Item 3 Stay with XML 1.0

Everything you need to know about XML 1.1 can be summed up in two rules.

1. Don't use it.
2. (For experts only) If you speak Mongolian, Yi, Cambodian, Amharic, Dhivehi, Burmese, or a very few other languages and you want to write your markup (not your text but your markup) in these languages, you can set the `version` attribute of the XML declaration to 1.1. Otherwise, refer to rule 1.

XML 1.1 does several things, one of them marginally useful to a few developers, the rest actively harmful.

- It expands the set of characters allowed as name characters.
- The C0 control characters (except for NUL) such as form feed, vertical tab, BEL, and DC1 through DC4 are now allowed in XML text provided they are escaped as character references.
- The C1 control characters (except for NEL) must now be escaped as character references.
- NEL can be used in XML documents but is resolved to a line feed on parsing.
- Parsers may (but do not have to) tell client applications that Unicode data was not normalized.
- Namespace prefixes can be undeclared.

Let's look at these changes in more detail.

New Characters in XML Names

XML 1.1 expands the set of characters allowed in XML names (that is, element names, attribute names, entity names, ID-type attribute values, and so forth) to allow characters that were not defined in Unicode 2.0, the version that was extant when XML 1.0 was first defined. Unicode 2.0 is fully adequate to cover the needs of markup in English, French, German, Russian, Chinese, Japanese, Spanish, Danish, Dutch, Arabic, Turkish, Hebrew, Farsi, Thai, Hindi, and most other languages you're likely to be familiar with as well as several thousand you aren't. However, Unicode 2.0 did miss a few important living languages including Mongolian, Yi, Cambodian, Amharic, Dhivehi, and Burmese, so if you want to write your markup in these languages, XML 1.1 is worthwhile.

However, note that this is relevant only if we're talking about *markup,* particularly element and attribute names. It is not necessary to use XML 1.1 to write XML data, particularly element content and attribute values, in these languages. For example, here's the beginning of an Amharic translation of the Book of Matthew written in XML 1.0.

```
<?xml version="1.0" encoding="UTF-8">
<book>
<title>የማቴዎስ ወንጌል</title>
<chapter number="፩">
<title>የኢየሱስ የትው ልድ ሐረግ</title>
<verse number="፩">
 የዳዊት ልጅ፥ የአከርሃም ልጅ የሁ ነው የኢየሱስ ክርስቶስ የትው ልድ ሐረግ የሚከተለው ነው፤
```

```
</verse>
<verse number="፪">
 ኣከርሃም ይስሐቅን ወለደ፤
 ይስሐቅ ያዕቆብን ወለደ፤
 ያዕቆብ ይሁዳንና ወንድሞቹን ወለደ፤
</verse>
</chapter>
</book>
```

Here the element and attribute names are in English although the content and attribute values are in Amharic. On the other hand, if we were to write the element and attribute names in Amharic, we would need to use XML 1.1.

```
<?xml version="1.1" encoding="UTF-8">
```

```
<መጽሐፉ>
<ኣርእስት>የማቴዎስ ወንጌል</ኣርእስት>
<ምዕራፉ ቁጥር="፩">
<ኣርእስት>የኢየሱስ የትው ልድ ሐረግ</ኣርእስት>
<ቤት ቁጥር="፩">
የዳዊት ልጅ፥ የኣከርሃም ልጅ የሁ ነው የኢየሱስ ከርስቶስ የትው ልድ ሐረግ የሚከተለው ነው፤
</ቤት>
<ቤት ቁጥር="፪">
 ኣከርሃም ይስሐቅን ወለደ፤
 ይስሐቅ ያዕቆብን ወለደ፤
 ያዕቆብ ይሁዳንና ወንድሞቹን ወለደ፤
</ቤት>
</ምዕራፉ>
</መጽሐፉ>
```

This is plausible. A native Amharic speaker might well want to write markup like this. However, the loosening of XML's name character rules have effects far beyond the few extra languages they're intended to enable. Whereas XML 1.0 is conservative (everything not permitted is forbidden),

XML 1.1 is liberal (everything not forbidden is permitted). XML 1.0 lists the characters you can use in names. XML 1.1 lists the characters you can't use in names. Characters XML 1.1 allows in names include:

- Symbols like the copyright sign (©)
- Mathematical operators such as ±
- Superscript 7 (^{7})
- The musical symbol for a six-string fretboard
- The zero-width space
- Private-use characters
- Several hundred thousand characters that aren't even defined in Unicode and probably never will be

XML 1.1's lax name character rules have the potential to make documents much more opaque and obfuscated.

