Chapter 5

Reading XML

Writing XML documents is very straightforward, as I hope Chapters 3 and 4 proved. Reading XML documents is not nearly as simple. Fortunately, you don't have to do all the work yourself; you can use an XML parser to read the document for you. The XML parser exposes the contents of an XML document through an API, which a client application then reads. In addition to reading the document and providing the contents to the client application, the parser checks the document for well-formedness and (optionally) validity. If it finds an error, it informs the client application.

InputStreams and Readers

It's time to reverse the examples of Chapters 3 and 4. Instead of putting information into an XML document, I'm going to take information out of one. In particular, I'm going to use an example that reads the response from the Fibonacci XML-RPC servlet introduced in Chapter 3. This document takes the form shown in Example 5.1.

Example 5.1 A Response from the Fibonacci XML-RPC Server

```
<?xml version="1.0"?>
<methodResponse>
```

```
    <params>
      <param>
        <value><double>28657</double></value>
      </param>
    </params>
</methodResponse>
```

The clients for the XML-RPC server developed in Chapter 3 simply printed the entire document on the console. Now I want to extract just the answer and strip out all of the markup. In this situation, the user interface will look something like this:

```
C:\XMLJAVA>java FibonacciClient 9
34
```

From the user's perspective, the XML is completely hidden. The user neither knows nor cares that the request is being sent and the response is being received in an XML document. Those are merely implementation details. In fact, the user may not even know that the request is being sent over the network rather than being processed locally. All the user sees is the very basic command line interface. Obviously you could attach a fancier GUI front end, but this is not a book about GUI programming, so I'll leave that as an exercise for the reader.

Given that you're writing a client to talk to an XML-RPC server, you know that the documents you're processing always take this form. You know that the root element is `methodResponse`. You know that the `methodResponse` element contains a single `params` element that in turn contains a `param` element. You know that this `param` element contains a single `value` element. (For the moment, I'm going to ignore the possibility of a fault response to keep the examples smaller and simpler. Adding a fault response would be straightforward, and we'll do that in later chapters.) The XML-RPC specification specifies all of this. If any of it is violated in the response you get back from the server, then that server is not sending correct XML-RPC. You'd probably respond to this by throwing an exception.

Given that you're writing a client to talk to the specific servlet at *http://www.elharo.com/fibonacci/XML-RPC,* you know that the `value` element contains a single `double` element that in turn contains a string representing a double. This isn't true for all XML-RPC servers, but it is true for this one. If the server returned a value with a type other than double, you'd probably respond by throwing an exception, just as you would if a local method you expected to return a `Double` instead returned a `String`. The only significant difference is that in the XML-RPC case, neither the compiler nor the virtual machine can do any type checking. Thus you may want to be a bit more explicit about handling a case in which something unexpected is returned.

The main point is this: Most programs you write are going to read documents written in a specific XML vocabulary. They will not be designed to handle absolutely any well-formed document that comes down the pipe. Your programs will make assumptions about the content and structure of those documents, just as they now make assumptions about the content and structure of external objects. If you are concerned that your assumptions may occasionally be violated (and you should be), then you can validate your documents against a schema of some kind so you know up front if you're being fed bad data. However, you do need to make some assumptions about the format of your documents before you can process them reasonably.

It's simple enough to hook up an `InputStream` and/or an `InputStreamReader` to the document, and read it out. For example, the following method reads an input XML document from the specified input stream and copies it to `System.out`:

```
public printXML(InputStream xml) {

  int c;
  while ((c = xml.read()) != -1) System.out.write(c);

}
```

To actually extract the information, a little more work is required. You need to determine which pieces of the input you actually want and separate those out from all the rest of the text. In the Fibonacci XML-RPC example, you need to extract the text string between the `<double>` and `</double>` tags and then convert it to a `java.math.BigInteger` object. (Remember, I'm using a `double` here only because XML-RPC's `int`s aren't big enough to handle Fibonacci numbers. However, all the responses should contain an integral value.)

The `readFibonacciXMLRPCResponse()` method in Example 5.2 does exactly this by first reading the entire XML document into a `StringBuffer`, converting the buffer to a `String`, and then using the `indexOf()` and `substring()` methods to extract the desired information. The `main()` method connects to the server using the `URL` and `URLConnection` classes, sends a request document to the server using the `OutputStream` and `OutputStreamWriter` classes, and passes `InputStream` containing the response XML document to the `readFibonacciXMLRPCResponse()` method.

Example 5.2 Reading an XML-RPC Response

```
import java.net.*;
import java.io.*;
import java.math.BigInteger;


public class FibonacciClient {

  static String defaultServer
   = "http://www.elharo.com/fibonacci/XML-RPC";

  public static void main(String[] args) {

    if (args.length <= 0) {
      System.out.println(
       "Usage: java FibonacciClient number url"
      );
      return;
    }

    String server = defaultServer;
    if (args.length >= 2) server = args[1];

    try {
      // Connect to the server
      URL u = new URL(server);
      URLConnection uc = u.openConnection();
      HttpURLConnection connection = (HttpURLConnection) uc;
      connection.setDoOutput(true);
      connection.setDoInput(true);
      connection.setRequestMethod("POST");
      OutputStream out = connection.getOutputStream();
      Writer wout = new OutputStreamWriter(out);

      // Write the request
      wout.write("<?xml version=\"1.0\"?>\r\n");
      wout.write("<methodCall>\r\n");
      wout.write(
       "  <methodName>calculateFibonacci</methodName>\r\n");
```

```
      wout.write("  <params>\r\n");
      wout.write("    <param>\r\n");
      wout.write("      <value><int>" + args[0]
       + "</int></value>\r\n");
      wout.write("    </param>\r\n");
      wout.write("  </params>\r\n");
      wout.write("</methodCall>\r\n");

      wout.flush();
      wout.close();

      // Read the response
      InputStream in = connection.getInputStream();
      BigInteger result = readFibonacciXMLRPCResponse(in);
      System.out.println(result);

      in.close();
      connection.disconnect();
    }
    catch (IOException e) {
      System.err.println(e);
    }

  }

  private static BigInteger readFibonacciXMLRPCResponse(
   InputStream in) throws IOException, NumberFormatException,
   StringIndexOutOfBoundsException {

    StringBuffer sb = new StringBuffer();
    Reader reader = new InputStreamReader(in, "UTF-8");
    int c;
    while ((c = in.read()) != -1) sb.append((char) c);

    String document = sb.toString();
    String startTag = "<value><double>";
    String endTag = "</double></value>";
    int start = document.indexOf(startTag) + startTag.length();
    int end = document.indexOf(endTag);
    String result = document.substring(start, end);
    return new BigInteger(result);
```

```
    }

}
```

Reading the response XML document is more work than writing the request document, but still plausible. This stream- and string-based solution is far from robust, however, and will fail if any one of the following conditions is present:

- The document returned is encoded in UTF-16 instead of UTF-8.
- An earlier part of the document contains the text "<value><double>," even in a comment.
- The response is written with line breaks between the `value` and `double` tags, like this:

```
<value>
  <double>28657</double>
</value>
```

- There's extra white space inside the `double` tags, like this:

```
<double >28657</double >
```

Perhaps worse than these potential pitfalls are all the malformed responses `FibonacciClient` will accept, even though it should recognize and reject them. And this is a simple example in which we just want one piece of data that's clearly marked up. The more data you want from an XML document, and the more complex and flexible the markup, the harder it is to find using basic string matching or even the regular expressions introduced in Java 1.4.

Straight text parsing is not the appropriate tool with which to navigate an XML document. The structure and semantics of an XML document are encoded in the document's markup, its tags, and its attributes; and you need a tool that is designed to recognize and understand this structure as well as reporting any possible errors in this structure. The tool you need is called an XML parser.

XML Parsers

To avoid the difficulties inherent in parsing raw XML input, almost all programs that need to process XML documents rely on an XML parser to actually read the document. The *parser* is a software library (in Java, it's a class) that reads the XML document and checks it for well-formedness. Client applications use method calls

defined in the parser API to receive or request information that the parser retrieves from the XML document.

The parser shields the client application from all of the complex and not particularly relevant details of XML, including

- Transcoding the document to Unicode
- Assembling the different parts of a document divided into multiple entities
- Resolving character references
- Understanding CDATA sections
- Checking hundreds of well-formedness constraints
- Maintaining a list of the namespaces in scope on each element
- Validating the document against its DTD or schema
- Associating unparsed entities with particular URLs and notations
- Assigning types to attributes

One of the original goals of XML was that it be simple enough that a "Desperate Perl Hacker" (DPH) be able to write an XML parser. The exact interpretation of this requirement varied from person to person. At one extreme, the DPH was assumed to be a web designer accustomed to writing CGI scripts without any formal training in programming, who was going to hack it together in a weekend. At the other extreme, the DPH was assumed to be Larry Wall and he was allowed two months for the task. The middle ground was a smart grad student with a couple of weeks.

