All about JAXP

Sun's Java API for XML parsing

Brett McLaughlin (brett@newInstance.com)
Enhydra Strategist, Lutris Technologies
November 2000

This close examination of JAXP, Sun's Java API for XML, helps clear up the confusion about the specific nature of JAXP and the purpose it serves. This article covers basic concepts of JAXP, demonstrates why it is needed in the XML parsing space, and shows how the parser used by JAXP can easily be changed. It also drills down into SAX and DOM, two popular JAXP-related Java and XML APIs.

Java and XML have made news in every area of technology and are arguably the most important developments of 1999 and 2000 for software developers. As a result, the number of Java and XML APIs has proliferated. The two most popular, DOM and SAX, have generated a tremendous amount of interest, while JDOM and data-binding APIs have followed. Understanding even one or two of these thoroughly is quite a task; using all of them correctly makes you quite a guru. But in the last year, another API has made a big impression: Sun's Java API for XML, more commonly known as JAXP. This development isn't too surprising considering Sun didn't have any XML-specific offerings for its platform. What is surprising is the lack of understanding about JAXP. Most developers who use it have a misconception about the very API they are depending on.

What's a JAXP?

In this article I assume basic knowledge of SAX and DOM. There simply isn't enough room here to explain SAX, DOM, and JAXP. If you're new to XML parsing, you may want to read up on SAX and DOM through online sources or skim through my book. (Links to the APIs and my book are available in the Resources section.) This article will make a lot more sense once you've picked up the basics.

API or abstraction?

Before diving into code, it's important to cover some basic concepts. Strictly speaking, JAXP is an API, but it is more accurately called an abstraction layer. It does not provide a new means of parsing XML, add to SAX or DOM, or provide new functionality to Java and XML handling. (If you're in disbelief at this point, you're reading the right article!) Instead, it makes it easier to deal with some difficult tasks with DOM and SAX. It also makes it possible to handle some vendor-specific tasks that are encountered when using the DOM and SAX APIs in a vendor-neutral way.

While I'll go through all of these features individually, the thing you really need to get a handle on is that JAXP *does not provide parsing functionality*! Without SAX, DOM, or another XML parsing API, you *cannot parse XML*. I have seen many requests for a comparison of DOM, SAX, or JDOM to JAXP. Making these comparisons is impossible because the first three APIs serve a completely different purpose than JAXP. SAX, DOM, and JDOM all parse XML. JAXP provides a means to get to these parsers and results. It doesn't offer a new way to parse the document itself. This is a critical distinction to make if you're going to use JAXP correctly. It will also most likely put you miles ahead of many of your fellow XML developers.

If you're still dubious (or think I'm on something) download the JAXP distribution from Sun's Web site (see the Resources section), and you'll get an idea of how basic JAXP is. In the included jar (`jaxp.jar`), you will find *only six classes*! How hard could this API be? All of these classes (part of the `javax.xml.parsers` package) sit on top of an existing parser. And two of these classes are for error handling. JAXP is a lot simpler than people think. So why all the confusion?

Sun's JAXP and Sun's parser

Sun's parser is included with the JAXP download. The parser classes are all in the `parser.jar` archive as part of the `com.sun.xml.parser` package and related subpackages. You should know that this parser (code-named Crimson) is *not* part of JAXP. It is part of the JAXP *distribution*, but it is not part of the JAXP *API*. Confusing? A little bit. Think about it this way: JDOM ships with the Apache Xerces parser. That parser isn't part of JDOM but is used by JDOM, so it's included to ensure that JDOM is usable "out of the box." The same principle applies for JAXP, but it isn't as clearly publicized: JAXP comes with Sun's parser so it can be used immediately. However, many people refer to the classes included in Sun's parser as part of the JAXP API itself. For example, a common question on newsgroups is, "How can I use the `XMLDocument` class that comes with JAXP? What is its purpose?" The answer is somewhat complicated.

First, the `com.sun.xml.tree.XMLDocument` class is not part of JAXP. It is part of Sun's parser. So the question is misleading from the start. Second, the whole point of JAXP is to provide vendor independence when dealing with parsers. The same code, using JAXP, could be used with Sun's XML parser, Apache's Xerces XML parser, and Oracle's XML parser. Using a Sun-specific class, then, is a bad idea. It violates the entire point of using JAXP. Are you starting to see how this subject has gotten muddied? The parser and the API in the JAXP distribution (at least the one from Sun) have been lumped together, and developers mistake classes and features from one as part of the other, and vice versa.