C0 Control Characters

The first 32 Unicode characters with code points from 0 to 31 are known as the C0 controls. They were originally defined in ASCII to control teletypes and other monospace dumb terminals. Aside from the tab, carriage return, and line feed they have no obvious meaning in text. Since XML is text, it does not include binary characters such as NULL (#x00), BEL (#x07), DC1 (#x11) through DC4 (#x14), and so forth. These noncharacters are historical relics. XML 1.0 does not allow them. This is a good thing. Although dumb terminals and binary-hostile gateways are far less common today than they were twenty years ago, they are still used, and passing these characters through equipment that expects to see plain text can have nasty consequences, including disabling the screen. (One common problem that still occurs is accidentally paging a binary file on a console. This is generally quite ugly and often disables the console.)

A few of these characters occasionally do appear in non-XML text data. For example, the form feed (#x0C) is sometimes used to indicate a page break. Thus moving data from a non-XML system such as a BLOB or CLOB field in a database into an XML document can unexpectedly cause malformedness errors. Text may need to be cleaned before it can be added to an XML document. However, the far more common problem is that a document's encoding is misidentified, for example, defaulted as UTF-8 when it's really UTF-16 or ISO-8859-1. In this case, the parser will notice unexpected nulls and throw a well-formedness error.

XML 1.1 fortunately still does not allow raw binary data in an XML document. However, it does allow you to use character references to escape the C0 controls such as form feed and BEL. The parser will resolve them into the actual characters before reporting the data to the client application. You simply can't include them directly. For example, the following document uses form feeds to separate pages.

```
<?xml version="1.1">
<book>
  <title>Nursery Rhymes</title>
  <rhyme>
    <verse>Mary, Mary quite contrary</verse>
    <verse>How does your garden grow?</verse>
  </rhyme>
  &#x0C;
  <rhyme>
    <verse>Little Miss Muffet sat on a tuffet</verse>
    <verse>Eating her curds and whey</verse>
  </rhyme>
  &#x0C;
  <rhyme>
    <verse>Old King Cole was a merry old soul</verse>
    <verse>And a merry old soul was he</verse>
  </rhyme>
</book>
```

However, this style of page break died out with the line printer. Modern systems use stylesheets or explicit markup to indicate page boundaries. For example, you might place each separate page inside a `page` element or add a `pagebreak` element where you wanted the break to occur, as shown below.

```
<?xml version="1.1">
<book>
  <title>Nursery Rhymes</title>
  <rhyme>
    <verse>Mary, Mary quite contrary</verse>
    <verse>How does your garden grow?</verse>
  </rhyme>
  <pagebreak/>
  <rhyme>
```

```
    <verse>Little Miss Muffet sat on a tuffet</verse>
    <verse>Eating her curds and whey</verse>
  </rhyme>
  <pagebreak/>
  <rhyme>
    <verse>Old King Cole was a merry old soul</verse>
    <verse>And a merry old soul was he</verse>
  </rhyme>
</book>
```

Better yet, you might not change the markup at all, just write a stylesheet that assigns each rhyme to a separate page. Any of these options would be superior to using form feeds. Most uses of the other C0 controls are equally obsolete.

There is one exception. You still cannot embed a null in an XML document, not even with a character reference. Allowing this would have caused massive problems for C, C++, and other languages that use null-terminated strings. The null is still forbidden, even with character escaping, which means it's still not possible to directly embed binary data in XML. You have to encode it using Base64 or some similar format first. (See Item 19.)

C1 Control Characters

There is a less common block of C1 control characters between 128 (#x80) and 159 (#x9F). These include start of string, end of string, cancel character, privacy message, and a few other equally obscure characters. For the most part these are even less useful and less appropriate for XML documents than the C0 control characters. However, they were allowed in XML 1.0 mostly by mistake. XML 1.1 rectifies this error (with one notable exception, which I'll address shortly) by requiring that these control characters be escaped with character references as well. For example, you can no longer include a "break permitted here" character in element content or attribute values. You have to write it as `&#x82;` instead.

This actually does have one salutary effect. There are a lot of documents in the world that are labeled as ISO-8859-1 but actually use the nonstandard Microsoft Cp1252 character set instead. Cp1252 does not include the C1 controls. Instead it uses this space for extra graphic characters such as €, Œ, and ™. This causes significant interoperability problems when

moving documents between Windows and non-Windows systems, and these problems are not always easy to detect.

