="male"/>
                    <xs:enumeration value="female"/>
                </xs:restriction>
            </xs:simpleType>
        </xs:element>
    </xs:sequence>
</xs:complexType>
```

The generated interfaces for person and gender would be organized in source something like the following. Of course, you wouldn't see the source, but you can see here that the Gender interface is nested with Person. Also, notice that it extends XmlNMTOKEN, mirroring the schema.

```
public interface Person extends XmlObject
{
    public interface Gender extends XmlNMTOKEN
    {
        // Methods omitted for this example
    }
    // Methods omitted for this example
}
```

You could create a new instance of the Gender type in this way (there are also various alternatives to this):

```
// Create a new Person instance.
Person person = Person.Factory.newInstance();
/*
 * Set the gender element's value using the
 * enumeration generated from the schema.
 */
person.setGender(Gender.FEMALE);
```

## User−Derived Simple Types

In addition to the 46 built−in simple types in XML schema, a schema can include its own custom simple types using xs:simpleType declarations. These user−derived simple types are always based on the built−in XML schema types. The built−in types can be modified by *restricting* them, taking *unions* of them, or making space−separated *lists* of them. Each XML simple type is translated into a Java type that provides access to the underlying data.

### Unions

In schema, you can use xs:union to specify a simple type that is allowed to contain values of a number of other simple types. XMLBeans generates a type for a union, just as it generates a type for any other schema type. At run time, you can discover the underlying type of an instance of a union type by calling the XmlObject interface's instanceType method. Once you have determined the type, you can cast an instance of a union type to the actual underlying instance type.

```
<xs:simpleType name="intOrString">
    <xs:union memberTypes="xs:int xs:string">
</xs:simpleType>
```

Given the preceding schema snippet, you could set the intOrString value to, say, 6 or "six". The union of xs:int and xs:string makes both allowable.

```
// Create a new instance of the type.
IntOrString intOrString = IntOrString.Factory.newInstance();
intOrString.set("5");
// This code prints "XmlInt" to the console.
System.out.println(intOrString.instanceType().getShortJavaName());
```

### Restrictions

XML schema restrictions on simple XMLBeans types are enforced. So, for example, it is illegal to set a number outside its restricted range.

#### Numeric Type Restrictions

In schema, you can restrict numeric types to allow, for example, only a particular range of values. For such a restriction, XMLBeans tailors the resulting natural Java alternative. For example, suppose you have the following element defined in schema:

```
<xs:element name="number">
    <xs:simpleType>
        <xs:restriction base="xs:integer">
            <xs:minInclusive value="1"/>
            <xs:maxInclusive value="1000000"/>
        </xs:restriction>
    </xs:simpleType>
</xs:element>
```

The type is restricted from xs:integer, but because the number's range is limited to between 1 and 1000000, it will fit into a Java int. A long or java.math.BigInteger would be too big for the need. In other words, the getNumber method generated for this type will return an int, rather than a BigInteger or a long.

By the same token, an long can be compiled to an int if the totalDigits attribute is <=9, or the min and max attribute values are within 32−bit 2s complement range.

The single primitive XML type xs:decimal can be restricted in several ways that influence the resulting natural Java type. For example, it can be:

- Compiled to a BigInteger if its fractionDigit attribute is set to 0.
- Compiled to a long if its totalDigits attribute is <=18, or the min and max are within 64−bit 2s complement range.

### Enumerations

In schema, you can derive a new type by restricting a built−in type so that only a finite set of values are allowable. Where schema does this by restricting xs:string, XMLBeans generates a special Enum type. With an Enum, you can select the enumerated value either by its String value or by a numeric index. The index's value is determined based on the String value's order in the schema. Having an index can be useful in Java switch statements.

For example, suppose you had a document containing price elements whose type was the priceType defined in the following schema snippet:

```
<xs:complexType name="priceType">
    <xs:sequence>
        <xs:element name="item" type="ps:itemType" minOccurs="0"
              maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:attribute name="threshold">
        <xs:simpleType>
            <xs:restriction base="xs:string">
                <xs:enumeration value="Below10Dollars"/>
                <xs:enumeration value="Between10And20Dollars"/>
                <xs:enumeration value="Above20Dollars"/>
            </xs:restriction>
        </xs:simpleType>
    </xs:attribute>
</xs:complexType>
```

Using types generated from the schema, you would be able to write the following Java code to "switch" on the threshold attribute's enumeration:

```
/*
 * Use the intValue method provided by the Enum type to determine the threshold
 * attribute's current enumeration value.
 */
switch(priceElements[i].getThreshold().intValue())
{
    // Use the Threshold type's enumeration values to test for an attribute value.
    case PriceType.Threshold.INT_BELOW_10_DOLLARS:
        zeroBuffer.append(item.getTitle() + "\n");
        break;
    case PriceType.Threshold.INT_BETWEEN_10_AND_20_DOLLARS:
        tenBuffer.append(item.getTitle() + "\n");
        break;
    case PriceType.Threshold.INT_ABOVE_20_DOLLARS:
        twentyBuffer.append(item.getTitle() + "\n");
        break;
```

