# Using the XDoc class to create and manipulate XML documents

**Assembly**: mindtouch.dream
**Namespace**: MindTouch.Dream

XDoc is the Swiss army knife for all things XML.  It is a lightweight wrapper around the XmlDocument, XmlElement, XmlText, and XmlNodeList classes, making it easy to create, query, and compose XML documents.  With XDoc, you don't have to know any of the .Net XML classes because all operations result in new XDoc instances, which keeps everything really simple!

Let's see how XDoc is used in practice.

## Creating a Document

Let's create the following document:

```
<doc />
```

With XDoc, this is straighforward:

```
XDoc d = new XDoc("doc");
```

Each XDoc corresponds to an XML element node.  Therefore, when creating an XDoc, you must specify the name of the root element node of the document.

## Adding Elements

Let's create the following document:

```
<doc>
        <tag>Some text</tag>
</doc>
```

With XDoc, this is straighforward:

```
XDoc d = new XDoc("doc").Elem("tag", "Some text");
```

-OR-

```
XDoc d = new XDoc("doc").Start("tag").Value("Some text").End();
```

All XDoc methods return an XDoc reference, making it easy to chain multiple operations together. Each XDoc instance represents an XML element in the document. The Start() method creates a new element, makes it the active element, and returns the current XDoc. Similarly, the End() method sets the active element to being the parent element. And the Value() method appends a new text node to the current element. The Value() method has several overloads to make it easy to append other kinds of data like booleans, integers, floating-point values, and date-time values. Adding multiple text nodes is the same as adding one text node with all strings concatenated together.

## Adding Attributes

Let's create the following document:

```
<doc>
        <tag name="value">Some text</tag>
</doc>
```

With XDoc, this is straighforward:

```
XDoc d = new XDoc("doc").Start("tag").Attr("name", "value").Value("Some text").End();
```

-OR-

```
XDoc d = new XDoc("doc").Start("tag").Value("Some text").Attr("name", "value").End();
```

Similarly to the Value() method, the Attr() method adds an XML attribute to the current element and returns the current element.

## Accessing Elements

Let's assume we have the following document in an XDoc variable called *d*:

```
<doc>
        <tag>FIRST</tag>
        <tag>SECOND</tag>
        <tag>THIRD</tag>
        <node>
                <entry>ALPHA</entry>
        </node>
        <node>
                <entry>BETA</entry>
        </node>
        <node>
                <entry>GAMMA</entry>
        </node>
</doc>
```

Now, let's look at some access expressions.  XDoc provides three means to refer to an element: by index, by name, and by path.  For the purpose of these examples, we'll use the Contents property to return the inner text of the selected element.

Access-by-index is straighforward.  The first child element has index 0 (zero).  An empty XDoc is returned when the index is greater than the index of the last child element.  The Count property returns the number of child elements for the current element.  Access-by-index is not recommended for most uses, because it is sensitive to the structure of the XML document.  However, from time to time, it is necessary to be able to access a child element without knowing its name.  In this case, access-by-index is very handy.

| Expression | Value |
|---|---|
| d[0].Contents | "FIRST" |
| d[3].Contents | "&lt;entry&gt;ALPHA&lt;/entry&gt;" |
| d[3][0].Contents | "ALPHA" |

Access-by-name is more flexible and more forgiving than access-by-index, because it doesn't care about the position of the child element.  Since XML allows multiple child elements with the same name to exist simultaneously, we also need the ability to select which child element we want.  By default, access-by-name will return the first element, if one exists.  Otherwise, an empty XDoc is returned.  An index can optionally be provided as a second argument to specify which occurrence to select.  However, this is rarely done.  In general, we either just care about the first child element or we want to iterate over all of them.  The next section shows how easy it is to iterate over all child elements.

| Expression | Value |
|---|---|
| d["tag"].Contents | "FIRST" |
| d["tag", 0].Contents | "FIRST" |
| d["tag", 1].Contents | "SECOND" |
| d["tag", 2].Contents | "THIRD" |
| d["tag", 3].Contents | "" |
| d["node"].Contents | "&lt;entry&gt;ALPHA&lt;/entry&gt;" |
| d["node"]["entry"].Contents | "ALPHA" |

Access-by-path is an extension of access-by-name.  It uses the same notation, but with a more complex selection string than just a name.  Instead, it allows any XPath 1.0 expression.  XPath is a powerful expression language that makes it easy to select specific XML elements.  Access-by-path is oftern preferred over two sequential access-by-name operations, because it's easier to read and type.

| Expression | Value |
|---|---|

| | |
|---|---|
| d["node/entry"].Contents | "ALPHA" |
| d["node/entry", 0].Contents | "ALPHA" |
| d["node/entry", 1].Contents | "BETA" |
| d["node/entry", 2].Contents | "GAMMA" |
| d["node/entry", 3].Contents | "" |
| d["node[entry='BETA']"].Contents | "<entry>BETA</entry>" |
| d["node[2]/entry"].Contents | "BETA" (NOTE: in XPath, the first child element in an expression has index 1) |

