

**LINQ to XML  
.NET Language Integrated Query  
for XML Data**

**February 2007**

This is a preliminary document and may be changed substantially prior to final commercial release of the software described herein.

The information contained in this document represents the current view of Microsoft Corporation on the issues discussed as of the date of publication. Because Microsoft must respond to changing market conditions, it should not be interpreted to be a commitment on the part of Microsoft, and Microsoft cannot guarantee the accuracy of any information presented after the date of publication.

This White Paper is for informational purposes only. MICROSOFT MAKES NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, AS TO THE INFORMATION IN THIS DOCUMENT.

Complying with all applicable copyright laws is the responsibility of the user. Without limiting the rights under copyright, no part of this document may be reproduced, stored in or introduced into a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photocopying, recording, or otherwise), or for any purpose, without the express written permission of Microsoft Corporation.

Microsoft may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this document. Except as expressly provided in any written license agreement from Microsoft, the furnishing of this document does not give you any license to these patents, trademarks, copyrights, or other intellectual property.

Unless otherwise noted, the companies, organizations, products, domain names, e-mail addresses, logos, people, places, and events depicted in examples herein are fictitious. No association with any real company, organization, product, domain name, e-mail address, logo, person, place, or event is intended or should be inferred.

 2007 Microsoft Corporation. All rights reserved.

Microsoft, MS-DOS, Windows, Windows Server, Windows Vista, Visual Studio are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

All other trademarks are property of their respective owners.

**Table of Contents**

1. Introduction 1

1.1 Sample XML 1

2. Programming XML with LINQ to XML 3

2.1 LINQ to XML Design Principles 3

2.1.1 Key Concepts 3

2.1.1.1 Functional Construction 3

2.1.1.2 Document "Free" 4

2.1.1.3 XML Names 6

2.1.1.4 Text as value 6

2.2 The LINQ to XML Class Hierarchy 7

2.3 XML Names 8

2.3.1.1 XML Prefixes and Output 9

2.4 Loading existing XML 10

2.5 Creating XML from Scratch 11

2.6 Traversing XML 12

2.6.1.1 Getting the Children of an XML Element 13

2.6.1.2 Getting the Parent and Document of an XML Element 14

2.7 Manipulating XML 14

2.7.1.1 Inserting XML 15

2.7.2 Deleting XML 15

2.7.3 Updating XML 16

2.7.4 Be careful with deferred query execution 16

2.8 Working with Attributes 17

2.8.1 Adding XML Attributes 17

2.8.2 Getting XML Attributes 18

2.8.3 Deleting XML Attributes 18

2.9 Working with other types of XML Nodes 18

2.10 Annotating nodes with user-defined information 19

2.11 Outputting XML 20

3. Querying XML with LINQ to XML 21

3.1 Querying XML 21

3.1.1 Standard Query Operators and XML 21

3.1.1.1 Creating multiple peer nodes in a select 22

3.1.1.2 Handling Null in a Transform 22

3.1.2 XML Query Extensions 23

3.1.2.1 Elements and Content 23

3.1.2.2 Descendants and Ancestors 24

3.1.2.3 Attributes 25

3.1.2.4 ElementsBeforeSelf, ElementsAfterSelf, NodesBeforeSelf, NodesAfterSelf 25

3.1.2.5 Technical Note: XML Query Extensions 26

3.1.3 XML Transformation 26

3.2 Using Query Expressions with XML 28

3.3 Using XPath and XSLT with LINQ to XML 29

Mixing XML and other data models 30

3.4 Reading from a database to XML 30

3.5 Reading XML and Updating a Database 31

4. Layered Technologies Over LINQ to XML 33

4.1 LINQ to XML in Visual Basic 9.0 33

4.1.1 XML Literals 33

4.1.2 Xml Axis Properties 34

4.1.3 Putting it all together 35

4.2 Schema aware XML Programming 36

5. February 2007 CTP Release Notes 39

5.1 Changes since the May 2006 CTP 39

5.1.1 Bridge Classes 39

5.1.2 Event Model 39

5.2 Non-exhaustive list of planned features in future releases 39

6. References 41

# Introduction

XML has achieved tremendous adoption as a basis for formatting data whether in Word files, on the wire, in configuration files, or in databases … XML seems to be everywhere. Yet, from a development perspective, XML is still hard to work with. If you ask the average software developer to work in XML you will likely hear a heavy sigh. The API choices for working with XML seem to be either aged and verbose such as DOM or XML specific such as XQuery or XSLT which require motivation, study, and time to master. LINQ to XML, a component of the LINQ project, aims to address this issue. LINQ to XML is a modernized in-memory XML programming API designed to take advantage of the latest .NET Framework language innovations. It provides both DOM and XQuery/XPath like functionality in a consistent programming experience across the different LINQ-enabled data access technologies.

There are two major perspectives for thinking about and understanding LINQ to XML. From one perspective you can think of LINQ to XML as a member of the LINQ Project family of technologies with LINQ to XML providing an XML Language Integrated Query capability along with a consistent query experience for objects, relational database (LINQ to SQL, LINQ to DataSet, LINQ to Entities), and other data access technologies as they become LINQ-enabled. From a another perspective you can think of LINQ to XML as a full feature in-memory XML programming API comparable to a modernized, redesigned Document Object Model (DOM) XML Programming API plus a few key features from XPath and XSLT.

LINQ to XML was developed with Language Integrated Query over XML in mind from the beginning. It takes advantage of the Standard Query Operators and adds query extensions specific to XML. From an XML perspective LINQ to XML provides the query and transformation power of XQuery and XPath integrated into .NET Framework languages that implement the LINQ pattern (e.g., C#, VB, etc.). This provides a consistent query experience across LINQ enabled APIs and allows you to combine XML queries and transforms with queries from other data sources. We will go in more depth on LINQ to XML’s query capability in section 3, "Querying XML with LINQ to XML".

Just as significant as the Language Integrated Query capabilities of LINQ to XML is the fact that LINQ to XML represents a new, modernized in-memory XML Programming API. LINQ to XML was designed to be a cleaner, modernized API, as well as fast and lightweight. LINQ to XML uses modern language features (e.g., generics and nullable types) and diverges from the DOM programming model with a variety of innovations to simplify programming against XML. Even without Language Integrated Query capabilities LINQ to XML represents a significant stride forward for XML programming. The next section of this document, "Programming XML", provides more detail on the in-memory XML Programming API aspect of LINQ to XML.

LINQ to XML is a language-agnostic component of the LINQ Project. The samples in most of this document are shown in C# for brevity. LINQ to XML can be used just as well with a LINQ-enabled version of the VB.NET compiler. Section 5.1, "LINQ to XML in Visual Basic 9.0" discusses VB specific programming with LINQ to XML in more detail.

## Sample XML

For the purposes of this paper let's establish a simple XML contact list sample that we can use throughout our discussion.

<contacts>  
 <contact>  
 <name>Patrick Hines</name>  
 <phone type="home">206-555-0144</phone>  
 <phone type="work">425-555-0145</phone>  
 <address>  
 <street1>123 Main St</street1>  
 <city>Mercer Island</city>   
 <state>WA</state>  
 <postal>68042</postal>  
 </address>  
 <netWorth>10</netWorth>  
 </contact>  
 <contact>  
 <name>Gretchen Rivas</name>  
 <phone type="mobile">206-555-0163</phone>  
 <address>  
 <street1>123 Main St</street1>  
 <city>Mercer Island</city>  
 <state>WA</state>  
 <postal>68042</postal>  
 </address>  
 <netWorth>11</netWorth>  
 </contact>  
 <contact>  
 <name>Scott MacDonald</name>  
 <phone type="home">925-555-0134</phone>  
 <phone type="mobile">425-555-0177</phone>  
 <address>  
 <street1>345 Stewart St</street1>  
 <city>Chatsworth</city>  
 <state>CA</state>  
 <postal>91746</postal>  
 </address>  
 <netWorth>500000</netWorth>  
 </contact>  
</contacts>

# Programming XML with LINQ to XML

This section details how to program with LINQ to XML independent of Language Integrated Query. Because LINQ to XML provides a fully featured in-memory XML programming API you can do all of the things you would expect when reading and manipulating XML. A few examples include the following:

* Load XML into memory in a variety of ways (file, XmlReader, etc.).
* Create an XML tree from scratch.
* Insert new XML Elements into an in-memory XML tree.
* Delete XML Elements out of an in-memory XML tree.
* Save XML to a variety of output types (file, XmlWriter, etc.).

And much more. You should be able to accomplish most XML programming tasks you run into using this technology.

## LINQ to XML Design Principles

LINQ to XML is designed to be a lightweight XML programming API. This is true from both a conceptual perspective, emphasizing a straightforward, easy to use programming model, and from a memory and performance perspective. Its public data model is aligned as much as possible with the W3C XML Information Set.

### Key Concepts

This section outlines some key concepts that differentiate LINQ to XML from other XML programming APIs, in particular the current predominant XML programming API, the W3C DOM.

#### Functional Construction

In object oriented programming when you create object graphs, and correspondingly in W3C DOM, when creating an XML tree, you build up the XML tree in a bottom-up manner. For example using XmlDocument (the DOM implementation from Microsoft) this would be a typical way to create an XML tree.

