
3

In this chapter:

- *Getting Prepared*
- *SAX Readers*
- *Content Handlers*
- *Error Handlers*
- *A Better Way to Load a Parser*
- *“Gotcha!”*
- *What’s Next?*

Parsing XML

With two solid chapters of introduction behind us, we are ready to code! By now you have seen the numerous acronyms that make up the world of XML, you have delved into the language itself, and you should be familiar with an XML document. This chapter takes the next step, and the first on our path of Java programming, by demonstrating how an XML document is parsed and how we can access the parsed data from within Java code.

One of the first things you will have to do when dealing with XML programmatically is take an XML document and parse it. As the document is parsed, the data in the document becomes available to the application using the parser, and suddenly we are within an XML-aware application! If this all sounds a little too simple to be true, it almost is. In this chapter, we will look closely at how an XML document is parsed. Using a parser within an application and how to feed that parser your document’s data will be covered. Then we will look at the various callbacks that are available within the parsing lifecycle. These events are the points where application-specific code can be inserted and data manipulation can occur.

In addition to looking at how parsers work, we will also begin our exploration of the Simple API for XML (SAX) in this chapter. SAX is what makes these parsing callbacks available. The interfaces provided in the SAX package will become an important part of our toolkit for handling XML. Even though the SAX classes are small and few in number, everything else in our discussions of XML is based on these classes. A solid understanding of how they help us access XML data is critical to effectively leveraging XML in your Java programs.

Getting Prepared

There are several items that we should take care of before beginning to code. First, you must obtain an XML parser. Writing a parser for XML is a serious task, and there are several efforts going on to provide excellent XML parsers. We are not going to detail the process of actually writing an XML parser here; rather, we will discuss the applications that wrap this parsing behavior, focusing on using existing tools to manipulate XML data. This results in better and faster programs, as we do not seek to reinvent what is already available. After selecting a parser, we must ensure that a copy of the SAX classes is on hand. These are easy to locate, and are key to our Java code being able to process XML. Finally, we will need an XML document to parse. Then, on to the code!

Obtaining a Parser

The first step in getting ready to code Java that uses XML is locating and obtaining the parser you want to use. We briefly talked about this process in Chapter 1, *Introduction*, and listed various XML parsers that could be used. To ensure that your parser works with all of the examples in the book, you should verify your parser's compliance with the XML specification. Because of the variety of parsers available and the rapid pace of change within the XML community, all of the details about which parsers have what compliance levels are beyond the scope of this book. You should consult the parser's vendor and visit the web sites previously given for this information.

In the spirit of the open source community, all of the examples in this book will use the Apache Xerces parser. Freely available in binary and source form at <http://xml.apache.org>, this C- and Java-based parser is already one of the most widely contributed-to parsers available. In addition, using an open source parser such as Xerces allows you to send questions or bug reports to the parser's authors, resulting in a better product, as well as helping you use the software quickly and correctly. To subscribe to the general list and request help on the Xerces parser, send a blank email to xerces-dev-subscribe@xml.apache.org. The members of this list can help if you have questions or problems with a parser not specifically covered in this book. Of course, the examples in this book all run normally on any parser that uses the SAX implementation covered here.

Once you have selected and downloaded an XML parser, make sure that your Java environment, whether it be an IDE (Integrated Development Environment) or a command line, has the XML parser classes in its class path. This will be a basic requirement for all further examples.

Getting the SAX Classes and Interfaces

Once you have your parser, you need to locate the SAX classes. These classes are almost always included with a parser when downloaded, and Xerces is no exception. If this is the case with your parser, you should be sure not to download the SAX classes explicitly, as your parser is probably packaged with the latest version of SAX that is supported by the parser. At the time of this writing, SAX 2.0 had just gone final. The SAX 2.0 classes are used throughout this book, and should come bundled with the latest version of the Apache Xerces parser.

If you are not sure whether you have the SAX classes, look at the *jar* file or class structure used by your parser. The SAX classes are packaged in the `org.xml.sax` structure. The latest version of these includes 17 classes in this root directory, as well as 9 classes in `org.xml.sax.helpers` and 2 in `org.xml.sax.ext`. If you are missing any of these classes, you should try to contact your parser's vendor to see why the classes were not included with your distribution. It is possible that some classes may have been left out if they are not supported in whole.* These class counts are for SAX 2.0 as well; fewer classes may appear if only SAX 1.0 is supported.

Finally, you may want to either download or bookmark the SAX API Javadocs on the Web. This documentation is extremely helpful in using the SAX classes, and the Javadoc structure provides a standard, simple way to find out additional information about the classes and what they do. This documentation is located at <http://www.megginson.com/SAX/SAX2/javadoc/index.html>. You may also generate Javadoc from the SAX source if you wish, by using the source included with your parser, or by downloading the complete source from <http://www.megginson.com/SAX/SAX2>.