```
    default:
        System.out.println("Yo! Something unexpected happened!");
        break;
}
```

Related Topics

XMLBeans Support for Built−In Schema Types

# Methods for Types Generated From Schema

As you may have seen in Getting Started with XMLBeans, you use the types generated from schema to access XML instances based on the schema. If you're familiar with the JavaBeans technology, the conventions used in the generated API will be recognizable.

In general, elements and attributes are treated as "properties" in the JavaBeans sense. In other words, as you would with JavaBeans properties, you manipulate parts of the XML through accessor methods such as getCustomer() (where you have a "customer" element), setId(String) (where you have an "id" attribute), and so on. However, because schema structures can be somewhat complex, XMLBeans provides several other method styles for handling those structures in XML instances.

## Prototypes for Methods in Generated Interfaces

Several methods are generated for each element or attribute within the complex type. This topic lists each method that could be generated for a given element or attribute.

Note that whether or not a given method is generated is based on how the element or attribute is defined in schema. For example, a customer element definition with a maxOccurs attribute value of 1 will result in a getCustomer method, but not a getCustomerArray method — after all, only one customer element is possible in an instance document.

Note, too, that there may be two sets of parallel methods: one whose prototype starts with an "x". An "x" method such as xgetName or xsetName would be generated for elements or attribute whose type is a simple type. A simple type may be one of the 44 built−in simple types or may be a restriction in schema of one of those built−in types. Of course, an attribute will always be of a simple type. For built−in simple types, an "x" method will get or set one of the types provided with XMLBeans, such as XmlString, XmlInteger, XmlGDay, and so on. For derived types, the "x" method will get or set a generated type.

### Single Occurrence Methods

Methods generated for elements or attributes that allow a single occurrence. An element is singular if it was declared with maxOccurs="1". An attribute is singular if it was not declared with use="prohibited".

```
Type getFoo()
void setFoo(Type newValue)
```

Returns or sets the value of Foo. Generated when Foo is an attribute, or is an element that can occur only once as a child.

---

```
XmlType xgetFoo()
void xsetFoo(XmlType newValue)
```

Returns or sets the value of Foo as an XMLBean simple type. These methods are generated if Foo's type is defined in schema as a simpleType.

---

```
boolean isNilFoo()
void setNilFoo()
```

Determines or specifies whether the Foo element is nil (in other words, "null" in schema terms), meaning it can be empty. A nil element looks something like this:

```
<foo/>
```

These methods are only generated when an element type is declared as nillable in schema — it has a nillable="true" attribute.

---

```
XmlType addNewFoo()
```

Adds a new Foo as an XMLBean simple to the document, or returns Foo's value if one exists already.

---

```
boolean isSetFoo()
void unSetFoo()
```

Determines whether the Foo element or attribute exists in the document; removes Foo. These methods are generated for elements and attributes that are optional. In schema, and optional element has an minOccurs attribute set to "0"; an optional attribute has a use attribute set to "optional".

---

### Multiple Occurrence Methods

Methods generated for elements that allow multiple occurrences.

An element may occur multiple times if it has a maxOccurs attribute set to "unbounded" or greater than 1. Attributes can't occur multiple times.

```
Type[] getFooArray()
void setFooArray(Type[] newValue)
```

Returns or sets all of the Foo elements.

```
// Get an array of the all of the purchase-order elements item children.
Item[] items = myPO.getItemArray();
```

---

```
Type getFooArray(int index)
void setFooArray(Type newValue, int index)
```

Returns or sets the Foo child element at the specified index.

```
// Sets the value of the third item child element.
myPO.setItem(newItem, 2);
```

---

```
int sizeOfFooArray()
```

Returns the number of Foo child elements.

```
// Returns the number of item child elements.
int itemCount = myPO.sizeOfItemArray();
```

---

```
void removeFoo(int index)
```

Removes the Foo child element at the specified index.

---

```
XmlType[] xgetFooArray()
void xsetFooArray(XmlType[] arrayOfNewValues)
```

Returns or sets all of the Foo elements as XMLBeans simple types. Generated only when the

Foo element is defined as a simple type.

```
/*
 * Returns values of all the phone child elements of an employee element,
 * where the phone element has been defined as xs:string.
 */
XmlString[] empPhones = currentEmployee.xGetPhoneArray();
```

---

```
XmlType xgetFooArray(int index)
void xsetFooArray(int index, XmlType newValue)
```

Returns or sets the Foo element at the specified index, using an XMLBeans simple type value. Generated for an element defined as a simple type in schema.

---

```
void insertFoo(int index, FooType newValue)
```

Inserts the specified Foo child element at the specified index.

---

```
void addFoo(FooType newValue)
```

Adds the specified Foo to the end of the list of Foo child elements.

---

```
XmlType insertNewFoo(int index)
```

Inserts a new Foo at the specified index, returning an XMLBeans simple type representing the new element; returns the existing Foo if there's already one at index.

---

```
XmlType addNewFoo()
```

Adds a new Foo element to the end of the list of Foo child elements, returning an XMLBeans simple type representing the newly added element.

---