Whichever way you interpreted the requirement, it wasn't met. In fact, it took Larry Wall more than a couple of months just to add the Unicode support to Perl that XML assumed. Java developers already had adequate Unicode support, however; and thus Java parsers were a lot faster out the gate. Nonetheless, it probably still isn't possible to write a fully conforming XML parser in a weekend, even in Java. Fortunately, however, you don't need to. There are several dozen XML parsers available under a variety of licenses that you can use. In 2002, there's very little need for any programmer to write his or her own parser. Unless you have very unusual requirements, the chance that you can write a better parser than Sun, IBM, the Apache XML Project, and numerous others have already written is quite small.

Java 1.4 is the first version of Java to include an XML parser as a standard feature. With earlier Java versions, you need to download a parser from the Web and install it in the usual way, typically by putting its `.jar` file in your `jre/lib/ext` directory. Even with Java 1.4, you may well want to replace the standard parser with a different one that provides additional features or is simply faster with your documents.

Caution

If you're using Windows, then chances are good you have two different `ext` directories, one where you installed the JDK, such as `C:\jdk1.3.1\jre\lib\ext`, and one in your `Program Files` folder, probably `C:\Program Files\Javasoft\jre\1.3.1\lib\ext`. The first is used for compiling Java programs, the second for running them. To install a new class library, you need to place the relevant JAR file in *both* directories. It is not sufficient to place the JAR archive in one and a shortcut in the other. You need to place full copies in each `ext` directory.

Choosing an XML API

The most important decision you'll make at the start of an XML project is choosing the application programming interface (API) that you'll use. Many APIs are implemented by multiple vendors, so if the specific parser gives you trouble, you can swap in an alternative, often without even recompiling your code. However, if you choose the wrong API, changing to a different one may well involve redesigning and rebuilding the entire application from scratch. Of course, as Fred Brooks taught us, "In most projects, the first system built is barely usable. It may be too slow, too big, awkward to use, or all three. There is no alternative but to start again, smarting but smarter, and build a redesigned version in which these problems are solved. . . . Hence *plan to throw one away; you will, anyhow*."[1] Still, it is much easier to change parsers than it is to change APIs.

There are two major standard APIs for processing XML documents with Java—the Simple API for XML (SAX) and the Document Object Model (DOM)—each of which comes in several versions. In addition there are a host of other, somewhat idiosyncratic APIs including JDOM, dom4j, ElectricXML, and XMLPULL. Finally, each specific parser generally has a native API that it exposes below the level of the standard APIs. For example, the Xerces parser has the Xerces Native Interface (XNI). However, picking such an API limits your choice of parser, and indeed may even tie you to one particular version of the parser, since parser vendors tend not to worry a great deal about maintaining native compatibility between releases. Each of these APIs has its own strengths and weaknesses.

SAX

SAX, the Simple API for XML, is the gold standard of XML APIs. It is the most complete and correct by far. Given a fully validating parser that supports all its optional features, there is very little you can't do with it. It has one or two holes,

1. Frederick Brooks. *The Mythical Man-Month, Anniversary Edition*. Reading, Mass.: Addison-Wesley, 1995, p. 116.

but those are really off in the weeds of the XML specifications, and you have to look pretty hard to find them. SAX is an event-driven API. The SAX classes and interfaces model the parser, the stream from which the document is read, and the client application receiving data from the parser. However, no class models the XML document itself. Instead the parser feeds content to the client application through a callback interface, much like the ones used in Swing and the AWT. This makes SAX very fast and very memory efficient (since it doesn't have to store the entire document in memory). However, SAX programs can be harder to design and code because you normally need to develop your own data structures to hold the content from the document.

SAX works best when processing is fairly *local*; that is, when all the information you need to use is close together in the document (for example, if you were processing one element at a time). Applications that require access to the entire document at once in order to take useful action would be better served by one of the tree-based APIs, such as DOM or JDOM. Finally, because SAX is so efficient, it's the only real choice for truly huge XML documents. Of course, "truly huge" needs to be defined relative to available memory. However, if the documents you're processing are in the gigabyte range, you really have no choice but to use SAX.

DOM

DOM, the Document Object Model, is a fairly complex API that models an XML document as a tree. Unlike SAX, DOM is a read-write API. It can both parse existing XML documents and create new ones. Each XML document is represented as a `Document` object. Documents are searched, queried, and updated by invoking methods on this `Document` object and the objects it contains. This makes DOM much more convenient when random access to widely separated parts of the original document is required. However, it is quite memory intensive compared with SAX, and not nearly as well suited to streaming applications.

JAXP

JAXP, the Java API for XML Processing, bundles SAX and DOM together along with some factory classes and the TrAX XSLT API. (TrAX is not a general-purpose XML API like SAX and DOM. I'll get to it in Chapter 17.) JAXP is a standard part of Java 1.4 and later. However, it is not really a different API. When starting a new program, you ask yourself whether you should choose SAX or DOM. You don't ask yourself whether you should use SAX or JAXP, or DOM or JAXP. SAX and DOM are part of JAXP.

JDOM

JDOM is a Java-native tree-based API that attempts to remove a lot of DOM's ugliness. The JDOM mission statement is, "There is no compelling reason for a Java

API to manipulate XML to be complex, tricky, unintuitive, or a pain in the neck." And for the most part JDOM delivers. Like DOM, JDOM reads the entire document into memory before it begins to work on it; and the broad outline of JDOM programs tends to be the same as for DOM programs. However, the low-level code is a lot less tricky and ugly than the DOM equivalent. JDOM uses concrete classes and constructors rather than interfaces and factory methods. It uses standard Java coding conventions, methods, and classes throughout. JDOM programs often flow a lot more naturally than the equivalent DOM program.

I think JDOM often does make the easy problems easier; but in my experience, JDOM also makes the hard problems harder. Its design shows a very solid understanding of Java, but the XML side of the equation feels much rougher. It's missing some crucial pieces, such as a common node interface or superclass for navigation. JDOM works well (and much better than DOM) on fairly simple documents with no recursion, limited mixed content, and a well-known vocabulary. It begins to show some weakness when asked to process arbitrary XML. When I need to write programs that operate on any XML document, I tend to find DOM simpler despite its ugliness.

dom4j

dom4j was forked from the JDOM project fairly early on. Like JDOM, it is a Java-native, tree-based, read-write API for processing generic XML. However, it uses interfaces and factory methods rather than concrete classes and constructors. This enables you to plug in your own node classes that put XML veneers on other forms of data such as objects or database records. (In theory, you could do this with DOM interfaces too; but in practice most DOM implementations are too tightly coupled to interoperate with each other's classes.) It does have a generic node type that can be used for navigation.

ElectricXML

ElectricXML is yet another tree-based API for processing XML documents with Java. It's quite small, which makes it suitable for use in applets and other storage-limited environments. It's the only API I mention here that isn't open source, and the only one that requires its own parser rather than being able to plug into multiple different parsers. It has gained a reputation as a particularly easy-to-use API. However, I'm afraid its perceived ease-of-use often stems from catering to developers' misconceptions about XML. It is far and away the least correct of the tree-based APIs. For example, it tends to throw away a lot of white space it shouldn't, and its namespace handling is poorly designed. Ideally, an XML API should be as simple as it can be *and no simpler*. In particular, it should not be simpler than XML itself is. ElectricXML pretends that XML is less complex than it really is, which may work for a while as long as your needs are simple, but will ultimately fail

when you encounter more complex documents. The only reason I mention it here is because the flaws in its design aren't always apparent to casual users, and I tend to get a lot of e-mail from ElectricXML users asking me why I'm ignoring it.

XMLPULL

SAX is fast and very efficient, but its callback nature is uncomfortable for some programmers. Recently some effort has gone into developing pull parsers that can read streaming content the way that SAX does, but only when the client application requests it. The recently published standard API for such parsers is XMLPULL. XMLPULL shows promise for the future (especially for developers who need to read large documents quickly but just don't like callbacks). However, pull parsing is still clearly in its infancy. On the XML side, namespace support is turned off by default. Even worse, XMLPULL ignores the `DOCTYPE` declaration, even the internal DTD subset, unless you specifically ask it to read it. From the Java side of things, XMLPULL does not take advantage of polymorphism, relying instead on such un-OOP constructs as `int` type codes to distinguish nodes instead of making them instances of different classes or interfaces. I don't think XMLPULL is ready for prime time quite yet. None of this is unusual for such a new technology, however. Some of the flaws I cite were also present in earlier versions of SAX, DOM, and JDOM and were only corrected in later releases. In the next couple of years, as pull parsing evolves, XMLPULL may become a much more serious competitor to SAX.

Data Binding

Recently, there has been a flood of so-called data-binding APIs that try to map XML documents into Java classes. Although DOM, JDOM, and dom4j all map XML documents into Java classes, these data-binding APIs attempt to go further—mapping a `Book` document into a `Book` class rather than just a generic `Document` class, for example. These are sometimes useful in very limited and predictable domains. But they tend to make too many assumptions that simply aren't true in the general case to make them broadly suitable for XML processing. In particular, these products tend to depend implicitly on one or more of the following common fallacies:

- Documents have schemas or DTDs.
- Documents that do have schemas and/or DTDs are valid.
- Structures are fairly flat and definitely not recursive; that is, they look pretty much like tables.
- Narrative documents aren't worth considering.
- Mixed content doesn't exist.