## Iterating over Elements

As shown in the previous section, when accessing an element by name or by path, we might actually have multiple elements that match our expression.  When we only expect at most one element to match, we don't really care about the other elements and XDoc takes on the value of the first one.  However, for many use cases, we expect more than one element to exist.  Fortunately, XDoc enables us to access these other elements without introducing another class like XmlNodeList.  Instead, XDoc takes on the traits of a *cursor*.  A cursor points to a current element and can be moved forward to point to the next element.  In the case of XDoc, we actually don't change the meaning of the XDoc instance, but instead invoke the Next property to get a new XDoc instance that represents the next element, if one exists. Otherwise, the returned XDoc is empty.

This might sound complicated, but it's actually very simple and intuitive.  Let's see how this looks in code when we want to iterate over all <tag> elements.

```
XDoc tag = d["tag"];
while(!tag.IsEmpty) {
        // do something
        tag = tag.Next;
}
```

-OR-

```
for(XDoc tag = d["tag"]; !tag.IsEmpty; tag = tag.Next) {
        // do something
}
```

As you can see, it's really quite simple, but XDoc makes it even easier than this.  XDoc can also be used in foreach statements:

```
foreach(XDoc tag in d["tag"]) {
        // do something
}
```

## Composing XDocs

We've seen how to make new XDoc instances from scratch, how to select parts of an XDoc, and how to iterate over them.  We now have the foundation for talking about making new XDocs from existing ones.  This is a very common operation as many services manipulate XML data from multiple sources and combine it into a new XML document.

Let's modify our XDoc instance by adding a new element at the end:

```
d.Start("node").Start("entry").Value("DELTA").End().End();
```

This yields:

```
<doc>
        <tag>FIRST</tag>
        <tag>SECOND</tag>
        <tag>THIRD</tag>
        <node>
                <entry>ALPHA</entry>
        </node>
        <node>
                <entry>BETA</entry>
        </node>
        <node>
                <entry>GAMMA</entry>
        </node>
        <node>
                <entry>DELTA</entry>
        </node>
</doc>
```

**NOTE:** you might have wondered earlier why XDoc requires a root element to be created from the get-go.  The reason is that an XML document can contain only one root element.  Thus, it's usuaully necessary to check if the current node is the document node and if it already has a root element.  And if it has, you cannot add another element.  However, with XDoc the root element is automatically created and all other elements are child elements.  Thus, adding a new element is always possible and you don't have to worry about where you are in the XDoc.

Now, instead of adding an element to the end of our document, let's add it to the <node> element that contains the <entry>BETA</entry> element.

```
d["node[entry='BETA']"].Start("annotation").Value("Second greek letter").End();
```

This yields:

```
<doc>
        <tag>FIRST</tag>
        <tag>SECOND</tag>
        <tag>THIRD</tag>
        <node>
                <entry>ALPHA</entry>
        </node>
        <node>
                <entry>BETA</entry>
                <annotation>Second greek letter</annotation>
        </node>
        <node>
                <entry>GAMMA</entry>
        </node>
</doc>
```

The combination of accessing elements and adding new elements can nicely be used in conjunction to make it easy to modify an XDoc. But how about using an existing XDoc to create a new one? Let's create a new XDoc with the contents of the first <node> element. We'll use the Add() method to add an entire XDoc.

```
XDoc f = new XDoc("new").Add(d["node"]);
```

This yields:

```
<new>
        <node>
                <entry>ALPHA</entry>
        </node>
</new>
```

Similarly to our previous example, we could have used an accessor to first select a specific element in an existing XDoc before using the Add() method. Thus, it is very simple to augment an existing XDoc with other XDocs.

### Added in Beryl

But what if we want not to just add one element, but multiple elements? The AddAll() method does exactly that:

```
XDoc f = new XDoc("new").AddAll(d["tag"]);
```

This yields:

```
<new>
        <tag>FIRST</tag>
        <tag>SECOND</tag>
        <tag>THIRD</tag>
</new>
```

## Conclusion

XDoc is designed to make interacting with XML documents really easy. No more is there a need to learn about XmlDocument, XmlElement, XmlNodeList, and so many other classes. XDoc builds on all of these and provides a simple, straightforward interface to create and access XML documents. With XDoc you don't have to worry about what an operation returns, it's always another XDoc. By default, XDoc represents the first element, but it's very simple to iterate over the remaining elements. An XDoc is always a valid XML document with one notable exception: an XDoc instance can be empty. There is no equivalent in XML for this. However, when designing XDoc we didn't want to throw exceptions when selecting none-existing elements.  Instead, we wanted to provide a safe operating experience.  By enabling this speclal case, developers don't have to worry about exceptions or *null* values suddently occurring that might themselves cause exceptions.