The old and the new

A final note about JAXP that you don't want to miss: Using JAXP comes with some pitfalls. For example, JAXP only supports SAX 1.0 and DOM Level 1. SAX 2.0 has been out in its final form since May 2000, and DOM Level 2 support has been in most parsers for even longer. DOM Level 2 is not yet final, but it is certainly stable enough for production use. The new versions of both APIs have significant improvements, most notably support for XML namespaces. This support also enables XML Schema validation, which is another hot technology related to XML. In fairness to Sun, neither SAX 2.0 nor DOM Level 1 were final when JAXP went out as a 1.0 final release. However, the failure to include the new versions has resulted in a significant drawback for developers.

JAXP is still usable, but you may want to wait for JAXP 1.1, which supports SAX 2.0 and DOM Level 2. Otherwise, you may find that the advantages JAXP provides come at the expense of functionality in SAX and DOM's latest versions, and make your applications more difficult to code. Regardless of whether you wait for the next release of JAXP, be aware of this problem. You can also run into issues with classpaths if you mix JAXP with parsers that support DOM and SAX versions greater than JAXP's support. So consider yourself forewarned, and upgrade to JAXP 1.1 as soon as it's available. With this basic understanding of JAXP, let's look at the APIs that JAXP depends on: SAX and DOM.

Starting with SAX

SAX, the Simple API for XML, is an event-driven methodology for processing XML. It's basically made up of lots of callbacks. For example, the `startElement()` callback is invoked every time a SAX parser comes across the opening tag of an element. The `characters()` callback is called for character data, and then `endElement()` is called for the end tag of the element. Many more callbacks are present for document processing, errors, and other lexical structures. You get the idea. The SAX programmer implements one of the SAX interfaces that defines these callbacks. SAX also provides a class called `HandlerBase` that implements all of these callbacks and provides default, empty implementations of all the callback methods. (I mention this because it will be important in DOM, which we look at next.) The SAX developer needs only extend this class, then implement methods that require insertion of specific logic. So the key in SAX is to provide code for these various callbacks, then allow a parser to trigger each of these callbacks when appropriate.

So here's the typical SAX rundown:

- Create a `SAXParser` instance using a specific vendor's parser implementation
- Register callback implementations (by using a class that extends `HandlerBase`, for example)
- Start parsing and sit back as your callback implementations are fired off

The SAX component of JAXP provides a simple means to do all of this. Without JAXP, a SAX parser instance either must be instantiated directly from a vendor class (such as `org.apache.xerces.parsers.SAXParser`), or it must use a SAX helper class called `ParserFactory`. The problem with the first methodology is obvious; it isn't vendor-neutral. The problem with the second is that the factory requires, as an argument, the String name of the parser class to use (that Apache class, `org.apache.xerces.parsers.SAXParser`, again). You can change the parser by passing in a different parser class as a `String`. With this approach, you wouldn't need to change any import statements, but you would still have to recompile the class. This is obviously not a best-case solution. It would be much easier to be able to change parsers *without* recompiling the class, wouldn't it?

JAXP provides that better alternative: It allows you to provide a parser as a Java system property. Of course, when you download a distribution from Sun, you get a JAXP implementation that uses Sun's parser. The same JAXP interfaces, but with an implementation built on Apache Xerces, can be downloaded from the Apache XML Web site. Therefore (in either case), changing the parser you are using requires you change a classpath setting, moving from one parser implementation to another, but it does *not* require code recompilation. And this is the magic, the abstraction, that JAXP is all about.

A look at the SAX parser factory
The JAXP `SAXParserFactory` class is the key to being able to easily change parser implementations. You must create a new instance of this class (which I'll look at in a moment). After the new instance is created, the factory provides a method to obtain a SAX-capable parser. Behind the scenes, the JAXP implementation takes care of the vendor-dependent code, keeping your code happily unpolluted. This factory provides some other nice features, as well.

In addition to the basic job of creating instances of SAX parsers, the factory allows configuration options to be set. These options affect all parser instances obtained through the factory. The two options available in JAXP 1.0 are to set namespace awareness (`setNamespaceAware` (boolean awareness)), and to turn on validation (`setValidating` (boolean validating)). Remember that once these options are set, they affect *all* instances obtained from the factory after the method invocation.

Once you have set up the factory, invoking `newSAXParser()` returns a ready-to-use instance of the JAXP `SAXParser` class. This class wraps an underlying SAX parser (an instance of the SAX class `org.xml.sax.Parser`). It also protects you from using any vendor-specific additions to the parser class. (Remember our earlier discussion about the `XmlDocument` class?) This class allows actual parsing behavior to be kicked off. The following listing shows how a SAX factory can be created, configured, and used.