- Choices don't exist; that is, elements with the same name tend to have the same children.
- Order doesn't matter.

The fundamental flaw in these schemes is an insistence on seeing the world through object-colored glasses. XML documents can be used for object serialization, and in that use case all of these assumptions are reasonably accurate—but XML is a lot more general than that. The vast majority of XML documents cannot plausibly be understood as serialized objects, although a lot of programmers come at it from that point of view because that's what they're familiar with. Once again, when you're an expert with a hammer, it's not surprising that the world looks like it's full of nails.

The truth is, XML documents are not objects and schemas are not classes. The constraints and structures that apply to objects simply do not apply to XML elements and vice versa. Unlike Java objects, XML elements routinely violate their declared types, if indeed they even have a type in the first place. Even valid XML elements often have different content in different locations. Mixed content is quite common. Recursive content isn't as common, but it does exist. A little more subtle but even more important, XML structures are based on hierarchy and position rather than on the explicit pointers of object systems. It is possible to map one to the other, but the resulting structures are ugly and fragile. When you're finished, what you've accomplished tends to merely reinvent DOM. XML needs to be approached and understood on its own terms, not Java's. Data-binding APIs are just a little too limited to interest me, and I do not plan to treat them in this book.

Choosing an XML Parser

When choosing a parser library, many factors come into play. These include what features the parser has, how much it costs, which APIs it implements, how buggy it is, and—last and certainly least—how fast the parser parses.

Features

The XML 1.0 specification does allow parsers some leeway in how much of the specification they implement. Parsers can be divided roughly into three categories:

1. Fully validating parsers
2. Parsers that do not validate, but do read the external DTD subset and external DTD parameter entity references, in order to supply entity replacement text and assign attribute types
3. Parsers that read only the internal DTD subset and do not validate

In practice there's also a fourth category of parsers that read the instance document but do not perform all of the mandated well-formedness checks. Technically such parsers are not allowed by the XML specification, but there are still a lot of them out there.

If the documents you're processing have DTDs, then you need to use a fully validating parser. You don't necessarily have to turn on validation if you don't want to. However, XML is designed such that you really can't be sure to get the full content of an XML document without reading its DTD. In some cases, the differences between a document whose DTD has been processed and the same document whose DTD has not been processed can be huge. For example, a parser that reads the DTD will report default attribute values, but one that doesn't won't. The handling of ignorable white space can vary between a validating parser and a parser that merely reads the DTD but does not validate. Furthermore, external entity references will be expanded by a validating parser, but not necessarily by a nonvalidating parser. You should use a nonvalidating parser only if you're confident none of the documents you'll process carry document type declarations. One situation in which this is reasonable is with a SOAP server or client, because SOAP specifically prohibits documents from using `DOCTYPE` declarations. (But even in that case, I still recommend that you check for a `DOCTYPE` declaration and throw an exception if you spot one.)

Beyond the lines set out by XML 1.0, parsers also differ in their support for subsequent specifications and technologies. In 2002, all parsers worth considering support namespaces and automatically check for namespace well-formedness as well as XML 1.0 well-formedness. Most of these parsers do allow you to disable these checks for the rare legacy documents that don't adhere to namespaces rules. Currently, Xerces and Oracle are the only Java parsers that support schema validation, although other parsers are likely to add this in the future.

Some parsers also provide extra information not required for normal XML parsing. For example, at your request, Xerces can inform you of the `ELEMENT`, `ATTLIST`, and `ENTITY` declarations in the DTD. Crimson will not do this, so if you needed to read the DTD you would pick Xerces over Crimson.

API Support

Most of the major parsers support both SAX and DOM. But a few parsers only support SAX, and at least a couple only support their own proprietary API. If you want to use DOM or SAX, make sure you pick a parser that can handle it. Xerces, Oracle, and Crimson can.

SAX actually includes a number of optional features that parsers are not required to support. These include validation, reporting comments, reporting declarations in the DTD, reporting the original text of the document before parsing,

and more. If any of these are important to you, you'll need to make sure that your parser supports them, too.

The other APIs, including JDOM and dom4j, generally don't provide parsers of their own. Instead they use an existing SAX or DOM parser to read a document, which they then convert into their own tree model. Thus they can work with any convenient parser. The notable exception here is ElectricXML, which does include its own built-in parser. ElectricXML is optimized for speed and size and does not interoperate well with SAX and DOM.

License

One often overlooked consideration when choosing a parser is the license under which the parser is published. Most parsers are free in the free-beer sense, and many are also free in the free-speech sense. However, license restrictions can still get in your way.

Because parsers are essentially class libraries that are dynamically linked to your code (as all Java libraries are), and because parsers are generally released under fairly lenient licenses, you don't have to worry about viral infections of your code with the GPL. In one case I'm aware of, Ælfred, any changes you make to the parser itself would have to be donated back to the community; but this would not affect the rest of your classes. That being said, you'll be happier and more productive if you do donate your changes back to the communities for the more liberally licensed parsers such as Xerces. It's better to have your changes rolled into the main code base than to have to keep applying them every time a new version is released.

There actually aren't too many parsers you can buy. If your company is really insistent about not using open source software, then you can probably talk IBM into selling you an overpriced license for their XML for Java parser (which is just an IBM-branded version of the open source Xerces). However, there isn't a shrink-wrapped parser you can buy, nor is one really needed. The free parsers are more than adequate.

Correctness

An often overlooked criterion for choosing a parser is correctness—how much of the relevant specifications are implemented and how well. All of the parsers I've used have had nontrivial bugs in at least some versions. Although no parser is perfect, some parsers are definitely more reliable than others.

I wish I could say that there was one or more good choices here, but the fact is that every single parser I've ever tried has sooner or later exhibited significant conformance bugs. Most of the time these fall into two categories:

1. Reporting correct constructs as errors
2. Failing to report incorrect syntax

It's hard to say which is worse. On one hand, unnecessarily rejecting well-formed documents prevents you from handling data others send you. On the other hand, when a parser fails to report incorrect XML documents, it's virtually guaranteed to cause problems for people and systems who receive the malformed documents and correctly reject them.

One thing I will say is that well-formedness is the most important criterion of all. To be seriously considered, a parser has to be absolutely perfect in this area, and many aren't. A parser must allow you to confidently determine whether a document is or is not well-formed. Validity errors are not quite as important, but they're still significant. Many programs can ignore validity and consequently ignore any bugs in the validator.

Continuing downward in the hierarchy of seriousness are failures to properly implement the standard SAX and DOM APIs. A parser might correctly detect and report all well-formedness and validity errors, but fail to pass on the contents of the document. For example, it might throw away ignorable white space rather than making it available to the application. Even less serious but still important are violations of the contracts of the various public APIs. For example, DOM guarantees that each `Text` object read from a parsed document will contain the longest-possible string of characters uninterrupted by markup. However, I have seen parsers that occasionally passed in adjacent text nodes as separate objects rather than merging them.

Java parsers are also subject to a number of edge conditions. For example, in SAX each attribute value is passed to the client application as a single string. Because the Java `String` class is backed by an array of chars indexed by an `int`, the maximum number of chars in a `String` is the same as the maximum size of an `int`, 2,147,483,647. However, there is no maximum number of characters that may appear in an attribute value. Admittedly a three-gigabyte attribute value doesn't seem too likely (perhaps a base64 encoded video?), and you'd probably run out of memory long before you bumped up against the maximum size of a string. Nonetheless, XML doesn't prohibit strings of such lengths, and it would be nice to think that Java could at least theoretically handle all XML documents within the limits of available memory.

Efficiency

The last consideration is efficiency, or how fast the parser runs and how much memory it uses. Let me stress that again: *Efficiency should be your* last *concern when choosing a parser*. As long as you use standard APIs and keep parser-dependent code to a minimum, you can always change the underlying parser later if the one you picked initially proves too inefficient.

The speed of parsing tends to be dominated by I/O considerations. If the XML document is served over the network, it's entirely possible that the bottleneck is

the speed with which data can move over the network, not the XML parsing at all. With situations in which the XML is being read from the disk, the time to read the data can still be significant even if not quite the bottleneck it is in network applications.

Anytime you're reading data from a disk or the network, remember to buffer your streams. You can buffer at the byte level with a `BufferedInputStream` or at the character level with a `BufferedReader`. Perhaps a little counter-intuitively, you can gain extra speed by double buffering with both byte and character buffers. However, most parsers are happier if you feed them a raw `InputStream` and let them convert the bytes to characters (parsers normally detect the correct encoding better than most client code). Therefore, I prefer to use just a `BufferedInputStream`, and not a `BufferedReader`, unless speed is very important and I'm very sure of the encoding in advance. If you don't buffer your I/O, then I/O considerations will limit total performance no matter how fast the parser is.

Complicated programs can also be dominated by processing that happens after the document is parsed. For example, if the XML document lists store locations, and the client application is attempting to solve the traveling salesman problem for those store locations, then parsing the XML document is the least of your worries. In such a situation, changing the parser isn't going to help very much at all. The time taken to parse a document normally grows only linearly with the size of the document.

