Chapter 1

Java and XML: Joining Forces in Search of the Holy Grail

In This Chapter

- Why Java and XML work well together
- Java tools for XML processing: an overview
- The big move toward Web services

At my home, we celebrate every year on the tenth of May. That's because May 10, 1869 was the day of the "Golden Spike" — the driving of the final stake into the Transcontinental Railroad. The completion of this nineteenth century engineering miracle — connecting the two ends of the continent by rail — was an event of enormous importance.

Of course, miracles don't come easily. This miracle faced some serious stumbling blocks along the way. As late as April 1869, the two railroad companies (one working from the west coast, and the other from the east coast) were competing for big government subsidies. The top dogs in each company wanted to lay more track than the people from the other company. So the two companies didn't agree on a common meeting point for the tracks.

Both companies laid tracks along their preferred routes and, instead of meeting, the routes actually passed one another. Imagine two stretches of railroad tracks, covering an overall distance of three thousand miles, and then missing one another near Promontory, Utah. The companies' work crews were close enough to fight with one another while they both laid track over the same two-hundred-mile stretch.

At this point, you may be asking yourself what this railroad story has to do with Java and XML. Well, many of my stories travel long journeys before they turn around and go home. But this railroad story isn't such a stretch. For me, the story illustrates the most blatant lack of standards in history.

It wasn't the absence of insight that kept these people from setting a standard. Quite the contrary, the whole point of the endeavor was to join the tracks. It wasn't a lack of communication, or a shortage of funds. These people simply refused to agree. And because they refused, their workers laid two hundred miles of parallel track. What a waste!

Well, if this story has a point, it's that standards are important. (Notice that I said "if.") Without standards, the modern world would be a no-tech zone. We'd be lighting candles because Brand A's bulbs didn't fit Brand B's sockets. And, if we could muster the know-how to build two computers, then one computer's software wouldn't communicate with the other computer's software.

That's good. I've finally reached a relevant point. Software standards are important. That's what a book on Java and XML is all about.

Why Software Doesn't Work

Long ago, in a mythical village named Philadelphia, some people named Mauchly and Eckert developed the first general-purpose, all-electronic computer. The computer weighed over 30 tons, and consumed enough electrical power to run 150 present-day homes. Some folks say that inventing a computer was the easy part. The hard part (a job that consumed peoples' time and energy for the rest of the millennium) was creating software that runs effectively on this modern marvel.

I don't want to bore you with historical details. (Well, actually, I'd *like* to bore you with 'em, but my editor won't let me!) Anyway, the problem with software is threefold:

- **There's always more than one way to express a solution to a programming problem.**

  Forget "skinning the cat" — how do you give someone instructions to clean a cat box? You could say, "First scoop the litter, then add more litter." You would start with imperative statements.

  But when you write a computer program, you don't necessarily start with imperative statements. In Java, for example, you start by constructing the *objects* — as in, "Create instances of the `CatBox` and `CatLitter` classes" — before you tell it what to do with them. Such is the approach used in object-oriented programming.

  It's taken decades for the world to understand the benefits of object-oriented programming. We've seen FORTRAN, C, and many other languages get pushed into the background as newer, more usable languages come to the fore. The ultimate challenge — finding the *best* way to issue instructions — is a never-ending battle. It's one developer against another, one company against another, one passionate believer against another.

- **Software is virtual, not physical.**

  That means you can build, rework, replace, and destroy it at lightning speed. If we could build bridges as quickly as we build computer programs, then we'd have billions of miles of shore-to-shore spans. The average hobbyist would build a bridge in minutes — and most of the world's bridges would be completely unusable — a vast tangle of kludges.

  Easily built structures tend to be complex and unwieldy. When you work hard to piece together a physical system (a real bridge, for instance), you're constrained by nature to make the system as simple and (we hope) reliable as possible. But when you piece together a large system with no physical effort, the system can sprout complexities till you don't fully understand its parts. Interactions among the parts become less predictable, the system becomes unreliable, and that's why so much software is so brittle.

- **Without rigorous standards, software isn't useful in more than one context.**

  In the olden days, every computer had its own, private version of FORTRAN or COBOL. This didn't work well, because one person's program couldn't run on another person's computer.