XmlDocument doc = new XmlDocument();  
XmlElement name = doc.CreateElement("name");  
name.InnerText = "Patrick Hines";  
XmlElement phone1 = doc.CreateElement("phone");  
phone1.SetAttribute("type", "home");  
phone1.InnerText = "206-555-0144";   
XmlElement phone2 = doc.CreateElement("phone");  
phone2.SetAttribute("type", "work");  
phone2.InnerText = "425-555-0145";   
XmlElement street1 = doc.CreateElement("street1");   
street1.InnerText = "123 Main St";  
XmlElement city = doc.CreateElement("city");  
city.InnerText = "Mercer Island";  
XmlElement state = doc.CreateElement("state");  
state.InnerText = "WA";  
XmlElement postal = doc.CreateElement("postal");  
postal.InnerText = "68042";  
XmlElement address = doc.CreateElement("address");  
address.AppendChild(street1);  
address.AppendChild(city);  
address.AppendChild(state);  
address.AppendChild(postal);  
XmlElement contact = doc.CreateElement("contact");  
contact.AppendChild(name);  
contact.AppendChild(phone1);  
contact.AppendChild(phone2);  
contact.AppendChild(address);  
XmlElement contacts = doc.CreateElement("contacts");  
contacts.AppendChild(contact);  
doc.AppendChild(contacts);

This style of coding provides few clues to the structure of the XML tree. LINQ to XML supports this approach to constructing an XML tree but also supports an alternative approach referred to as functional construction. Here is how you would construct the same XML tree by using LINQ to XML functional construction.

XElement contacts =  
 new XElement("contacts",  
 new XElement("contact",  
 new XElement("name", "Patrick Hines"),  
 new XElement("phone", "206-555-0144",   
 new XAttribute("type", "home")),  
 new XElement("phone", "425-555-0145",  
 new XAttribute("type", "work")),  
 new XElement("address",  
 new XElement("street1", "123 Main St"),  
 new XElement("city", "Mercer Island"),  
 new XElement("state", "WA"),  
 new XElement("postal", "68042")  
 )  
 )  
 );

Notice that by indenting (and squinting a bit) the code to construct the XML tree shows the structure of the underlying XML.

Functional construction is described further section 2.5, "Creating XML from Scratch".

#### Document "Free"

When programming XML your primary focus is usually on XML elements and perhaps attributes. This makes sense because an XML tree, other than at the leaf level, is composed of XML elements and your primary goal when working with XML is traversing or manipulating the XML elements that make up the XML tree. In LINQ to XML you can work directly with XML elements in a natural way. For example you can do the following:

* Create XML elements directly (without an XML document involved at all)
* Load them from XML that exists in a file
* Save (write) them to a writer

Compare this to W3C DOM, in which the XML document is used as a logical container for the XML tree. In DOM XML nodes, including elements and attributes, must be created in the context of an XML document. Here is a fragment of the code from the previous example to create a name element:

XmlDocument doc = new XmlDocument();  
XmlElement name = doc.CreateElement("name");

Note how the XML document is a fundamental concept in DOM. XML nodes are created in the context of the XML document. If you want to use an element across multiple documents you must import the nodes across documents. This is an unnecessary layer of complexity that LINQ to XML avoids.

In LINQ to XML you create XML elements directly:

XElement name = new XElement("name");

You do not have to create an XML Document to hold the XML tree. The LINQ to XML object model does provide an XML document to use if necessary, for example if you have to add a comment or processing instruction at the top of the document. The following is an example of how to create an XML Document with an XML Declaration, Comment, and Processing Instruction along with the contacts content.

XDocument contactsDoc =  
 new XDocument(  
 new XDeclaration("1.0", "utf-8", "yes"),  
 new XComment("LINQ to XML Contacts XML Example"),  
 new XProcessingInstruction("MyApp", "123-44-4444"),  
 new XElement("contacts",  
 new XElement("contact",  
 new XElement("name", "Patrick Hines"),   
 new XElement("phone", "206-555-0144"),  
 new XElement("address",  
 new XElement("street1", "123 Main St"),  
 new XElement("city", "Mercer Island"),  
 new XElement("state", "WA"),  
 new XElement("postal", "68042")  
 )  
 )  
 )  
 );

After this statement contactsDoc contains:

<?xml version="1.0" encoding="utf-8" standalone="yes"?>  
<!--LINQ to XML Contacts XML Example-->  
<?MyApp 123-44-4444?>  
<contacts>  
 <contact>  
 <name>Patrick Hines</name>  
 <phone>206-555-0144</phone>  
 <address>  
 <street1>123 Main St</street1>  
 <city>Mercer Island</city>  
 <state>WA</state>  
 <postal>68042</postal>  
 </address>  
 </contact>  
</contacts>

#### XML Names

LINQ to XML goes out of its way to make XML names as straightforward as possible. Arguably, the complexity of XML names, which is often considered an advanced topic in XML literature, comes not from namespaces, which developers use regularly in programming, but from XML prefixes. XML prefixes can be useful for reducing the keystrokes required when inputting XML or making XML easier to read, however prefixes are just a shortcut for using the full XML Namespace. On input LINQ to XML resolves all prefixes to their corresponding XML Namespace and prefixes are not exposed in the programming API. In LINQ to XML, an XName represents a full XML name consisting of an XNamespace object and the local name. Developers will usually find it more convenient to use the XNamespace object rather than the namespace URI string.

For example, to create an XElement called contacts that has the namespace "http://mycompany.com" you could use the following code:

XNamespace ns = "http://mycompany.com";

XElement contacts = new XElement(ns + "contacts");

Conversely, W3C DOM exposes XML names in a variety of ways across the API. For example, to create an XmlElement, there are three different ways that you can specify the XML name. All of these allow you to specify a prefix. This leads to a confusing API with unclear consequences when mixing prefixes, namespaces, and namespace declarations (xmlns attributes that associate a prefix with an XML namespace).

LINQ to XML treats XML namespace prefixes as serialization options and nothing more. When you read XML, all prefixes are resolved, and each named XML item has a fully expanded name containing the namespace and the local name. On output, the XML namespace declarations (xmlns attributes) are honored and the appropriate prefixes are then displayed. If you need to influence prefixes in the XML output, you can add xmlns attributes in the appropriate places in the XML tree. See Section 2.3, “XML Names,” for more information.

#### Text as value

Typically, the leaf elements in an XML tree contain values such as strings, integers, and decimals. The same is true for attributes. In LINQ to XML, you can treat elements and attributes that contain values in a natural way, simply cast them to the type that they contain. For example, assuming that name is an XElement that contains a string, you could do the following:

string nameString = (string) name;

Usually this will show up in the context of referring to a child element directly like this:

string name = (string) contact.Element("name");

Explicit cast operators are provided for string, bool, bool?, int, int?, uint, uint?, long, long?, ulong, ulong?, float, float?, double, double?, decimal, decimal?, DateTime, DateTime?, TimeSpan, TimeSpan?, and GUID, GUID?.

In contrast, the W3C DOM *always* treats text as an XML node. Consequently in many DOM implementations the only way to read and manipulate the underlying text of a leaf node is to read the text node children of the leaf node. For example just to read the value of the name element you would need to write code similar to the following:

XmlNodeList children = name.ChildNodes;  
string nameValue = "";  
foreach (XmlText text in children) {  
 nameValue = nameValue + text.Value;  
}  
Console.WriteLine(nameValue);

This has been simplified in some W3C DOM implementations, such as the Microsoft XmlDocument API, by using the InnerText method.. With LINQ to XML, there is an XText class, but it is used only to let you work with mixed content and CData sections. Developers of applications that do not use these features of XML don’t have to worry about text nodes in most cases. You can usually work directly with the basic .NET Framework-based types, reading them and adding them directly to the XML. In general, it is best to ignore the existence of XText nodes unless you are working with mixed content or CData sections. If you must work with text nodes, do not re-use them or assume that a reference to a text node will contain the correct data after changes are made to the tree.

## The LINQ to XML Class Hierarchy

In Figure 1 LINQ to XML Class Hierarchy" you can see the major classes defined in LINQ to XML.



Figure LINQ to XML Class Hierarchy

Note the following about the LINQ to XML class hierarchy:

* Although XElement is low in the class hierarchy, it is the fundamental class in LINQ to XML. XML trees are generally made up of a tree of XElements. XAttributes are name/value pairs associated with an XElement. XDocuments are created only if necessary, such as to hold a DTD or top level XML processing instruction (XProcessingInstruction). All other XNodes can only be leaf nodes under an XElement, or possibly an XDocument (if they exist at the root level).
* XAttribute and XNode are peers and not derived from a common base class (other than object). This reflects the fact that XML attributes are really name value pairs associated with an XML element not nodes in the XML tree. Contrast this with W3C DOM.
* XText and XCData are exposed in this version of LINQ to XML, but as discussed above, it is best to think of them as a semi-hidden implementation detail except when exposing text nodes is necessary. As a user, you can get back the value of the text within an element or attribute as a string or other simple value.
* The only XNode that can have children is an XContainer, meaning either an XDocument or XElement. An XDocument can contain an XElement (the root element), an XDeclaration, an XDocumentType, or an XProcessingInstruction. An XElement can contain another XElement, an XComment, an XProcessingInstruction, and text (which can be passed in a variety of formats, but will be represented in the XML tree as text).