By making escaping of the C1 controls mandatory, such mislabeled documents will now be obvious to parsers. Any document that contains an unescaped C1 character labeled as ISO-8859-1 is malformed. Documents that correctly identify themselves as Cp1252 are still allowed.

The downside to this improvement is that there is now a class of XML documents that is well-formed XML 1.0 but not well-formed XML 1.1. XML 1.1 is not a superset of XML 1.0. It is neither forward nor backward compatible.

NEL Used as a Line Break

The fourth change XML 1.1 makes is of no use to anyone and should never have been adopted. XML 1.1 allows the Unicode next line character (#x85, NEL) to be used anywhere a carriage return, line feed, or carriage return–line feed pair is used in XML 1.0 documents. Note that a NEL doesn't mean anything different than a carriage return or line feed. It's just one more way of adding extra white space. However, it is incompatible not only with the installed base of XML software but also with all the various text editors on UNIX, Windows, Mac, OS/2, and almost every other non-IBM platform on Earth. For instance, you can't open an XML 1.1 document that uses NELs in emacs, vi, BBEdit, UltraEdit, jEdit, or most other text editors and expect it to put the line breaks in the right places. Figure 3–1 shows what happens when you load a NEL-delimited file into emacs. Most other editors have equal or bigger problems, especially on large documents.

If so many people and platforms have such problems with NEL, why has it been added to XML 1.1? The problem is that there's a certain huge monopolist of a computer company that doesn't want to use the same standard everyone else in the industry uses. And—surprise, surprise—its name isn't Microsoft. No, this time the villain is IBM. Certain IBM mainframe software, particularly console-based text editors like XEdit and OS/390 C compilers, do not use the same two line-ending characters (carriage return and line feed) that everybody else on the planet has been using for at least the last twenty years. Instead those text editors use character #x85, NEL.

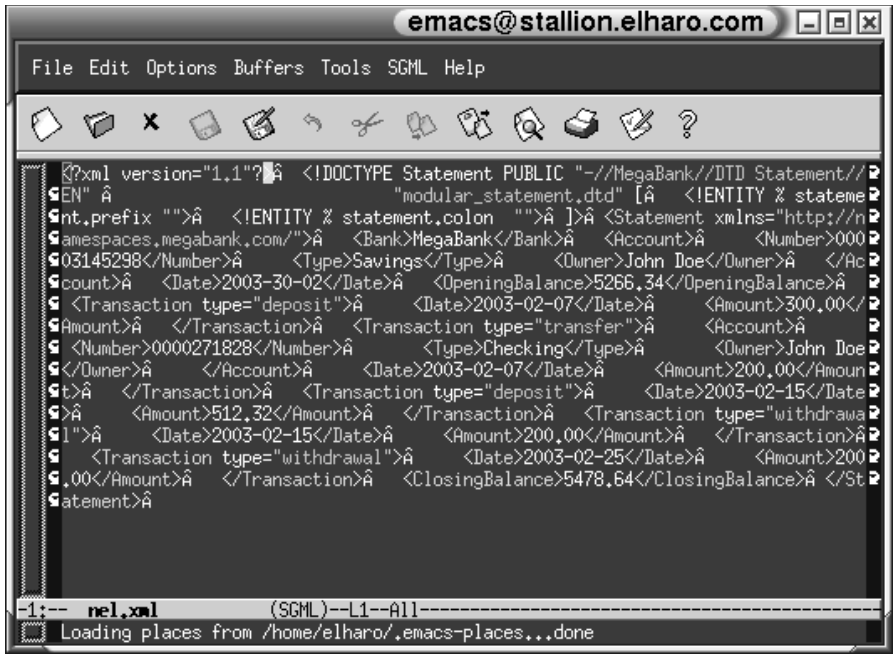

Figure 3–1 | Loading a NEL-Delimited File into a Non-IBM Text Editor

If you're one of those few developers writing XML by hand with a plain console editor on an IBM mainframe, you should upgrade your editor to support the line-ending conventions the rest of the world has standardized on. If you're writing C code to generate XML documents on a mainframe, you just need to use \x0A instead of \n to represent the line end. (Java does not have this problem.) If you're reading XML documents, the parser should convert the line endings for you. There's no need to use XML 1.1.

Unicode Normalization

For reasons of compatibility with legacy character sets such as ISO-8859-1 (as well as occasional mistakes) Unicode sometimes provides multiple representations of the same character. For example, the e with accent acute (é) can be represented as either the single character #xE9 or with the two characters #x65 (e) followed by #x301 (combining accent acute). XML 1.1 suggests that all generators of XML text should normalize such

alternatives into a canonical form. In this case, you should use the single character rather than the double character.

