**Chapter 4. XML and Rational Functional Tester**

**Jeffrey R. Bocarsly**

*XML is important for Rational Functional Tester users for two reasons. First, data is now frequently formulated in an XML format, either for persistence in a file, a database, or to be sent (usually via HTTP) to another application. Verifying the data content of XMLs is often an important software quality task. So, parsing XMLs to capture data to compare it to baseline data is a common software testing need. Second, Rational Functional Tester employs the XML format to persist its own data. This chapter doesn’t discuss the details of Rational Functional Tester’s use of XML; rather, it covers all of the core XML-handling tasks that are needed to test XML data and to manipulate Rational Functional Tester XMLs. For discussions of how Rational Functional Tester uses XML, see* [*Chapter 10*](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch10.html#ch10)*, “*[*Advanced Scripting with Rational Functional Tester Test-Objects*](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch10.html#ch10)*,” and* [*Chapter 16*](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch16.html#ch16)*, “*[*Internationalized Testing with Rational Functional Tester*](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch16.html#ch16)*.”*

*For this chapter, we assume a basic knowledge of XML, its syntax, and terminology. If you’re new to XML, consult an XML primer. (See the end of the chapter for some suggestions.)*

*As you read this chapter, you’ll see code snippets for the Java and Eclipse flavors of Rational Functional Tester and the VB.NET and Visual Studio flavors. Each language has a set of class libraries that it uses to manipulate XML. However, the implementations do differ, and there are numerous syntax differences in the same general API paradigm.*

**Handling XML in Rational Functional Tester**

This chapter uses a simple sample XML, but one that demonstrates all the basic moves you’ll need to make. It follows:



Our discussion of XML handling in Rational Functional Tester starts with a brief overview of the two main XML-handling standards, DOM (Document Object Model) and SAX (Simple API for XML). Both DOM and SAX are W3C standards; in Java, DOM and SAX are implemented in the org.w3c.dom and org.xml.sax packages. In VB.NET, the System.Xml libraries implement DOM and a SAX-like parser.

**DOM and SAX**

DOM and SAX are fundamentally different in that the DOM loads and persists an entire XML document in memory (in a tree-structure form), whereas SAX is event-driven, meaning that a SAX parser fires off a sequence of events reflecting XML structure and content as it scans through an XML document. A SAX parser never holds a whole document in memory. You see output from a SAX parser sooner than from a DOM for equal tasks on equal documents because the SAX parser fires off its events as it encounters them during its scan. In the DOM approach, the entire document is parsed and loaded in memory before any processing can occur.

Because of this key difference in structure, the DOM demands more memory than a SAX parser does for an equivalent document. On the other hand, the DOM provides random access to all document nodes at will because they are all in memory. One of the major factors in the choice of which to use is the size and complexity of the largest document that will have to be parsed relative to the available memory. The DOM is most useful when an XML should be persisted in memory for repeated access. SAX is strongest for processing large XMLs quickly for specific data content where it is not necessary to keep the full XML in memory.

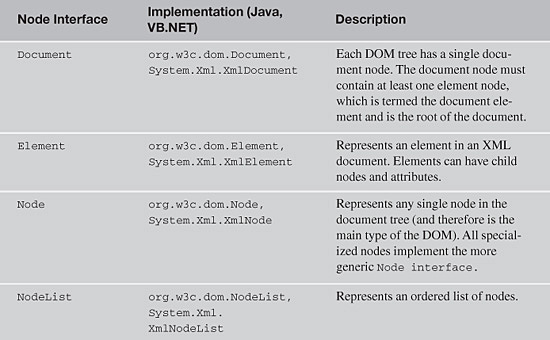
As noted previously, a major intersection of test automation and XML is data content. Code to validate the data content of XML documents is mostly what you need to write, and the most direct route to this is through the DOM. The issue is not that data content can’t be validated with SAX, but more that the DOM is the path of least resistance; the code to extract the data is simpler. So, if your XMLs do not eat up too much memory, or they are not large enough to put you in the slow processing regime, DOM is the easiest route to go. If you are parsing large documents, then SAX becomes an attractive choice.

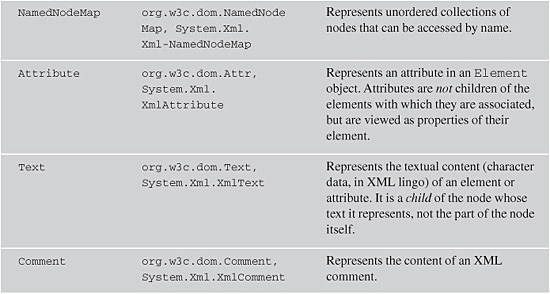
**Using the XML DOM with Rational Functional Tester**

We begin our foray into the DOM with a brief summary of the logical structure of a DOM tree. An exhaustive discussion of DOM concepts is beyond the scope of this book, but this chapter discusses some of the most common structures and concepts so that the basic DOM manipulations are intelligible (see <http://www.w3.org/TR/2004/REC-DOM-Level-3-Core-20040407/> for the w3C DOM specification).

From the DOM perspective, every XML document has a tree structure. The tree is composed of nodes, of which there are 12 specific types. Each node has a well-defined relationship to the surrounding nodes and is accessed through a specific interface. In addition, all nodes implement the more general Node interface. Collections of nodes are represented by the NodeList and NamedNodeMap interfaces. [Table 4.1](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch04.html#ch04tab01) shows the most common Node interfaces.

**Table 4.1** Common XML DOM Node Interfaces and Node Collection Interfaces





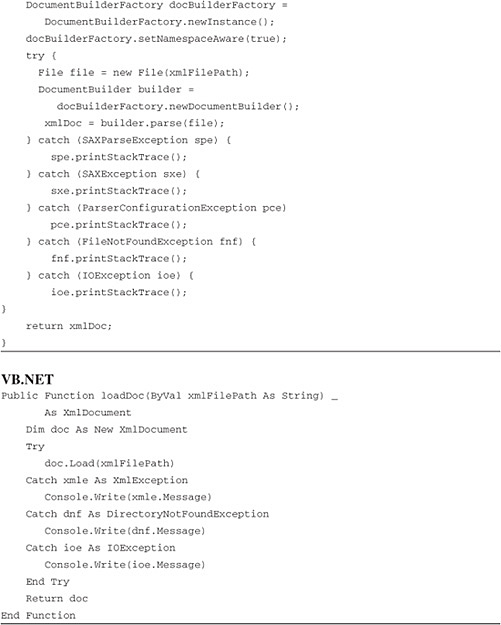
\*Less commonly used Node interfaces are: DocumentFragment, DocumentType, EntityReference, ProcessingInstruction, CDATASection, Entity, Notation

**Loading a Document into the XML DOM**

The DOM Document provides the highest-level access to the DOM object tree, and most likely, all of your DOM manipulations will start by getting a reference to a Document. [Listing 4.1](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch04.html#ch04ex01) shows a simple method to load an XML file on disk to the DOM using a standard factory object (in Java) or standard object creation (in VB.NET). Note that this method uses an elaborate cascade of catch clauses, illustrating the range of exceptions that might be thrown. These include parser exceptions, even though the code uses the DOM. Parsing is an underlying step that occurs prior to loading an XML into the DOM. This is clearest in the Java code, although it also reflects what is happening under the hood in VB.NET. The methods return a Document (Java) or XmlDocument (VB.NET) reference that is used for access to the Document’s node tree.