## XML Names

XML names, often a complex subject in XML programming APIs, are represented simply in LINQ to XML. An XML name is represented by an XNamespace object (which encapsulates the XML namespace URI) and a local name. An XML namespace serves the same purpose that a namespace does in your .NET Framework-based programs, allowing you to uniquely qualify the names of your classes. This helps ensure that you don’t run into a name conflict with other users or built-in names. When you have identified an XML namespace, you can choose a local name that needs to be unique only within your identified namespace. For example, if you want to create an XML element with the name contacts, you would likely want to create it within an XNamespace with a URI such as http://yourCompany.com/ContactList.

Another aspect of XML names is XML namespace prefixes. XML prefixes cause most of the complexity of XML names. In XML syntax, prefixes allow you to create a shortcut for an XML namespace, which makes the XML document more concise and understandable. XML prefixes depend on their context to have meaning. The XML prefix myPrefix could be associated with one XML namespace in one part of an XML tree, but be associated with a completely different XML namespace in a different part of the XML tree.

LINQ to XML simplifies XML names by removing XML prefixes from the XML Programming API and encapsulates them in XNamespace objects. When reading in XML, each XML prefix is resolved to its corresponding XML namespace. Therefore, when developers work with XML names they are working with a fully qualified XML name: an XML namespace, and a local name.

In LINQ to XML, the class that represents XML names is XName, consisting of an XNamespace object and the local name. For example, to create an XElement called contacts that has the namespace "http://mycompany.com" you could use the following code:

XNamespace ns = "http://mycompany.com";

XElement contacts = new XElement(ns + "contacts");

XML names appear frequently throughout the LINQ to XML API, and wherever an XML name is required, you will find an XName parameter. However, you seldom work directly with an XName. XName contains an implicit conversion from string.

The string representation of an XName is referred to as an expanded name. An expanded name looks like the following:

{NamepaceURI}LocalName

An expanded name with the XML namespace http://yourCompany.com and the local name contacts looks like the following:

{http://myCompany.com}contacts

It is possible to use this expanded name syntax rather than constructing XNamespace objects any time an XName is required. For example, the constructor for XElement takes an XName as its first argument:

XElement contacts = new XElement("{http://myCompany.com}contacts", … );

You do not have to type the XML namespace every time you use an XML name. You can use the facilities of the language itself to make this easier. For example, you can use the following common pattern:

XNamespace myNs = "http://mycompany.com";  
  
XElement contacts =  
 new XElement(myNs + "contacts",  
 new XElement(myNs + "contact",  
 new XElement(myNs + "name", "Patrick Hines"),  
 new XElement(myNs + "phone", "206-555-0144",   
 new XAttribute("type", "home")),  
 new XElement(myNs + "phone", "425-555-0145",  
 new XAttribute("type", "work")),  
 new XElement(myNs + "address",  
 new XElement(myNs + "street1", "123 Main St"),  
 new XElement(myNs + "city", "Mercer Island"),  
 new XElement(myNs + "state", "WA"),  
 new XElement(myNs + "postal", "68042")  
 )  
 )  
 );

The resulting XML will look like:

<contacts xmlns="http://mycompany.com">  
 <contact>  
 <name>Patrick Hines</name>  
 <phone type="home">206-555-0144</phone>  
 <phone type="work">425-555-0145</phone>  
 <address>  
 <street1>123 Main St</street1>  
 <city>Mercer Island</city>  
 <state>WA</state>  
 <postal>68042</postal>  
 </address>  
 </contact>  
</contacts>

#### XML Prefixes and Output

Earlier in this section we mentioned that, when reading in XML, prefixes are resolved to their corresponding XML namespaces. But what happens on output? What if you need or want to influence prefixes when outputting the XML? You can do this by creating xmlns attributes (XML namespace declarations) that associate a prefix to an XML namespace. For example:

XNamespace ns = "URI";

XElement e =  
 new XElement(ns + "e",  
 new XAttribute(XNamespace.Xmlns + "p", ns)  
 );

The snippet would generate:

<p:e xmlns:p="URI"/>

Therefore, if you have a specific output in mind, you can manipulate the XML to have the XML namespace declarations with your desired prefixes exactly where you want them.

## Loading existing XML

You can load existing XML into an LINQ to XML XML tree so that you can read it or manipulate it. LINQ to XML provides multiple input sources, including a file, an XmlReader, a TextReader, or a string. To input a string, you use the Parse method. Here is an example of the Parse method:

XElement contacts = XElement.Parse(  
 @"<contacts>  
 <contact>  
 <name>Patrick Hines</name>  
 <phone type=""home"">206-555-0144</phone>  
 <phone type=""work"">425-555-0145</phone>  
 <address>  
 <street1>123 Main St</street1>  
 <city>Mercer Island</city>   
 <state>WA</state>  
 <postal>68042</postal>  
 </address>  
 <netWorth>10</netWorth>  
 </contact>  
 <contact>  
 <name>Gretchen Rivas</name>  
 <phone type=""mobile"">206-555-0163</phone>  
 <address>  
 <street1>123 Main St</street1>  
 <city>Mercer Island</city>  
 <state>WA</state>  
 <postal>68042</postal>  
 </address>  
 <netWorth>11</netWorth>  
 </contact>  
 <contact>  
 <name>Scott MacDonald</name>  
 <phone type=""home"">925-555-0134</phone>  
 <phone type=""mobile"">425-555-0177</phone>  
 <address>  
 <street1>345 Stewart St</street1>  
 <city>Chatsworth</city>  
 <state>CA</state>  
 <postal>91746</postal>  
 </address>  
 <netWorth>500000</netWorth>  
 </contact>  
 </contacts>");

To input from any of the other sources, you use the Load method. For example, to load XML from a file:

XElement contactsFromFile = XElement.Load(@"c:\myContactList.xml");

## Creating XML from Scratch

LINQ to XML provides a powerful approach to creating XML elements. This is referred to as functional construction. Functional construction lets you create all or part of your XML tree in a single statement. For example, to create a contacts XElement, you could use the following code:

XElement contacts =  
 new XElement("contacts",  
 new XElement("contact",  
 new XElement("name", "Patrick Hines"),   
 new XElement("phone", "206-555-0144"),  
 new XElement("address",  
 new XElement("street1", "123 Main St"),  
 new XElement("city", "Mercer Island"),  
 new XElement("state", "WA"),  
 new XElement("postal", "68042")  
 )  
 )  
 );

By indenting, the XElement constructor resembles the structure of the underlying XML. Functional construction is enabled by an XElement constructor that takes a params object.

public XElement(XName name, params object[] contents)

The contents parameter is extremely flexible, supporting any type of object that is a legitimate child of an XElement. Parameters can be any of the following:

* A string, which is added as text content. This is the recommended pattern to add a string as the value of an element; the LINQ to XML implementation will create the internal XText node.
* An XText, which can have either a string or CData value, added as child content. This is mainly useful for CData values; using a string is simpler for ordinary string values.
* An XElement, which is added as a child element
* An XAttribute, which is added as an attribute
* An XProcessingInstruction or XComment, which is added as child content
* An IEnumerable, which is enumerated, and these rules are applied recursively
* Anything else, ToString() is called and the result is added as text content
* null, which is ignored

In the above example showing functional construction, a string ("Patrick Hines") is passed into the name XElement constructor. This could have been a variable (for example, new XElement("name", **custName**)), it could have been a different type besides string (for example, new XElement("quantity", 55)), it could have been the result of a function call like this

{  
...  
 XElement qty = new XElement("quantity", **GetQuantity()**);  
...  
}

public int GetQuantity() { return 55; }

or it could have even been the an IEnumerable<XElement>. For example, a common scenario is to use a query within a constructor to create the inner XML. The following code reads contacts from an array of Person objects into a new XML element contacts.

class Person  
{  
 public string Name;  
 public string[] PhoneNumbers;  
}

var persons = new[] {  
 new Person {  
 Name = "Patrick Hines",  
 PhoneNumbers = new[] { "206-555-0144", "425-555-0145" }  
 },  
 new Person {  
 Name = "Gretchen Rivas",  
 PhoneNumbers = new[] { "206-555-0163" }  
 }  
};

XElement contacts =  
 new XElement("contacts",  
 **from p in persons  
 select new XElement("contact",  
 new XElement("name", p.Name),  
 from ph in p.PhoneNumbers  
 select new XElement("phone", ph)  
 )**  
 );

Console.WriteLine(contacts);

This gives the following output:

<contacts>  
 <contact>  
 <name>Patrick Hines</name>  
 <phone>206-555-0144</phone>  
 <phone>425-555-0145</phone>  
 </contact>  
 <contact>  
 <name>Gretchen Rivas</name>  
 <phone>206-555-0163</phone>  
 </contact>  
</contacts>

Notice how the inner body of the XML, the repeating contact element, and, for each contact, the repeating phone were generated by queries that return an IEnumerable.

When an objective of your program is to create an XML output, functional construction lets you begin with the end in mind. You can use functional construction to shape your goal output document and either create the subtree of XML items inline, or call out to functions to do the work.

Functional construction is instrumental in transforms, which are described in more detail in section 3.1.4, “XML Transformation.” Transformation is a key usage scenario in XML, and functional construction is well-suited for this task.