  Along came standard FORTRAN and standard COBOL, and the software world rolled along nicely for a while. But then other languages came on the scene and muddled it up again. Your machine ran only FORTRAN or C, while my machine ran only Algol, PL/1, or COBOL. If you wanted to share a FORTRAN program with me, then I'd have to spend time and money configuring a FORTRAN compiler to run on my computer. Of course, no one worried too much about this problem because the notion of sharing code seemed like sharing profits with competitors — not a very popular idea.

  These days we have the Internet, and open-source software is all the rage; everyone wants to run everyone else's code. The more you share, the more your paid services become desirable. But with half a million programmers in the United States alone, each one writing code in his or her own way, how do you coordinate all that programming activity?

At the start of the third millennium, we've found partial solutions to each of these problems. Here's what we know so far:

- **On expressing a programming solution:**

  Some programming languages are better than others. Some ways of representing data are better than others. For many applications, object-oriented programming is better than straight procedural programming. Everyone has his or her favorite language, and everyone has his or her favorite programming style. While no one is rational and unbiased, everyone is ready to fight for his or her choice.

- **On software's being virtual, complex, and generally unreliable:**

  The people who study software engineering have some clever tactics for tackling this problem. According to these folks, the answer lies in a standardized discipline for analyzing, designing, coding, and maintaining a software project, with attention to every detail in the software life cycle.

  The promise of software engineering has had mixed success. Some people swear by it, some cast asparagus on it, and others ignore it. One way or another, software engineering is a worthwhile and noble effort.

- **On standards and portability:**

  As offensive as it may be to the freethinkers of this world, experience has shown that industry-wide standards make life easier.

  If everyone uses a standard programming language, then your code has a fighting chance of running on my computer, and vice versa; code has become *portable* from platform to platform. Maybe you want to sell your code to me, or maybe you realize that, when I run your code, I can build on top of it, make enhancements to it, and eventually make your code more salable.

  If everyone uses a standard format for representing data, then my programs can read your data, and your programs can read mine — the data has become portable. Sure, this situation isn't always desirable, but with sharable data, the potential benefit is mind-boggling. Share product information with visitors to your Web site. Share the terms of a financial agreement with partners and clients. Share tidbits with friends using Comics Markup Language.* The possibilities are endless.

Portable Code and Portable Data

Java is portable, XML is portable — so what's the big deal? Well, in the realm of portability, the chain is as strong as its weakest link. Take, for instance, the following scenario:

Your company forms a partnership with Joe's Hardware — a company with roughly five thousand employees. The two companies' products complement one another, so an increase in either company's sales will benefit the other company. For as long as you've been in business (two whole years, including the dot-com downturn), you've been storing your data in your company's homegrown format. Now it's time to share data with your partner Joe.

Fortunately, your homegrown data format is based on XML. Getting Joe's computers to read your data is easy. You just juggle some XML elements.

*For information on ComicsML, visit `www.xml.com/pub/a/2001/04/18/comicsml.html`.

Anyway, you and Joe agree on a procedure for transforming data. You send representatives to work on a pair of transformation programs. One program transforms your data into an intermediate format, and then another program transforms the intermediate data into Joe's format. The intermediate format is encrypted for safe passage across public Internet lines. (The encryption process is a resource hog, but it's a necessary component to ensure the data's safety.)

Your transformation program is written in Java for a Unix machine, and Joe's program is written in C++ for a Windows machine. That's okay, because each program runs in its own environment.

Everything is rosy until Joe sends you a memo. The memo reads as follows:

To: You, CEO

Your High-Rolling Company

From: Joe

Your Not-So-Esteemed Partner

My assets are sinking,

Of mergers I'm thinking.

A marriage! Yes that would be nice.

With champagne a flowing,

And bridesmaids all glowing,

And shareholders throwing the rice.

Sincerely,

Poor, Undernourished Joe

Suddenly, the bits hit the fan. You'll have one company, with one information technology department, and one set of computers. The combined company will run your Unix-based computers. But what about all that data? Remember the costly intermediate format used to transfer encrypted information across the Internet? All the data will be transferred internally. You can save a bundle by eliminating that transfer step. (Two can live cheaper than one. That's what mergers are for.)