**Listing 4.1** Loading an XML into the DOM

image



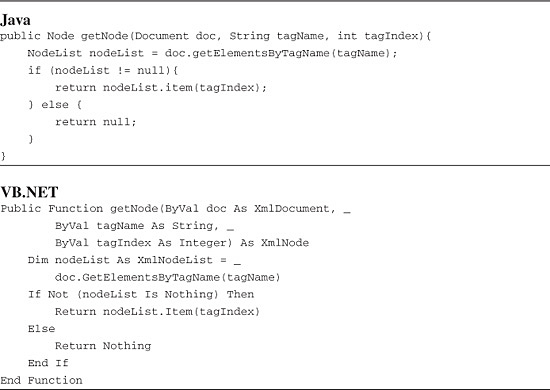
It is relatively unlikely that you are going to encounter all of the exceptions illustrated here when loading a Document. In your own work, you might want to explicitly handle the more common exceptions (SAXParseException and SAXException in Java, and XmlException in VB.NET) and make the remaining exception handling more generic.

**Capturing Element Values Using the XML DOM**

One of the most important tasks you have to perform with XML is the extraction of data content from XML in preparation for comparing the data to baseline values, and output of the results to the Rational Functional Tester log. Data resides in XML in one of two places: either it appears as the text content of a tag (<sometag>your\_text\_here</sometag>) or it appears as an attribute on a tag (<anothertag myattrib='attribvalue'/>). This chapter looks at ways to pull data out of both of these locations.

Probably the most common place to find data is as the text of a tag. To start extracting the text of a tag, you build simple methods to pull target elements out of the XML structure. The simplest way to identify an element is by its tag name and its index. Remember that XML syntax allows you to have as many identically named tags as you’d like, so identifying a specific element by name isn’t enough information to uniquely find a tag. Asking the DOM for an element by tag name and index fully specifies a given tag. When you specify your indices, remember that, as usual, indexing uses a zero-based count. To find the tag, use the DOM’s getElementsByTagName() method of the Document object to generate a NodeList (Java) or an XmlNodeList (VB.NET), and then simply ask for the specified list element from the list. [Listing 4.2](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch04.html#ch04ex02) shows the methods.

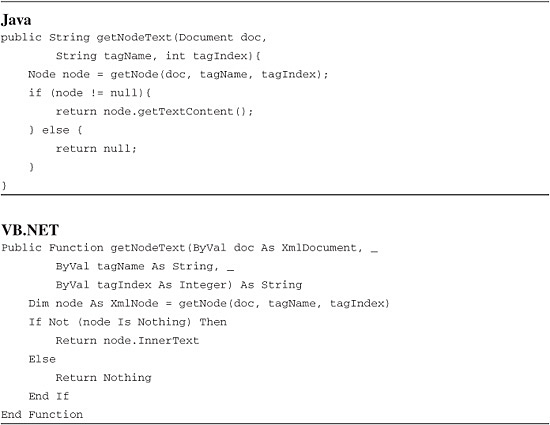
**Listing 4.2** Methods to return a specific XML Node reference based on name and index



Note that a null reference is returned (either null in Java or Nothing in VB.NET) if the call fails. This is because getElementsByTagName() does not throw an exception if you specify an element that does not exist (either by invalid tag name or index). This method simply returns a null reference, indicating that no list items were returned. This makes it convenient to simply check whether the list is null or not, but of course, it places the burden for checking for nulls on the calling code.

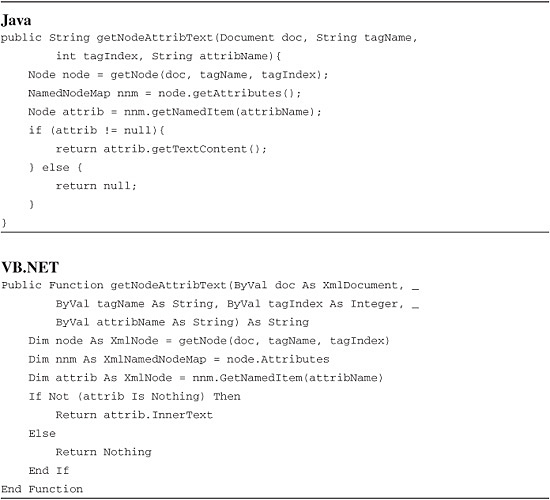
After you find the element you want, you can capture its text content by calling the getTextContent() method on the org.w3c.dom.Node class in Java or accessing the InnerText property of the System.Xml.XmlNode class in VB.NET. This is shown in [Listing 4.3](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch04.html#ch04ex03).

**Listing 4.3** Method to return the text of an XML Node reference



The final task in this section is to capture data from the other place it might be found, in an element’s attribute. The syntax around attributes is not identical to that regarding DOM nodes, but it is similar. To capture attribute data, you not only need to specify the same information as you did previously to specify an element, but you also need to then get a list of the target element’s attributes, by getting a NamedNodeMap reference (Java) or an XmlNamedNodeMap (VB.NET). You then use the NamedNodeMap to specify which attribute you want to get the text content of by specifying the attribute name. Attribute names must be unique in an element (unlike the element names themselves in the document), so you have to know which attribute you want by name. With the attribute name, you can call getNamedItem() to return a Node reference to the attribute, and then you retrieve the attribute’s text data as in [Listing 4.3](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch04.html#ch04ex03). [Listing 4.4](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch04.html#ch04ex04) shows the methods.