One area in which parser choice does make a significant difference is in the amount of memory used. SAX is generally quite efficient no matter which parser you pick. However, DOM is exactly the opposite. Building a DOM tree can easily eat up as much as ten times the size of the document itself. For example, given a one-megabyte document, the DOM object representing it could be ten megabytes. If you're using DOM or any other tree-based API to process large documents, then you want a parser that uses as little memory as possible. The initial batch of DOM-capable parsers were not really optimized for space, but more recent versions are doing a lot better. With some testing you should be able to find parsers that use only two to three times as much memory as the original document. Still, it's pretty much guaranteed that the memory usage will be larger than the document itself.

Available Parsers

I now want to discuss a few of the more popular parsers and the relative advantages and disadvantages of each.

Xerces-J

I'll begin with my parser of choice, Xerces-J *[http://xml.apache.org/xerces-j/]* from the Apache XML Project. In my experience, this very complete, validating parser

has the best conformance to the XML 1.0 and Namespaces in XML specifications I've encountered. It fully supports the SAX2 and DOM Level 2 APIs, as well as JAXP, although I have encountered a few bugs in the DOM support. The latest versions feature experimental support for parts of the DOM Level 3 working drafts. Xerces-J is highly configurable and suitable for almost any parsing need. Xerces-J is also notable for being the first parser to support the W3C XML Schema Language, although that support is not yet 100 percent complete or bug-free.

The Apache XML Project publishes Xerces-J under the very liberal open source Apache license. Essentially, you can do anything you like with it except use the Apache name in your own advertising. Xerces-J 1.x was based on IBM's XML for Java parser *[http://www.alphaworks.ibm.com/tech/xml4j]*, whose code base IBM donated to the Apache XML Project. Today, the relationship is reversed, and XML for Java is based on Xerces-J 2.x. However, in neither version is there significant *technical* difference between Xerces-J and XML for Java. The real difference is that if you work for a large company with a policy that prohibits using software from somebody you can't easily sue, then you can probably pay IBM a few thousand dollars for a support contract for XML for Java. Otherwise, you might as well just use Xerces-J.

Note

The Apache XML Project also publishes Xerces-C, an open source XML parser written in C++, which is based on IBM's XML for C++ product. But because this is a book about Java, all future references to the undifferentiated name *Xerces* should be understood as referring strictly to the Java version (Xerces-J).

Crimson

Crimson, previously known as Java Project X, is the parser Sun bundles with the JDK 1.4. Crimson supports more or less the same APIs and specifications as Xerces does—SAX2, DOM2, JAXP, XML 1.0, Namespaces in XML, and so on—with the notable exception of schemas. In my experience, Crimson is somewhat buggier than Xerces. I've encountered well-formed documents that Crimson incorrectly reported as malformed but which Xerces could parse without any problems. All of the bugs I've encountered with Xerces related to validation, not to the more basic criterion of well-formedness.

The reason Crimson exists is that some Sun engineers disagreed with some IBM engineers about the proper internal design for an XML parser. (Also, the IBM code was so convoluted that nobody outside of IBM could figure it out.) Crimson was supposed to be significantly faster, more scalable, and more memory efficient than Xerces—and not get soggy in milk either. However, whether it's actually faster than Xerces (much less significantly faster) is questionable. When first released,

Sun claimed that Crimson was several times faster than Xerces. But IBM ran the same benchmarks and got almost exactly opposite results, claiming that Xerces was several times faster than Crimson. After a couple of weeks of hooting and hollering on several mailing lists, the true cause was tracked down. Sun had heavily optimized Crimson for the Sun virtual machine and just-in-time compiler and naturally ran the tests on Sun virtual machines. IBM publishes its own Java virtual machine and was optimizing for and benchmarking on that.

To no one's great surprise, Sun's optimizations didn't perform nearly as well when run on non-Sun virtual machines; and IBM's optimizations didn't perform nearly as well when run on non-IBM virtual machines. To quote Donald Knuth, "Premature optimization is the root of all evil."[2] Eventually both Sun and IBM began testing on multiple virtual machines and watching out for optimizations that were too tied to the architecture of any one virtual machine; and now both Xerces-J and Crimson seem to run about equally fast on equivalent hardware, regardless of the virtual machine.

The real benefit to Crimson is that it's bundled with the JDK 1.4. (Crimson does work with earlier virtual machines back to Java 1.1. It just isn't bundled with them in the base distribution.) Thus if you know you're running in a Java 1.4 or later environment, you don't have to worry about installing extra JAR archives and class libraries just to parse XML. You can write your code to the standard SAX and DOM classes and expect it to work out of the box. If you want to use a parser other than Crimson, then you can still install the JAR files for Xerces-J or some other parser and load its implementation explicitly. However, in Java 1.3 and earlier (which is the vast majority of the installed base at the time of this writing), you have to include some parser library with your own application.

Going forward, Sun and IBM are cooperating on Xerces-2, which will probably become the default parser in a future release of the JDK. Crimson is unlikely to be developed further or gain support for new technologies like XInclude and schemas.

Ælfred

The GNU Classpath Extensions Project's Ælfred *[http://www.gnu.org/software/classpathx/jaxp/]* is actually two parsers, `gnu.xml.aelfred2.SAXDriver` and `gnu.xml.aelfred2.XmlReader`. `SAXDriver` aims for a small footprint rather than a large feature set. It supports XML 1.0 and Namespaces in XML. However, it implements the minimum part of XML 1.0 needed for conformance. For example, it neither resolves external entities nor validates them. Nor does it make all the well-formedness checks it should make, and it can miss malformed documents. It supports

2. Donald Knuth. Structured programming with `go to` statements. *Computing Surveys* 6 (1974):261–301.

SAX but not DOM. Its small size makes it particularly well suited for applets. For less resource constrained environments, Ælfred provides `XmlReader`, a fully validating, fully conformant parser that supports both SAX and DOM.

Ælfred was originally written by the now defunct Microstar, which placed it in the public domain. David Brownell picked up development of the parser and brought it under the aegis of the GNU Classpath Extensions Project *[http://www.gnu.org/software/classpathx/]*, an attempt to reimplement Sun's Java extension libraries (the `javax` packages) as free software. Ælfred is published under the GNU General Public License with library exception. In brief, this means that as long as you only call Ælfred through its public API and don't modify the source code yourself, the GPL does not infect your code.

Piccolo

Yuval Oren's Piccolo *[http://piccolo.sourceforge.net/]* is the newest entry into the parser arena. It is a very small, very fast, open source, nonvalidating XML parser. However, it does read the external DTD subset, apply default attribute values, and resolve external entity references. Piccolo supports the SAX API exclusively; it does not have a DOM implementation.

SAX

SAX, the Simple API for XML, was the first standard API shared across different XML parsers. SAX is unique among XML APIs in that it models the parser rather than the document. In particular the parser is represented as an instance of the `XMLReader` interface. The specific class that implements this interface varies from parser to parser. Most of the time, you access it only through the common methods of the `XMLReader` interface.

A parser reads a document from beginning to end. As it does so, it encounters start-tags, end-tags, text, comments, processing instructions, and more. In SAX, the parser tells the client application what it sees as it sees it by invoking methods in a `ContentHandler` object. `ContentHandler` is an interface the client application implements to receive notification of document content. The client application will instantiate a client-specific instance of the `ContentHandler` interface and register it with the `XMLReader` that's going to parse the document. As the reader reads the document, it calls back to the methods in the registered `ContentHandler` object. The general pattern is very similar to how events are handled in the AWT and Swing.

Example 5.3 is a simple SAX program that communicates with the XML-RPC servlet from Chapter 3. It sends the request document using basic output stream techniques and then receives the response through SAX.

Example 5.3 A SAX-Based Client for the Fibonacci XML-RPC Server

```
import java.net.*;
import java.io.*;
import java.math.BigInteger;
import org.xml.sax.*;
import org.xml.sax.helpers.*;

public class FibonacciSAXClient {

  public final static String DEFAULT_SERVER
   = "http://www.elharo.com/fibonacci/XML-RPC";

  public static void main(String[] args) {

    if (args.length <= 0) {
      System.out.println(
       "Usage: java FibonacciSAXClient number url"
      );
      return;
    }

    String server = DEFAULT_SERVER;
    if (args.length >= 2) server = args[1];

    try {
      // Connect to the server
      URL u = new URL(server);
      URLConnection uc = u.openConnection();
      HttpURLConnection connection = (HttpURLConnection) uc;
      connection.setDoOutput(true);
      connection.setDoInput(true);
      connection.setRequestMethod("POST");
      OutputStream out = connection.getOutputStream();
      Writer wout = new OutputStreamWriter(out);

      // Transmit the request XML document
      wout.write("<?xml version=\"1.0\"?>\r\n");
      wout.write("<methodCall>\r\n");
      wout.write(
```

```
          "  <methodName>calculateFibonacci</methodName>\r\n");
        wout.write("  <params>\r\n");
        wout.write("    <param>\r\n");
        wout.write("      <value><int>" + args[0]
         + "</int></value>\r\n");
        wout.write("    </param>\r\n");
        wout.write("  </params>\r\n");
        wout.write("</methodCall>\r\n");

        wout.flush();
        wout.close();

        // Read the response XML document
        XMLReader parser = XMLReaderFactory.createXMLReader(
          "org.apache.xerces.parsers.SAXParser"
        );
        // There's a name conflict with java.net.ContentHandler
        // so we have to use the fully package qualified name.
        org.xml.sax.ContentHandler handler
         = new FibonacciHandler();
        parser.setContentHandler(handler);

        InputStream in = connection.getInputStream();
        InputSource source = new InputSource(in);
        parser.parse(source);
        System.out.println();

        in.close();
        connection.disconnect();
      }
      catch (Exception e) {
        System.err.println(e);
      }

    }

}
```

Because SAX is a read-only API, I used the same code as before to write the request sent to the server. The code for reading the response, however, is quite different. Rather than reading directly from the stream, SAX bundles the `InputStream`

in an `InputSource`, a generic wrapper for all the different places an XML document might be stored—`InputStream`, `Reader`, `URL`, `File`, and so on. This `InputSource` object is then passed to the `parse()` method of an `XMLReader`.