Listing 1. Using the SAXParserFactory

```
import java.io.File;
import java.io.IOException;
import java.io.OutputStreamWriter;
import java.io.Writer;
// JAXP
import javax.xml.parsers.FactoryConfigurationError;
import javax.xml.parsers.ParserConfigurationException;
import javax.xml.parsers.SAXParserFactory;
import javax.xml.parsers.SAXParser;
// SAX
import org.xml.sax.AttributeList;
import org.xml.sax.HandlerBase;
import org.xml.sax.SAXException;
public class TestSAXParsing {
    public static void main(String[] args) {
        try {
            if (args.length != 1) {
                System.err.println ("Usage: java TestSAXParsing [filename]");
                System.exit (1);
            }
            // Get SAX Parser Factory
            SAXParserFactory factory = SAXParserFactory.newInstance();
            // Turn on validation, and turn off namespaces
            factory.setValidating(true);
            factory.setNamespaceAware(false);
            SAXParser parser = factory.newSAXParser();
            parser.parse(new File(args[0]), new MyHandler());
        } catch (ParserConfigurationException e) {
            System.out.println("The underlying parser does not support " +
                               " the requested features.");
        } catch (FactoryConfigurationError e) {
            System.out.println("Error occurred obtaining SAX Parser Factory.");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
class MyHandler extends HandlerBase {
    // SAX callback implementations from DocumentHandler, ErrorHandler, etc.
}
```

Notice in this code that two JAXP-specific problems can occur in using the factory: the inability to obtain or configure a SAX factory, and the inability to configure a SAX parser. The first of these problems, `FactoryConfigurationError`, usually occurs when the parser specified in a JAXP implementation or system property cannot be obtained. The second problem, `ParserConfigurationException`, occurs when a requested feature is not available in the parser being used. Both are easy to deal with and shouldn't pose as any difficulty in using JAXP.

A `SAXParser` is obtained once you get the factory, turn off namespaces, and turn on validation; then parsing begins. Notice that the `parse()` method of the SAX parser takes an instance of the SAX `HandlerBase` class that I mentioned earlier. (You can view the implementation of this class with the complete Java listing.) You also pass in the `File` to parse. However, the `SAXParser` class contains much more than just this single method.

Working with the SAX parser
Once you have an instance of the `SAXParser` class, you can do a lot more with it than solely passing it a `File` to parse. Because of the way application components in large applications communicate now, it is not always safe to assume that the creator of an object instance is its user. In other words, one component may create the `SAXParser` instance, while another component (perhaps coded by another developer) may need to use that same instance. For this reason, methods are provided to determine the setting of the parser. The two methods that do this are `isValidating()`, which informs the caller that the parser will, or will not, perform validation, and `isNamespaceAware()`, which returns an indication that the parser can, or cannot, process namespaces in an XML document. While these methods can give you information about what the parser can do, you do not have the means to change these features. You must do this at the parser factory level.

Additionally, there are a variety of ways to request parsing of a document. Instead of just accepting a `File` and a SAX `HandlerBase` instance, the SAXParser's `parse()` method can also accept a SAX `InputSource`, a Java `InputStream`, or a URL in `String` form, all with a `HandlerBase` instance. So different types of input documents can be treated to different means of parsing.

Finally, the underlying SAX parser (an instance of `org.xml.sax.Parser`) can be obtained and used directly through the SAXParser's `getParser()` method. Once this underlying instance is obtained, the usual SAX methods are available. The next listing shows examples of the various uses of the `SAXParser` class, the core class in JAXP for SAX parsing.

Listing 2. Using the JAXP SAXParser

```
    // Get a SAX Parser instance
    SAXParser saxParser = saxFactory.newSAXParser();
    // Find out if validation is supported
    boolean isValidating = saxParser.isValidating();
    // Find out if namespaces is supported
    boolean isNamespaceAware = saxParser.isNamespaceAware();
    // Parse, in a variety of ways
    // Use a file and a SAX HandlerBase instance
    saxParser.parse(new File(args[0]), myHandlerBaseInstance);
    // Use a SAX InputSource and a SAX HandlerBase instance
    saxParser.parse(mySaxInputSource, myHandlerBaseInstance);
    // Use an InputStream and a SAX HandlerBase instance
    saxParser.parse(myInputStream, myHandlerBaseInstance);
    // Use a URI and a SAX HandlerBase instance
    saxParser.parse("http://www.newInstance.com/xml/doc.xml", myHandlerBaseInstance);
    // Get the underlying (wrapped) SAX parser
    org.xml.sax.Parser parser = saxParser.getParser();
    // Use the underlying parser
    parser.setContentHandler(myContentHandlerInstance);
    parser.setErrorHandler(myErrorHandlerInstance);
    parser.parse(new org.xml.sax.InputSource(args[0]));
```