```
boolean isNilFooArray(int index)
void setNilFooArray(int index)
```

Determines or specifies whether the Foo element at the specified index is nil.

---

Related Topics

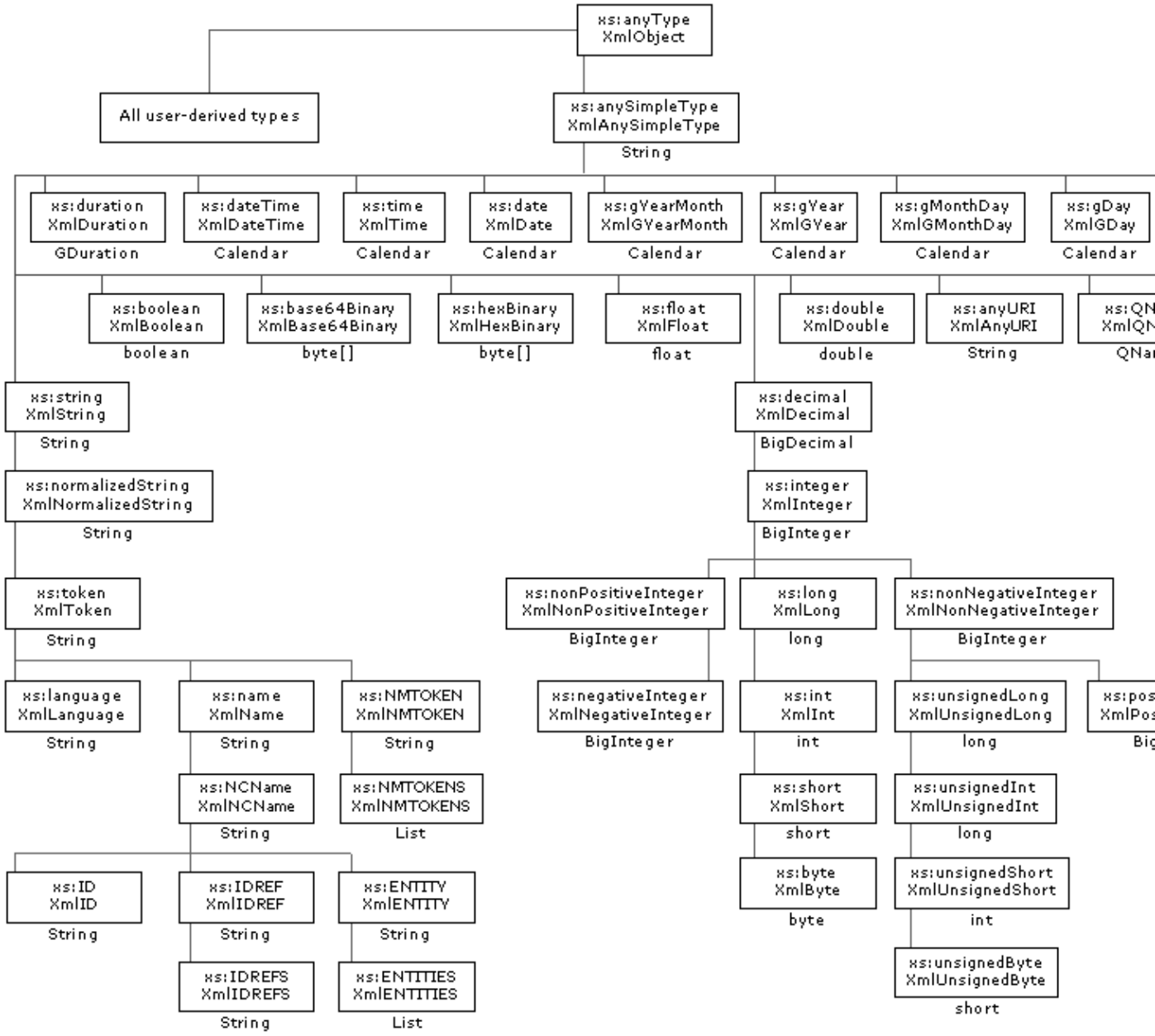
Java Types Generated from User−Derived Schema Types

## XMLBeans Support for Built−In Schema Types

Just as with types in XML schema, the XMLBeans types based on schema are arranged in a hierarchy. In general, the hierarchy of XMLBean types mirrors the hierarchy of the schema types themselves. XML schema types all inherit from xs:anyType (which is at the root of the hierarchy); XMLBeans types inherit from XmlObject.

In XML schema, xs:anyType is the root type for two type categories: built−in and user−derived. Built−in schema types are common types defined by the XML schema specification. User−derived types are those you define in your schema. This topic lists the built−in types provided with XMLBeans. For information about user−derived types and the Java types generated from them, see Java Types Generated from User−Derived Schema Types.

The following figure illustrates the hierarchy of types, showing both built−in schema types and their XMLBeans parallels. Notice, too, that nearly all of the built−in types are also available as natural Java types.



## Java Types Representing Built−In Schema Types

As you can see from the figure above, all 46 built−in schema types are represented by an XMLBean type provided with XMLBeans. When using built−in XMLBean types, keep in mind that:

- Since XmlObject corresponds to the xs:anyType, all Java XMLBean types inherit from XmlObject. Therefore all XMLBean types share the XmlObject interface's ability to create an XmlCursor at the position of the object. They can also render the XML as strings or streams.
- Where there is a corresponding natural Java type, the XMLBeans type provides a way to get the underlying XML's value as the natural type. See the table in this topic for information on how natural Java types map to XMLBeans types.
- Values assigned to built−in XMLBeans types are validated in keeping with the rules of the schema type they represent. So, for example, an attempt to assign a negative integer value to an XmlPositiveInteger will throw an XmlValueOutOfRange exception.

The following table lists the built−in schema types, along with their XMLBeans and natural Java parallels. Unless otherwise noted, natural Java types are in the java.lang package; XMLBean types are in the com.bea.xml package.

| ***Built−In Schema Type*** | ***XMLBean Type*** | ***Natural Java Type*** |
|---|---|---|
| xs:anyType | XmlObject | com.bea.xml.XmlObject |
| xs:anySimpleType | XmlAnySimpleType | String |
| xs:anyURI | XmlAnyURI | String |
| xs:base64Binary | XmlBase64Binary | byte[] |
| xs:boolean | XmlBoolean | boolean |
| xs:byte | XmlByte | byte |
| xs:date | XmlDate | java.util.Calendar |
| xs:dateTime | XmlDateTime | java.util.Calendar |
| xs:decimal | XmlDecimal | java.math.BigDecimal |
| xs:double | XmlDouble | double |
| xs:duration | XmlDuration | com.bea.xml.GDuration |
| xs:ENTITIES | XmlENTITIES | String |
| xs:ENTITY | XmlENTITY | String |
| xs:float | XmlFloat | float |
| xs:gDay | XmlGDay | java.util.Calendar |
| xs:gMonth | XmlGMonth | java.util.Calendar |
| xs:gMonthDay | XmlGMonthDay | java.util.Calendar |
| xs:gYear | XmlGYear | java.util.Calendar |
| xs:gYearMonth | XmlGYearMonth | java.util.Calendar |
| xs:hexBinary | XmlHexBinary | byte[] |
| xs:ID | XmlID | String |
| xs:IDREF | XmlIDREF | String |
| xs:IDREFS | XmlIDREFS | String |
| xs:int | XmlInt | int |
| xs:integer | XmlInteger | java.math.BigInteger |

| | | |
|---|---|---|
| xs:language | XmlLanguage | String |
| xs:long | XmlLong | long |
| xs:Name | XmlName | String |
| xs:NCName | XmlNCNAME | String |
| xs:negativeInteger | XmlNegativeInteger | java.math.BigInteger |
| xs:NMTOKEN | XmlNMTOKEN | String |
| xs:NMTOKENS | XmlNMTOKENS | String |
| xs:nonNegativeInteger | XmlNonNegativeInteger | java.math.BigInteger |
| xs:nonPositiveInteger | XmlNonPositiveInteger | java.math.BigInteger |
| xs:normalizedString | XmlNormalizedString | String |
| xs:NOTATION | XmlNOTATION | Not supported |
| xs:positiveInteger | XmlPositiveInteger | java.math.BigInteger |
| xs:QName | XmlQName | javax.xml.namespace.QName |
| xs:short | XmlShort | short |
| xs:string | XmlString | String |
| xs:time | XmlTime | java.util.Calendar |
| xs:token | XmlToken | String |
| xs:unsignedByte | XmlUnsignedByte | short |
| xs:unsignedInt | XmlUnsignedInt | long |
| xs:unsignedLong | XmlUnsignedLong | java.math.BigInteger |
| xs:unsignedShort | XmlUnsignedShort | int |

Related Topics

Getting Started with XMLBeans

# Selecting XML with XQuery and XPath

You can use XQuery and XPath to retrieve specific pieces of XML as you might retrieve data from a database. XQuery and XPath provide a syntax for specifying which elements and attributes you're interested in. The XMLBeans API provides two methods for executing XQuery and XPath expressions, and two differing ways to use them. The methods are selectPath and execQuery, and you can call them from XmlObject (or an object inheriting from it) or XmlCursor. The results for the methods differ somewhat.

## Using the selectPath Method

The selectPath method is the most efficient way to execute XPath expressions. The selectPath method is optimized for XPath. When you use XPath with the selectPath method, the value returned is an array of values from the *current document*. In contrast, when you use execQuery, the value returned is a *new document*.

### Calling from XmlObject

When called from XmlObject (or a type that inherits from it), this method returns an array of objects. If the expression is executed against types generated from schema, then the type for the returned array is one of the Java types corresponding to the schema.

For example, imagine you have the following XML containing employee information. You've compiled the schema describing this XML and the types generated from schema are available to your code.