However, both forms are still accepted. Neither is malformed. Furthermore, parsers are explicitly prohibited from doing the normalization for the client program. They may merely report a nonfatal error if the XML is found to be unnormalized. In fact, this is nothing that parsers couldn't have done with XML 1.0, except that it didn't occur to anyone to do it. Normalization is more of a strongly recommended best practice than an actual change in the language.

Undeclaring Namespace Prefixes

There's one other new feature that's effectively part of XML 1.1: namespaces 1.1, which adds the ability to undeclare namespace prefix mappings. For example, consider the following API element.

```
<?xml version="1.0" encoding="UTF-8">
<API xmlns:public="http://www.example.com"
      xmlns:private="http://www.example.org" >
  <title>Geometry</title>
  <cpp xmlns:public="" xmlns:private="">
    class CRectangle {
      int x, y;
      public:void set_values (int,int);
      private:int area (void); }
  </cpp>
</API>
```

A system that was looking for qualified names in element content might accidentally confuse the `public:void` and `private:int` in the `cpp` element with qualified names instead of just part of C++ syntax (albeit ugly C++ syntax that no good programmer would write). Undeclaring the public and private prefixes allows them to stand out for what they actually are, just plain unadorned text.

In practice, however, very little code looks for qualified names in element content. Some code does look for these things in attribute values, but in those cases it's normally clear whether or not a given attribute can contain qualified names. Indeed this example is so forced precisely because prefix undeclaration is very rarely needed in practice and never needed if you're only using prefixes on element and attribute names.

That's it. There is nothing else new in XML 1.1. It doesn't move namespaces or schemas into the core. It doesn't correct admitted mistakes in the design of XML such as attribute value normalization. It doesn't simplify XML by removing rarely used features like unparsed entities and notations. It doesn't even clear up the confusion about what parsers should and should not report. All it does is change the list of name and white space characters. This very limited benefit comes at an extremely high cost. There is a huge installed base of XML 1.0–aware parsers, browsers, databases, viewers, editors, and other tools that don't work with XML 1.1. They will report well-formedness errors when presented with an XML 1.1 document.

The disadvantages of XML 1.1 (including the cost in both time and money of upgrading all your software to support it) are just too great for the extremely limited benefits it provides most developers. If you're more comfortable working in Mongolian, Yi, Cambodian, Amharic, Dhivehi, or Burmese and you only need to exchange data with other speakers of one of these languages (for instance, you're developing a system exclusively for a local Amharic-language newspaper in Addis Ababa where everybody speaks Amharic), you can set the `version` attribute of the XML declaration to 1.1. Everyone else should stick to XML 1.0.

Item 29 Always Use a Parser

XML documents are just too rich in syntax sugar to be processed by anything short of a full-blown XML parser. I've seen many hackish systems held together by string and bailing wire based on regular expressions, grep, sed, raw stream processing, and other tools. These are extremely brittle and rarely able to handle the full panoply of documents they encounter. Problems include:

- Detecting the encoding, including handling multibyte character sets
- Comments that contain tags

- Processing instructions that contain tags
- CDATA sections
- Unexpected placement of spaces and line breaks within tags
- Default attribute values applied from the internal DTD subset
- Character references like ` ` and ` `
- Predefined entity references such as `&` and `>`
- Malformedness errors
- Empty-element tags
- Internal DTD subsets that define default attribute values

These all have little to nothing to do with the semantic content or structure of a document. They have a great deal to do with syntax. A parser knows how to resolve all of these into the actual intended content. Very few other processes do. In fact, if you were to write your own program that handled all of this correctly, you'd be very close to inventing your own XML parser. The fact is, nothing short of a real XML parser can truly handle XML. Any program you write to process XML documents needs to sit on top of a real XML parser.

There are two main reasons developers invent their own systems based on regular expressions or other tools instead of using an XML parser.