## Traversing XML

When you have XML available to you in-memory, the next step is often to navigate to the XML elements that you want to work on. Language Integrated Query provides powerful options for doing just this, as described in section 3, "Querying XML with LINQ to XML", this section describes more traditional approaches to walking through an XML tree.

#### Getting the Children of an XML Element

LINQ to XML provides methods for getting the children of an XElement. To get all of the children of an XElement (or XDocument), you can use the Nodes() method. This returns IEnumerable<object> because you could have text mixed with other LINQ to XML types. For example, you might have the following XML loaded into an XElement called contact:

<contact>  
 Met in 2005.  
 <name>Patrick Hines</name>  
 <phone>206-555-0144</phone>  
 <phone>425-555-0145</phone>  
 <!--Avoid whenever possible-->  
</contact>

Using Nodes(), you could get all of the children and output the results by using this code fragment:

foreach (c in contact.Nodes()) {  
 Console.WriteLine(c);  
}

The results would show on the console as:

Met in 2005.  
<name>Patrick Hines</name>  
<phone>206-555-0144</phone>  
<phone>425-555-0145</phone>  
<!--Avoid whenever possible-->

The first child was the string, "Met in 2005.", the second child was the XElement name, the third child was the first phone XElement, the fourth child was the second phone XElement, and the fifth child was an XComment with the value "Avoid whenever possible". Notice that ToString() on an XNode (XElement, for example) returns a formatted XML string based on the node type. This is a great convenience, and we will use this many times in this document.

If you want to be more specific, you can ask for content nodes of an XElement of a particular type. For example, you might want to get the XElement children for the contact XElement only. In this case, you can specify a parameterized type:

foreach (c in contact.Nodes().OfType<XElement>()) {  
 Console.WriteLine(c)  
}

And you would only get the element child written to the console:

<name>Patrick Hines</name>  
<phone>206-555-0144</phone>  
<phone>425-555-0145</phone>

Because XML Elements are prevalent and important in most XML scenarios, there are methods for navigating to XElements directly below a particular XElement in the XML tree. The method Elements() returns IEnumerable<XElement>, and is a shortcut for Nodes().OfType<XElement>(). For example, to get all of the element children of contact, you would do the following:

foreach (x in contact.Elements()) {  
 Console.WriteLine(x);  
}

Again, only the XElement children would be output:

<name>Patrick Hines</name>  
<phone>206-555-0144</phone>  
<phone>425-555-0145</phone>

If you want to get all XElements with a specific name, you can use the Elements(XName) overload that takes an XName as a parameter. For example, to get only the phone XElements, you could do the following:

foreach (x in contact.**Elements("phone")**) {  
 Console.WriteLine(x);  
}

This would write all of the phone XElements to the console.

<phone>206-555-0144</phone>  
<phone>425-555-0145</phone>

If you know that there is only one child element with a particular name, you can use the Element(XName) (not plural) method, which returns a single XElement. If there is more than one element with this name, you will get the first one. For example, to get the name XElement, you could do the following:

XElement name = contact.**Element("name")**;

Or, you could get the value of name like this:

string name = (string) contact.Element("name");

Nodes(), Elements(), Elements(XName), and Element(XName) are the basic methods for simple traversal of XML. If you are familiar with XPath, these methods are analogous to child::node(), child::\*, child::name, and child::name[1], respectively. XML Query extensions such as Descendants() and Ancestors() as discussed in section 3, "Querying XML with LINQ to XML", serve a similar traversal purpose and are often combined with the basic traversal methods.

#### Getting the Parent and Document of an XML Element

To traverse upwards in the XML tree, you can use the Parent property of XElement. For example, if you had a phone XElement, you retrieve the associated contact with the following:

XElement contact = phone**.Parent**;

Note that the Parent property of a root element is null. It is not the associated document as it is in some other XML APIs. In LINQ to XML, the XML document is not considered a part of the XML tree. If you want the document associated with an XElement (or any XNode), you can get to it from the Document property. If you want to associate an XElement as the root element of a document, you can pass the element into the XDocument constructor or you can add the root to the document as a child element. For example, to establish the contacts XElement as the root element of a contactsDoc XDocument, you could do the following:

XDocument contactsDoc = new XDocument(contacts);

or

XDocument contactsDoc = new XDocument();  
contactsDoc.Add(contacts);

## Manipulating XML

LINQ to XML provides a full set of methods for manipulating XML. You can insert, delete, copy, and update XML content.

#### Inserting XML

You can easily add content to an existing XML tree. To add another phone XElement by using the Add() method:

XElement mobilePhone = new XElement("phone", "206-555-0168");  
contact.**Add**(mobilePhone);

This code fragment will add the mobilePhone XElement as the last child of contact. If you want to add to the beginning of the children, you can use AddFirst(). If you want to add the child in a specific location, you can navigate to a child before or after your target location by using AddBeforeSelf() or AddAfterSelf(). For example, if you wanted mobilePhone to be the second phone you could do the following:

XElement mobilePhone = new XElement("phone", "206-555-0168");  
XElement firstPhone = contact.Element("phone");  
firstPhone.**AddAfterSelf**(mobilePhone);

The Add methods work similarly to the XElement and XDocument (actually XContainer) constructors so you can easily add full XML subtrees using the functional construction style. For example, you might want to add an Address to a contact.

contact.**Add**(new XElement("address",  
 new XElement("street", "123 Main St"),  
 new XElement("city", "Mercer Island"),  
 new XElement("state", "WA"),  
 new XElement("country", "USA"),  
 new XElement("postalCode", "68042")  
));

Let's look a little deeper at what is happening behind the scenes when adding an element child to a parent element. When you first create an XElement it is unparented. If you check its Parent property you will get back null.

XElement mobilePhone = new XElement("phone", "206-555-0168");  
Console.WriteLine(mobilePhone.Parent); // will print out null

When you use Add to add this child element to the parent, LINQ to XML checks to see if the child element is unparented, if so, LINQ to XML parents the child element by setting the child's Parent property to the XElement that Add was called on.

contact.Add(mobilePhone);  
Console.WriteLine(mobilePhone.Parent); // will print out contact

This is a very efficient technique which is extremely important since this is the most common scenario for constructing XML trees.

To add mobilePhone to another contact:

contact2.Add(mobilePhone);

Again, LINQ to XML checks to see if the child element is parented. In this case, the child is already parented. If the child is already parented, LINQ to XML clones the child element under subsequent parents. The previous example is the same as doing the following:

contact2.Add(new XElement(mobilePhone));

### Deleting XML

To delete XML, navigate to the content you want to delete and call Remove(). For example, if you want to delete the first phone number for a contact:

contact.Element("phone").**Remove()**;

Remove() also works over an IEnumerable, so you could delete all of the phone numbers for a contact in one call.

contact.Elements("phone").Remove();

You can also remove all of the content from an XElement by using the RemoveContent() method. For example you could remove the content of the first contact's first address with this statement:

contacts.Element("contact").Element("address").**RemoveContent()**;

Another way to remove an element is to set it to null using SetElement, which we talk further about in the next section.

### Updating XML

To update XML, you can navigate to the XElement whose contents you want to replace, and then use the ReplaceContent() method. For example, if you wanted to change the phone number of the first phone XElement of a contact, you could do the following:

contact.Element("phone").**ReplaceContent**("425-555-0155");

You can also update an XML subtree using ReplaceContent(). For example, to update an address we could do the following:

contact.Element("address").**ReplaceContent**(  
 new XElement("street", "123 Brown Lane"),  
 new XElement("city", "Redmond"),  
 new XElement("state", "WA"),  
 new XElement("country", "USA"),  
 new XElement("postalCode", "68072")  
);

ReplaceContent() is general purpose. SetElement() is designed to work on simple content. You call ReplaceContent() on the element itself; with SetElement(), you operate on the parent. For example, we could have performed the same update we demonstrated above on the first phone number by using this statement:

contact.**SetElement**("phone", "425-555-0155");

The results would be identical. If there had been no phone numbers, an XElement named “phone” would have been added under contact. For example, you might want to add a birthday to the contact. If a birthday is already there, you want to update it. If it does not exist, you want to insert it.

contact.SetElement("birthday", "12/12");

Also, if you use SetElement() with a value of null, the XElement will be deleted. You can remove the birthday element completely by:

contact.SetElement("birthday", **null**);

Attrributes have a symmetric method called SetAttribute() which is discussed in section 2.8, "Working with Attributes".

### Be careful with deferred query execution

Keep in mind when manipulating XML that in most cases query operators work on a "deferred execution" basis (also called "lazy"), meaning the queries are resolved as requested rather than all at once at the beginning of the query. For example take this query which attempts to remove all of the phone elements in the contacts list:

// Don't do this! NullReferenceException  
foreach (var phone in contacts.Descendants("phone")) {  
 phone.Remove();  
}

The query will fail with a NullReferenceException when it tries to iterate on the phone that you just deleted. You can resolve this issue by forcing resolution of the entire sequence using ToList() or ToArray(). For example, this approach will work.

foreach (var phone in contacts.Descendants("phone").ToList()) {  
 phone.Remove();  
}

This will cache up the list of phones so that there will be no problem iterating through them and deleting them.