But the two programs to blend the companies' XML formats aren't compatible. Your code is written in Java; Joe's code is written in C++. Melding the two programs to eliminate the bottleneck in the middle will be a living, breathing nightmare. Try compiling the Windows C++ code on your Unix machine. What do you get? You get one warning after another. Try running the C++ code on your Unix machine. What do you get?

```
Bus error (core dumped)
```

What an unpleasant situation! What was once beneficial data compatibility has turned into a long-term maintenance headache. And why did things turn out so badly? Because Joe's C++ code isn't portable, that's why. Joe's data is portable, but his code isn't.

Consider the facts

In case you're not convinced that both XML and Java enjoy cross-platform portability, look over these facts about the two technologies:

- **In its brief lifetime, XML has become the worldwide standard for representing structured, self-describing data.**

  The XML registry, housed at `www.xml.org`, lists over one hundred XML data formats. They include formats for financial data, healthcare, arts and entertainment, human resources, multimedia, and many other domains. The XML standard encapsulates almost any kind of data in a way that's flexible, extensible, and easy to maintain.

- **Java runs as bytecode on a virtual machine.**

  A "compiled" Java class file that runs on Windows will run the same way on Linux, on Windows, or on whatever platform supports the Java Virtual Machine.

  With Java, there's no such thing as platform-specific code. When you go from a `.java` source file to a `.class` bytecode file, you don't lose portability. To run the `.class` file, all you need is an operating system that can support a Java Virtual Machine. And versions of the Java Virtual Machine are available for at least twenty different operating systems.

- **Java is based on object-oriented programming technology.**

  Java code is reusable. You can call methods from existing classes, extend classes, or stretch and bend classes to meet your specialized needs. If someone writes a wonderful XML-handling package in Java, and the package has bits and pieces that you can use in your own work, you can import the package and extend the classes to solve exactly the problems that you need to solve.

  This cooperative model works both ways. When you create a package for your own anticipated needs, other developers can adopt your package, enhance your package, and spread the good word about your code.

Taken together, these factors eventually ensure that software written in one environment can run in all other environments. Instead of reinventing the wheel, programmers reuse the wheel. This ideal — the seamless integration of parts from many sources to build large, reliable software systems — has been the Holy Grail of computing for the past several decades. Now portable code and portable data put the ideal within reach.

The partnership between Java and XML

Java and XML work well together. Taken together, Java and XML form the virtual equivalent of a well-oiled machine. Why do I say this?

Well, for starters, much of the code created for processing XML *is written in Java.* I have no hard statistics to prove this, but I visited the utilities page at `www.xmlsoftware.com`. On that page, I found references to 79 utilities, of which 10 were written in C++, 7 were written in Python, 6 were written in Perl, and 9 were written in other non-Java languages. A whopping 47 utilities were written in Java. Clearly the XML developer community has an investment in Java — for many good reasons, of which the likely best one is that both Java and XML are streamlined for the Internet.

Since its humble beginnings in the 1990s, Java has been an Internet-ready language. When it first hit the scene, Java was viewed primarily as a tool for building applets and other Web-client applications. Java's core *API* (*Application Programming Interface*) included a package named `java.net`. This package contained support for URLs, sockets, authentication, and other necessities of network coding.

As time went on, people saw more and more uses for server-side Java.

- The first big push came in 1997, when Sun released the Java Servlet API. With a servlet, you respond dynamically to a request for your Web site's services. (For instance, you can build a customized Web page on the fly to accommodate a particular visitor's needs.)
- In 1998, Sun Microsystems started developing the JavaServer Pages specifications. With JavaServer Pages, you create a Web page that includes both HTML tags and Java program logic.
- In 1999, Sun announced support for JavaServer Pages as part of the ever-popular Apache Web server.

XML was developed (in part) to address the weaknesses of HTML, the lingua franca of the Internet. The whole push for XML has been based on the desirability of sharing of data. Company A's software examines the data made public by Company B. Company A's software can read Company B's data because the data is stored in an XML document. The infrastructure for the exchange of data becomes the entire Internet.

Starting with version 1.4, Java's core API includes packages devoted exclusively to the processing of XML documents. These packages help solidify the bond between Java and XML.

Java Tools for XML Processing

In this section, you get a ten-cent tour. The tour includes descriptions of several useful Java APIs. Each API is freely available for download and use. (Most of them can be downloaded from `java.sun.com/xml`.)