Have an XML Document on Hand

You should also make sure that you have an XML document to parse. The output shown in the examples is based on parsing the XML document we discussed in Chapter 2, *Creating XML*. Save this file as *contents.xml* somewhere on your local hard drive. We highly recommend that you follow what we're doing in this file. You can simply type the file in from the book, or you may download the XML file from the book's web site, <http://www.oreilly.com/catalog/javaxml>. You are encouraged to take the time to type in the example, though, as it will almost certainly familiarize you with XML syntax more than a quick download will.

* Supporting SAX in whole is a very important item for a parser. Although you are certainly welcome to use any parser you like, if your parser does not have complete SAX 2.0 support, many of the examples in this book will not work. In addition, your parser is not keeping up with the latest XML developments. For either or both reasons, you may want to consider at least trying the Xerces parser for the duration of this book.

In addition to downloading or creating the XML file, you need to make a couple of small modifications. Because we haven't covered or discussed how to constrain and transform documents, our programs only parse XML in this chapter. To prevent errors, we need to remove the references within the XML document to an external DTD, which constrains the XML, and the XSL stylesheets that transform it. You should comment out these two lines in the XML document, as well as the processing instruction to Cocoon requesting XSL transformation:

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

<!-- We don't need these yet
  <?xml-stylesheet href="XSL\JavaXML.html.xsl" type="text/xsl"?>
  <?xml-stylesheet href="XSL\JavaXML.wml.xsl" type="text/xsl"
    media="wap"?>
  <?cocoon-process type="xslt"?>
  <!DOCTYPE JavaXML:Book SYSTEM "DTD\JavaXML.dtd">
-->

<!-- Java and XML -->
<JavaXML:Book xmlns:JavaXML="http://www.oreilly.com/catalog/javaxml/">
```

Once these lines are commented, note the full path to the XML document. You will need to supply that path to our programs in this and later chapters.

Finally, we need to comment out our reference to the `OREillyCopyright` external entity that would be used to load a file from the filesystem with the needed copyright information. Without a DTD to define how to resolve this entity reference, we will receive unwanted errors. In the next chapter, we will look at how to resolve this reference for the XML document.

```
</JavaXML:Contents>

<!-- Leave out until DTD Section
  <JavaXML:Copyright>&OREillyCopyright;</JavaXML:Copyright>
-->

</JavaXML:Book>
```

SAX Readers

Without spending any further time on the preliminaries, let's begin to code. Our first program will be able to take an XML file as a command-line parameter, and parse that file. We will build document callbacks into the parsing process so that we can display events in the parsing process as they occur, which will give us a better idea of what exactly is going on "under the hood."

The first thing we need to do is get an instance of a class that conforms to the SAX `org.xml.sax.XMLReader` interface. This interface defines parsing behavior and allows us to set features and properties, which we will look at in Chapter 5, *Validating XML*. For those of you familiar with SAX 1.0, this interface replaces the `org.xml.sax.Parser` interface.

Instantiating a Reader

SAX provides an interface that all SAX-compliant XML parsers should implement. This allows SAX to know exactly what methods are available for callback and use within an application. For example, the Xerces main SAX parser class, `org.apache.xerces.parsers.SAXParser`, implements the `org.xml.sax.XMLReader` interface. If you have access to the source of your parser, you should see the same interface implemented in your parser's main SAX parser class. Each XML parser must have one class (sometimes more!) that implements this interface, and that is the class we need to instantiate to allow us to parse XML:

```
XMLReader parser =
    new SAXParser();

// Do something with the parser
parser.parse(uri);
```

For those of you new to SAX entirely, it may be a bit confusing not to see the instance variable we used named `reader` or `XMLReader`. While that would be a normal convention, the SAX 1.0 classes defined the main parsing interface as `Parser`, and a lot of legacy code has variables named `parser` because of that naming. This interface was deprecated because of the large number of changes required for namespace and feature and properties support, but the naming convention is still a good one, as `parser` does indicate the purpose of the instance variable.

With that in mind, let's look at a small program to start up and instantiate a SAX parser. This program, shown in Example 3-1, won't actually parse a document, but sets up the skeleton within which we can work for the rest of the chapter; we will add the actual parsing behavior in the next chapter.

Example 3-1. SAX Parser Example

```
import org.xml.sax.XMLReader;

// Import your vendor's XMLReader implementation here
import org.apache.xerces.parsers.SAXParser;

/**
 * <b><code>SAXParserDemo</code></b> will take an XML file and parse it
```

Example 3-1. SAX Parser Example (continued)

```
* using SAX, displaying the callbacks in the parsing lifecycle.
*
* @author
*   <a href="mailto:brettmclaughlin@earthlink.net">Brett McLaughlin</a>
* @version 1.0
*/
public class SAXParserDemo {