The query extension Remove() is one of the few extension methods that does not use deferred execution and uses exactly this ToList() approach to cache up the items targeted for deletion. We could have written the previous example as:

contacts.Descendants("phone").Remove();

While removal is the most obvious situation where the combination of data manipulation operations and deferred query execution can create problems, it is not the only one. A few words of advice:

* Understand that this complex interaction between lazy evaluation and data manipulation is not a “bug” in LINQ to XML, it is a more fundamental issue in computer science (often referred to as the “Halloween Problem”).
* In general, LINQ to XML’s minimalist design philosophy precludes extensive analysis and optimization to keep users from stumbling over these problems. You need to determine, for your own application, what the appropriate tradeoff between making a static copy of a region of an XML document before manipulating it without fear of the Halloween Problem, and carefully working around the reality that that data manipulation operations can change the definition of the results of a query in ways that are not easy to anticipate.
* Consider using a “functional” transformation approach rather than an in-place updating approach when designing your data manipulation logic. LINQ to XML’s functional constructors make it quite easy to dynamically produce a new document with structures and values defined as transformations of some input document. You don’t need to learn an event-oriented API or XSLT to build efficient XML transformation pipeline, you can do it all with LINQ to XML.

## Working with Attributes

There is substantial symmetry between working with XElement and XAttribute classes. However, in the LINQ to XML class hierarchy, XElement and XAttribute are quite distinct and do not derive from a common base class. This is because XML attributes are not nodes in the XML tree; they are unordered name/value pairs associated with an XML element. LINQ to XML makes this distinction, but in practice, working with XAttribute is quite similar to working with XElement. Considering the nature of an XML attribute, where they diverge is understandable.

### Adding XML Attributes

Adding an XAttribute is very similar to adding a simple XElement. In the sample XML, notice that each phone number has a type attribute that states whether this is a home, work, or mobile phone number:

<contacts>  
 <contact>  
 <name>Patrick Hines</name>  
 <phone type="home">206-555-0144</phone>  
 <phone type="work">425-555-0145</phone>  
 </contact>  
 ...

You create an XAttribute by using functional construction the same way you would create an XElement with a simple type. To create a contact using functional construction:

XElement contact =  
 new XElement("contact",  
 new XElement("name", "Patrick Hines"),  
 new XElement("phone",  
 **new XAttribute("type", "home")**,  
 "206-555-0144"  
 ),  
 new XElement("phone",  
 **new XAttribute("type", "work")**,  
 "425-555-0145"  
 )  
 );

Just as you use SetElement to update, add, or delete elements with simple types, you can do the same using the SetAttribute(XName, object) method on XElement. If the attribute exists, it will be updated. If the attribute does not exist, it will be added. If the value of the object is null, the attribute will be deleted.

### Getting XML Attributes

The primary method for accessing an XAttribute is by using the Attribute(XName) method on XElement. For example, to use the type attribute to obtain the contact’s home phone number:

foreach (p in contact.Elements("phone")) {  
 if ((string)p.**Attribute**("type") == "home")  
 Console.Write("Home phone is: " + (string)p);  
}

Notice how the Attribute(XName) works similarly to the Element(XName) method. Also, notice that there are identical explicit cast operators, which lets you cast an XAttribute to a variety of simple types (see section 2.1.1.4, Text as value for a list of the types defined for explicit casting from XElements and XAttributes).

### Deleting XML Attributes

If you want to delete an attribute you can use Remove or use SetAttribute(XName, object) passing null as the value of object. For example to delete the type attribute from the first phone using Remove.

contact.Elements("phone").First().Attribute("type").Remove();

Or using SetAttribute:

contact.Elements("phone").First().**SetAttribute**("type", **null**);

## Working with other types of XML Nodes

LINQ to XML provides a full set of the different types of XML nodes that appear in XML. To illustrate this, we can create a document that uses all of the different XML node types:

XDocument xdoc =  
 new XDocument(  
 new XDeclaration("1.0", "UTF-8", "yes"),  
 new XProcessingInstruction("myApp", "My App Data"),  
 new XComment("My comment"),  
 new XElement("rootElement",  
 new XAttribute("myAttribute", "att"),  
 1234,  
 new XCData("Text with a <left> bracket"),  
 "mystring"  
 )  
 );

When you output xdoc you get:

<?xml version="1.0" standalone="yes"?>  
<!--DOCTYPE-->  
<?myApp My App Data?>  
<!--My comment-->  
<rootElement myAttribute="att">  
 1234<![CDATA[Text with a <left> bracket]]>mystring  
</rootElement>

LINQ to XML makes it as easy as possible to work with XML elements and attributes, but other XML node types are ready and available if you need them.

## Annotating nodes with user-defined information

LINQ to XML gives you the ability associate some application-specific information with a particular node in an XML tree. Examples include the line number range in the source file from which an element was parsed, the post schema validation type of the element, a business object that contains the data structures into which the XML information was copied and the methods for working with it (e.g. a real invoice object with data in CLR and application defined types), and so on.

LINQ to XML accommodates this need by defining methods on the XContainer class that can annotate an instance of the class with one or more objects, each of some unique type. Conceptually, the set of annotations on an XContainer object is akin to a dictionary, with the type being the key and the object itself being the value.

To add an annotation to an XElement or XDocument object:

XElement contact = new XElement(...);  
LineNumberInfo linenum = new LineNumberInfo(...);  
contact.AddAnnotation(linenum);

where LineNumberInfo is an application defined class for storing line number information. The annotation can be retrieved with:

LineNumberInfo annotation = contact.GetAnnotation<LineNumberInfo>();

The GetAnnotation() method returns null if the element does not have an annotation of the given type. The annotation is removed with:

contact.RemoveAnnotation<LineNumberInfo>();

There are a couple caveats: Annotation lookup is based on type identity; it doesn’t know about interfaces, inheritance, etc. For example, if you add an annotation with an object of type Customer which derives from type Person (or implements a Person interface), a call to GetAnnotation<Person>() won’t find it. Thus, when you annotate an XElement object, it should be with an instance of a private class of a type that you are sure will be unique.

## Outputting XML

After reading in your XML or creating some from scratch, and then manipulating it in various ways, you will probably want to output your XML. To accomplish this, you can use one of the overloaded Save() methods on an XElement or XDocument to output in a variety of ways. You can save to a file, a TextWriter, or an XmlWriter. For example, to save the XElement named contacts to a file:

contacts.**Save**(@"c:\contacts.xml");

## Validating XML

You can validate an XElement tree against an XML schema via extensions method in the System.Xml.Schema namespace. This is exactly the same functionality that was shipped in .NET 2.0, with only a “bridge” to expose the classes in that namespace to LINQ to XML.

To bring it into scope, use:

using System.Xml.Schema;

Use the .NET 2.0 classes and methods to populate XmlSchemaObject / XmlSchemaSet objects. There will be methods available to Validate()XElement, XAttribte, or XDocument objects against the schema and optionally populate a post schema validation infoset as annotations on the LINQ to XML tree.

# Querying XML with LINQ to XML

The major differentiator for LINQ to XML and other in-memory XML programming APIs is Language Integrated Query. Language Integrated Query provides a consistent query experience across different data models as well as the ability to mix and match data models within a single query. This section describes how to use Language Integrated Query with XML. The following section contains a few examples of using Language Integrated Query across data models.

The Standard Query Operators form a complete query language for IEnumerable<T>. The Standard Query Operators show up as extension methods on any object that implements IEnumerable<T> and can be invoked like any other method. This approach, calling query methods directly, can be referred to as “explicit dot notation.” In addition to the Standard Query Operators are query expressions for five common query operators:

* Where
* Select
* SelectMany
* OrderBy
* GroupBy

Query expressions provide an ease-of-use layer on top of the underlying explicit dot notation similar to the way that foreach is an ease-of-use mechanism that consists of a call to GetEnumerator() and a while loop. When working with XML, you will probably find both approaches useful. An orientation of the explicit dot notation will give you the underlying principles behind XML Language Integrated Query, and help you to understand how query expressions simplify things.

## Querying XML

We encourage you to review the reference materials in section 7, “References,” for in-depth information about Language Integrated Query. This section describes Language Integrated Query from a usage perspective, focusing on XML querying patterns and providing examples along the way.

The LINQ to XML integration with Language Integrated Query is apparent in three ways:

* Leveraging the Standard Query Operators
* Using XML Query extensions
* Using XML Transformation

The first is common with any other Language Integrated Query enabled data access technology and contributes to a consistent query experience. The last two provide XML-specific query and transform features.

### Standard Query Operators and XML

LINQ to XML fully leverages the Standard Query Operators in a consistent manner exposing collections that implement the IEnumerable interface. Review the Standard Query Operator document (see section 7 "References ") for details on how to use the Standard Query Operators. In this section we will cover two scenarios that occasionally arise when using Standard Query Operators.