1. They're simply not familiar with parsers and their APIs.
2. They find parsing to be too slow.

If it's simply a question of developer familiarity, the solution is simple. Learn to use SAX, DOM, JDOM, or some other API that sits on top of a parser. Numerous books can help you, including my own *Processing XML with Java* (Boston, MA: Addison-Wesley, 2002).[1]

The question of performance is more fundamental. However, fortunately it's often a canard. Before resorting to brittle non-XML tools for processing data, measure the real speed of the parser-based equivalent. Often parsing is not the bottleneck. Even if it is, the parser-based program may still be fast enough for your needs. If it isn't, you can often improve performance by moving to a different parser. For instance, Piccolo is often noticeably faster than Xerces, though it's not quite as feature rich. The slowdown may be the parser's fault but not the API's. A different parser with the same API may well do better. If it is the API's fault, you may be

1. See http://www.cafeconleche.org/books/xmljava/ for more information.

able to switch to a different API that performs better on your class of documents. (Items 32 and 33 discuss which APIs are appropriate for which tasks.) Finally, you may be able to live without some optional features like external entity resolution and validation that increase the cost of parsing.

However, let's assume that it is indeed the parser's fault. You're using the fastest API and parser available, and you still can't get the performance you want. Is it then acceptable to write a quick and dirty program that saves time by skipping a lot of mandated well-formedness checks and not processing all the syntax sugar? Is it acceptable to write your own mini-parser that properly handles only a subset of XML? I think the answer is no, it is not acceptable. I tend to side with Bertrand Meyer here. Although not specifically addressing XML, his more general point is correct:

> Necessary as tradeoffs between quality factors may be, one factor stands out from the rest: correctness. There is never any justification for compromising correctness for the sake of other concerns such as efficiency. If the software does not perform its function, the rest is useless.[2]

Developers think they can get away with compromising correctness because they assume they know the input format. They know the documents will always be well formed. They know all the element names in advance. They know the documents don't use CDATA sections, document type declarations, or processing instructions. Sometimes, as in SOAP, this is even required by the specification.

Nonetheless, relying on such assumptions is dangerous. In a heterogeneous, distributed, network environment, it's insane. Sooner or later (and more likely sooner) these assumptions will be violated. SOAP messages are sent with processing instructions, the specification not withstanding. Authors do use character and entity references even when they're told not to. Programmers put in document type declarations for testing and then forget to take them out in production. An upgraded library may begin inserting character and entity references whereas before it used literal characters. Any syntax that can be used will be used, and programs need to be ready for this.

Often developers object that they're only using the XML documents internally, on their intranets. These are never passed through the firewall.

---

2. Meyer, Bertrand. *Object-Oriented Software Construction,* 2nd ed. Upper Saddle River, NJ: Prentice Hall, 1997, p. 15.

Thus they have absolute confidence that the documents will always adhere to the constraints their homegrown systems require. These developers have been fortunate enough never to work with Wally or have a pointy-haired boss, but sooner or later we all have to deal with Wally. Assume nothing! Verify everything, even if you're only on an intranet. Sooner or later somebody or something is going to violate your assumptions.

At the absolute extreme, documents are passed between two well-tested and debugged computer processes on the same computer that never talk to anybody else. The output of one process is tied very closely to the input of the other. No human ever intervenes and the code is never changed; or if it is changed, it's only changed in sync with the other system. In this case, it seems perfectly reasonable to make additional assumptions about the format of the data being read. For instance, if you know the sending process never generates comments, you don't need to write the code to handle them. Indeed, if there were such processes in the real world, this might be true. However, in practice nothing is ever so clean. It may not happen today, it may not happen next week, it may not happen before you jump ship to a company with fewer pointy-haired bosses, but sooner or later the sending process is going to change the documents it sends. Perhaps this will happen because the new programmer who took your place is modifying the system but managed to misplace all the detailed documentation you left behind. (And if you aren't the sort of programmer who leaves behind documentation, they have an even bigger problem.) It may happen after a library is upgraded, and the new version uses entity references instead of character references or just puts in a comment identifying itself as the generator of the XML document. It may even happen because some programmer is using telnet to manually insert documents into the system to figure out what it does. Do you want to tell your CIO that because your program didn't use an XML parser, it missed a well-formedness error in the input data and consequently the database running all the stores in the tri-state area was corrupted and crashed at 1:22 P.M. on Christmas Eve?

Hopefully by now you're convinced that you just can't do better than a real XML parser. But what should you do if your systems are still too slow? I suppose you could always throw hardware and memory at the problem. Sometimes that's enough. However, you may reach a point where you have to admit that XML is not the right approach for your system. If you really do have an unfixable performance problem, you might need to consider using a simpler format that requires less work from the

parser, such as tab-delimited text. This loses many of the well-known benefits of XML, but if you're considering throwing away XML syntax and well-formedness rules to gain speed, you've lost those already. What you're processing may look like XML, but it isn't, not really. However, this doesn't happen often. Most systems can optimize the XML parsing to the point where it is no longer a crippling deficiency.