```
<xq:employees xmlns:xq="http://openuri.org/selectPath">
    <xq:employee>
        <xq:name>Fred Jones</xq:name>
        <xq:address location="home">
            <xq:street>900 Aurora Ave.</xq:street>
            <xq:city>Seattle</xq:city>
            <xq:state>WA</xq:state>
            <xq:zip>98115</xq:zip>
        </xq:address>
        <xq:address location="work">
            <xq:street>2011 152nd Avenue NE</xq:street>
            <xq:city>Redmond</xq:city>
            <xq:state>WA</xq:state>
            <xq:zip>98052</xq:zip>
        </xq:address>
        <xq:phone location="work">(425)555-5665</xq:phone>
        <xq:phone location="home">(206)555-5555</xq:phone>
        <xq:phone location="mobile">(206)555-4321</xq:phone>
    </xq:employee>
</xq:employees>
```

If you wanted to find the phone numbers whose area code was 206, you could capture the XPath expression in this way:

```
String queryExpression =
    "declare namespace xq='http://openuri.org/selectPath'" +
    "$this/xq:employees/xq:employee/xq:phone[contains(., '(206)')]"
```

Notice in the query expression that the variable $this represents the current context node (the XmlObject that you are querying from). In this example you are querying from the document level XmlObject.

You could then print the results with code such as the following:

```
/*
 * Retrieve the matching phone elements and assign the results to the corresponding
 * generated type.
 */
PhoneType[] phones = (PhoneType[])empDoc.selectPath(queryExpression);
/*
 * Loop through the results, printing the value of the phone element.
 */
for (int i = 0; i < phones.length; i++)
{
    System.out.println(phones[i].stringValue());
}
```

### Calling from XmlCursor

When called from an XmlCursor instance, the selectPath method retrieves a list of selections, or locations in the XML. The selections are remembered by the cursor instance. You can use methods such as toNextSelection to navigate among them.

The selectPath method takes an XPath expression. If the expression returns any results, each of those results is added as a selection to the cursor's list of selections. You can move through these selections in the way you

might use java.util.Iterator methods to move through a collection.

For example, for a path such as $this/employees/employee, the results would include a selection for each employee element found by the expression. Note that the variable $this is always bound to the current context node, which in this example is the document. After calling the selectPath method, you would use various "selection"−related methods to work with the results. These methods include:

- getSelectionCount() to retrieve the number of selections resulting from the query.
- toNextSelection() to move the cursor to the next selection in the list (such as to the one pointing at the next employee element found).
- toSelection(int) to move the cursor to the selection at the specified index (such as to the third employee element in the selection).
- hasNextSelection() to find out if there are more selections after the cursor's current position.
- clearSelections() clears the selections from the current cursor. This doesn't modify the document (in other words, it doesn't delete the selected XML); it merely clears the selection list so that the cursor is no longer keeping track of those positions.

The following example shows how you might use selectPath, in combination with the push and pop methods, to maneuver through XML, retrieving specific values.