**Listing 4.4** Method to return an XML Attribute value

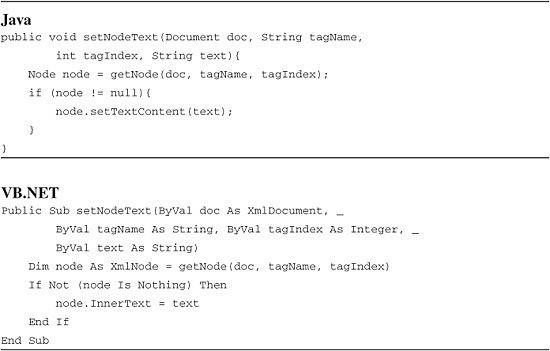


**Changing Element Data Using the XML DOM**

This section briefly discusses changing element data because it can be useful for tasks such as manipulating XML configuration files. If part of your test bed setup involves XML configuration files, this type of code can be a time saver, enabling you to automate configuration changes during your test runs. In addition, as mentioned before, Rational Functional Tester persists project information in XML format and manipulation of this data in a script can sometimes provide a powerful technique for extending projects (see [Chapter 16](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch16.html#ch16) for an example).

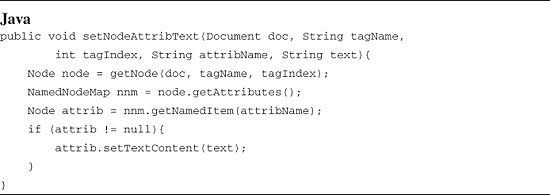
[Listing 4.2](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch04.html#ch04ex02) contains the core of what is needed to change element data. The method getNode() is used to obtain a reference to the node whose data you want to change. The Node reference has a method (Java) or property (VB.NET) to change its text value. [Listing 4.5](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch04.html#ch04ex05) shows these methods.

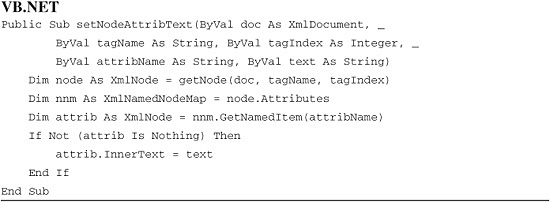
**Listing 4.5** Modifying XML node text



To change attribute text, you can use exactly the same approach as you just saw for the Node text; this is shown in [Listing 4.6](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch04.html#ch04ex06).

**Listing 4.6** Modifying XML attribute values





**Finding Elements Using XPath**

The previous discussion illustrated how to find XML nodes based on tag name and index. From the perspective of these examples, searching for specific nodes in a DOM tree is a process of iterating through all candidate nodes, and checking data or metadata, to find the ones you need. Although this works for many needs, there is a more powerful way to query XML documents for their content: XML Path language (XPath).

XPath is an XML query language based on the notion of paths through an XML tree. It does not use an XML notation, but rather a path-like notation similar to that used to describe directory paths. These paths can be modified with a range of qualifiers (which brings in considerable complexity), so that precise queries can be constructed. It can be extraordinarily useful to query XML documents with a query language format when evaluating data content. A full discussion of XPath is beyond the scope of this book (you’ll see that comment a number of times in this chapter), but some of the basics and some examples that show how valuable XPath can be for the tasks involved in XML data validation are discussed.

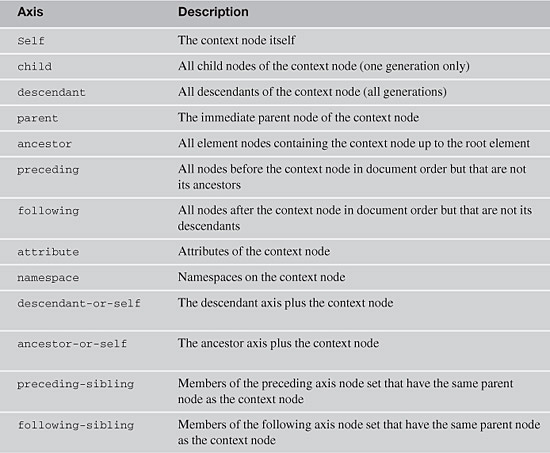
The current XPath standard XPath 1.0 is part of Java and comes with the Eclipse version of Rational Functional Tester. Likewise, it is part of VS.NET and is available in the VB.NET flavor of Rational Functional Tester. However, XPath 2.0 and something called XQuery 1.0, both significant additions to XML query domain, are on the horizon. While current XPath 1.0 syntax should be supported in these new standards, many additional query styles and functionalities will be supported. The discussion here is based only on XPath 1.0; however, when XPath 2.0 becomes available, you’ll want to check out the many new features that can enhance your testing queries.

XPath gives you the ability to query for XML elements from a document in a way broadly analogous to how you query a database with SQL for rows. The notion of a query and the syntax of XPath is, however, completely different from that of SQL. XPath queries are defined from the perspective of a specific node in a given XML. This node is called the context node. From the context node, you can move along a location path to select a specific set of nodes from a document. The grammar of location paths allows several different kinds of phrases: a *location step*, a *location path*, a *node test*, and a *predicate*.

**Location Steps**

A location step is a step from the context node to another node that has a specific relationship to the context node. For example, you could step from the context node to its child node. This step is considered a movement along the child “axis.” XPath, in fact, defines 13 different types of axes along which location steps can be taken. Of the 13 axes, 9 can be conceptualized as “fundamental” axes, while the remaining four are composite axes, meaning that they are conceptually related to two of the fundamental axes. The XPath axes are shown in [Table 4.2](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch04.html#ch04tab02) along with their definitions.

**Table 4.2** XPath Axes



The basic syntax for a location step is: “*axis::nodeName.*” For example, using our sample XML, with the Document node as the context node, the following location step returns the document element:

"child::ibmTools"

If you used the following location step with the document element as context node, XPath would return all three <tool> nodes:

"descendant::tool"

Finally, using the first <tool> node as the context node, the value of the first rxt attribute is returned by the following location step:

"attribute::rxt"

**Location Paths**

Location steps are combined to create location paths with a forward slash (“/”). The node-set that is selected by the first location step becomes the context node-set for the second step, down through the full path. So, for example, using the Document node as context node, the following path returns the three <tool> nodes because the <ibmTools> node is a child of the Document node:

"ibmTools/tool"

Location paths can be made into *absolute location paths* by prepending a forward slash (“/”) to the path. An absolute location path starts with the root node of a document, regardless of the context node. So, the location path used with the first <tool> node as context node ("ibmTools/tool") returns an empty node set (because <ibmTools> is the document element), but the location path "/ibmTools/tool" (forward slash prepended) returns all three <tool> nodes because the search starts at the Document node despite the fact that a <tool> node is the context node.