Item 37 Validate Inside Your Program with Schemas

Rigorously testing preconditions is an important characteristic of robust, reliable software. Schemas make it very easy to define the preconditions for XML documents you parse and the postconditions for XML documents you write. Even if the document itself does not have a schema, you can write one and use it to test the documents before you operate on them. It is quite hard to attach a DTD to a document inside a program. Fortunately, however, most other schema languages are much more flexible about this.

For example, let's suppose you're in charge of a system at *TV Guide* that accepts schedule information from individual stations over the Web. Information about each show arrives as an XML document formatted as shown in Example 37–1.

Example 37–1 | An XML Instance Document Containing a Television Program Listing

```
<Program xmlns="http://namespaces.example.com/tvschedule"
  <Title>Reality Bites</Title>
  <Description>
   Elimination tournament in which contestants eat a
   succession of gross items until only one is left standing.
   Tonight's episode features rancid apples, insects, and
   McDonald's Happy Meals.
  </Description>
  <Date>2003-11-21</Date>
  <Start>08:00:00-05:00</Start>
  <Duration>PT30M</Duration>
  <Station>KFOX</Station>
</Program>
```

Every day, around the clock, stations from all over the country send schedule updates like this one that you need to store in a local database. Some of these stations use software you sold them. Some of them hire interns to type the data into a password-protected form on your web site. Others use custom software they wrote themselves. There may even be a few hackers typing the information into text files using emacs and then telnetting to your web server on port 80, where they paste in the data. There are about a dozen different places where mistakes can creep in. Therefore, before you even begin to think about processing a submission, you want to verify that it's correct. In particular, you want to verify the following.

- The root element of the document is `Program`.
- All required elements are present.
- No more than one of each element is present.
- The `Title` element is not empty.
- The date is a legal date in the future.
- The `Start` element contains a sensible time.
- The duration looks like a period of time.
- The station identifier is a four-letter code beginning with either K or W.
- The station identifier maps to a known station somewhere in the country, which can be determined by looking it up in a database running on a different machine in your intranet.

You could write program code to verify all of these statements after the document was parsed. However, it's much easier to write a schema that

describes them declaratively and let the parser check them. The W3C XML Schema Language, RELAX NG, and Schematron can all handle about 85% of these requirements. They all have problems with the requirement that the date be in the future and that the station be listed in a remote database. These will have to be checked using real programming code written in Java, C++, or some other language after the document has been parsed. However, we can make the other checks with a schema. Example 37–2 shows one possible W3C XML Schema Language schema that tests most of the above constraints.

Example 37–2 | A W3C XML Schema for Television Program Listings

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <xsd:element name="Program">
    <xsd:complexType>
      <xsd:all>
        <xsd:element name="Title">
          <xsd:simpleType>
            <xsd:restriction base="xsd:string">
              <xsd:minLength value="1"/>
            </xsd:restriction>
          </xsd:simpleType>
        </xsd:element>
        <xsd:element name="Description" type="xsd:string"/>
        <xsd:element name="Date"        type="xsd:date"/>
        <xsd:element name="Start"       type="xsd:time"/>
        <xsd:element name="Duration"    type="xsd:duration"/>
        <xsd:element name="Station">
          <xsd:simpleType>
            <xsd:restriction base="xsd:token">
              <xsd:pattern value="(W|K)[A-Z][A-Z][A-Z]"/>
            </xsd:restriction>
          </xsd:simpleType>
        </xsd:element>
      </xsd:all>
    </xsd:complexType>
  </xsd:element>

</xsd:schema>
```

For simplicity, I'll assume this schema resides at the URL `http://www.example.com/tvprogram.xsd` in the examples that follow, but you can store it anywhere convenient.

There are several different ways to programmatically validate a document, depending on the schema language, the parser, and the API. Here I'll demonstrate two: Xerces-J using SAX properties and DOM Level 3 validation.

Xerces-J

The Xerces-J SAX parser supports validation with the W3C XML Schema Language. By default, it reads the schema with which to validate documents from the `xsi:schemaLocation` and `xsi:noNamespaceSchemaLocation` attributes in the instance document. However, you can override these with the `http://apache.org/xml/properties/schema/external-schemaLocation` and `http://apache.org/xml/properties/schema/external-noNamespaceSchemaLocation` SAX properties. In this example, the documents being validated have namespaces, so we'll set `http://apache.org/xml/properties/schema/external-schemaLocation` to `http://www.example.com/tvprogram.xsd`. Then, we'll turn on schema validation by setting the `http://apache.org/xml/features/validation/schema` feature to true.