Up to now, I've talked a lot about SAX, but I haven't unveiled anything remarkable or even that surprising. The fact is, the functionality of JAXP is fairly minor, especially when SAX is involved. This is fine because having minimal functionality means your code is more portable and can be used by other developers, either freely (through open source, it is hoped), or commercially, with any SAX-compliant XML parser. That's it. There's nothing more to using SAX with JAXP. If you already know SAX, you're about 98 percent of the way there. You just need to learn two new classes and a couple of Java exceptions, and you're ready to roll. If you've never used SAX, it's easy enough to start now.

Dealing with DOM

If you're thinking you need to take a break in order to gear up for the challenge of DOM, you can save yourself some rest. The process of using DOM with JAXP is nearly identical to that with SAX; all you do is change two class names and a return type, and you are pretty much there. If you understand how SAX works and understand what DOM is, you won't have any problem.

The primary difference between DOM and SAX is the structure of the APIs themselves. While SAX consists of an event-based set of callbacks, DOM has an in-memory tree structure. In other words, with SAX there's never a data structure to work on (unless the developer creates one manually). SAX, therefore, does not offer the ability to modify an XML document. DOM provides exactly this type of functionality. The `org.w3c.dom.Document` class represents an XML document and is made up of DOM *nodes* that represent the elements, attributes, and other XML constructs. So JAXP doesn't have to fire SAX callbacks, it is only responsible for returning a DOM `Document` object from a parsing.

A look at the DOM parser factory

With this basic understanding of DOM and the differences between DOM and SAX, there is little else to say. The following code will look remarkably similar to our SAX code. First, a `DocumentBuilderFactory` is obtained (in the same way that it was in SAX). Then the factory is configured to handle validation and namespaces (in the same way that it was in SAX). Next, a `DocumentBuilder`, the analog to `SAXParser`, is retrieved from the factory (in the same way . . . well, you get the idea). Parsing can then occur, and the resultant DOM `Document` object is handed off to a method that prints the DOM tree.

Listing 3. Using the document builder factory

```
import java.io.File;
import java.io.IOException;
import java.io.OutputStreamWriter;
import java.io.Writer;
// JAXP
import javax.xml.parsers.FactoryConfigurationError;
import javax.xml.parsers.ParserConfigurationException;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.DocumentBuilder;
// DOM
import org.w3c.dom.Document;
import org.w3c.dom.DocumentType;
import org.w3c.dom.NamedNodeMap;
import org.w3c.dom.Node;
import org.w3c.dom.NodeList;
public class TestDOMParsing {
    public static void main(String[] args) {
        try {
            if (args.length != 1) {
                System.err.println ("Usage: java TestDOMParsing [filename]");
                System.exit (1);
            }
            // Get Document Builder Factory
            DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
            // Turn on validation, and turn off namespaces
            factory.setValidating(true);
            factory.setNamespaceAware(false);
            DocumentBuilder builder = factory.newDocumentBuilder();
            Document doc = builder.parse(new File(args[0]));
            // Print the document from the DOM tree and
            //   feed it an initial indentation of nothing
            printNode(doc, "");
            // Modifications to the DOM Document could also be made here.
        } catch (ParserConfigurationException e) {
            System.out.println("The underlying parser does not support the requested
features.");
        } catch (FactoryConfigurationError e) {
            System.out.println("Error occurred obtaining Document Builder Factory.");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
    private static void printNode(Node node, String indent)  {
        // print the DOM tree
    }
}
```

Two different problems can arise with this code (as with SAX in JAXP): a `FactoryConfigurationError` and a `ParserConfigurationException`. The cause of each is the same as it was in SAX. Either there's a problem present in the implementation classes (`FactoryConfigurationError`), or the parser provided doesn't support the requested features (`ParserConfigurationException`). The only difference between DOM and SAX is that with DOM you substitute `DocumentBuilderFactory` for `SAXParserFactory`, and `DocumentBuilder` for `SAXParser`. It's that simple! (You can view the complete code listing, which includes the method used to print out the DOM tree.)

Working with the DOM parser
Once you have a DOM factory, you can obtain a `DocumentBuilder` instance. The methods available to a `DocumentBuilder` instance are very similar to those available to its SAX counterpart. The major difference is that variations of the `parse()` do not take an instance of the SAX `HandlerBase` class. Instead they return a DOM `Document` instance representing the XML document that was parsed. The only other difference is that two methods are provided for SAX-like functionality: `setErrorHandler()`, which takes a SAX `ErrorHandler` implementation to handle problems that may arise in parsing, and `setEntityResolver)_`, which takes a SAX `EntityResolver` implementation to handle entity resolution. If these concepts are unfamiliar to you, now's the time to check out SAX either online or in my book. The following listing shows examples of these methods in action.