Several exceptions can be thrown at various points in this process. For example, a `IOException` will be thrown if the socket connecting the client to the server is broken. A `SAXException` will be thrown if the `org.apache.xerces.parsers.SAXParser` class can't be found somewhere in the class path. A `SAXParseException` will be thrown if the server returns malformed XML. For now, Example 5.3 lumps all of these together in one generic `catch` block. Later chapters go into the various exceptions in more detail.

There's no code in this class to actually find the double response and print it on the console. Yet, when run it produces the expected response:

```
C:\XMLJAVA>java FibonacciSAXClient 42
267914296
```

The real work of understanding and processing documents in this particular format takes place inside the `ContentHandler` object. The specific implementation of the `ContentHandler` interface used here is `FibonacciHandler`, shown in Example 5.4. In this case I chose to extend the `DefaultHandler` adapter class rather than implement the `ContentHandler` interface directly. The pattern is similar to using `WindowAdapter` instead of `WindowListener` in the AWT. It avoids having to implement a lot of do-nothing methods that don't matter in this particular program.

Example 5.4 The ContentHandler for the SAX Client for the Fibonacci XML-RPC Server

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


public class FibonacciHandler extends DefaultHandler {

  private boolean inDouble = false;

  public void startElement(String namespaceURI, String localName,
   String qualifiedName, Attributes atts) throws SAXException {

    if (localName.equals("double")) inDouble = true;

  }
```

```
    public void endElement(String namespaceURI, String localName,
     String qualifiedName) throws SAXException {

      if (localName.equals("double")) inDouble = false;

    }

    public void characters(char[] ch, int start, int length)
    throws SAXException {

      if (inDouble) {
        for (int i = start; i < start+length; i++) {
          System.out.print(ch[i]);
        }
      }

    }

  }
```

What this `ContentHandler` needs to do is recognize and print the contents of the single `double` element in the response while ignoring everything else. Thus, when the `startElement()` method (which the parser invokes every time it encounters a start-tag or an empty-element tag) sees a start-tag with the name `double`, it sets a private `boolean` field named `inDouble` to true. When the `endElement()` method sees an end-tag with the name `double`, it sets the same field back to false. The `characters()` method prints whatever it sees on `System.out`, but only when `inDouble` is true.

Unlike the earlier stream- and string-based solution, this program will detect any well-formedness errors in the document. It will not be tripped up by the unexpected appearance of `<double>` tags in comments, or processing instructions, or ignorable white space between tags. This program would not detect problems that occurred as a result of multiple `double` elements or other invalid markup. However, in later chapters I'll show you how to use a schema to add this capability. The parser-based client is much more robust than the one in Example 5.2, and it's almost as simple. As the markup becomes more complex and the amount of information you need to extract from the document grows, parser-based solutions become far easier and cheaper to implement than any alternative.

The big advantage to SAX compared with other parser APIs is that it's quite fast and extremely memory efficient. You only need to store in memory those parts of the document you actually care about. You can ignore the rest. DOM, by contrast,

must keep the entire document in memory at once.[3] Furthermore, the DOM data structures tend to be substantially less efficient than the serialized XML itself. A DOM `Document` object can easily take up ten times as much memory as would be required just to hold the characters of the document in an array. This severely limits the size of documents that can be processed with DOM and other tree-based APIs. SAX, by contrast, can handle documents that vastly exceed the amount of available memory. If your documents cross the gigabyte threshold, there is really no alternative to SAX.

Furthermore, SAX works very well in streaming applications. A SAX program can begin working with the start of a document before the parser has reached the middle. This is particularly important in low-bandwidth, high-latency environments, such as most network applications. For example, if a client sent a brokerage an XML document containing a list of stocks to buy, the brokerage could execute the first trade before the entire document had been received or parsed. Multithreading can be especially useful here.

The downside to SAX is that most programs are more concerned with XML documents than with XML parsers. In other words, a class hierarchy that models the XML document is a lot more natural and closer to what you're likely to need than a class hierarchy that models parsers would be. SAX programs tend to be more than a little obtuse. It's rare that SAX gives you all the information you need at the same time. Most of the time you find yourself building data structures in memory to store the parts of the document you're interested in until you're ready to use them. At worst, you can end up inventing your own tree model for the entire document, in which case you're probably better off just using DOM or one of the other tree models in the first place, and saving yourself the work.

■ DOM

The Document Object Model, DOM, is the second major standard API for XML parsers, and the first tree-based API I'll consider. Most major parsers implement both SAX and DOM. DOM programs start off similarly to SAX programs, by having a parser object read an XML document from an input stream or other source. However, whereas the SAX parser returns the document broken up into a series of small pieces, the equivalent DOM method returns an entire `Document` object that contains everything in the original XML document. You read information from the document by invoking methods on this `Document` object or on the other objects it contains. This makes DOM much more convenient when random access to widely

3. Xerces does give you the option of using a lazy DOM that only parses parts of the document as they're needed to help reduce memory usage.

separated parts of the original document is required. However, it is quite memory intensive compared with SAX, and not nearly as well suited to streaming applications.

A second advantage to DOM is that it is a read-write API. Whereas SAX can only parse existing XML documents, DOM can also create them. Documents created in this fashion are automatically well-formed. Attempting to create a malformed document throws an exception. Example 5.5 is a DOM-based program for connecting to the Fibonacci XML-RPC servlet. The request is formed as a new DOM document, and the response is read as a parsed DOM document.

Example 5.5 A DOM-Based Client for the Fibonacci XML-RPC Server

```
import java.net.*;
import java.io.*;
import org.w3c.dom.*;
import org.apache.xerces.dom.*;
import org.apache.xerces.parsers.*;
import org.apache.xml.serialize.*;
import org.xml.sax.InputSource;


public class FibonacciDOMClient {

  public final static String DEFAULT_SERVER
   = "http://www.elharo.com/fibonacci/XML-RPC";

  public static void main(String[] args) {

    if (args.length <= 0) {
      System.out.println(
       "Usage: java FibonacciDOMClient number url"
      );
      return;
    }

    String server = DEFAULT_SERVER;
    if (args.length >= 2) server = args[1];

    try {
```

```
// Build the request document
DOMImplementation impl
 = DOMImplementationImpl.getDOMImplementation();

Document request
 = impl.createDocument(null, "methodCall", null);

Element methodCall = request.getDocumentElement();

Element methodName = request.createElement("methodName");
Text text = request.createTextNode("calculateFibonacci");
methodName.appendChild(text);
methodCall.appendChild(methodName);

Element params = request.createElement("params");
methodCall.appendChild(params);

Element param = request.createElement("param");
params.appendChild(param);

Element value = request.createElement("value");
param.appendChild(value);

// Had to break the naming convention here because of a
// conflict with the Java keyword int
Element intElement = request.createElement("int");
Text index = request.createTextNode(args[0]);
intElement.appendChild(index);
value.appendChild(intElement);

// Transmit the request document
URL u = new URL(server);
URLConnection uc = u.openConnection();
HttpURLConnection connection = (HttpURLConnection) uc;
connection.setDoOutput(true);
connection.setDoInput(true);
connection.setRequestMethod("POST");
OutputStream out = connection.getOutputStream();

OutputFormat fmt = new OutputFormat(request);
XMLSerializer serializer = new XMLSerializer(out, fmt);
```

```
            serializer.serialize(request);

            out.flush();
            out.close();

            // Read the response
            DOMParser parser = new DOMParser();
            InputStream in = connection.getInputStream();
            InputSource source = new InputSource(in);
            parser.parse(source);
            in.close();
            connection.disconnect();

            Document doc = parser.getDocument();
            NodeList doubles = doc.getElementsByTagName("double");
            Node datum = doubles.item(0);
            Text result = (Text) datum.getFirstChild();
            System.out.println(result.getNodeValue());
          }
          catch (Exception e) {
            System.err.println(e);
          }