```
XMLReader parser = XMLReaderFactor.createXMLReader(
 "org.apache.xerces.parsers.SAXParser");
parser.setProperty(
"http://apache.org/xml/properties/schema/external-
schemaLocation",
  "http://namespaces.example.com/tvschedule"
  + " http://www.example.com/tvprogram.xsd");
parser.setFeature(
  "http://apache.org/xml/features/validation/schema",
  true);
```

We'll also have to register an `ErrorHandler` to receive any validation errors that are detected. Because validity errors aren't necessarily fatal unless we make them so, we'll rethrow the `SAXParseException` passed to the `error()` method. Example 37–3 shows an appropriate `ErrorHandler` class.

Example 37–3 | A SAX `ErrorHandler` That Makes Validity Errors Fatal

```
import org.xml.sax.*;

public class ErrorsAreFatal implements ErrorHandler {

  public void warning(SAXParseException exception) {
    // Ignore warnings
  }

  public void error(SAXParseException exception)
   throws SAXException {

    // A validity error; rethrow the exception.
    throw exception;

  }

  public void fatalError(SAXParseException exception)
   throws SAXException {

    // A well-formedness error
    throw exception;

  }

}
```

This `ErrorHandler` also needs to be installed with the parser.

```
parser.setErrorHandler(new ErrorsAreFatal());
```

Finally, the document can be parsed. The parser checks it against the schema as it parses. At the same time, the `ContentHandler` methods accumulate the data into the fields. Since SAX parsing interleaves parser operation with client code, all the data collected should be stored until the complete document has been validated. Only then can you be sure the document is valid and the information should be committed. Example 37–4 demonstrates one way to build a `TVProgram` object that stores this data. The constructor is private, so the only way to build such an object is by passing an `InputStream` containing a `TVProgram` document to the `readTVProgram()` method. The `TVProgram` object is actually created before the parsing starts. However, it's not returned to anything outside this class until the input document has been parsed and any constraints verified. If a constraint is violated, then an exception is thrown.

Example 37–4 | A Program That Validates against a Schema

```
import java.util.*;
import java.io.*;
import org.xml.sax.*;
import org.xml.sax.helpers.*;

public class TVProgram extends DefaultHandler {

  private String title;
  private String description;
  private Date   startTime; // includes both date and time
  private int    duration;  // rounded to nearest second
  private String station;   // rounded to nearest second

  private TVProgram() {
    // Data will be initialized in the readTVProgram() method
  }

  private static XMLReader parser;

  // Initialization block. No need to load a new parser for
  // each document.
  static {
    try {
      parser = XMLReaderFactory.createXMLReader(
       "org.apache.xerces.parsers.SAXParser");
      parser.setProperty(
"http://apache.org/xml/properties/schema/external-schemaLocation",
       "http://namespaces.example.com/tvschedule"
       + " http://www.example.com/tvprogram.xsd");
      parser.setFeature(
       "http://apache.org/xml/features/validation/schema",
       true);
      parser.setErrorHandler(new ErrorsAreFatal());
    }
    catch (SAXException e) {
     throw new RuntimeException(
       "Handling exceptions in static initializers is tricky");
    }
  }
```

```
  public static TVProgram readTVProgram(String systemID)
   throws SAXException, IOException {

    TVProgram program = new TVProgram();
    parser.setContentHandler(program);
    parser.parse(systemID);

    // If no exception has been thrown yet, then the document
    // must be valid. However, we still have to check the
    // constraints the schema couldn't:
    checkDateInFuture(program.startTime);
    checkStationExists(program.station);

    // If we get here, everything's fine.
    return program;

  }

  private static void checkDateInFuture(Date date)
   throws SAXException {

    // Java code to compare the date to the current time

  }

  private static void checkStationExists(String station)
   throws SAXException {

    // JDBC code to look up the station call letters in our
    // database

  }

  // Various ContentHandler methods that will fill in the fields
  // of this object. This could be a separate class instead...

  // Various setter and getter and other methods...