    /**
     * <p>
     * This parses the file, using registered SAX handlers, and outputs
     * the events in the parsing process cycle.
     * </p>
     *
     * @param uri <code>String</code> URI of file to parse.
     */
    public void performDemo(String uri) {
        System.out.println("Parsing XML File: " + uri + "\n\n");

        // Instantiate a parser
        XMLReader parser =
            new SAXParser();
    }

    /**
     * <p>
     * This provides a command-line entry point for this demo.
     * </p>
     */
    public static void main(String[] args) {
        if (args.length != 1) {
            System.out.println("Usage: java SAXParserDemo [XML URI]");
            System.exit(0);
        }

        String uri = args[0];
        SAXParserDemo parserDemo = new SAXParserDemo();
        parserDemo.performDemo(uri);
    }
}
```

You should be able to load and compile this program if you made the preparations talked about earlier to ensure the SAX classes are in your class path. This simple program doesn't do much yet; in fact, if you run it and supply a bogus file-name or URI as an argument, it should happily grind away and do nothing, other than print out the initial "Parsing XML file" message. That's because we have only instantiated a parser, not requested that our XML document be parsed.

NOTE If you have trouble compiling this source file, you most likely have problems with your IDE or system's class path. First, make sure you obtained the Apache Xerces parser (or your vendor's parser). For Xerces, this involves downloading a *jar* file. This archive can then be extracted, and will contain a *xerces.jar* file; it is this *jar* file that contains the compiled class files for the program. Add this archive to your class path. You should then be able to compile the source file listing.

Parsing the Document

Once a parser is loaded and ready for use, we can instruct it to parse our document. This is conveniently handled by the `parse()` method of `org.xml.sax.XMLReader`, and this method can accept either an `org.xml.sax.InputSource`, or a simple string URI. For now, we will defer talking about using an `InputSource` and look at passing in a simple URI. Although this URI could be a network-accessible address, we will use the full path to the XML document we prepared for this use earlier. If you did choose to use a URL for network-accessible XML documents, you should be aware that the application would have to resolve the URL before passing it to the parser (generally this requires only some form of network connectivity).

We need to add the `parse()` method to our program, as well as two exception handlers. Because the document must be loaded, either locally or remotely, a `java.io.IOException` can result, and must be caught. In addition, the `org.xml.sax.SAXException` can be thrown if problems occur while parsing the document. So we can add two more import statements and a few lines of code, and have an application that parses XML ready to use:

```
import java.io.IOException;

import org.xml.sax.SAXException;
import org.xml.sax.XMLReader;

// Import your vendor's XMLReader implementation here
import org.apache.xerces.parsers.SAXParser;

...

/**
 * <p>
 * This parses the file, using registered SAX handlers, and outputs
 * the events in the parsing process cycle.
 * </p>
 */
```

```
* @param uri <code>String</code> URI of file to parse.
*/
public void performDemo(String uri) {
    System.out.println("Parsing XML File: " + uri + "\n\n");

    try {
        // Instantiate a parser
        XMLReader parser =
            new SAXParser();

        // Parse the document
        parser.parse(uri);

    } catch (IOException e) {
        System.out.println("Error reading URI: " + e.getMessage());
    } catch (SAXException e) {
        System.out.println("Error in parsing: " + e.getMessage());
    }
}
```

Compile these changes and you are ready to execute the parsing example. You should specify the full path to your file as the first argument to the program:

```
D:\prod\JavaXML> java SAXParserDemo D:\prod\JavaXML\contents\contents.xml
Parsing XML File: D:\prod\JavaXML\contents\contents.xml
```

This rather uninteresting output may make you doubt that anything has happened. However, if you lean nice and close, you may hear your hard drive spin briefly (or you can just have faith in our bytecode). In fact, the XML document is parsed, and if you pass in an invalid file URI, the parser will throw an exception letting you know it couldn't locate a file to parse. However, we have not set up any callbacks to tell SAX to take action during the parsing process and let us know what is going on. Without these callbacks, a document is parsed quietly and without application intervention. Of course, we want to intervene in that process, so we must next look at creating some parser callback methods. This intervention is the most important part of using SAX. Parser callbacks let us insert action into the program flow, and turn our rather boring, quiet parsing of an XML document into an application that can react to the data, elements, attributes, and structure of the document being parsed, as well as interact with other programs and clients along the way.

Using an InputSource

Instead of using a full URI, the `parse()` method may also be invoked with an `org.xml.sax.InputSource` as an argument. There is actually remarkably little to comment on in regard to this class; it is used as a helper and wrapper class more than anything else. An `InputSource` simply encapsulates information about a single

object. While this isn't very helpful in our example, in situations where a system identifier, public identifier, or a stream may all be tied to one URI, using an `InputSource` for encapsulation can become very handy. The class has accessor and mutator methods for its system ID and public ID, a character encoding, a byte stream (`java.io.InputStream`), and a character stream (`java.io.Reader`). Passed as an argument to the `parse()` method, SAX also guarantees that the parser will never modify the `InputSource`. This ensures that the original input to a parser is still available unchanged after its use by a parser or XML-aware application. While we do not spend any further time looking at this utility class here, many of the applications we look at later in the book use the `InputSource` class as input to SAX parsers rather than a specific URI.

Content Handlers

In order to let our application do something useful with XML data as it is being parsed, we must register *handlers* with the SAX parser. A handler is nothing more than a set of callbacks that SAX defines to let us interject application code at important events within a document's parsing. Realize that these events will take place as the document is parsed, not after the parsing has occurred. This is one of the reasons that SAX is such a powerful interface: it allows a document to be handled sequentially, without having to first read the entire document into memory. We will later look at the Document Object Model (DOM), which has this limitation.

There are four core handler interfaces defined by SAX 2.0: `org.xml.sax.ContentHandler`, `org.xml.sax.ErrorHandler`, `org.xml.sax.DTDHandler`, and `org.xml.sax.EntityResolver`. In this chapter, we discuss `ContentHandler`, which allows standard data-related events within an XML document to be handled, and take a first look at `ErrorHandler`, which receives notifications from the parser when errors in the XML data are found. `DTDHandler` will be examined in Chapter 5. We briefly discuss `EntityResolver` at various points in the text; it is enough for now to understand that `EntityResolver` works just like the other handlers, and is built specifically for resolving external entities specified within an XML document. Custom application classes that perform specific actions within the parsing process can implement each of these interfaces. These implementation classes can be registered with the parser with the methods `setContentHandler()`, `setErrorHandler()`, `setDTDHandler()`, and `setEntityResolver()`. Then the parser invokes the callback methods on the appropriate handlers during parsing.

For our example, we want to implement the `ContentHandler` interface. This interface defines several important methods within the parsing lifecycle that our application can react to. First we need to add the appropriate `import` statements to our source file (including the `org.xml.sax.Locator` and `org.xml.sax.Attributes` class and interface, which we will discuss in a moment), as well as a

new class that will implement these callback methods. This new class can be added at the end of your source file, *SAXParserDemo.java*:

```
import java.io.IOException;

import org.xml.sax.Attributes;
import org.xml.sax.ContentHandler;
import org.xml.sax.Locator;
import org.xml.sax.SAXException;
import org.xml.sax.XMLReader;

// Import your vendor's XMLReader implementation here
import org.apache.xerces.parsers.SAXParser;

...

/**
 * <b><code>MyContentHandler</code></b> implements the SAX
 * <code>ContentHandler</code> interface and defines callback
 * behavior for the SAX callbacks associated with an XML
 * document's content.
 */
class MyContentHandler implements ContentHandler {

    /** Hold onto the locator for location information */
    private Locator locator;

    /**
     * <p>
     * Provide reference to <code>Locator</code> which provides
     * information about where in a document callbacks occur.
     * </p>
     *
     * @param locator <code>Locator</code> object tied to callback
     * process
     */
    public void setDocumentLocator(Locator locator) {
    }

    /**
     * <p>
     * This indicates the start of a Document parse—this precedes
     * all callbacks in all SAX Handlers with the sole exception
     * of <code>{@link #setDocumentLocator}</code>.
     * </p>
     *
     * @throws <code>SAXException</code> when things go wrong
     */
    public void startDocument() throws SAXException {
    }
}
```

```
/**
 * <p>
 * This indicates the end of a Document parse—this occurs after
 * all callbacks in all SAX Handlers.</code>.
 * </p>
 *
 * @throws <code>SAXException</code> when things go wrong
 */
public void endDocument() throws SAXException {
}

/**
 * <p>
 * This indicates that a processing instruction (other than
 * the XML declaration) has been encountered.
 * </p>
 *
 * @param target <code>String</code> target of PI
 * @param data <code>String</code> containing all data sent to the PI.
 * This typically looks like one or more attribute value
 * pairs.
 * @throws <code>SAXException</code> when things go wrong
 */
public void processingInstruction(String target, String data)
    throws SAXException {
}

/**
 * <p>
 * This indicates the beginning of an XML Namespace prefix
 * mapping. Although this typically occurs within the root element
 * of an XML document, it can occur at any point within the
 * document. Note that a prefix mapping on an element triggers
 * this callback <i>before</i> the callback for the actual element
 * itself (<code>{@link #startElement}</code>) occurs.
 * </p>
 *
 * @param prefix <code>String</code> prefix used for the namespace
 * being reported
 * @param uri <code>String</code> URI for the namespace
 * being reported
 * @throws <code>SAXException</code> when things go wrong
 */
public void startPrefixMapping(String prefix, String uri) {
}

/**
 * <p>
 * This indicates the end of a prefix mapping, when the namespace
```

```
* reported in a @link #startPrefixMapping</code> callback
* is no longer available.
* </p>
*
* @param prefix String</code> of namespace being reported
* @throws SAXException</code> when things go wrong
*/
public void endPrefixMapping(String prefix) {
}