        }

      }
```

In DOM the request document is built as a tree. Each thing in the document is a node in this tree, including not only elements but also text nodes, comments, processing instructions, and more. The document serves as a factory for creating the various kinds of node objects. Each node in this tree belongs to exactly one document. After being created the node object is appended to the child list of its parent node.

Once the `Document` object has been created and populated, it needs to be serialized onto the output stream of the `URLConnection`. Unfortunately, there is no standard parser-independent way to do this in DOM2. This will be added in DOM3. In the meantime, you will need to resort to parser-specific classes and methods. Here I've used Xerces's `org.apache.xml.serialize` package. The basic design is that an `XMLSerializer` object is connected to an output stream. Options for serialization such as where to place line breaks and what character encoding to use are specified

by an `OutputFormat` object. Here I just used the default `OutputFormat`. The document is written onto the stream using the `XMLSerializer`'s `serialize()` method.

Once the server receives and parses the request, it calculates and transmits its response as an XML document. This document must be parsed to extract the single string you actually want. DOM includes a number of methods and classes to extract particular parts of a document without necessarily walking down the entire tree. The one I use here is the `Document` class's `getElementsByTagName()` method. This returns a `NodeList` containing one `Node` object for each element in the input document that has the name `double`. In this case there's exactly one of those, so it's extracted from the list. I then get the first child of that node, which happens to be a `Text` node that contains the value I want. This value is retrieved by the `getNodeValue()` method.

The first problem with DOM should now be apparent. It's more than a little complex, even for very simple problems like this one. However, DOM does have an internal logic; and once you become accustomed to it, you'll find it's actually not that hard to use. Still, the learning curve is quite steep, and frequent reference to the documentation is a necessity.

The second downside to DOM is that it does not expose as much of the information in an XML document as SAX does. Although the basic content of elements, text, and attributes is well supported by both DOM and SAX, there are many more esoteric aspects of XML documents that SAX provides but DOM does not. These include unparsed entities, notations, attribute types, and declarations in the DTD. Some of this will be provided in DOM3.

The third downside to DOM is that it's not as complete as SAX. Much of the code in Example 5.5 is actually part of the Xerces parser rather than standard DOM. Such parser-specific code is virtually impossible to avoid when programming in DOM. That's because DOM doesn't give you any way to create a new XML document, create a new parser, or write a `Document` onto an output stream. All of these have to be provided by the parser. If I were to port Example 5.5 to Crimson or GNU JAXP, I'd have to rewrite about half of it. DOM3 is going to fill in a lot of these holes. However, because DOM3 is still just a working draft with little parser support, I chose to stick to DOM2 for the time being. JAXP can also plug a few of these holes.

■ JAXP

Starting with Java 1.4, Sun bundled the Crimson XML parser and the SAX2, DOM2, and TrAX APIs into the standard Java class library. (TrAX is an XSLT API that sits on top of XML APIs like SAX and DOM. We'll get to it in Chapter 17.) Sun also threw in a couple of factory classes, and called the whole thing the Java API for XML Processing (JAXP).

Crimson is a reasonably fast parser and worth your consideration. The fact that this implementation is bundled with Sun's virtual machine is a major plus. It allows you to distribute Java programs that use XML without having to add several megabytes of your own parser and interface classes. However, API-wise there isn't a whole lot new here. As I said before, when starting a new program you ask yourself whether you should choose SAX or DOM. You don't ask yourself whether you should use SAX or JAXP, or DOM or JAXP. SAX and DOM are part of JAXP. If you know SAX, DOM, and TrAX, you know 99 percent of JAXP.

The only public part of JAXP that isn't part of its component APIs are the factory classes in `javax.xml.parsers`. You can use these to create new documents in memory and load existing documents from text files and streams. You can also use the TrAX API to do some simple serialization by copying a document from a DOM object to a stream. Putting them all together, JAXP can replace most of the parser-dependent part of DOM. Example 5.6 is a JAXP client for the XML-RPC server. All the DOM standard code is the same as in Example 5.5. However, the parser-dependent parts from the `org.apache` packages have been replaced with JAXP classes.

Example 5.6 A JAXP-Based Client for the Fibonacci XML-RPC Server

```
import java.net.*;
import java.io.*;
import org.w3c.dom.*;
import javax.xml.parsers.*;
import javax.xml.transform.*;
import javax.xml.transform.stream.StreamResult;
import javax.xml.transform.dom.DOMSource;

public class FibonacciJAXPClient {

  private static String DEFAULT_SERVER
   = "http://www.elharo.com/fibonacci/XML-RPC";

  public static void main(String[] args) {

    if (args.length <= 0) {
      System.out.println(
       "Usage: java FibonacciJAXPClient number url"
      );
      return;
    }
```

```
String server = DEFAULT_SERVER;
if (args.length >= 2) server = args[1];

try {
  // Build the request document
  DocumentBuilderFactory builderFactory
   = DocumentBuilderFactory.newInstance();
  DocumentBuilder builder
   = builderFactory.newDocumentBuilder();
  Document request = builder.newDocument();

  Element methodCall = request.createElement("methodCall");
  request.appendChild(methodCall);

  Element methodName = request.createElement("methodName");
  Text text = request.createTextNode("calculateFibonacci");
  methodName.appendChild(text);
  methodCall.appendChild(methodName);

  Element params = request.createElement("params");
  methodCall.appendChild(params);

  Element param = request.createElement("param");
  params.appendChild(param);

  Element value = request.createElement("value");
  param.appendChild(value);

  // Had to break the naming convention here because of a
  // conflict with the Java keyword int
  Element intElement = request.createElement("int");
  Text index = request.createTextNode(args[0]);
  intElement.appendChild(index);
  value.appendChild(intElement);

  // Transmit the request document
  URL u = new URL(server);
  URLConnection uc = u.openConnection();
  HttpURLConnection connection = (HttpURLConnection) uc;
  connection.setDoOutput(true);
  connection.setDoInput(true);
```

```
        connection.setRequestMethod("POST");
        OutputStream out = connection.getOutputStream();

        TransformerFactory xformFactory
         = TransformerFactory.newInstance();
        Transformer idTransform = xformFactory.newTransformer();
        Source input = new DOMSource(request);
        Result output = new StreamResult(out);
        idTransform.transform(input, output);

        out.flush();
        out.close();

        // Read the response
        InputStream in = connection.getInputStream();
        Document response = builder.parse(in);
        in.close();
        connection.disconnect();

        NodeList doubles = response.getElementsByTagName("double");
        Node datum = doubles.item(0);
        Text result = (Text) datum.getFirstChild();
        System.out.println(result.getNodeValue());
      }
      catch (Exception e) {
        System.err.println(e);
      }

    }

  }
```

Once again, the request document is built as a tree. However, this time a `DocumentBuilderFactory` from JAXP does the building, instead of the Xerces-specific `DOMImplementationImpl` class. Unlike `DOMImplementation`, `DocumentBuilder` creates the `Document` without a root element. Consequently the next step is to create the root `methodCall` element and append it to the list of children in the `Document`. The next lines after this are the same as in the `FibonacciDOMClient`.

When it becomes time to serialize the `Document`, the JAXP solution once again diverges. Here, `FibonacciDOMClient` used Xerces-specific classes, whereas `FibonacciJAXPClient` uses TrAX. Specifically it creates a `Transformer` object

initialized to perform an identity transformation. It sets the source for the transformation to the original DOM request document and the result to the output stream connected to the server; and the document is transformed from one into the other. It's a little roundabout, but it works.

Finally, parsing the server response is much the same as before. However, this time instead of using the Xerces-specific `DOMParser` class, I used the same `DocumentBuilder` that created the request document. `DocumentBuilder` may delegate the parsing to Xerces anyway, depending on which classes are where in the class path, and how certain environment variables are set. However, there's no need for code at this level to know that implementation detail. If it becomes important to specify the parser for reasons of performance or conformance, you can set the `javax.xml.parsers.DocumentBuilderFactory` system property to the name of the factory class you want to load, which then chooses the parser class. If this property is not set, a reasonable default class is used (most likely `org.apache.crimson.jaxp.DocumentBuilderFactoryImpl` from Crimson, or `org.apache.xerces.jaxp.DocumentBuilderFactoryImpl` from Xerces).

The `javax.xml.parsers` package does fill a hole in DOM2. However, it's not as well designed as the similar classes that are coming down the road in DOM3. The `SAXBuilderFactory` class is completely redundant with the much better designed `XMLReaderFactory` class that's a standard part of SAX2. Frankly, `javax.xml.parsers` is just a legacy package dating from the days of SAX1 and DOM2, which did not provide this functionality. Forward-looking developers can ignore it. What JAXP really is, is a bundle of the standard APIs. However, it is not a new API itself, and it is not an alternative to SAX or DOM.

■ JDOM

A large portion of DOM's complexity and unintuitiveness derives from the fact that it's not actually a Java API. DOM is defined in the Interface Definition Language (IDL) from the Object Management Group (OMG) and then compiled to Java. This means that DOM must use interfaces and factory methods instead of classes and constructors. DOM classes must always cater to the lowest common denominator of programming languages. For example, there are no overloaded methods in DOM because JavaScript doesn't support method overloading. It doesn't matter that in Java and C++ a lot of the DOM could be expressed more concisely with overloaded methods.

JDOM is a more intuitive, tree-based API designed just for Java. It makes no compromises to support other languages. It includes Java-specific features such as `equals()` and `hashCode()` methods. Where appropriate it makes free use of standard Java classes such as `List` and `String`. Consequently JDOM feels a lot more

intuitive to most Java programmers than DOM does. Take a look at Example 5.7. I think you'll agree that it's a lot easier to comprehend at first glance than the DOM equivalent.

Example 5.7 A JDOM-Based Client for the Fibonacci XML-RPC Server