- **JAXP: the Java API for XML Processing**

  The name JAXP is a catchall term for several of Java's XML tools that form the backbone of the Java XML strategy. JAXP is the most mature of all the toolsets described in this book.

  JAXP is actually a collection of APIs — in particular, SAX, DOM, and XSLT.

  - **SAX: The Simple API for XML**

    SAX represents a general-purpose approach to the handling of XML documents. Using SAX, you can do almost anything with an existing XML document, because the SAX approach to XML is very low-level. (This API has a nickname. It's QDAX — the Quick-and-Dirty API for XML.)

    SAX views an XML document as a sequence of tags. You assign an action to each kind of tag, and perform the appropriate action for each tag in a document. This tag-hunting approach makes SAX a real speed demon. If you have lots of work to do, you can count on SAX to do the work in record time.

    You'll find material on SAX in Chapters 3 through 6.

  - **DOM: The Document Object Model**

    Like SAX, the DOM API is an all-purpose tool. You can perform almost any XML task with DOM, because DOM isn't targeted toward specific XML applications.

    DOM doesn't view a document as a collection of tags. Instead, DOM works with XML *elements*. To do this, a DOM program makes a big copy of a document, and stores the copy in the computer's memory. So DOM isn't fast. If you run DOM on a large XML document, then the run takes a long time. That's the price you pay for dealing with elements instead of tags.

    In this book, Chapters 7 through 9 cover the DOM API.

  - **XSLT: Extensible Stylesheet Language Transformations**

    With XSLT, you can turn any XML document into almost any other form. You don't have to get lost in procedural (do-this-and-then-do-that) code. Instead, you create templates. If part of a document matches a template, then the computer uses that part of the document to compose its output.

    In this book, I cover XSLT in Chapters 12 and 13.

You can get JAXP on your computer in one of two ways:

- You can visit `java.sun.com/xml/jaxp` and download the JAXP API on its own.
- You can get JAXP as part of the Java 2 Platform Standard Edition, version 1.4 or later. The last time I looked, this was a 40MB download — but it's worth every byte. The link to the download is at `java.sun.com/j2se`.

SAX, DOM, and XSLT are all included in Sun's JAXP pack. The next two items are not:

- **JDOM: The Java Document Object Model**

  Neither SAX nor DOM is specific to Java. You can write DOM programs in C++, in Perl, and in a number of other programming languages. When you write DOM programs in Java, the core Java API gets rusty from disuse. DOM is all things to all languages, so DOM can be awkward and cumbersome to use.

  To remedy this and other DOM shortcomings, two guys named Jason Hunter and Brett McLaughlin created JDOM. In many respects, JDOM is a reworking of the ideas behind DOM. The big difference is, JDOM takes full advantage of the power of Java, and uses a sleek intuitive tree structure that's missing from DOM.

  Yes, JDOM is streamlined for Java. But no, JDOM isn't part of Sun's Java toolset. Instead, you download JDOM by visiting `www.jdom.org`. (To learn about JDOM, visit Chapters 10 and 11 in this book.)
- **JAXB: The Java API for XML Binding**

  The first time I saw some JAXB code, I was jealous. Why didn't I think of that? With JAXB, you take an XML document, and turn it into a Java class. If the document has a `Sale` element, then your Java code can have a `Sale` class. If the `Sale` element has a `quantity` attribute, then the `Sale` class has methods `getQuantity` and `setQuantity`. What could be simpler?

  JAXB is part of Sun's XML suite, but it's not currently bundled with JAXP or with the Java 2 core API. So, to get JAXB, you have to do a separate download. The home page for JAXB is `java.sun.com/xml/jaxb`. For your reading pleasure, I cover JAXB in Chapters 14 and 15.

I have three more APIs to tell you about. But first, I have to introduce you to the star of the show — Web services.

Web Services (Hot Stuff)

The Internet era is marked by several stiff competitions, all going on at roughly the same time. One competition was called the "Browser War." It was

Netscape versus Microsoft, and everyone knows who won. Later came the brawl between Microsoft and the U.S. Department of Justice. This brawl raised the possibility that Microsoft holds an unfair monopoly in the software market. (Well, that issue seems to have been settled.) These days, another competition is vying for center stage — the scramble for pieces of the Web-services-software pie — potentially a big, big deal.