Listing 4. Using the JAXP DocumentBuilder

```
    // Get a DocumentBuilder instance
    DocumentBuilder builder = builderFactory.newDocumentBuilder();
    // Find out if validation is supported
    boolean isValidating = builder.isValidating();
    // Find out if namespaces is supported
    boolean isNamespaceAware = builder.isNamespaceAware();
    // Set a SAX ErrorHandler
    builder.setErrorHandler(myErrorHandlerImpl);
    // Set a SAX EntityResolver
    builder.setEntityResolver(myEntityResolverImpl);
    // Parse, in a variety of ways
    // Use a file
    Document doc = builder.parse(new File(args[0]));
    // Use a SAX InputSource
    Document doc = builder.parse(mySaxInputSource);
    // Use an InputStream
    Document doc = builder.parse(myInputStream, myHandlerBaseInstance);
    // Use a URI
    Document doc = builder.parse("http://www.newInstance.com/xml/doc.xml");
```

If you got a little bored in this section on DOM, you're not alone; it was a little boring to write because it really is that straightforward to take what you've learned about SAX and apply it to DOM. So make your bar bets with friends and coworkers on how using JAXP is a piece of cake.

Changing the parser

The last topic we need to address in dealing with JAXP is the ability to easily change out the parser used by the factory classes. Changing the parser used by JAXP actually means changing the parser factory, because all `SAXParser` and `DocumentBuilder` instances come from these factories. Since the factories determine which parser is loaded, it's the factories that must change. The implementation of the `SAXParserFactory` interface to be used can be changed by setting the Java system property `javax.xml.parsers.SAXParserFactory`. If this property isn't defined, then the default implementation (whatever parser your vendor specified) is returned. The same principle applies for the `DocumentBuilderFactory` implementation you use. In this case, the `javax.xml.parsers.DocumentBuilderFactory` system property is queried. And as simple as that, we have gone through it all! This is the whole scope of JAXP: provide hooks into SAX, provide hooks into DOM, and allow the parser to easily be changed out.

Summary

As you can see, there's very little tricky material to wade through. Changing a system property, setting validation through a factory instead of a parser or builder, and getting clear on what JAXP actually *isn't* are the most difficult parts of putting JAXP to work. Other than not having SAX 2.0 and DOM Level 2 support, JAXP provides a helpful pluggability layer over two popular Java and XML APIs. It makes your code vendor-neutral and allows you to change from parser to parser without ever recompiling your parsing code. So download JAXP from Sun, Apache XML, or anywhere else you can get your hands on it, and go to it! Stay tuned for JAXP 1.1, with important additions like support for SAX 2 and DOM 2, XSLT, and more. You'll get the news here first, so keep an eye on *developerWorks*.

Resources

- Take a look at JAXP revisited, Brett's follow-up article on JAXP.
- Read the JAXP 1.0 specification to get the details from the source.
- Check out all of Sun's XML activity at their Java and XML headquarters.
- Get the Apache JAXP implementation at Apache XML.
- Find out more about the APIs under the covers. Start with SAX 2 for Java, at the SAX Web site.
- For another view of XML supported by SAX, take a look at DOM at the W3C Web site.
- For 490 or so more pages of expert XML advice, check out Java and XML, published by O'Reilly, Brett's book on the hottest technologies around.
- Join the *developerWorks* XML tools and APIs newsgroup for Java language developers
- Need a more basic introduction to XML? Try the *developerWorks* Intro to XML tutorial and other educational offerings, which cover the most fundamental topics.

About the author

Brett McLaughlin works as Enhydra strategist at Lutris Technologies and specializes in distributed systems architecture. He is author of Java and XML (O'Reilly). He is involved in technologies such as Java servlets, Enterprise JavaBeans technology, XML, and business-to-business applications. Along with Jason Hunter, he recently founded the JDOM project, which provides a simple API for manipulating XML from Java applications. He is also an active developer on the Apache Cocoon project, EJBoss EJB server, and a co-founder of the Apache Turbine project. Brett is currently on the expert group working on the JAXP 1.1 specification and release. You can contact him at brett@newInstance.com.

PDF e-mail it!

What do you think of this article?

○ Killer! (5) ○ Good stuff (4) ○ So-so; not bad (3) ○ Needs work (2) ○ Lame! (1)

Comments?

Submit feedback