/**
* <p>
* This reports the occurrence of an actual element. It includes
* the element's attributes, with the exception of XML vocabulary
* specific attributes, such as
* xmlns:[namespace prefix]</code> and
* xsi:schemaLocation</code>.
* </p>
*
* @param namespaceURI String</code> namespace URI this element
* is associated with, or an empty
* String</code>
* @param localName String</code> name of element (with no
* namespace prefix, if one is present)
* @param rawName String</code> XML 1.0 version of element name:
* [namespace prefix]:[localName]
* @param atts Attributes</code> list for this element
* @throws SAXException</code> when things go wrong
*/
public void startElement(String namespaceURI, String localName,
    String rawName, Attributes atts)
    throws SAXException {
}

/**
* <p>
* Indicates the end of an element
* (<![CDATA[</code>] is reached. Note that
* the parser does not distinguish between empty
* elements and non-empty elements, so this occurs uniformly.
* </p>
*
* @param namespaceURI String</code> URI of namespace this
* element is associated with
* @param localName String</code> name of element without prefix
* @param rawName String</code> name of element in XML 1.0 form
* @throws SAXException</code> when things go wrong
*/
public void endElement(String namespaceURI, String localName,
```

```
        String rawName)
        throws SAXException {
    }

/**
 * <p>
 * This reports character data (within an element).
 * </p>
 *
 * @param ch <code>char[]</code> character array with character data
 * @param start <code>int</code> index in array where data starts.
 * @param end <code>int</code> index in array where data ends.
 * @throws <code>SAXException</code> when things go wrong
 */
public void characters(char[] ch, int start, int end)
        throws SAXException {
    }

/**
 * <p>
 * This reports whitespace that can be ignored in the
 *   originating document. This is typically invoked only when
 *   validation is occurring in the parsing process.
 * </p>
 *
 * @param ch <code>char[]</code> character array with character data
 * @param start <code>int</code> index in array where data starts.
 * @param end <code>int</code> index in array where data ends.
 * @throws <code>SAXException</code> when things go wrong
 */
public void ignorableWhitespace(char[] ch, int start, int end)
        throws SAXException {
    }

/**
 * <p>
 * This reports an entity that is skipped by the parser. This
 *   should only occur for non-validating parsers, and then is still
 *   implementation-dependent behavior.
 * </p>
 *
 * @param name <code>String</code> name of entity being skipped
 * @throws <code>SAXException</code> when things go wrong
 */
public void skippedEntity(String name) throws SAXException {
    }
}
```

We have added empty implementations for all the methods defined in the `ContentHandler` interface, which allows our source file to compile. Of course, these empty implementations don't provide any feedback for us, so we will walk through each of these required methods now.

The Document Locator

The first method we need to define is one that sets an `org.xml.sax.Locator` for any SAX event. When a callback event occurs, a class that implements a handler often needs access to the location within an XML file of the SAX parser. This can then be used to help the application make decisions about the event and its location within the XML document. The `Locator` class has several useful methods such as `getLineNumber()` and `getColumnNumber()` that return the current location within an XML file when invoked. Because this location is only valid for the current parsing lifecycle, the `Locator` should only be used within the scope of the `ContentHandler` implementation. Since we may want to use this later, we save the provided `Locator` instance to a member variable, as well as printing out a message indicating that the callback has occurred. This will help outline the order and occurrence of SAX events:

```
/** Hold onto the locator for location information */
private Locator locator;

/**
 * <p>
 * Provide reference to <code>Locator</code>, which provides
 * information about where in a document callbacks occur.
 * </p>
 *
 * @param locator <code>Locator</code> object tied to callback
 * process
 */
public void setDocumentLocator(Locator locator) {
    System.out.println("    * setDocumentLocator() called");
    // We save this for later use if desired.
    this.locator = locator;
}
```

Later, we can add details to this method if we need to act upon information about the origin of events; in this example, we merely want to show information about what is occurring in the parsing process. However, if we wanted to show information about where in the document events were occurring, such as the line number an element appeared on, we would want to assign this `Locator` to a member variable for later use within the class.

The Start and the End of a Document

In any lifecycle process, there must always be a beginning and an end. These important events should both occur once, the former before all other events, and the latter after all other events. This obvious fact is critical to applications, as it allows them to know exactly when parsing begins and exactly when it ends. SAX provides callback methods for each of these events, `startDocument()` and `endDocument()`.

The first method, `startDocument()`, is called before any other callbacks, including the callback methods within other SAX handlers, such as `DTDHandler`. In other words, `startDocument()` is not only the first method called within `ContentHandler`, but also within the entire parsing process, aside from the `setDocumentLocator()` method we just discussed. This ensures a finite beginning to parsing, and lets the application perform any tasks it needs to before parsing takes place.

The second method, `endDocument()`, is always the last method called, again across all handlers. This includes situations in which errors occur that cause parsing to halt. We will discuss errors later, but there are both recoverable errors and unrecoverable errors. If an unrecoverable error occurs, the `ErrorHandler`'s callback method will be invoked, and then a final call to `endDocument()` completes the attempted parsing.

In our example, we want to output to the console when both these events occur to further illustrate the parsing lifecycle:

```
/**
 * <p>
 * This indicates the start of a Document parse—this precedes
 * all callbacks in all SAX Handlers with the sole exception
 * of <code>{@link #setDocumentLocator}</code>.
 * </p>
 *
 * @throws SAXException when things go wrong
 */
public void startDocument() throws SAXException {
    System.out.println("Parsing begins...");
}

/**
 * <p>
 * This indicates the end of a Document parse - this occurs after
 * all callbacks in all SAX Handlers.</code>.
 * </p>
 *
 * @throws SAXException when things go wrong
 */
```

```
 */
public void endDocument() throws SAXException {
    System.out.println("...Parsing ends.");
}
```

Both of these callback methods can throw `SAXExceptions`. These are the only types of exceptions that SAX events ever throw, and they provide another standard interface to the parsing behavior. However, these exceptions often wrap other exceptions that are indicative of what problems occur. For example, if an XML file was being parsed over the network via a URL, and the connection suddenly became invalid, an `IOException` would result. However, an application using the SAX classes should not have to catch this exception, because it should not have to know where the XML resource is located. Instead, the application can catch the single `SAXException`. Within the SAX parser, the original exception is caught and re-thrown as a `SAXException`, with the originating exception stuffed inside the new one. This allows applications to have one standard exception to trap for, while allowing specific details of what errors occurred within the parsing process to be wrapped and made available to the calling program through this standard exception. The `SAXException` class provides a method, `getException()`, which returns the underlying `Exception`.

Processing Instructions

You should recall that we talked about processing instructions (PIs) within XML as a bit of a special case. They were not considered XML elements, and were handled differently by being passed to the calling application. Because of these special characteristics, SAX defines a specific callback for handling processing instructions. This method receives the target of the processing instruction and any data sent to the PI. For our example, we want to echo this information to the screen to notify us when a callback is made:

```
/**
 * <p>
 * This indicates that a processing instruction (other than
 * the XML declaration) has been encountered.
 * </p>
 *
 * @param target <code>String</code> target of PI
 * @param data <code>String</code> containing all data sent to the PI.
 *           This typically looks like one or more attribute-value
 *           pairs.
 * @throws <code>SAXException</code> when things go wrong
 */
public void processingInstruction(String target, String data)
    throws SAXException {
```

```
        System.out.println("PI: Target:" + target + " and Data:" + data);
    }
```

In a real application that is using XML data, this is where an application could receive instructions and set variable values or execute methods to perform application-specific processing. For example, the Apache Cocoon publishing framework might set flags to perform transformations on the data once it is parsed, or to display the XML as a specific content type. This method, like the other SAX callbacks, throws a `SAXException` when errors occur.

You may also remember that in our discussion of PIs we mentioned the XML declaration. This special processing instruction gives the version and optional information about the encoding of the document and whether it is a standalone document:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
```

This instruction is specifically for the XML parser, allowing the parser to report an error, like a version that is not supported, at the outset of parsing. Because this instruction is only intended to be used by the parser, it does not initiate a callback to `processingInstruction()`. Be sure not to build application code that expects this instruction or version information, because the application will never receive a callback for this PI. In fact, it is only the parser that should have much interest in the encoding and version of an XML document, as these items are used in parsing. Once the data is available to you through Java APIs, these details are generally irrelevant.

Namespace Callbacks

By the amount of discussion (and confusion) we have already encountered about namespaces in XML, you should be starting to realize their importance and impact on parsing and handling XML. Alongside XML Schema, XML Namespaces is easily the most significant concept added to XML since the original XML 1.0 Recommendation. With SAX 2.0, support for namespaces was introduced at the element level. This allows a distinction to be made between the namespace of an element, signified by an element prefix and an associated namespace URI, and the local name of an element. In this case, we use *local name* to refer to the unprefix name of an element. For example, the local name of `JavaXML:Book` is simply `Book`. The namespace prefix is `JavaXML`, and the namespace URI (in our example) is declared as `http://www.oreilly.com/catalog/javaxml`.

There are two SAX callbacks specifically dealing with namespaces (although the element callbacks use them as well). These callbacks are invoked when the parser reaches the start and end of a *prefix mapping*. Although this is a new term, it is not a new concept; a prefix mapping is simply an element that uses the `xmlns` attribute to declare a namespace. This is often the root element (which may have multiple

mappings), but can be any element within an XML document that declares an explicit namespace. For example:

```
<root>
  <element1>
    <myNamespace:element2 xmlns:myNamespace="http://myUrl.com">
      <myNamespace:element3>Here is some data</myNamespace:element3>
    </myNamespace:element2>
  </element1>
</root>
```

In this case, an explicit namespace is declared several element nestings deep within the document.