So what's the fuss all about? What are Web services, and why do they concern XML hounds like you and me? Well, the answer is illustrated in Figure 1-1.

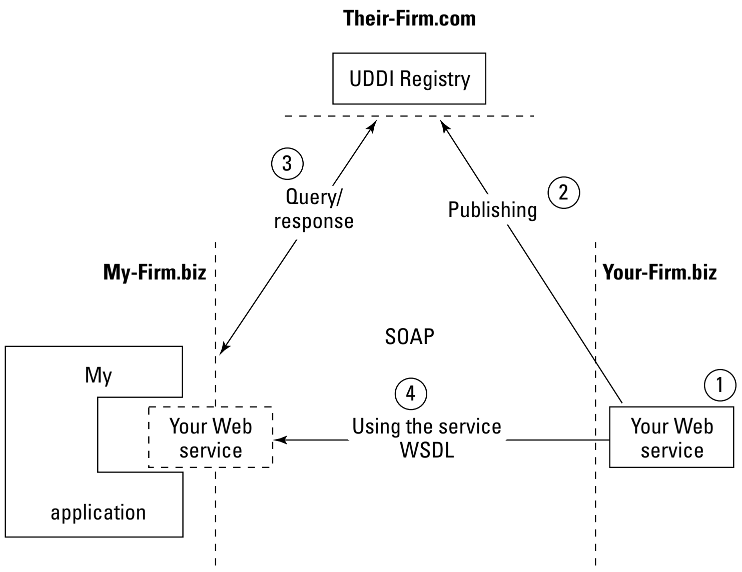

Figure 1-1: The life cycle of a Web service.

Of course, I have a story to go with Figure 1-1.

Your company creates software

You write a useful piece of software. (See the bubble numbered 1 in Figure 1-1.) Your software is so useful that you decide to offer it to others. (You offer it for a fee.)

You want to tell the world about the availability of your software. How can you do that? Well, you can take an ad out in the local newspaper, create a commercial for night-owl TV, or wear a sandwich board around town. But it's better to put information about your company and its software on the Internet. A place to deposit such information is called a *registry*.

How do you get information about your company into a registry? Once again, there are clunky alternatives. You can write a nice letter to the registry manager, or spend hours online filling out cumbersome Web-based forms. But the best procedure is one that's completely automated. You need a standardized protocol for describing things about your company. You use the protocol to describe your company and its software. Then you automate the sending of the description to the registry. (This sending of information is called *publishing* to the registry. If you're following along in Figure 1-1, we're up to the bubble numbered 2.)

Well, there just so happens to be a standard protocol for describing companies and their services. That protocol is named *UDDI* (*Universal Description, Discovery, and Integration*). So now, this thing that we've been calling a "registry" has a more specific name. It's called a *UDDI registry*. In case you haven't guessed, UDDI is based on XML.

In fact, at this point in the story, XML is useful on two different layers. On an upper layer, you describe your company and its services with XML-based UDDI. On a lower layer, you need a standard for transporting this UDDI document to a registry on a remote computer. That standard has to work with Windows, Unix, Java, C#, Fred's Private Programming Language, or anything else that's at the other end of the Internet pipe.

Well, you're in luck. There's a standard for sending information, and it's called *SOAP* — the *Simple Object Access Protocol*. Of course, SOAP is based on XML. The idea behind SOAP is to wrap a message in an XML envelope. Then you send this envelope across the Internet, and have the receiver unwrap the message. With the SOAP standard, a Java program on a Windows computer can send stuff to a Perl program on a Linux box. The underlying platform is irrelevant.

My company gets wind of your software

You've published information on a UDDI registry. Now you wait while others visit the registry. In the meantime, my firm develops a very specialized need. I've developed software that has a big, fat gap in the middle of it. I suspect that some pluggable components can fill that gap. (For instance, I may have a Web site that's crying out for a currency converter. I know there must be currency converters that I can use.)

To help me fill the gap, I send an automated query to the UDDI registry. This query can be automated because the query/response mechanisms are part of the UDDI specification. Anyway, I send a query to the UDDI registry and what do I get as part of the response? I get information about your software. (See step 3 in Figure 1-1.)