}
```

Presumably, after such an object has been read, other code will store it in a database or otherwise work with it. And, of course, building an object that exactly matches the data in the document is far from the only way to model the data. All these details will depend on the business logic in the rest of the program. However, the input checking through validation will normally be similar to what's shown here.

DOM Level 3 Validation

DOM Level 3 provides a detailed API for validation. This API can be used to validate against any schema language the parser supports, although DTDs and W3C XML Schema Language schemas are certainly the most common options.

Caution *This section is based on working drafts of the relevant specifications and experimental software. The broad picture presented here is correct, but a lot of details are likely to change before DOM Level 3 is finalized.*

Unlike SAX, DOM objects can be validated when the document is first parsed or at any later point. You can also validate individual nodes rather than validating the entire document. To validate while parsing, you set the following features on the document or document builder's `DOMConfiguration` object.

- `schema-type`: A URI identifying the schema language used to validate. Values include `http://www.w3.org/2001/XMLSchema` for the W3C XML Schema Language and `http://www.w3.org/TR/REC-xml` for DTDs.
- `schema-location`: A white-space-separated list of URLs for particular schema documents used to validate.
- `validate`: If true, all documents should be validated. If false, no documents should be validated unless `validate-if-schema` is true.
- `validate-if-schema`: Validate only if a schema (in whatever language) is available, either one set by the `schema-location` and `schema-type` parameters or one specified in the instance document using a mechanism such as a `DOCTYPE` declaration or an `xsi:schemaLocation` attribute.

For example, here's the DOM Level 3 code to parse the document at `http://www.example.net/kfox.xml` while validating it against the schema at `http://www.example.com/tvprogram.xsd`.

```
DOMImplementation impl = DOMImplementationRegistry
    .getDOMImplementation("XML 1.0 LS-Load 3.0");
if (impl == null || !impl.hasFeature("Core", "3.0") {
  throw new Exception("DOM Level 3 not supported");
}
```

```
DOMImplementationLS implLS = impl.getInterface("LS-Load", "3.0");
DOMBuilder builder = implLS.createDOMBuilder(
  DOMBuilder.MODE_SYNCHRONOUS,
  "http://www.w3.org/2001/XMLSchema");
DOMConfiguration config = builder.getConfig();
config.setParameter("validate", Boolean.TRUE);
config.setParameter("schema-location",
  "http://www.example.com/tvprogram.xsd");
config.setParameter("schema-type",
  "http://www.w3.org/2001/XMLSchema");
builder.setErrorHandler(new DOMErrorHandler() {
  public boolean handleError(in DOMError error) {
    System.err.println(error.getMessage());
  }
});
Document doc = builder.parseURI(
  "http://www.example.net/kfox.xml");
```

Currently, this API is only experimentally supported by Xerces and the Xerces-derived XML for Java, but more parsers should support it in the future.

If you make modifications to a document, DOM3 allows you to revalidate it to make sure it's still valid. This is an optional feature, and not all DOM Level 3 implementations support it. If one does, each `Document` object will be an instance of the `DocumentEditVal` interface as well. Just cast the object to this type and invoke the `validateDocument()` method as shown below.

```
if (doc instanceof DocumentEditVal) {
  DocumentEditVal docVal = (DocumentEditVal) doc;
  try {
    boolean valid = docVal.validateDocument();
  }
  catch (ExceptionVAL ex) {
    // This document doesn't have a schema
  }
}
```

You can even continuously validate a document as it is modified. If any change makes the document invalid, the problem will be reported to

the registered `DOMErrorHandler`. Just set the `continuousValidityChecking` attribute to true.

```
docVal.setContinuousValidityChecking(true);
```

This is particularly useful if the modifications are not driven by the program but by a human using an editor. In this case, you can even check the data input for validity before allowing the changes to be made.

If you need to change the schema associated with a document, set the `schema-location` and `schema-type` parameters on the document's `DOMConfiguration` object.

```
DOMConfiguration config = doc.getConfig();
config.setParameter("schema-type",
                    "http://www.w3.org/2001/XMLSchema");
config.setParameter("schema-location",
                    "http://www.example.com/schema.xsd");
```

To validate this document, you would then call `validateDocument()` as described above.

Validation with DOM differs from validation with SAX in that you don't actually begin working with the document until after it has been validated. Thus there's no need to worry about committing the data in pieces. This is a common difference between SAX and DOM programs. A second advantage is that DOM validation can be reversed so that you build the document in memory and then check for validity before outputting it. You can even check every node you add to the `Document` object for adherence to a schema immediately and automatically.

Whether you validate with SAX or DOM, whether you do so continuously or just once when the document is first parsed, and whether the schema is a DTD, a W3C XML Schema Language schema, or something else, validation is an extremely useful tool. Even if you don't reject invalid documents, you can still use the result of validity checking to determine what to do with any given document. For instance, you might validate documents against several known schemas to identify the document's type and dispatch the document to the method that processes that type. Validation is an essential component of robust, reliable systems.