The `startPrefixMapping()` callback is given the namespace prefix as well as the URI associated with that prefix. The mapping is considered “closed” or “ended” when the element that declared the mapping is closed. The only twist to this callback is that it doesn’t quite behave in the sequential manner in which SAX usually is structured; the prefix mapping callback occurs directly *before* the callback for the element that declares the namespace. We look at this callback now:

```
/**
 * <p>
 * This will indicate the beginning of an XML Namespace prefix
 * mapping. Although this typically occurs within the root element
 * of an XML document, it can occur at any point within the
 * document. Note that a prefix mapping on an element triggers
 * this callback <i>before</i> the callback for the actual element
 * itself ({@link #startElement}) occurs.
 * </p>
 *
 * @param prefix String prefix used for the namespace
 *              being reported
 * @param uri String URI for the namespace
 *           being reported
 * @throws SAXException when things go wrong
 */
public void startPrefixMapping(String prefix, String uri) {
    System.out.println("Mapping starts for prefix " + prefix +
        " mapped to URI " + uri);
}
```

In our document, the only mapping we have is declared as an attribute of the root element. That means we should expect to see this callback invoked before the first element callback (which we look at next), although still after the `startDocument()` callback as well as any PIs we have at the top of our document. The other half of this namespace pair of callbacks is invoked to signify the end of

the mapping, and appears directly *after* the closing tag of the element declaring the mapping:

```
/**
 * <p>
 * This indicates the end of a prefix mapping, when the namespace
 * reported in a <code>{@link #startPrefixMapping}</code> callback
 * is no longer available.
 * </p>
 *
 * @param prefix <code>String</code> of namespace being reported
 * @throws <code>SAXException</code> when things go wrong
 */
public void endPrefixMapping(String prefix) {
    System.out.println("Mapping ends for prefix " + prefix);
}
```

For the XML document fragment above, we could then expect the following output when the `element2` element was reached:

```
Mapping starts for prefix myNamespace mapped to URI http://myUrl.com
```

This lets us know the prefix being mapped, and what URI that prefix is associated with.

Element Callbacks

By now you are probably ready to actually get to the data in our XML document. It is true that over half of the SAX callbacks have nothing to do with XML elements, attributes, and data. This is because the process of parsing XML is intended to do more than simply provide your application with the XML data; it should give the application instructions from XML PIs so your application can know what actions to take, let the application know when parsing starts and when it ends, and even tell it when there is whitespace that can be ignored! If some of these callbacks don't make much sense, keep reading. We'll explain more here, as well as in Chapter 5, when we look at how validation of XML fits into the picture.

Still, there certainly are SAX callbacks intended to give you access to the XML data within your documents. The three primary events you will concern yourself with to get that data are the start and end of elements and the `characters()` callback. These tell you when an element is parsed, the data within that element, and when the closing tag for that element is reached. The first of these, `startElement()`, gives an application information about an XML element and any attributes it may have. The parameters to this callback are the name of the element (in various forms), and an `org.xml.sax.Attributes` instance (remember our import statement earlier?). This helper class holds references to all of the attributes within an element. It allows easy iteration through the element's attributes in a form similar

to a `Vector`. In addition to being able to reference an attribute by its index (used when iterating through all attributes), it is possible to reference an attribute by its name. Of course, by now you should be a bit cautious when you see the word “name” referring to an XML element or attribute, as it can mean various things. In this case, either the complete name of the attribute (with a namespace prefix, if any), called its “raw” name, can be used, or the combination of its local name and namespace URI if a namespace is used. There are also helper methods such as `getURI(int index)` and `getLocalName(int index)` that help give additional namespace information about an attribute. Used as a whole, the `Attributes` interface can be a comprehensive set of information about an element’s attributes.

In addition to the element attributes, we mentioned you get several forms of the element’s name. This again is in deference to XML namespaces. The namespace URI of the element is first supplied. This places the element in its correct context across the complete document’s set of namespaces. Then the local name of the element is supplied, which we mentioned is the unprefix element name. In addition (and for backwards compatibility), the “raw” name of the element is supplied. This is the unmodified, unchanged name of the element, which includes a namespace prefix if present. In other words, this is exactly what was in the XML document, and so it would be `JavaXML:Book` for our `Book` element. With these three types of names supplied, you should be able to describe an element with or without respect to its namespace.