#### Creating multiple peer nodes in a select

Creating a single XElement with the Select Standard Query Operator works as you would expect when doing a transform into XML but what if you need to create multiple peer elements within the same Select? For example let's say that we want to flatten out our contact list and list the contact information directly under the root <contacts> element rather than under individual <contact> elements. Like this:

<contacts>  
 <!-- contact -->  
 <name>Patrick Hines</name>  
 <phone type="home">206-555-0144</phone>  
 <phone type="work">425-555-0145</phone>  
 <address>  
 <address>  
 <state>WA</state>  
 </address>  
 </address>  
 <!-- contact -->  
 <name>Gretchen Rivas</name>  
 <address>  
 <address>  
 <state>WA</state>  
 </address>  
 </address>  
 <!-- contact -->  
 <name>Scott MacDonald</name>  
 <phone type="home">925-555-0134</phone>  
 <phone type="mobile">425-555-0177</phone>  
 <address>  
 <address>  
 <state>CA</state>  
 </address>  
 </address>  
</contacts>

To do this, you can use this query:

new XElement("contacts",  
 from c in contacts.Elements("contact")  
 select new object[] {  
 new XComment("contact"),  
 new XElement("name", (string)c.Element("name")),  
 c.Elements("phone"),  
 new XElement("address", c.Element("address"))  
 }  
);

Notice that we used an array initializer to create the sequence of children that will be placed directly under the contacts element.

#### Handling Null in a Transform

When you are writing a transform in XML using functional construction, you sometimes encounter situations where an element is optional, and you do not want to create some part of the target XML if the element is not there. For example, the following is a query that gets names and phone numbers putting the phone numbers under a wrapping element <phoneNumbers>.

new XElement("contacts",   
 from c in contacts.Elements("contact")  
 select new XElement("contact",  
 c.Element("name"),  
 new XElement("phoneNumbers", c.Elements("phone"))  
 )  
);

If the contact has no phone numbers, the phoneNumbers wrapping element will exist, but there will be no phone child elements. The following example demonstrates how to resolve this situation:

new XElement("contacts",   
 from c in contacts.Elements("contact")  
 select new XElement("contact",  
 c.Element("name"),  
 c.Elements("phone").Any() ?  
 new XElement("phoneNumbers", c.Elements("phone")) :  
 null  
 )  
);

Functional construction has no problem with null, so using the ternary operator inline (c.Elements("phone").Any() ? ... : null) lets you suppress the phoneNumber if the contact has no phone numbers. This same result could be achieved without using the ternary operator by calling out to a function from the query:

new XElement("contacts",   
 from c in contacts.Elements("contact")  
 select new XElement("contact",  
 c.Element("name"),  
 GetPhoneNumbers(c)  
 )  
);

...

static XElement GetPhoneNumbers(XElement c) {  
 if (c.Elements("phone").Any())  
 return new XElement("phoneNumbers", c.Elements("phone"));  
 else  
 return null;  
}

### XML Query Extensions

XML-specific query extensions provide you with the query operations you would expect when working in an XML tree data structure. These XML-specific query extensions are analogous to the XPath axes. For example, the Elements method is equivalent to the XPath \* (star) operator. The following sections describe each of the XML-specific query extensions in turn.

#### Elements and Content

The Elements query operator returns the child elements for each XElement in a sequence of XElements (IEnumerable<XElement>). For example, to get the child elements for every contact in the contact list, you could do the following:

foreach (XElement x in contacts.Elements("contact").Elements())  
{  
 Console.WriteLine(x);  
}

Note that the two Elements() methods in this example are different, although they do identical things. The first Elements is calling the XElement method Elements(), which returns an IEnumerable<XObject> containing the child elements in the single XElement contacts. The second Elements() method is defined as an extension method on IEnumerable<XObject>. It returns a sequence containing the child elements of every XElement in the list. The results of the above query look like this:

<name>Patrick Hines</name>  
<phone type="home">206-555-0144</phone>  
<phone type="work">425-555-0145</phone>  
<address>  
 <street1>123 Main St</street1>  
 <city>Mercer Island</city>  
 <state>WA</state>  
 <postal>68042</postal>  
</address>  
<netWorth>10</netWorth>  
<name>Gretchen Rivas</name>  
<phone type="mobile">206-232-4444</phone>  
<address>  
 <street1>123 Main St</street1>  
 <city>Mercer Island</city>  
 <state>WA</state>  
 <postal>68042</postal>  
</address>  
<netWorth>11</netWorth>  
<name>Scott MacDonald</name>  
<phone type="home">925-555-0134</phone>  
<phone type="mobile">425-555-0177</phone>  
<address>  
 <street1>345 Stewart St</street1>  
 <city>Chatsworth</city>  
 <state>CA</state>  
 <postal>92345</postal>  
</address>  
<netWorth>500000</netWorth>

If you want all of the children with a particular name, you can use the Elements(XName) overload. For example:

foreach (XElement x in contacts.Elements("contact").Elements("phone"))  
{  
 Console.WriteLine(x);  
}

This would return:

<phone>206-555-0144</phone>  
<phone>425-555-0145</phone>  
<phone>925-555-0134</phone>  
<phone>425-555-0177</phone>

#### Descendants and Ancestors

The Descendants and Ancestors query operators let you query down and up the XML tree, respectively. Descendants with no parameters gives you all the child content of an XElement and, in turn, each child's content down to the leaf nodes (the XML subtree). Optionally, you can specify an XName (Descendants(XName)) and retrieve all of the descendants with a specific name, or specify a type (Descendants<T>) and retrieve all of the descendants of a specified LINQ to XML type (for example, XComment).

For example, to get all of the phone numbers in our contact list, you could do the following:

contacts.Descendants("phone");

Descendants and Ancestors do not include the current node. If you use Descendants() on the root element, you will get the entire XML tree except the root element. If you want to include the current node, use SelfAndDescendants, which lets you specify an XName or type.

Ancestors and SelfAndAncestors work similarly to Descendants and SelfAndDescendants; they just go up the XML tree instead of down. For example, you can retrieve the first phone number in the contacts XML tree, and then print out its ancestors:

XElement phone = contacts.Descendants("phone").First();  
foreach (XElement a in phone.Ancestors()) {  
 Console.WriteLine(a.Name);  
};

The results will show:

contact  
contacts

If you do the same thing with SelfAndAncestors, the output will also show phone:

XElement phone = contacts.Descendants("phone").First();  
foreach (XElement a in phone.**SelfAndAncestors()**) {  
 Console.WriteLine(a.Name);  
};

The results will show:

Phone  
contact  
contacts

The Descendants and Ancestors XML query extensions can greatly reduce the code needed to traverse an XML tree. You will find that you use them often for quick navigation in an XML tree.

#### Attributes

The Attributes XML query extension is called on an IEnumerable<XElement> and returns a sequence of attributes (IEnumerable<XAttribute>). Optionally, you can specify an XName to return only attributes with that name. For example, you could get a list of the distinct types of phone numbers that are in the contact list:

contacts.Descendants("phone").  
 **Attributes(**"type").Select(t => t.Value).Distinct();

which will return:

home  
work  
mobile

#### ElementsBeforeSelf, ElementsAfterSelf, NodesBeforeSelf, NodesAfterSelf

If you are positioned on a particular element, you sometimes want to retrieve all of the child elements or content before that particular element, or the child elements or content after that particular element. The ElementsBeforeSelf query extension returns an IEnumerable<XElement> containing the sibling elements that occur before that element. ElementsAfterSelf returns the sibling elements that occur after that element. The NodesBeforeSelf query extension returns the previous siblings of any type (e.g., string, XComment, XElement, etc.). Consequently, it returns an IEnumerable<XNode>. Similarly, NodesAfterSelf returns the following siblings of any type.

#### Technical Note: XML Query Extensions

The LINQ to XML specific extension methods are found in the XElementSequence class. Just as the Standard Query Operators are generally defined as extension methods on IEnumerable<T>, the XML query operators are generally defined as extension methods on IEnumerable<XElement>. XElementSequence is just a container class to hold these extension methods. Most likely you will never call these static methods through XElementSequence – but you could. For example, consider the following query to get all of the phone numbers in the contact list.

IEnumerable<XElement> phones =   
 contacts.Elements("contact").Elements("phone");

This could be rewritten using the static extension method Elements(this IEnumerable<XElement> source, XName name) in ElementSequence like this:

IEnumerable<XElement> phones =   
 XElementSequence.Elements(contacts.Elements("contact"), "phone");

You can learn more about the technical details of query extensions in the C# 3.0 Overview document (see section 7, "References").

### XML Transformation

Transforming XML is a very important XML usage scenario. It is so important that it is a critical feature in two key XML technologies: XQuery and XSLT. While XSLT is accessible in LINQ to XML, the “pure” LINQ way to do transformation, which works for all types of input data, is via functional construction. Most transformations to an XML document can be thought of in terms of functionally constructing your target XML. In other words, you can “begin with the end in mind,” shaping your goal XML and filling in chunks of the XML by using combinations of queries and functions as needed.

For example, you might want to transform the format of the contact list to a customer list. Beginning with the end in mind, the customer list needs to look something like this:

<Customers>  
 <Customer>  
 <Name>Patrick Hines</Name>  
 <PhoneNumbers>  
 <Phone type="home">206-555-0144</Phone>  
 <Phone type="work">425-555-0145</Phone>  
 </PhoneNumbers>  
 </Customer>  
</Customers>

Using functional construction to create this XML would look like this:

new XElement("Customers",  
 new XElement("Customer",  
 new XElement("Name", "Patrick Hines"),  
 new XElement("PhoneNumbers",  
 new XElement("Phone",   
 new XAttribute("type", "home"),  
 "206-232-2222"),  
 new XElement("Phone",  
 new XAttribute("type", "work"),  
 "425-555-0145")  
 )  
 )  
);

To transform our contact list to this new format, you would do the following:

new XElement("Customers",  
 from c in contacts.Elements("contact")  
 select new XElement("Customer",  
 new XElement("Name", (string) c.Element("name")),  
 new XElement("PhoneNumbers",  
 from ph in c.Elements("phone")  
 select new XElement("phone", (string) ph,  
 ph.Attribute("type")  
 )  
 )  
 )  
 );

Notice how the transformation aligns with the structure of our target document. You start by creating the outer, root element of the target XML:

new XElement("Customers", ...