Location paths can be used with steps. For example, you can specify a descendant of the Document node to find a child node, and then step to its attribute node. The following query returns the rxt attribute nodes on all <tool> nodes:

"descendant::tool/attribute::rxt"

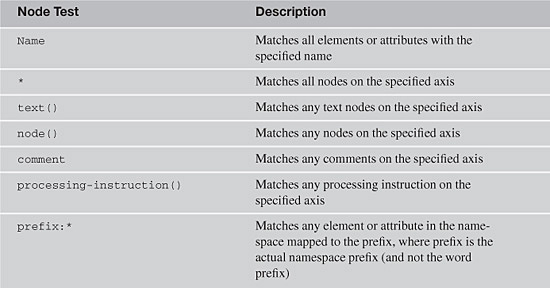
Location paths can also be combined using the pipe operator (“|”), which indicates a logical “or.” So, for example, if you wanted to return a node set that contained all <name> and <version> nodes, you could write the following XPath expression:

"ibmTools/tool/version|ibmTools/tool/name"

**Node Tests**

Node tests are a mechanism for limiting the nodes on an axis that are selected and returned by an XPath query. [Table 4.3](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch04.html#ch04tab03) shows the XPath node tests that are available.

**Table 4.3** XPath Node Tests



So, if you wanted to match all the descendants of the context node, you could write:

"descendant::\*"

If the context node is the Document node, the following test returns all of the text nodes of the document:

"descendant::text()"

However, the following test returns nothing when used in the Document context because the Document node has no direct text children:

"child::text()"

XPath has an abbreviated notation for several commonly used axis and Node test expressions. A selection of the most commonly-used abbreviations follows:

• "Name" = "child::Name"

• "@Name" = "attribute::Name"

• "." = "self::node()"

• ".." = "parent::node()"

• "//" = "descendant-or-self::node()"

**Predicates**

A predicate is a logical XPath expression that is evaluated for each node selected by a location step. If a predicate evaluates to true for a specific node, the node is retained in the returned node set; otherwise, it is omitted. Predicates are enclosed in square brackets.

Sample XPath queries that use predicates include the following (the Document node is the context node):

• "descendant::tool[1]"—returns the first <tool> node

• "descendant::tool[child::name='Rational Performance Tester']"—Returns the <tool> node that has a <name> child with text Rational Performance Tester

• "descendant::\*[@rxt]"—Returns all three descendant nodes with an rxt attribute

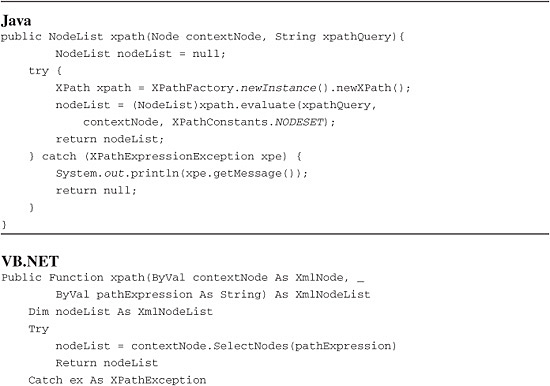
• "//tool[attribute::rxt='RFT']"—Returns all <tool> nodes whose rxt attribute is RFT

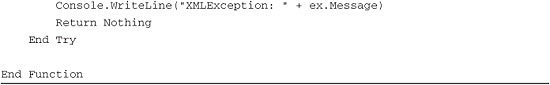
• "//version[text()='8.1.0']"—Returns all <version> nodes whose text is 8.1.0

**Code for XPath Searches**

The code required for XPath searching is relatively straightforward, especially if namespaces are not involved. There are a number of packages that offer XPath searching; however, our illustrations are with the core classes that distribute with the tool languages (javax.xml.xpath.\*, System.Xml). [Listing 4.7](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch04.html#ch04ex07) shows the basic form of the invocation for searches on namespace unaware XMLs.

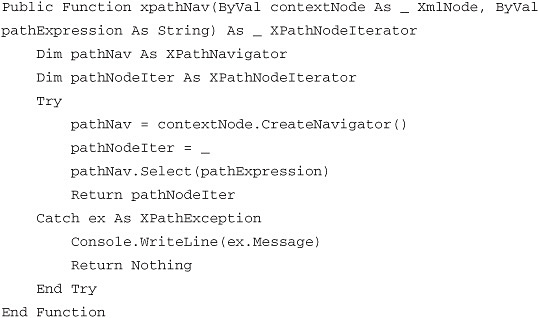
**Listing 4.7** A simple XPath search implementation





In [Listing 4.7](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch04.html#ch04ex07), note that the results of the XPath queries are always returned as list objects, even if only a single node in the result is expected. This is done purely for the sake of simplicity. If you want to write some additional xpath() method implementations that return only a single node instead of a list, you can do it easily—in Java, by passing XPathConstants.NODE to the execute() method, and in VB.NET, by calling the method SelectSingleNode() on the context node in place of SelectNodes().

Visual Studio has another mechanism for handling XPath queries—a set of classes that are optimized for fast query execution. These classes do not load the full functionality of the XML DOM objects, so memory is not wasted on unneeded instructions when the main focus is on querying the XML. The approach starts with the XPathNavigator class, a reference to which is obtained by calling the CreateNavigator() method on a regular DOM object. Following is a simple method that illustrates this coding approach to XPath queries.



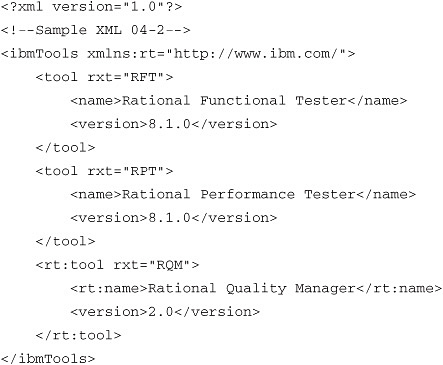
**A Word about Namespaces**

XML uses the concept of a namespace as a mechanism to avoid name collisions between XML documents designed by different organizations employing tags with the same name. A *namespace* is simply a unique identifier (typically based on an organization’s URL) that can be applied to any or all tags or attributes in an XML. A namespace can be declared as a default namespace, applying to all children of the element in which it is declared. A namespace can also be declared as a nondefault namespace with a tag name prefix, and namespace members include only those elements using the prefix with their tag names. Namespaces are declared using special attributes on an element; in the following example, the namespace rt is declared on the root of the sample XML: <ibmTools xmlns:rt="http://www.ibm.com"/>. The namespace declaration is composed of the xmlns designation, the namespace prefix (rt), and the namespace URI (Universal Resource Indicator, “[http://www.ibm.com](http://www.ibm.com/)”). The URI provides uniqueness to the namespace.

To apply the rt namespace to an XML element, rt is used as a colon-separated prefix to the tag name (for example, <rt:tool rt:rxt="RQM">. When an element is included in a namespace, its tag name without the prefix is referred to as the local name (tool is the local name in this example). Either elements or attributes can be namespace members, as in: <tool rt:rxt="RQM">.