```
import java.net.*;
import java.io.*;
import org.jdom.*;
import org.jdom.output.XMLOutputter;
import org.jdom.input.SAXBuilder;

public class FibonacciJDOMClient {

  public final static String DEFAULT_SERVER
   = "http://www.elharo.com/fibonacci/XML-RPC";

  public static void main(String[] args) {

    if (args.length <= 0) {
      System.out.println(
       "Usage: java FibonacciJDOMClient number url"
      );
      return;
    }

    String server = DEFAULT_SERVER;
    if (args.length >= 2) server = args[1];

    try {
      // Build the request document
      Element methodCall = new Element("methodCall");
      Element methodName = new Element("methodName");
      methodName.setText("calculateFibonacci");
      methodCall.addContent(methodName);
      Element params = new Element("params");
      methodCall.addContent(params);
      Element param = new Element("param");
      params.addContent(param);
      Element value = new Element("value");
```

```
    param.addContent(value);
    // Had to break the naming convention here because of a
    // conflict with the Java keyword int
    Element intElement = new Element("int");
    intElement.setText(args[0]);
    value.addContent(intElement);
    Document request = new Document(methodCall);

    // Transmit the request document
    URL u = new URL(server);
    URLConnection uc = u.openConnection();
    HttpURLConnection connection = (HttpURLConnection) uc;
    connection.setDoOutput(true);
    connection.setDoInput(true);
    connection.setRequestMethod("POST");
    OutputStream out = connection.getOutputStream();

    XMLOutputter serializer = new XMLOutputter();
    serializer.output(request, out);

    out.flush();
    out.close();

    // Read the response
    InputStream in = connection.getInputStream();
    SAXBuilder parser = new SAXBuilder();
    Document response = parser.build(in);
    in.close();
    connection.disconnect();

    // Walk down the tree
    String result = response.getRootElement()
                    .getChild("params")
                    .getChild("param")
                    .getChild("value")
                    .getChild("double")
                    .getText();
    System.out.println(result);
  }
  catch (Exception e) {
    System.err.println(e);
```

```
        }

      }

    }
```

In JDOM the request document is built as a tree. Rather than factory methods, however, JDOM uses classes and constructors. Furthermore, it's possible to create elements that are not attached to any document. Elements and most other nodes are added to a document by their parent's `addContent()` method. However, if an element contains only text and no markup, then its entire content may be set by passing a string to its `setText()` method instead.

To serialize documents, JDOM provides the `XMLOutputter` class. This class combines formatting options such as encoding and the actual writing of the document onto a stream or writer. This is a standard part of JDOM that is used the same way no matter which parser you used underneath JDOM.

Once it has received the response, JDOM parses it into a `Document` object using the `SAXBuilder` class. This is the class that connects JDOM to an underlying parser that implements the SAX API. JDOM does not provide convenient methods for extracting subsets of the nodes in a document as DOM does. Consequently to get to the element we want, we must walk the tree. Because the document is very structured, this is straightforward. We get first the root element of the document, then its `params` child, then its `param` child, then its `value` child, then its `double` child, and finally its text content. The result is the string value we need. Although this string value happens to represent a double, JDOM has no way of knowing this (nor does any other XML API).

JDOM is definitely easier than DOM for simple problems like this. But I think it fails when it comes to more complex problems such as processing large narrative documents with lots of mixed content, or working with XPath-based technologies like XSLT, XML Pointer Language (XPointer), XInclude, and XQuery. JDOM often chooses simplicity and intuitiveness over power and completeness. The incomplete tree model really begins to hurt as the problems grow more complex. Any operation that requires you to walk the entire document tree recursively is more complex in JDOM than it is in DOM. JDOM is much better suited to data-oriented documents with well-known record-like structures that do not contain any mixed content.

■ dom4j

dom4j *[http://www.dom4j.org/]* began as a fork of the JDOM project by one developer, James Strachan, who disagreed with several aspects of the JDOM architecture, including the decision to use classes instead of interfaces. dom4j has since gone its own way to feature a very different approach to a pure-Java, tree-based API than either DOM or JDOM. Notable unique features in dom4j include integrated XPath and XSLT support and optional DOM compatibility (for example, you can wrap a dom4j `Element` inside a DOM `Element`).

Example 5.8 shows the dom4j version of our standard example. It's structured pretty much like the previous two tree-based APIs. First a `Document` object representing the XML-RPC request is constructed in memory, and then it is serialized onto the `URLConnection`'s output stream. Finally, the response document is parsed to find the desired value.

Example 5.8 A dom4j-Based Client for the Fibonacci XML-RPC Server

```
import java.net.*;
import java.io.*;
import org.dom4j.*;
import org.dom4j.io.*;


public class Fibonaccidom4jClient {

  public final static String DEFAULT_SERVER
   = "http://www.elharo.com/fibonacci/XML-RPC";

  public static void main(String[] args) {

    if (args.length <= 0) {
      System.out.println(
       "Usage: java Fibonaccidom4jClient number url"
      );
      return;
    }

    String server = DEFAULT_SERVER;
    if (args.length >= 2) server = args[1];
```

```
try {
  // Build request document
  Document request = DocumentHelper.createDocument();
  Element methodCall = request.addElement("methodCall");
  Element methodName = methodCall.addElement("methodName");
  methodName.addText("calculateFibonacci");
  Element params = methodCall.addElement("params");
  Element param = params.addElement("param");
  Element value = param.addElement("value");
  // Had to break the naming convention here because of a
  // conflict with the Java keyword int
  Element intElement = value.addElement("int");
  intElement.addText(args[0]);

  // Transmit the request document
  URL u = new URL(server);
  URLConnection uc = u.openConnection();
  HttpURLConnection connection = (HttpURLConnection) uc;
  connection.setDoOutput(true);
  connection.setDoInput(true);
  connection.setRequestMethod("POST");
  OutputStream out = connection.getOutputStream();

  XMLWriter serializer = new XMLWriter(out);
  serializer.write(request);

  out.flush();
  out.close();

  // Read the response
  InputStream in = connection.getInputStream();
  SAXReader reader = new SAXReader();
  Document response = reader.read(in);
  in.close();
  connection.disconnect();

  // Use XPath to find the element we want
  Node node = response.selectSingleNode(
   "/methodResponse/params/param/value/double"
  );
```

```
            String result = node.getStringValue();
            System.out.println(result);

          }
          catch (Exception e) {
            System.err.println(e);
          }

        }

      }
```

In dom4j the request `Document` object is initially created by the `DocumentHelper.createDocument()` factory method. The contents of the document are formed at the same time they're added to their parent, so that no nodes are ever orphaned from the tree. The `XMLWriter` class performs the serialization to the server.

Once dom4j receives the response, it parses it into a `Document` object using the `SAXReader` class that connects dom4j to an underlying parser such as Crimson or Xerces. XPath extracts the `double` element from this document. If you're comfortable with XPath, this is a very useful and convenient feature.

■ ElectricXML

The Mind Electric's ElectricXML *[http://www.themindelectric.com/exml/]* is yet another tree-based API for processing XML documents with Java. It has a reputation for being particularly easy to use. It's also small. The JAR archive for ElectricXML 4.0 is about 10 percent the size of the JAR archive for dom4j 1.3. Finally, the Mind Electric has claimed in the past that ElectricXML is much faster and more memory efficient than DOM and JDOM. However, their benchmarks were a little too scant and a little too contrived to convince me.

Example 5.9 shows this chapter's standard example program implemented in ElectricXML. As with the previous examples, the input number is read from the command line. This is wrapped up in an XML request document built in memory using the ElectricXML API, which is then serialized onto an output stream connected to the server. Finally the server's response document is read and parsed.

Example 5.9 An ElectricXML-Based Client for the Fibonacci XML-RPC Server

```
import java.net.*;
import java.io.*;
import electric.xml.*;


public class FibonacciElectricXMLClient {

  private static String defaultServer
   = "http://www.elharo.com/fibonacci/XML-RPC";

  public static void main(String[] args) {

    if (args.length <= 0) {
      System.out.println(
       "Usage: java FibonacciElectricXMLClient number url"
      );
      return;
    }

    String server = defaultServer;
    if (args.length >= 2) server = args[1];

    try {
      // Build request document
      Document request = new Document();
      request.setRoot("methodCall");
      Element methodCall = request.getRoot();
      Element methodName = methodCall.addElement("methodName");
      methodName.setText("calculateFibonacci");
      Element params = methodCall.addElement("params");
      Element param = params.addElement("param");
      Element value = param.addElement("value");
      // Had to break the naming convention here because of a
      // conflict with the Java keyword int
      Element intElement = value.addElement("int");
      intElement.setText(args[0]);
```