```
public void printZipsAndWorkPhones(XmlObject xml)
{
    // Declare the namespace that will be used.
    String xqNamespace =
        "declare namespace xq='http://openuri.org/selectPath'";

    // Insert a cursor and move it to the first element.
    XmlCursor cursor = xml.newCursor();
    cursor.toFirstChild();
    /*
     * Save the cursor's current location by pushing it
     * onto a stack of saved locations.
     */
    cursor.push();
    // Query for zip elements.
    cursor.selectPath(xqNamespace + "$this//xq:zip");
    /*
     * Loop through the list of selections, getting the value of
     * each element.
     */
    while (cursor.toNextSelection())
    {
        System.out.println(cursor.getTextValue());
    }
    // Pop the saved location off the stack.
    cursor.pop();
    // Query again from the top, this time for work phone numbers.
    cursor.selectPath(xqNamespace + "$this//xq:phone[@location='work']");
    /*
     * Move the cursor to the first selection, them print that element's
     * value.
     */
    cursor.toNextSelection();
    System.out.println(cursor.getTextValue());
    // Dispose of the cursor.
    cursor.dispose();
}
```

Using selections is somewhat like tracking the locations of multiple cursors with a single cursor. This becomes especially clear when you remove the XML associated with a selection. When you do so the selection itself remains at the location where the removed XML was, but now the selection's location is immediately before the XML that was after the XML you removed. In other words, removing XML created a kind of vacuum that was filled by the XML after it, which shifted up into the space — up into position immediately after the selection location. This is exactly the same as if the selection had been another cursor.

Finally, when using selections keep in mind that the list of selections is in a sense "live". The cursor you're working with is keeping track of the selections in the list. In other words, be sure to call the clearSelections method when you're finished with the selections, just as you should call the XmlCursor.dispose() method when you're finished using the cursor.

## Using the execQuery Method

Use the execQuery method to execute XQuery expressions that are more sophisticated than paths. These expressions include more sophisticated loops and FLWR (For, Let, Where, and Results) expressions.

***Note:*** Be sure to see the simpleExpressions sample in the SamplesApp application for a sampling of XQuery expressions in use.

### Calling from XmlObject

Unlike selectPath, calling execQuery from an XmlObject instance will return an XmlObject array. If the XmlObject instances resulting from the XQuery match a recognized XMLBeans type (the namespace and top level element name match up with an XMLBeans type) then the XmlObject will be typed; otherwise the XmlObject will be untyped.

### Calling from XmlCursor

Calling execQuery from an XmlCursor instance returns a new XmlCursor instance. The cursor returned is positioned at the beginning of a new xml document representing the query results, and you can use it to move through the results, cursor−style (for more information, see Navigating XML with Cursors). If the document resulting from the query execution represents a recognized XMLBeans type (the namespace and top level element name match up with an XMLBeans type) then the document resulting from the xquery will have that Java type; otherwise the resulting document will be untyped.

## Related Topics

Getting Started with XMLBeans

# Navigating XML with Cursors

XML cursors are a way to navigate through an XML instance document. Once you load an XML document, you can create a cursor to represent a specific place in the XML. Because you can use a cursor with or without a schema corresponding to the XML, cursors are an ideal way to handle XML without a schema.

With an XML cursor, you can:

- Use the token model to move through XML in small increments, or in a manner similar to using a DOM−based model.

- Get and set values within the XML.
- Change the structure of an XML document by inserting, removing, and moving elements and attributes.
- Execute XQuery expressions against the XML represented by the cursor.
- Insert bookmarks to mark locations in XML.

When you're finished using a cursor, your code should call its dispose method.

## Creating and Moving a Cursor

With an XML instance document bound to XmlObject (or a type inheriting from it), you create a new cursor by calling the newCursor method. The XmlCursor interface represents a cursor. From a cursor standpoint, an XML document is a collection of *tokens* that represent the kinds of things that can appear in XML. These include attributes, the start and end of elements, comments, and so on. Each piece of information in XML is represented by a *token type*.

***Note:*** For a more complete description of XML tokens, see Understanding XML Tokens.

For example, the following code loads the XML instance described above from a File object, then creates a new cursor. The toFirstChild takes the cursor to the start tag of the batchWidgetOrder document element. The code then prints the type for the token at the cursor's location, along with the XML the cursor represents—in other words, Token type: START / and the batchWidgetOrderElement and its contents.

```
public static void insertCursor(File orderFile) throws Exception
{
    BatchWidgetOrderDocument xmlDoc = BatchWidgetOrderDocument.Factory.parse(orderFile);
    XmlCursor orderCursor = xmlDoc.newCursor();
    orderCursor.toFirstChild();
    System.out.println("Token type: " + orderCursor.currentTokenType() +
        " / " + orderCursor.xmlText());
}
```

***Note***: The XmlCursor interface provides many methods you can use to put a cursor where you want it. For a list of those methods, see XmlCursor Interface.

## Adding Elements and Attributes

The XmlCursor interface provides several methods you can use to add elements and attributes to XML.

One way to add new XML is with the beginElement method. This method is designed to insert a new element at the cursor's location, and do it so the cursor ends up between the new element's START and END tokens. From this position, you can insert attributes (they're automatically placed in the start tag, where they belong) and insert a value. Here's an example:

```
// Create a new chunk of XML.
XmlObject newXml = XmlObject.Factory.newInstance();
/*
 * Insert a new cursor and move it to the first START token (where the
 * XML actually begins.
 */
XmlCursor cursor = newXml.newCursor();
cursor.toNextToken();
// Begin a new item element whose namespace URI is "http://openuri.org".
```

```
cursor.beginElement("item", "http://openuri.org/");
// Insert an ID attribute on the item element, along with an attribute value.
cursor.insertAttributeWithValue("id", "4056404");
// Insert "bicycle" as an element value.
cursor.insertChars("bicycle");
cursor.dispose();
```

This example results in something like the following:

```
<ns1:item id="4056404" xmlns:ns1="http://openuri.org/">bicycle</ns1:item>
```

## Using Stored Cursor Locations with push() and pop()

When you want to move a cursor around, but want to keep track of a former location, you can use the XmlCursor interface's push and pop methods. The push method pushes the cursor's current location onto a stack of locations maintained for that particular cursor; the pop method removes the location from the top of the stack and moves the cursor to that location.

For example, consider the following <employee> element, used in the example below.

```
<employee>
    <name>Gladys Kravitz</name>
    <address location="home">
        <street>1313 Mockingbird Lane</street>
        <city>Seattle</city>
        <state>WA</state>
        <zip>98115</zip>
    </address>
    <address location="work">
        <street>2011 152nd Avenue NE</street>
        <city>Redmond</city>
        <state>WA</state>
        <zip>98052</zip>
    </address>
    <phone location="work">(425) 555-6897</phone>
    <phone location="home">(206) 555-6594</phone>
    <phone location="mobile">(206) 555-7894</phone>
</employee>
```

The following Java code illustrates how you can use push and pop to put the cursor back to a saved location after a bit of traveling.

```
/**
 * Pass to the trySelectPath method an XmlObject instance that contains
 * the XML above.
 */
public void trySelectPath(XmlObject xml)
{
    /*
     * Inserts the cursor at the STARTDOC token (the very beginning,
     * before any elements).
     */
    XmlCursor cursor = xml.newCursor();
    // Moves the cursor to just before <employee>
    cursor.toFirstChild();
    // Pushes the cursor's current location onto the stack.
    cursor.push();
```