All of the XPath examples thus far are *not* namespace-aware. XPath, of course, does support namespace-aware queries. Some of the mechanisms of support have already been mentioned: there is a namespace axis ([Table 4.2](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch04.html#ch04tab02)) and a prefix:\* node test ([Table 4.3](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch04.html#ch04tab03)).

If your XML uses namespaces, your XPath queries will, of course, have to reflect the namespace a given node belongs to. So, for example, if you modify the sample XML to declare the rt namespace in the root tag, and put the final <tool> element and its <name> child in the namespace, you get the following XML, which is used to illustrate some common namespace-aware queries.



The first query to consider uses a simple location path. Using the document element as context, the following query returns the tool node in the rt namespace (i.e., <rt:tool>):

"ibmTools/rt:tool"

An equivalent query that returns the same result can be built with an axis:

"descendant::rt:tool"

The following related query (again, with the Document element as the context node) returns all of the nodes in the rt namespace:

"descendant::rt:\*"

This query returns two nodes: <rt:tool> and <rt:name>.

Finally, there is an XPath function called local-name(), so that if you wanted to write a query to return nodes that are in a namespace, but without using the namespace prefix, you could write:

"descendant::\*[local-name()='tool']"

This query returns all nodes on the descendant axis whose local name equals 'tool'. In our case, three nodes are returned: two <tool> nodes and one <rt:tool> node.

Note that if you request the following, the XPath engine returns to you the rxt attribute rxt=RQM:

"ibmTools/rt:tool/attribute::rxt"

However, if you request the following, then nothing is returned, because the rt namespace on the <tool> tag is not automatically inherited by the tag’s attribute.:

"ibmTools/rt:tool/attribute::rt:rxt"

**Code for Namespace-Aware XPath Searches**

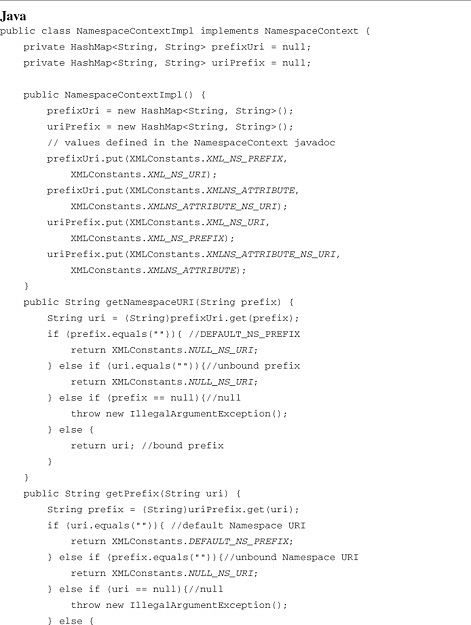
For namespace-aware XMLs in Java, there are some complexities that must be dealt with. First, you must make sure that your XML documents are loaded using a DocumentBuilderFactory that is has namespace-awareness ‘turned on’. This appears in the loadDoc() method shown in [Listing 4.1](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch04.html#ch04ex01) in the following line:

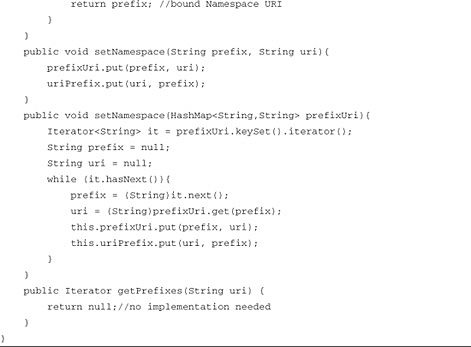
docBuilderFactory.setNamespaceAware(true);

It is important to check this line of code because if it is omitted, the default is false, and the parser ignores any namespaces you use. Setting the value to true has no effect for namespace-*un*aware XML shown earlier, so leaving setNamespaceAware(true) should be safe.

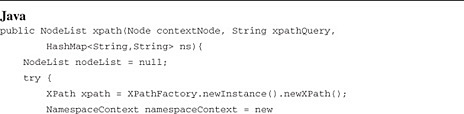
Another important note in setting up namespace-aware XPath searching is that Java 5 and Java 6 (also known as 1.5 and 1.6) require you to supply a helper class for namespace-aware XPath queries. This class must implement the interface NamespaceContext, which enables the XPath engine to look up prefixes from URIs and URIs from prefixes. Without a NamespaceContext implementation, the exception XPathStylesheetDOM3Exception is thrown when you try to execute your query. (In an upcoming Java release, there should be a default NamespaceContext implementation, so eventually, you will not have to supply this class). After you have built an implementation for NamespaceContext, you can write code to execute your namespace-aware XPath queries. [Listing 4.8](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch04.html#ch04ex08) shows code for a simple version of the NamespaceContext-implementing class (it persists only one namespace per XML), and [Listing 4.9](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch04.html#ch04ex09) shows calling code for a namespace-aware XPath query.

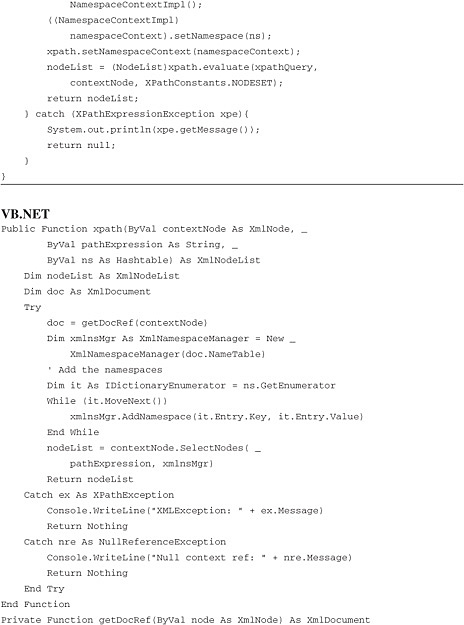
**Listing 4.8** NamespaceContextImpl: implementation of interface NamespaceContext

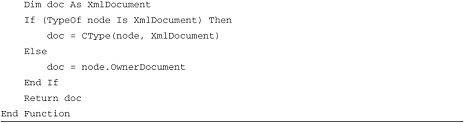




**Listing 4.9** Sample calling code for a namespace-aware XPath query







As noted earlier, [Listing 4.9](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch04.html#ch04ex09) shows calling code for a namespace-aware XPath query. The signature of this method differs from the one shown in [Listing 4.7](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch04.html#ch04ex07) because namespace-aware queries require the namespaces in use to be explicitly set. This is done using a HashMap in Java or a Hashtable in VB.NET, so that the calling code (in Java) for a query on our sample XML looks like the following:

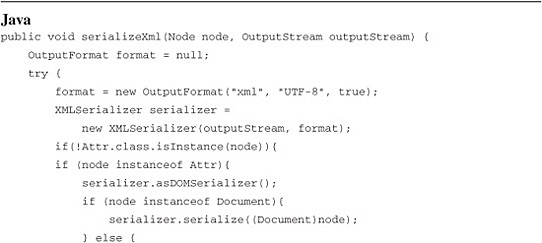
image

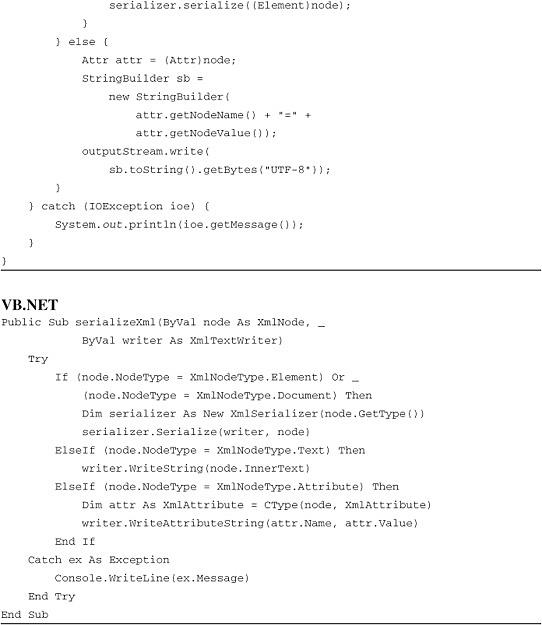
This discussion should give you a sense of the power that XPath has for test automation. This is not a complete discussion of XPath; so if you are inclined to use it, get more resources! Broad queries can be made that return large node sets, and highly specific queries can be made that return just a small number of desired nodes. The tremendous flexibility of XPath can make the job of XML data validation much quicker, with code that is more readable and maintainable.

**Serializing XML to a String or to a File**

One of the more practical and useful things you need to do is to take an XML DOM involved in a test and change it into a form in which it can either be saved or printed. This is useful for debugging a test or for documenting defects. In other words, you need to serialize your DOM either to a file on disk or to a String. This can be done easily with the XMLSerializer class (named the same in Java and VB.NET). In Java, you use an OutputStream to achieve serialization. (Writers can be used, but they do not let you handle internationalized characters.) In VB.NET, StringWriter or StreamWriter can be used. A core method that takes a generic OutputStream argument (Java) or XmlTextWriter argument (VB.NET) is shown in [Listing 4.10](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch04.html#ch04ex10). Note that in both cases, you must check whether the node you are serializing is an Attribute node because these require special handling.

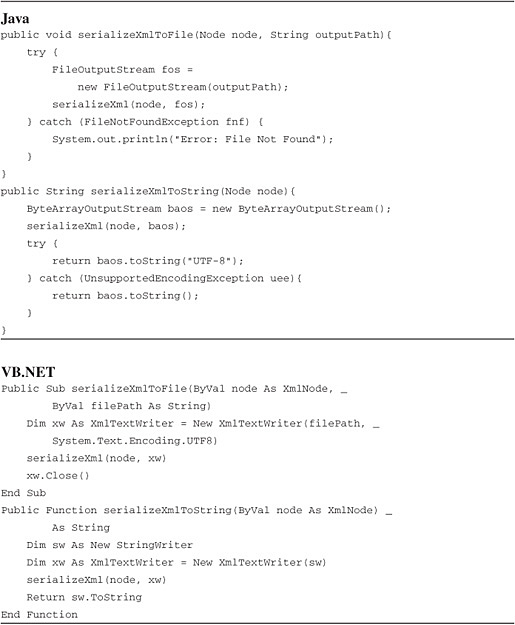
**Listing 4.10** Serializing XML nodes via a Writer





You can write some wrapper methods for serializeXml() that set up the kind of object you are going to use and handle the output. Note that because our output handles the actual output, there is no return value from serializeXml(). [Listing 4.11](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch04.html#ch04ex11) shows method calls for string and file output.

**Listing 4.11** Serializing XML to a file or a string



As a side note, for convenience, you can easily override serializeXmlToFile() to take a File argument instead of a String argument.

**Processing XML with SAX in Rational Functional Tester**

As previously mentioned, SAX parsing requires a more elaborate setup than DOM parsing of XML, but it becomes attractive when you are dealing with large XMLs. What constitutes a large XML depends on your box’s hardware spec, of course, but if you are using a box of relatively recent vintage, large is probably somewhere in the multiple megabyte range, and small is in the range of hundreds of kilobytes or smaller. Because setting up DOM code is simpler, it is probably wisest to try that first on your largest XMLs and see if the performance outcome is acceptable. If you need something faster, SAX is recommended.

Before delving into SAX, it is important to note that SAX is chiefly a creature of Java. While there is an open-source port of the SAX API specification to C# (see <http://sourceforge.net/projects/saxdotnet>), the .NET framework itself doesn’t contain an implementation of SAX. Rather, it comes with a parser implemented in the System.Xml.XmlReader class (a pull parser instead of SAX’s push parser). For this reason, this chapter does not examine SAX in .NET, but it does include a short example using XmlReader.

**Implementing with SAX**

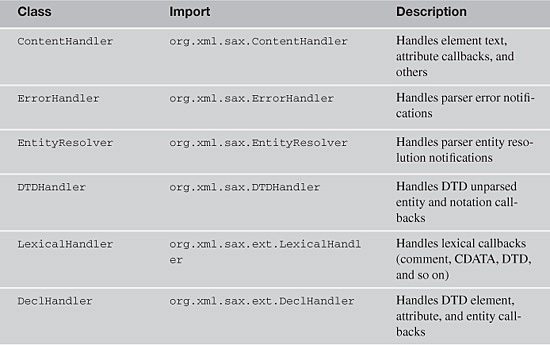
To use SAX, you need to build custom classes that contain event handlers for the various SAX events. The SAX parser calls these handlers as it scans through an XML document, firing off events in response to document contents.

Note

You can keep up with new developments in SAX at the SAX website [www.saxproject.org/](http://www.saxproject.org/), if that is where your interests or needs take you.

There are six SAX handler classes, four of which are considered standard classes, and two of which are extensions to SAX; [Table 4.4](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch04.html#ch04tab04) lists the SAX handler classes.

**Table 4.4** SAX 2.0 Handler Classes



\*SAX also offers a DefaultHandler (org.xml.sax.helpers.DefaultHandler), which contains implementations of the first four interfaces in this table. You can extend the DefaultHandler in a custom class and override just the methods you need.