```
        // Transmit the request documentf
        URL u = new URL(server);
        URLConnection uc = u.openConnection();
        HttpURLConnection connection = (HttpURLConnection) uc;
        connection.setDoOutput(true);
        connection.setDoInput(true);
        connection.setRequestMethod("POST");
        OutputStream out = connection.getOutputStream();
        request.write(out);
        out.flush();
        out.close();

        // Read the response
        InputStream in = connection.getInputStream();
        Document response = new Document(in);
        in.close();
        connection.disconnect();

        // Walk down the tree
        String result = response.getRoot()
                         .getElement("params")
                         .getElement("param")
                         .getElement("value")
                         .getElement("double")
                         .getTextString();
        System.out.println(result);

      }
      catch (Exception e) {
        System.err.println(e);
      }

    }

}
```

By this point, you should be experiencing a sense of déjà vu. The creation of the XML-RPC request document is very similar to the way it's done in dom4j. Navigation of the response document to locate the `double` element is very similar to the way it's done in JDOM. There are only so many plausible ways to design a tree-based API for processing XML. The most original part of the ElectricXML API is

that no separate serializer class is used to write the document onto the `OutputStream`. Instead the `Document`, `Element` and other classes have a `write()` method. The `OutputStream` onto which the node will be serialized is passed to this method. Each node takes responsibility for its own serialization.

My major technical concern about ElectricXML is that it has achieved its reputation for ease of use primarily by catering to developers' preconceptions and prejudices about XML. In other words, the API is designed around what developers think the XML specification says, rather than what it actually does say. For example, ElectricXML routinely deletes white space in element content (sometimes mistakenly called ignorable white space), even though the XML specification explicitly states, "An XML processor must always pass all characters in a document that are not markup through to the application." The client application is free to ignore white space characters if it doesn't need them, just as it is free to ignore anything else it doesn't need; but the parser is not free to make that decision for the client application. Worse yet, the ElectricXML namespace model focuses on namespace prefixes rather than namespace URIs. This certainly matches how most developers expect namespaces to work, but it is not in fact how they do work. I agree that the XML's namespace syntax is needlessly complicated and confusing. Nonetheless, an XML API cannot fix the problem by pretending that namespaces are less complicated than they really are. ElectricXML may feel easier at first than more XML-compatible APIs such as SAX, DOM, and JDOM, but it's bound to cause more pain in the long run.

I also have one major nontechnical concern about ElectricXML. Whereas all of the other APIs discussed here are released as various forms of open source, ElectricXML is not. The license *[http://www.themindelectric.com/products/xml/licensing.html]* limits what you're allowed to do with the software, including preventing you from competing with it, thereby prohibiting necessary forks. Still, ElectricXML is free beer, and source code is provided.

XMLPULL

A *pull parser* works in streaming mode the way that SAX does. However, the client specifically requests the next chunk of data from the parser when it is ready for it, rather than taking it whenever the parser is ready to provide it. Furthermore, the client has a limited ability to filter out some normally insignificant pieces, such as comments and processing instructions. This can be as fast and as memory efficient as SAX parsing, and perhaps more familiar to many developers.

The developing standard API for pull parsing is known as XMLPULL *[http://www.xmlpull.org]*. This is currently implemented by two parsers, Enhydra's kXML2

[http://kxml.org/] and Aleksander Slominski's MXP1 *[http://www.extreme.indiana.edu/xgws/xsoap/xpp/mxp1/]*. The client application loads a parser-specific instance of the `XmlPullParser` interface. It sets the `InputStream` or `Reader` from which the parser will read the document. However, the parser does not actually read the document until the client calls the `next()` method. Then the parser reads just enough to report the next start-tag, text string, or end-tag from the document. (Actually, the parser may well read quite a bit more and buffer the result, but that's an implementation detail that's invisible to the client.) When the client has processed that unit of data, it asks the parser for the next unit. This continues until `next()` indicates that it's at the end of the document. Example 5.10 demonstrates with an XMLPULL client for the Fibonacci XML-RPC server.

Example 5.10 An XMLPULL-Based Client for the Fibonacci XML-RPC Server

```
import org.xmlpull.v1.*;
import java.io.*;
import java.net.*;


public class FibonacciPullClient {

  public final static String DEFAULT_SERVER
   = "http://www.elharo.com/fibonacci/XML-RPC";

  public static void main(String[] args) {

    if (args.length <= 0) {
      System.out.println(
       "Usage: java FibonacciPullClient number url"
      );
      return;
    }

    String server = DEFAULT_SERVER;
    if (args.length >= 2) server = args[1];

    try {
      // Connect to the server
      URL u = new URL(server);
      URLConnection uc = u.openConnection();
      HttpURLConnection connection = (HttpURLConnection) uc;
```

```
connection.setDoOutput(true);
connection.setDoInput(true);
connection.setRequestMethod("POST");
OutputStream out = connection.getOutputStream();
Writer wout = new OutputStreamWriter(out);

// Transmit the request XML document
wout.write("<?xml version=\"1.0\"?>\r\n");
wout.write("<methodCall>\r\n");
wout.write(
 "  <methodName>calculateFibonacci</methodName>\r\n");
wout.write("  <params>\r\n");
wout.write("    <param>\r\n");
wout.write("      <value><int>" + args[0]
 + "</int></value>\r\n");
wout.write("    </param>\r\n");
wout.write("  </params>\r\n");
wout.write("</methodCall>\r\n");

wout.flush();
wout.close();

// Load the parser
XmlPullParserFactory factory
 = XmlPullParserFactory.newInstance();
XmlPullParser parser = factory.newPullParser();

// Read the response XML document
InputStream in = connection.getInputStream();
parser.setInput(in, null);
//                  ^^^^
// Encoding is not known; the parser should guess it based
// on the content of the stream.

int event;
while (
 (event = parser.next()) != XmlPullParser.END_DOCUMENT) {
  if( event ==  XmlPullParser.START_TAG) {
    if ( "double".equals(parser.getName()) ) {
      String value = parser.nextText();
      System.out.println(value);
```

```
            } // end if
          } // end if
        } // end while

        in.close();
        connection.disconnect();

      }
      catch (Exception e) {
        System.err.println(e);
      }

    }

}
```

Like SAX, XMLPULL is a read-only API. Thus this program begins by sending the request to the server by writing strings onto an `OutputStream`, just as in the SAX client. The style for reading the response is quite new, however. The pull parser's `next()` method is repeatedly invoked. Each time it returns an `int` type code: `XmlPullParser.START_TAG`, `XmlPullParser.TEXT`, `XmlPullParser.END_TAG`, and so forth. The loop exits when the `next()` method returns `XmlPullParser.END_DOCUMENT`. Until that happens, we look at each start-tag returned to see if its name is `double`. If it is, then the program invokes `nextText()` to return the complete text content of that element as a `String`. This is printed on `System.out`.

■ Summary

Reading an XML document is a complicated, error-prone operation, which is why it's fortunate that you never have to write the code that does this. Instead you install an XML parser class library and only access the document through the parser's API. Numerous XML parsers for Java are available as free beer, open source, and payware, including the Apache XML Project's Xerces, Sun's Crimson, Enhydra's kXML, the GNU Classpath Extensions Project's Ælfred, and Yuval Oren's Piccolo. Most XML parsers implement one or more of the standard APIs such as SAX, DOM, and XMLPULL. As long as you only write code to the standard APIs, you can change the underlying parser with relatively little effort.

Which API you pick is largely a matter of personal taste and appropriateness for the problem at hand. SAX is an event-based API that presents the contents of a

document from beginning to end in the order that the parser encounters them. XMLPULL is a similar read-only, streaming API, but it's based on the Iterator design pattern rather than the Observer design pattern. That is, in XMLPULL the client must ask the parser for the next chunk of data, whereas in SAX the parser tells the client when it has a new chunk of data. Both SAX and XMLPULL can manipulate documents larger than available memory.

SAX and XMLPULL both work well for documents in which the processing is local; that is, where all the information the program needs at one time tends to be close together in the document. Programs that need random access to widely separated parts of the document are often better served by one of the tree-based APIs, such as DOM or JDOM. These often feel more natural to developers than XMLPULL and SAX, mostly because DOM models the XML document whereas while SAX and XMLPULL model the XML parser. And most developers are more interested in what's in the document than how the parser behaves. The flip side is that SAX and XMLPULL are much easier APIs for parser vendors to implement, so that some parsers implement SAX or XMLPULL but not DOM. JAXP bundles both SAX and DOM together so that they can become a standard part of the JDK. But JAXP isn't really a separate API in itself.

In addition to DOM, there are several alternative tree-based APIs that layer on top of an underlying parser. These include JDOM, ElectricXML, and dom4j. These all correct various perceived flaws in the DOM API and often seem easier to use to a Java developer with little background in XML. However, in my experience these APIs are much more limited than DOM and SAX and leave out some important information from an XML document that more complex applications need. Worse yet, they occasionally increase ease-of-use by catering to common preexisting developer misconceptions about XML syntax and grammar, and thus help propagate the mistaken beliefs. SAX and DOM are both much more in sync with the broad family of XML specifications from XML 1.0 on up. These days I try to stick with standard SAX and DOM for most of my programs.