```
    // Moves the cursor to just before the "work" <phone> element.
    cursor.toChild(2);
    // Moves the cursor to just before the "home" <phone> element.
    cursor.toNextSibling();
    // Moves the cursor back to just before <employee>
    cursor.pop();
}
```

Of course, you can call push and pop multiple times. Each new call to the push method pushes the current location onto the stack. As you call the pop method, you're always getting what's on top of the stack. So if you called push three times before calling pop — 1, 2, 3 — calling pop three times would get those locations in reverse order — 3, 2, 1.

The push and pop methods can be handy as an alternative to creating new cursors that are designed simply to mark a particular location while you move another cursor around. The resources required to maintain a location stack through push and pop are far less than those needed by cursors.

## Disposing of a Cursor

When you're through with a cursor, your code should call its dispose method to indicate that it's no longer needed.

Related Topics

Understanding XML Tokens

Getting Started with XMLBeans

# Understanding XML Tokens

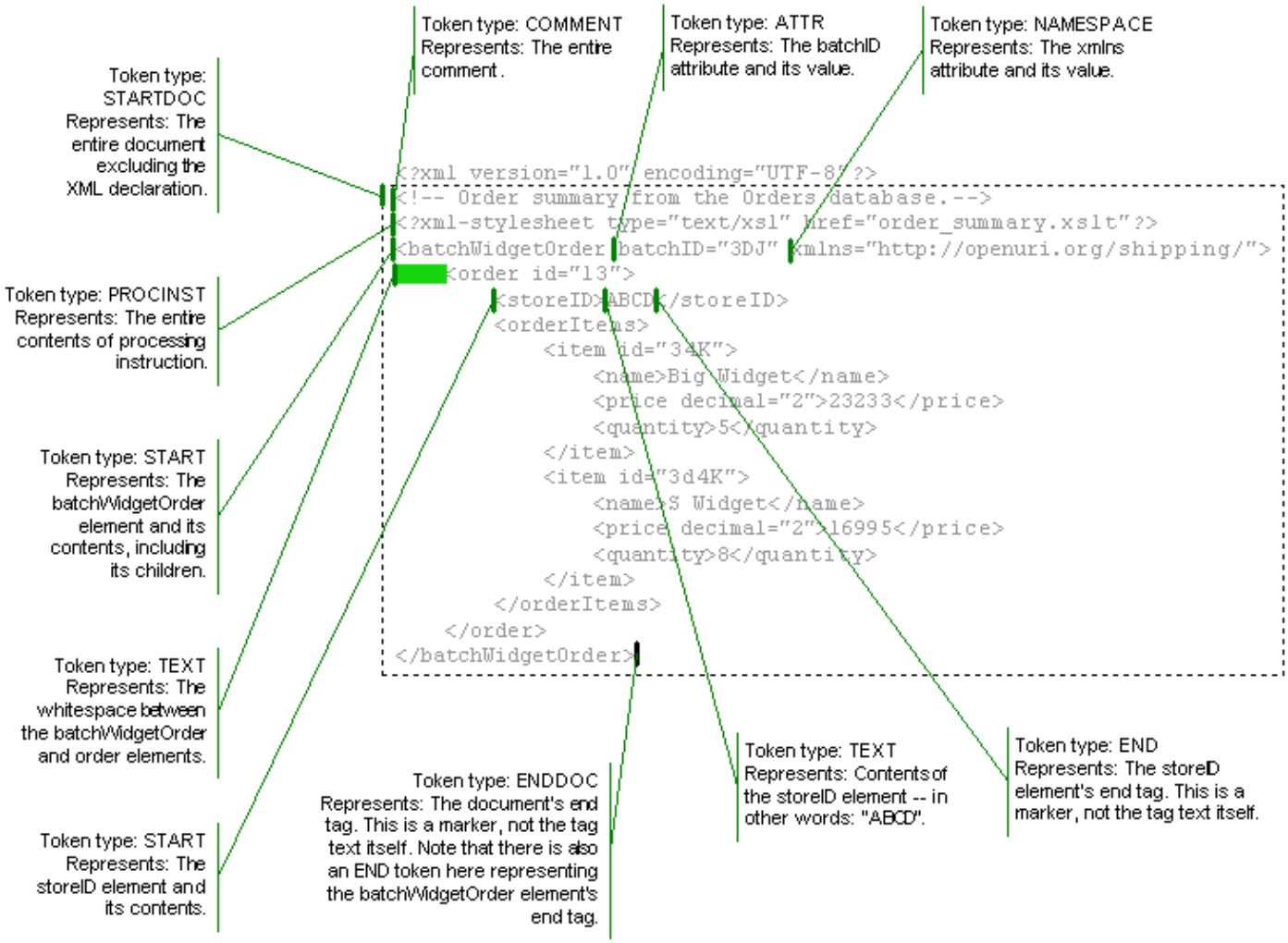
An XML cursor (an instance of the XmlCursor interface) moves from token to token as your code moves the cursor. When you move a cursor using a method such as toParent, toFirstAttribute, toPrevSibling, and so on, the cursor moves to the token fitting the description. If there is no appropriate token to move to, the cursor remains where it is, and false is returned to indicate that it didn't move. For example, if the cursor is at the ENDDOC token (the last tag in the document), a call to the toNextSibling method will not move the cursor and will return false to indicate that the move was unsuccessful.

Note that while you can call the XmlCursor.currentTokenType method to find out which token type the cursor is at, you might find it more convenient to use a method such as isEnddoc. The XmlCursor interface provides several methods that make it easy to discover whether the cursor you're moving is at the token you're looking for. These methods, such as isStart, isAttr, isText, and so on, return a boolean value that indicates whether the cursor is at the token type in question.

Tokens are represented by constants in the TokenType class, an inner class of the XmlCursor interface. Each has a constant you can use in switch statements to perform by−token actions. The following table lists the token types:

| *Token Type* | *Switch Constant* | *Description* |
|---|---|---|
| STARTDOC | INT_STARTDOC | Represents the start of the XML. Always the first token. The document element itself is represented by a START token, not the STARTDOC token. |
| ENDDOC | INT_ENDDOC | Represents the end of the XML. Always the last token. |
| START | INT_START | Represents the start of an element. |
| END | INT_END | Represents the end of an element. The END token has no value, but marks the element's end. |
| TEXT | INT_TEXT | Represents text. |
| ATTR | INT_ATTR | Represents an attribute. ATTR tokens are allowed to appear after a STARTDOC or START token. |
| NAMESPACE | INT_NAMESPACE | Represents a namespace (xmlns) attribute. Also only allowed after START or STARTDOC tokens. |
| COMMENT | INT_COMMENT | Represents a comment. |
| PROCINST | INT_PROCINST | Represents a processing instruction. |

As you use a cursor to navigate through XML, you can use one of the convenience methods described above to discover whether you're at the token you're looking for, or use the XmlCursor.currentTokenType method to discover the current token's type. The following figure illustrates example locations for token types:




## Switching to Test for Token Types

Here's a bit of code illustrating how you might use a Java switch statement to test for the START token type.

```
// Take an incoming XmlObject and insert a cursor.
XmlCursor documentCursor = xmlDoc.newCursor();

/*
 * Loop through the document, passing the cursor when it stops at each token
 * to a function designed to discover the token type. Continue the loop
 * as long at the cursor is at a token (until it reaches the end).
 */
while (!documentCursor.toNextToken().isNone())
{
    /*
     * Use the intValue method to return the int corresponding to the
         * current token type. If it is the value for INT_START,
         * then you have a match.
         */
    switch (cursor.currentTokenType().intValue())
    {
        case TokenType.INT_START:
        // Print out the token type and a message.
        System.out.println(cursor.currentTokenType() +
```

```
            "; cursor is at the start of an element.");
        break;
    }
}
// Be sure to dispose of a cursor when you're finished.
documentCursor.dispose();
```

The scope of an XML cursor is the XML document in which it is created. For example, you can create a cursor at the orderItem element in the example earlier in this topic. If you then use that XmlCursor instance's toNextToken method to move the cursor until it won't move any further, you'll have reached the ENDDOC token. In this example, that's at the </batchWidgetOrder> tag. In other words, the cursor's scope is not limited to the element at which it was created.

Related Topics

XmlCursor.TokenType Class

Navigating XML with Cursors

# Using Bookmarks to Annotate XML

You can use a cursor to insert bookmarks that annotate XML with markers containing information you design. These bookmarks aren't written into the XML itself, but in a sense "hang" from the location where it was inserted. In this way you can associate arbitrary pieces of information with specific parts of the XML.

You design your own bookmark classes by extending XmlBookmark, a nested class of XmlCursor. You can design your bookmark class to contain information specific to your needs.

In the following example, OrderBookmark is an inner class that extends XmlCursor.XmlBookmark. It merely stores a piece of text.