To handle standard XML-related testing tasks, you need to implement only SAX event-handling methods that help to pull data out of target XMLs. This can be done with just two classes: one that implements both the ContentHandler and LexicalHandler interfaces and one that implements the ErrorHandler interface. For the methods not needed for data-related testing tasks, you can provide stubs. (You could also implement with a DefaultHandler; the approach shown here is selected because it is simpler.)

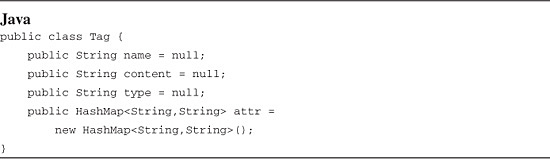
Typically, actual transactional data in XML is found either in element text (as character text or as a CDATA section) or in element attributes. It is also possible that data that might be significant for testing could appear in a processing instruction or in an XML comment. Data from all of these different element types can be captured with five event handlers: three are specified by the ContentHandler interface [element name, text, attributes, and processing instructions are handled with ContentHandler methods startElement(), characters(), and processingInstruction()], and the other two by the LexicalHandler with comments and CDATA sections being handled by LexicalHandler events endCDATA() and comment().

The class RFTSAXContentHandler will implement the ContentHandler and LexicalHandler interfaces and will provide implementations of the five data-capture event handlers. The five events that will handle the capture of XML data are startElement(), characters(), processingInstruction(), endCDATA(), and comment().

The ErrorHandler-implementing class is RFTSAXErrorHandler. The ErrorHandler interface requires three methods: warning(), error(), and fatalError().

Before examining the actual RFTSAXContentHandler code, there is a small design consideration: how to handle the XML data when the SAX parser fires its events. The data needs to be packaged as the SAX engine parses it on the fly. To do this, we’ll use a simple class, in which all of the data for any specific XML element can be held. The data for each XML element is loaded into an object of this type on parsing, and the full data set from a parsing run is managed with an ArrayList of these objects. A simple class that meets these requirements is the Tag class shown in [Listing 4.12](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch04.html#ch04ex12).

**Listing 4.12** The Tag class



The Tag class contains a set of fields sufficient to describe the data from a single XML tag. The name field corresponds to the element name in the case of a tag, the target in the case of a processing instruction or “comment” for comment tags. The content field holds whatever content is associated with the tag, either text, processing instruction data, or an XML comment. The type field keeps an indication of the data type (use the standard DOM type notations, #text, #cdata-section, and #comment, and add #processing-instruction). Finally, attr is a HashMap to store any attributes found.

To manage Tag objects, you need some custom members of RFTSAXContentHandler. These are shown in [Listing 4.13](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch04.html#ch04ex13).

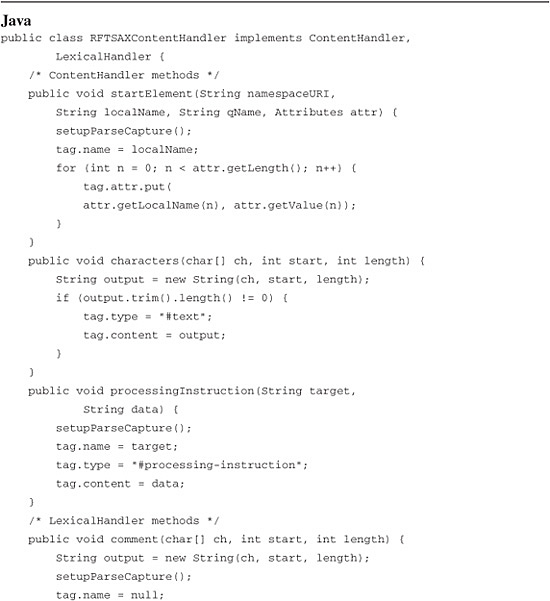
**Listing 4.13** Custom members of RFTSAXContentHandler for management of Tags

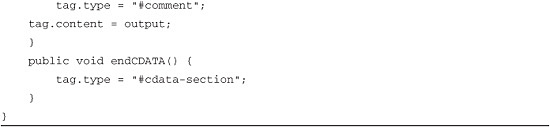


The private field tag that you’ll see in RFTSAXContentHandler corresponds to the current Tag instance holding the data of the XML element currently being parsed; the tags ArrayList is the container in which all the Tag instances for a parsing run are loaded for return to the calling class. The setupParseCapture() method creates a new Tag instance and loads it into the tags ArrayList. This method has to appear in any event handler that fires at the beginning of a tag reading. Finally, the class needs a method to return the tags ArrayList to the calling class: getTagData().

This example does not explicitly deal with namespace issues, but as you can see from the startElement() signature, the SAX parser includes namespace information in its callbacks. If you need to work with namespaces, you can handle them with some fairly straightforward changes to the sample code shown in this discussion. With these custom bits in hand, you can look at the key event handlers in the RFTSAXContentHandler class, which is shown in [Listing 4.14](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch04.html#ch04ex14).

**Listing 4.14** Event handlers in RFTSAXContentHandler (implements ContentHandler and LexicalHandler)





Some comments about these methods are necessary. First, it is interesting to note that in startElement(), the tag *text content* is not reported; only the tag name and attributes are. Text content is handled by the characters() method.

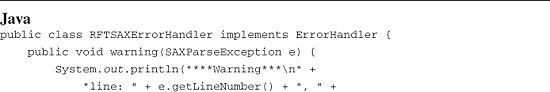
The characters() callback has some interesting behavior. First, the content data is passed back to the event handler as a char array, along with start and length parameters. The parser is required only to ensure that the data for the event is *within* the reported length. The char array might legally be longer than the reported length, and if you read past the length indicated, you might get “gibberish” data back. Furthermore, parsers have the option of passing back element content in either one call to characters() or in multiple calls. Calls can also include whitespace. You should become familiar with your parser implementation on these points, so that you can fashion your methods to interact properly with the parser behavior.

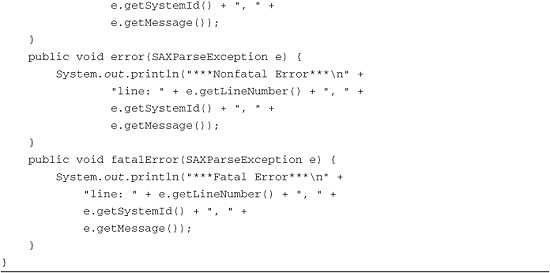
The final ContentHandler method is processingInstruction(). This callback method receives a processing instruction target and data. The target is defined as the first term in the processing instruction; the rest is data, regardless of the punctuation used.