In an ideal world, I would skip immediately to the next automated step. But in the real world, I probably hold six or seven boring meetings. My associates compare alternatives, schmooze with people over lunch, and spend money on expensive consultants. When all is said and done, we decide to plug your software into my system. This is step 4 in Figure 1-1.

By now, it's safe to stop calling your stuff "software," and start calling your stuff a *Web service*. After all, you're making your stuff available through Web-based protocols, and my firm will avail itself of your firm's service.

So to make your service work with my software, we use XML tools. One of the tools is called *WSDL* (the *Web Services Description Language*). WSDL is an XML standard for describing the way in which your software can get called into action by my software. (Like UDDI, a WSDL description draws a picture of the service that you're offering. But unlike UDDI, the WSDL description goes into detail about under-the-hood software interfaces.)

On the registry, you've posted a WSDL document describing the technical details of your Web service. My software can download the WSDL document, and decide on its own how to mesh with your software. Because my software configures itself, fewer people need spend hours reinventing wheels.

I use your software

The day for deployment has finally arrived. To use your service on my Web site, I have to call your class's methods. I call methods that run on your computer, and the end result is no different from calling methods on my own computer. This is bubble 4 in Figure 1-1. It's where your Web service plugs seamlessly into my company's application.

As usual, the process needs rules and regulations. Exactly how do these method calls work? The answer brings us back to SOAP. With SOAP, we transport UDDI documents, WSDL documents, method calls, and many other things. So, you see, it all comes down to SOAP. And SOAP is an incarnation of XML.

That's the Web services story in 1000 words or less. That's what all the fuss is about.

More Java Tools

Earlier in this chapter, I described some Java APIs for use with XML. Now, with a basic understanding of Web services, you're ready to read about three additional APIs.

- **JAXM: The Java API for XML Messaging**

  A SOAP message is an ordinary XML document with some special demands on the kinds of elements that it contains. For instance, a SOAP message has `envelope` and `body` elements. So some people working with SOAP messages asked themselves an important question. "Why don't we create an API that has classes and methods just for `envelope` and `body` elements?" As a result, they created JAXM, with classes and methods that target all the special features of a SOAP document.

  Sun's Web page for JAXM is `java.sun.com/xml/jaxm`. I cover the JAXM API in Chapter 16 of this book.

- **JAXR: The Java API for XML Registries**

  Let's face it. Everything in this world is becoming highly specialized. We have pills to make us happy, pills to make us happy without making us drowsy, and pills to make us happily drowsy (without making us drowsily happy).

  Well, the Java tools for XML are also becoming specialized. Take, for instance, the API for XML Registries. With this API, you do some of the things shown in Figure 1-1. You publish services on registries, and you query the registries. In essence, you send SOAP messages, but your program doesn't have to concern itself with envelopes and bodies. In fact, you can write JAXR programs without knowing squat about SOAP elements. With JAXR, you call methods named `findOrganizations` and `createService`. The JAXR API composes SOAP messages behind the scenes.

  You can read the official story about JAXR by visiting `java.sun.com/xml/jaxr`. You can read the unofficial story by visiting Chapter 17 in this book.

- **JAX-RPC: The Java API for XML-based Remote Procedure Calls**

  To bring a published Web service into my own software environment, I call procedures that live somewhere else on the Internet. And these days, the gold standard for reaching another place on the Internet is to communicate using SOAP. If I need prices, weather information, or song titles, then I use SOAP to reach out and call your `getPrice`, `getRainfall`, or `getSongTitle` methods.

  I can compose SOAP messages from scratch, but it's better to use an API that can call remote methods on my behalf. That API is called *JAX-RPC*, and it's described in Chapter 18. (The home page for this API is `java.sun.com/xml/jaxrpc`.)

I've made a decision: When I finish this book, I'm going to retire my computer's "J" and "X" keys. I'll work with pencil and paper to create something the world really needs — a *triply*-nested acronym. I'm thinking about integrating SAX and DOM. It'll be called "DSDA," which stands for "Dummies SAX and DOM API," which stands for "Dummies Simple API for XML and Document Object Model API," which stands for "Dummies Simple Application Programming Interface for Extensible Markup Language and Document Object Model Application Programming Interface." People will call it the "DSDA API" (right before they start gibbering helplessly) — but I'll remind them that the "A" in "DSDA" already stands for "API," and that'll probably put 'em over the edge. After all, nature abhors redundancy.