You will need to create a Customer XElement that corresponds to every contact in the original XML. To do this, you would retrieve all the contact elements under contacts, because you have to select what you need for each contact.

... from c in contacts.Elements("contact")...

The Select begins another functional construction block that will be executed for each contact.

select new XElement("Customer",

You now construct the <Customer> part of the target XML. You start by creating a Customer XElement:

select new XElement("Customer",  
 new XElement("Name", (string) c.Element("name")),

The <PhoneNumbers> child is more complex because the phone numbers in the contact list are listed directly under the contact:

<contact><phone>...</phone><phone>...</phone></contact>

To accomplish this, query the phone numbers for the contact and put them as children under the <PhoneNumbers> element:

...  
new XElement("PhoneNumbers",  
 from ph in c.Elements("phone")  
 select new XElement("phone", (string) ph,  
 ph.Attribute("type")  
 )  
)

In this code, you query the contact's phone numbers, c.Elements("phone"), for each phone. We also create a new XElement called Phone with same type attribute as the original phone, and with the same value.

You will often want to simplify your transformations by having functions that do the work for portions of your transformation. For example, you could write the above transformation using more functions to break up the transformation. Whether you decide to this is completely up to you, just as you might or might not decide to break up a large, complex function based on your own design sensibility. One approach to breaking up a complex function looks like this:

new XElement("Customers", **GetCustomers(contacts)**);

static IEnumerable<XElement> GetCustomers(XElement contacts) {  
 return from c in contacts.Elements("contact")  
 select FormatCustomer(c);  
 }

static XElement FormatCustomer(XElement c) {  
 return new XElement("Customer",  
 new XElement("Name", (string) c.Element("name"),  
 GetPhoneNumbers(c)));  
}

static XElement GetPhoneNumbers(XElement c) {  
 return !c.Elements("phone").Any() ? null :  
 new XElement("PhoneNumbers",  
 from ph in c.Elements("phone")  
 select new XElement("Phone",   
 ph.Attribute("type"),  
 (string) ph)  
 );  
}

This example shows a relatively trivial instance of the power of transformation in .NET Framework Language Integrated Query. With functional construction and the ability to incorporate function calls, you can create arbitrarily complex documents in a single query/transformation. You can just as easily include data from a variety of data sources, as well as XML.

## Using Query Expressions with XML

There is nothing unique in the way that LINQ to XML works with query expressions so we will not repeat information in the reference documents here. The following shows a few simple examples of using query expressions with LINQ to XML.

This query retrieves all of the contacts from Washington, orders them by name, and then returns them as string (the result of this query is IEnumerable<string>).

from c in contacts.Elements("contact")  
where (string) c.Element("address").Element("state") == "WA"  
orderby (string) c.Element("name")  
select (string) c.Element("name");

This query retrieves the contacts from Washington that have an area code of 206 ordered by name. The result of this query is IEnumerable<XElement>.

from c in contacts.Elements("contact"),  
 ph in c.Elements("phone")  
where (string) c.Element("address").Element("state") == "WA" &&  
 ph.Value.StartsWith("206")  
orderby (string) c.Element("name")  
select c;

Here is another example retrieving the contacts that have a net worth greater than the average net worth.

from c in contacts.Elements("contact"),  
 average = contacts.Elements("contact").  
 Average(x => (int) x.Element("netWorth"))  
where (int) c.Element("netWorth") > average  
select c;

## Using XPath and XSLT with LINQ to XML

LINQ to XML supports a set of “bridge classes” that allow it to work with existing capabilities in the System.Xml namespace, including XPath and XSLT. (Note that System.Xml supports only the 1.0 version of these specifications in “Orcas”).

Extension methods supporting XPath are enabled by referencing the System.Xml.XPath namespace

using System.Xml.XPath;

This brings into scope CreateNavigator overloads to create XpathNavigator objects, XPathEvaluate overloads to evaluate an XPath expression, and XPathSelectElement[s] overloads that work much like SelectSingleNode and XPatheXelectNodes methods in the System.Xml DOM API. To use namespace-qualified XPath expressions, it is necessary to pass in a NamespaceResolver object, just as with DOM.

For example, to display all elements with the name “phone”:

foreach (var phone in contacts.XPathSelectElements("//phone"))

Console.WriteLine(phone);

Likewise, extension methods supporting XSLT are enabled by referencing the System.Xml.Xsl namespace

using System.Xml.Xsl;

That enables the familiar System.Xml classes such as XslTransform, XslCompiledTransform, XsltSettings, etc. to be used to transform LINQ to XML objects.

# Mixing XML and other data models

Language Integrated Query provides a consistent query experience across different data models via the Standard Query Operators and the use of Lambda Expressions. It also provides the ability to mix and match Language Integrated Query enabled data models/APIs within a single query. This section provides a simple example of two common scenarios that mix relational data with XML, using the Northwind sample database.

We will use the Northwind sample database and for these examples.

## Reading from a database to XML

The following is a simple example of reading from the Northwind database (using LINQ to SQL) to retrieve the customers from London, and then transforming them into XML:

XElement londonCustomers =   
 new XElement("Customers",   
 from c in db.Customers  
 where c.City == "London"  
 select new XElement("Customer",  
 new XAttribute("CustomerID", c.CustomerID),  
 new XElement("Name", c.ContactName),  
 new XElement("Phone", c.Phone)  
 )  
 );

Console.WriteLine(londonCustomers);

The resulting XML output is this:

<Customers>  
 <Customer CustomerID="AROUT">  
 <Name>Mark Harrington</Name>  
 <Phone>(171) 555-0188</Phone>  
 </Customer>  
 <Customer CustomerID="BSBEV">  
 <Name>Michelle Alexander</Name>  
 <Phone>(171) 555-0112</Phone>  
 </Customer>  
 <Customer CustomerID="CONSH">  
 <Name>Nicole Holliday</Name>  
 <Phone>(171) 555-0182</Phone>  
 </Customer>  
 <Customer CustomerID="EASTC">  
 <Name>Kim Ralls</Name>  
 <Phone>(171) 555-0197</Phone>  
 </Customer>  
 <Customer CustomerID="NORTS">  
 <Name>Scott Culp</Name>  
 <Phone>(171) 555-0173</Phone>  
 </Customer>  
 <Customer CustomerID="SEVES">  
 <Name>Deepak Kumar</Name>  
 <Phone>(171) 555-0117</Phone>  
 </Customer>  
</Customers>

## Reading XML and Updating a Database

You can also read XML and put that information into a database. For this example, assume that you are getting a set of customer updates in XML format. For simplicity, the update records contain only the phone number changes.

The following is the sample XML:

<customerUpdates>  
 <customerUpdate>  
 <custid>ALFKI</custid>  
 <phone>206-555-0103</phone>  
 </customerUpdate>  
 <customerUpdate>  
 <custid>EASTC</custid>  
 <phone>425-555-0143</phone>  
 </customerUpdate>  
</customerUpdates>

To accomplish this update, you query for each customerUpdate element and call the database to get the corresponding Customer record. Then, you update the Customer column with the new phone number.

foreach (var cu in customerUpdates.Elements("customerUpdate")) {  
 Customer cust = db.Customers.  
 First(c => c.CustomerID == (string)cu.Element("custid"));  
 cust.Phone = (string)cu.Element("phone");  
}

db.SubmitChanges();

These are just a few examples of what you can do with Language Integerated Query across data models. For more examples of using LINQ to SQL, see the LINQ to SQL Overview document (see section 7, “[References](#_References).”).

# Layered Technologies Over LINQ to XML

The LINQ to XML XML Programming API will be the foundation for a variety of layered technologies. Two of these technologies are discussed below.

## LINQ to XML in Visual Basic 9.0

VB 9.0 will provide deep support for LINQ to XML***.*** Instead of using methods to construct and navigate XML, VB 9.0 uses XML literals for construction and *Xml Axis Properties*for navigation. This is an important distinction and is closer to the design center of VB. XML literals allow VB developers to construct LINQ to XML objects such as XDocument and XElement directly using familiar XML syntax. Values within these objects can be created with expression evaluation and variable substitution. *Xml Axis Properties*will allow developers to access XML nodes directly by special syntax that include the Xml axis and the element or attribute name, rather than indirectly using method calls. These two features will provide deep, explicit, easy to use and powerful support for XML and LINQ to XML programming in VB.