```
/*
 * The OrderBookmark class includes a constructor through which you will
 * insert your bookmark's content. It also includes get and set methods
 * you can use to retrieve and change the content.
 */
static class OrderBookmark extends XmlCursor.XmlBookmark {
    public OrderBookmark (String text) { TEXT = text; }
    public String TEXT;
    public String getText() { return TEXT; }
    public void setText(String newText) { TEXT = newText; }
}
```

You can use instances of this class to store bookmarks at places in your XML. The following excerpt of code creates a cursor and an instance of the OrderBookmark class. Then it uses the cursor to insert the bookmark at the cursor's current location.

```
    XmlCursor orderCursor = xmlDoc.newCursor();
    OrderBookmark thisBookmark = new OrderBookmark("foo");
    orderCursor.setBookmark(thisAnnotation);
```

Related Topics

Navigating XML with Cursors

## Introduction to Schema Type Signatures

When you compile schema, the API generated from your schema is integrated with the XMLBeans type system that represents the underlying XML schema. All together, these types make up the *schema type system* to which your code has access. When handling XML based on the schema, you typically call methods of the API generated when you compiled the schema. However, for the cases when you want to get information about the schema itself, you use the schema type system API.

In the XMLBeans API, you have access to the system itself through SchemaTypeSystem and related classes. These make up a kind of "meta−API," or a view on the schema. You can use the schema type system API to discover the type system at run time. See the reference topic on that interface for an overview of the schema type system.

## Schema Type "Signatures"

A schema is made up of schema components. Schema components are the pieces of a schema, such as a type definition, an element declaration, attribute declaration, and so on. To mirror these in the schema type system, a SchemaComponent instance represents a component in the underlying schema; separate components have corresponding types. For example you would have a SchemaType object for a CustomerType your schema defined, or a SchemaGlobalElement object for a global PurchaseOrder element. You would also have a SchemaType for built−in schema types, such as xs:string or xs:datetime. XMLBean provides a "signature" to describe each type. You can retrieve this signature by calling the SchemaType class's toString method.

The toString method returns XMLBeans' version of a unique signature for a schema type. This string is useful for debugging because it describes a given type even when the type doesn't have a name.

***Note:*** It's important to remember that this signature is an XMLBeans convention, rather than a standard from the schema working group. The working group has not yet standardized a signature for XML schema types. As a result the signature you'll see from XMLBeans is subject to change — whatever the schema working group comes up with in the end (if anything) is probably what will be used by this API. In other words, don't write a program to decode the signature.

You can use the following description to understand how a signature is constructed.

- ***Global types.*** If the type has a name, it's a global type. The following form is used:
  `T=<localname>@<targetNamespace>`

  The "T" is for "type," of course. "localname" is a convention used by qnames (qualified names), which include a local name and the namespace URI (if any). So an example might be:

  ```
  T=customer@openuri.org
  ```
- ***Document types and global attribute types.*** These correspond to a special anonymous type containing one global element or attribute. These special types are generated by XMLBeans to represent global types declared with the <element> or <attribute> tag in schema. Because such types are types, but are declared as elements or attributes, they require special treatment. The following signature form is used:
  ```
  D=<document−element−name>@<targetNamespace>
  R=<attribute−type−name>@<targetNamespace>
  ```

  Note that these are also the signatures of a type returned by a FooDocument.type or FooAttribute.type method call.
- ***Anonymous types.*** If the type is anonymous, it is defined as an element or attribute, or within a further anonymous type. In this case, the signature is built by establishing the local context (in order words, what is the anonymous type nested in?). From the local context, the larger context is built recursively. In other words, the signature is built by giving not only the anonymous type itself, but also by describing its context.

  The following rules are used for building a signature for an anonymous type.

  - ♦ It might be an anonymous type defined inside a local element or attribute, which in turn is defined within something else:

    If the element is form="qualified" (the usual default):

```
E=<eltname>|<signature of the type within which the elt is defined>
```

If the element is form="unqualified":
```
U=<eltname>|<signature of the type within which the elt is defined>
```

If the attribute is form="unqualified" (the usual default):
```
A=<attrname>|<signature of the type within the attr is defined>
```

if the attribute is form="qualified":
```
Q=<attrname>|<signature of the type within the attr is defined>
```

- It might be an anonymous type defined a simple restriction, union, or list definition:
```
M=#|<signature of the containing union type>
```

(The # is a number indicating which union member it is, by source order — such as 0,1,2, etc.)

```
B=|<signature of the containing base type for a restriction>
```

```
I=|<signature of the containing list type>
```

- In the future if anonymous types are allowed in some other context, there may be more codes.

## An Example

So, for example, if you have a type that describes the list items within an attribute of an instance that looks like this:

```
<root mylist="432 999 143 123"/>
```

The schema, if done with lots of nested types, could look something like this:

```
<schema targetNamespace="myNamespace" elementFormDefault="qualified">
    <element name="root">
        <complexType>
            <attribute name="mylist">
                <simpleType>
                    <list>
                        <simpleType> <!--This is the type that the signature is for -->
                            <restriction base="xs:nonNegativeInteger">
                                <totalDigits value="3"/>..
```

The signature for the simpleType indicated in the example would be:
```
I=|A=mylist|E=root|D=root@myNamespace
```

You could read this as:

> "The type of the list item | within the type of the mylist attribute's type | within the type of the root element | within the document type for <root> documents | in the myNamespace namespace".

Note that the signature structure mirrors the Java class structure generated by XMLBeans when compiling the schema. In other words, if you were to compile a schema that included the preceding snippet, you would be able to access an instance of the schema with Java code like the following:

```
SchemaType sType = RootDocument.Root.MyList.Item.type;
```

Related Topics

Getting Started with XMLBeans