There are two LexicalHandler methods: one for XML comments and the other for CDATA notifications. The comment() method works much like the characters() method. Finally, if you parse a CDATA-containing tag, the tag content is captured by characters(), and you can use endCDATA() to change the tag type from #text to #cdata-section. Note that neither endCDATA() nor startCDATA() handle the actual content from the CDATA section.

Finally, note that the ContentHandler and LexicalHandler interfaces require several more methods each than what is shown in [Listing 4.14](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch04.html#ch04ex14). These are stubbed in the full RFTSAXContentHandler class; they are omitted from [Listing 4.14](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch04.html#ch04ex14) for clarity. This ends the work implementing ContentHandler and LexicalHandler callbacks. All you need to do next is implement an ErrorHandler, and you are ready to parse. [Listing 4.15](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch04.html#ch04ex15) shows the code for RFTSAXErrorHandler.

**Listing 4.15** Class RFTSAXErrorHandler (implements ErrorHandler)





The SAX ErrorHandler interface specifies three callbacks for three levels of errors: a warning, a nonfatal error, and a fatal error. Each of these events receives a SAXParserException, which returns the line number where the error occurred, the document URI, and an error message. All three ErrorHandler methods can be implemented with identical code, as shown in [Listing 4.15](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch04.html#ch04ex15).

With the ErrorHandler implemented, you are ready to write some parsing code. To start a parser, four tasks need to be accomplished:

• An XMLReader must be created with the createXMLreader() method.

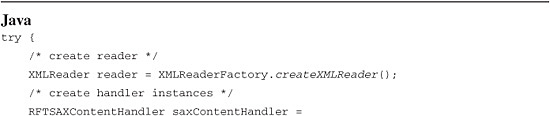
• Instances of the handler classes must be created.

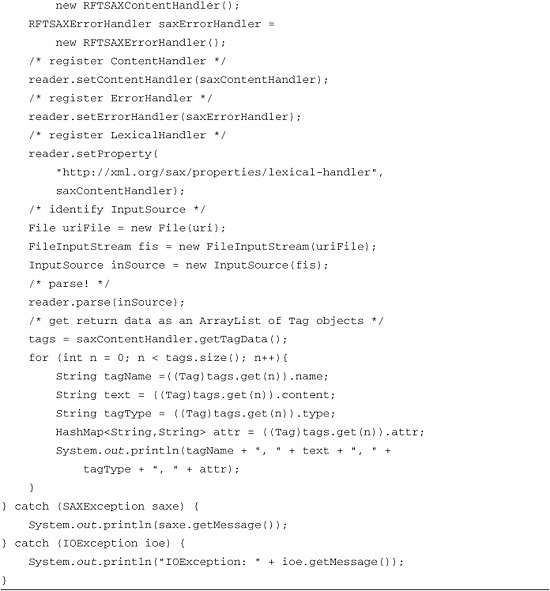
• Handlers must be registered.

• An InputSource must be identified.

When these tasks are finished, you are ready to parse. You can place all three classes designed in this exercise in the same file as your TestScript class, or you can place them in separate files in the same package in the classpath. A basic calling script for parsing with the required exception handling is shown in [Listing 4.16](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch04.html#ch04ex16).

**Listing 4.16** An RFT script for SAX parsing



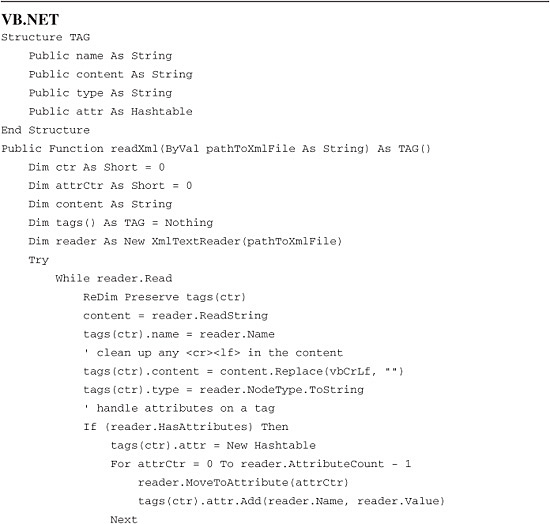


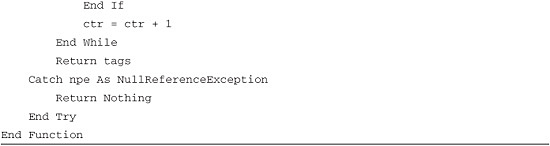
As discussed previously, there is more going on with SAX than what is described in this introduction. If you’re interested in pursuing a broader understanding of SAX, there are excellent publications available (some of them are found in the literature section at the end of this chapter) and in resources on the Web.

**Implementing with .NET’s XmlReader**

As noted previously, SAX emerged from the Java world and is not part of the .NET world, at least as far as the standard .NET libraries are concerned. Instead, .NET provides a conceptually related but different implementation for fast parsing in the XmlReader class. Setting up a basic parsing routine with XmlReader is simpler than with SAX—custom classes do not need to be supplied. Instead, all you need is to instantiate an XmlReader, and use a few of its methods to trap XML tag values as the parser scans through them. You can capture tag values in a VB.NET structure (an extension of the concept of a C-type struct) and return tag data as an array of structures. [Listing 4.17](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch04.html#ch04ex17) shows the structure and a simple parsing method.

**Listing 4.17** A simple XmlReader implementation





**Summary**

As you can see, XML manipulation for testing purposes can range across the entire XML universe; almost every aspect of XML can come into play when it comes to testing. Using the techniques described in this chapter, you can compare test XMLs against baseline XMLs or against data from Rational Functional Tester datapools, flat files, SQL calls, or practically any data source or repository to which you have access. With XML testing using Rational Functional Tester, you are limited by little beyond your imagination.

**For Further Information**

Harold, Rusty Eliotte and W. Scott Means. *XML in a Nutshell*, Third Edition. Sebastopol, CA: O’Reilly Media, Inc., 2004.

Kay, Michael. *XPath 2.0 Programmer’s Reference*. Indianapolis, IN: Wrox, 2004.

Bornstein, Niel M. .*NET and XML*. Sebastopol, CA: O’Reilly Media, Inc., 2003.

Wahlin, Dan. *XML for ASP.NET Developers*. Indianapolis, IN: Sams Publishing, 2001.

Harold, Eliotte Rusty. *Processing XML with Java*. Boston, MA: Addison-Wesley Professional, 2002.

[Prev](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch03.html)

[Chapter 3. General Script Enhancements](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch03.html)

[Next](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch05.html)

[Chapter 5. Managing Script Data](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch05.html)