### XML Literals

Let us revisit the first example in this paper, (section 2.1.1.1), but this time written in VB. The syntax is very similar to the existing C# syntax:

Dim contacts As XElement = \_  
 New XElement("contacts", \_  
 New XElement("contact", \_  
 New XElement("name", "Patrick Hines"), \_  
 New XElement("phone", "206-555-0144", \_  
 New XAttribute("type", "home")), \_  
 New XElement("phone", "425-555-0145", \_  
 New XAttribute("type", "work")), \_  
 New XElement("address", \_  
 New XElement("street1", "123 Main St"), \_  
 New XElement("city", "Mercer Island"), \_  
 New XElement("state", "WA"), \_  
 New XElement("postal", "98040"))))

The above VB statement initializes the value of the variable contacts to be a new object of type XElement using the traditional API approach. VB allows us to go one-step further than calling the LINQ to XML APIs to create new objects; it lets us write the XML inline using actual XML syntax:

Dim contacts As XElement = \_  
 <contacts>  
 <contact>  
 <name>Patrick Hines</name>  
 <phone type="home">206-555-0144</phone>  
 <phone type="work">425-555-0145</phone>  
 <address>  
 <street1>123 Main St</street1>  
 <city>Mercer Island</city>  
 <state>WA</state>  
 <postal>98040</postal>  
 </address>  
 </contact>  
 </contacts>

The XML structure of the result XElement is obvious, which makes the VB code easy to read and maintain. The VB compiler translates the XML literals on the right-hand side of the statement into the appropriate calls to the LINQ to XML APIs, producing the exact same code as in the first example. This ensures full interoperability between Visual Basic and other languages that use LINQ to XML.

Note that we do not need line continuations in XML literals. This allows developers to copy and paste XML from/to any XML source document.

Let us take another example where we create the same contact object but use variables instead. VB allows embedding expressions in the XML literals that create the XML values at run time. For example suppose that the contact name was stored in a variable called MyName. Now we may write as follows:

Dim myName = "Patrick Hines"  
Dim contact As XElement = <contact>  
 <name><%=myName %></name>   
 </contact>

People familiar with ASP.NET will immediately recognize the "<%=" and "%>" syntax. This syntax is used to bracket VB expressions, whose values will become the element content. Substituting the value of a variable like MyName is only one example, the expression could just as easily have been a database lookup, an array access, a library function call, that return a type that is valid element content such as string, List of XElement etc.

The same expression hole syntax is used within the angle brackets of XML syntax. In the following example, the value of the attribute "type" is set from an expression:

Dim phoneType = IIf(i = 1, "home", "work")   
Dim contact = <contact>  
 <phone type=<%= phoneType %>>206-555-0144</phone>  
 </contact>

Similarly, the name of an element can be computed from an expression:

Dim MyName = "Patrick Hines"  
Dim elementName = "contact"  
Dim contact As XElement = <<%=elementName %>>  
 <name><%= MyName %></name>   
 </>

Note that it is valid to use "</>" to close an element. This is a very convenient feature, especially when the element name is computed.

### Xml Axis Properties

In addition to using XML literals for constructing XML, VB 9.0 also simplifies accessing and *navigating* XML structures via Xml axis properties that can be used with XElement and XDocument types. That is, instead of calling explicit methods to navigate and locate elements and attributes, we can use Xml axis properties as LINQ to XML object properties. For example:

* use the child axis contact.<phone> to get all "phone" elements from the contact element,
* use the attribute axis phone.@type to get the "type" attributes of the phone element,
* use the descendants axis contact...<city> – written literally as three dots in the source code – to get all "city" children of the contact element, no matter how deeply in the hierarchy they occur,
* use the Value extension property to get the string value of the first object in the IEnumerable that is returned from the Xml axis properties.
* and finally, use the extension indexer on IEnumerable(Of T) to select the first element of the resulting sequence.

We put all these innovations together to make the code simpler, for example printing the phone's type and the contact's city looks as follows:

For Each phone In contact.<phone>  
 Console.WriteLine(phone.@type.Value)  
 Next  
 Console.WriteLine(contacts...<city>.Value)

The compiler knows to use Xml axis properties over XML when the target expression is of type XElement, XDocument, or a collection of these types.

The compiler translates the Xml axis propoerties as follows:

* the child-axis expression contact.<phone> into the raw LINQ to XML call contact.Elements("phone"), which returns the collection of all child elements named "phone" of the contact element,
* the attribute axis expression phone.@type into phone.Attributes("type"), which returns IEnumerable of the attributes named "type" of phone,
* and finally, the descendant axis contact...<city> expression into a combination of steps, first it calls the contact.Descendants("city") method, which returns the collection of all elements named city at any depth below contact, then it get gets the first one and if it exists it calls the Value property on that element.

The equivalent code after translation into LINQ to XML calls is as below:

For Each Dim phone In contact.Element("phone")  
 Console.WriteLine(CStr(phone.Attribute("type")))  
Next

If Any(contact.Descendants("city")) Then  
 Console.WriteLine(ElementAt(contact.Descendants("city"),0)).Value)

End If

### Putting it all together

Used together, Language Integrated Query and the new XML features in VB 9.0, provides a simple but powerful way to perform many common Xml programming tasks. Let us examine the query in 3.1.1.1 that creates a flattened contact list and removes the contact element:

<contacts>  
 <!-- contact -->  
 <name>Patrick Hines</name>  
 <phone type="home">206-555-0144</phone>  
 <phone type="work">425-555-0145</phone>  
 <address>  
 <address>  
 <state>WA</state>  
 </address>  
 </address>  
 </contacts>

The following is the C# version:

XElement contacts =   
 new XElement("contacts",  
 from c in contacts.Elements("contact")  
 select new object[] {  
 new XComment("contact"),  
 new XElement("name", (string)c.Element("name")),  
 c.Elements("phone"),  
 new XElement("address", c.Element("address"))  
 }  
 );

In VB 9.0 it can be written as follows:

Dim contacts as XElement = \_  
 <contacts>  
 <%= From c In contacts \_  
 Select \_  
 <>  
 <!-- contact -->   
 <name><%= c.<name>.Value %> </name>  
 <%= c.<phone> %>   
 <address><%= c.<address> %> </address>  
 </>  
 %>  
 </contacts>

## Schema aware XML Programming

As discussed in previous section Visual Basic will be providing late bound access and navigation of LINQ to XML by translating operators into LINQ to XML queries at run time. This section discusses the opportunity to take advantage of XML Schema to provide strong typing at compile time.

You may have noticed in the examples throughout this paper that the LINQ to XML code has a significant number of quotes and casts in it. Take the following code sample which will total orders for a specific zip code.

public static double GetTotalByZip(XElement root, int zip) {  
 return (from order in root.Elements(**"order"**),  
 item in order.Elements(**"item"**),  
 where **(int)**order.Element(**"address"**).Element(**"postal"**) == zip  
 select (**(double)**price \* **(int)**qty)).Sum();  
 }

The underlying LINQ to XML XML programming API knows nothing about the shape of the XML, it is not aware that there will be a zip attribute under an order element and that its type is int. Consequently, you as a developer must know and assert that information (using quotes and casts).

This situation can be improved substantially if there is meta-data around that can be used to generate Common Language Runtime types that contain the knowledge of how the XML is structured and the appropriate simple types. XML Schema can be leveraged for exactly this purpose. We are prototyping strongly typed, schema based support on top of LINQ to XML. For example, with XML Schema meta-data for the above XML it would be possible to write code like:

public static double GetTotalByZip(Orders root, int zip) {  
 return (from order in root.OrderCollection,  
 item in order.Items  
 where order.Address.Postal == zip  
 select (item.Price \* item.Quantity)).Sum();  
 }

Instead of quotes and casts you are working with types such as Orders and Items, and properties such as Price, and Quantity.

Plans, timelines, and preview schedules for a potential product based on this investigation have not been determined.

# February 2007 CTP Release Notes

The LINQ to XML specification is still evolving, and will continue to evolve before it is released. We release previews of this technology to get comments from potential users. The changes in this CTP reflect feedback from the previous CTPs, and subsequent releases will reflect feedback from this CTP.

## Changes since the May 2006 CTP

### Bridge Classes

These implementations of System.Xml interfaces such as XmlReader, XmlWriter, and XPathNavigator will allow XPath / XSLT to be used over LINQ to XML trees, allow XSLT transformations to produce an LINQ to XML tree, and allow efficient data interchange between DOM and LINQ to XML applications. There is also a Validate extension method to validate an XElement tree against an XML Schema.

### Event Model

This allows LINQ to XML trees to be efficiently synchronized with a GUI, e.g. a Windows Presentation Foundation application

### XObject class

There is a new base class for both XElement and XAttribute, introduced largely to support the event model.

### XStreamingElement class removed

This was done because there was uncertainty about the design of various features to support efficient production and consumption of large XML documents. The result of the design exercise was to confirm the original design of XStreamingElement, so it will be put back in the next release.

## Non-exhaustive list of planned features in future releases

* The IXmlSerializable interface will be supported.
* The XStreamingElement class will be added back to allow trees of enumerators to be defined that can be “lazily” serialized as XML in a deferred manner.
* The IXmlSerializeable interface will be implemented to facililtate the use of LINQ to XML in conjunction with the Microsoft web services APIs.

# References

These documents can be found online at the [Project LINQ website](http://msdn.microsoft.com/data/linq):

* The LINQ Project Overview, .NET Language Integrated Query
* The .NET Standard Query Operators
* C# Version 3.0 Specification
* Overview of VB 9.0
* LINQ to SQL .NET Language Integrated Query for Relational Data

Other documents, samples, and tutorials are also available.