Now that we’ve seen how an element and its attributes are made available, let’s look at an implementation of the SAX callback that prints this information out to the screen when it is invoked. In this example, we see if the element name has a namespace URI associated with it; if so, we print out that namespace; if not, we print a message stating that the element has no namespace associated with it:

```
/**
 * <p>
 * This reports the occurrence of an actual element. It will include
 * the element's attributes, with the exception of XML vocabulary
 * specific attributes, such as
 * <code>xmlns:[namespace prefix]</code> and
 * <code>xsi:schemaLocation</code>.
 * </p>
 *
 * @param namespaceURI <code>String</code> namespace URI this element
 * is associated with, or an empty
 * <code>String</code>
 * @param localName <code>String</code> name of element (with no
 * namespace prefix, if one is present)
 * @param rawName <code>String</code> XML 1.0 version of element name:
 * [namespace prefix]:[localName]
 * @param atts <code>Attributes</code> list for this element
```

```

* @throws <code>SAXException</code> when things go wrong
*/
public void startElement(String namespaceURI, String localName,
                        String rawName, Attributes atts)
    throws SAXException {

    System.out.print("startElement: " + localName);
    if (!namespaceURI.equals("")) {
        System.out.println(" in namespace " + namespaceURI +
                           " (" + rawName + ")");
    } else {
        System.out.println(" has no associated namespace");
    }

    for (int i=0; i<atts.getLength(); i++)
        System.out.println(" Attribute: " + atts.getLocalName(i) +
                           "=" + atts.getValue(i));
}

```

SAX makes this process very simple and straightforward. One final thing to notice when looking at the `startElement()` callback and attributes in particular is that attributes do not remain ordered. When iterating through an `Attributes` implementation, the attributes will not necessarily be available in the order in which they were parsed, which is the order in which they were written. This means it is not a good idea to depend on the ordering of attributes, due to XML not requiring this ordering to be maintained by XML parsers. While there are some parsers that implement an ordering, it often is not included in a parser's feature set.

The closing half of an element callback is the `endElement()` method. This simple callback is fairly self-explanatory, and only the name of the element is sent to the callback, allowing that name to be matched with the appropriate element name passed earlier to a `startElement()` callback. The main purpose of this callback is to signify the close of an element, and let an application know that further characters are part of another scope, rather than the element now being closed. We make note of this in our example by printing out the name of an element when it is closed:

```

/**
 * <p>
 * Indicates the end of an element
 * (<code>&lt;[/element name]&gt;</code>) is reached. Note that
 * the parser does not distinguish between empty
 * elements and non-empty elements, so this will occur uniformly.
 * </p>
 *
 * @param namespaceURI <code>String</code> URI of namespace this
 * element is associated with
 * @param localName <code>String</code> name of element without prefix

```

```

    * @param rawName <code>String</code> name of element in XML 1.0 form
    * @throws <code>SAXException</code> when things go wrong
    */
    public void endElement(String namespaceURI, String localName,
        String rawName)
        throws SAXException {

        System.out.println("endElement: " + localName + "\n");
    }

```

Element Data

Once the beginning and end of an element block are identified and the element's attributes are enumerated for an application, the next piece of important information is the actual data contained within the element itself. This generally consists of additional elements, textual data, or a combination of the two. When other elements appear, the callbacks for those elements are initiated, and a type of pseudo-recursion happens: elements nested within elements results in callbacks "nested" within callbacks. At some point, textual data will be encountered. This is typically the most important information to an XML client, as this data is usually either what is shown to the client or what is processed to generate a client response.

In XML, textual data within elements is sent to a wrapping application via the `characters()` callback. This method provides the wrapping application with an array of characters as well as a starting and ending index from which to read the relevant textual data:

```

/**
 * <p>
 * This will report character data (within an element).
 * </p>
 *
 * @param ch <code>char[]</code> character array with character data
 * @param start <code>int</code> index in array where data starts.
 * @param end <code>int</code> index in array where data ends.
 * @throws <code>SAXException</code> when things go wrong
 */
    public void characters(char[] ch, int start, int end)
        throws SAXException {

        String s = new String(ch, start, end);
        System.out.println("characters: " + s);
    }

```

Seemingly a simple callback, this method often results in a significant amount of confusion because the SAX interface and standards do not strictly define how this

callback must be used for lengthy pieces of character data. In other words, a parser may choose to return all contiguous character data in one invocation, or split this data up into multiple method invocations. For any given element, this method will be called not at all (if no character data is present within the element) or one or more times. Different parsers will implement this behavior differently, often using algorithms designed to increase parsing speed. You should never count on having all the textual data for an element within one callback method; conversely, you should never assume that multiple callbacks would result for one element's contiguous character data.

As you are writing your SAX event handlers, you should also be sure to keep your mind in a hierarchical mode. In other words, you should not get in the habit of thinking that any element owns its data and *child elements*, but only that it serves as a parent. Also keep in mind that the parser is moving along, handling elements, attributes, and data as it comes across them. This can make for some surprising results. Consider the following XML document fragment:

```
<parent>This is<child>embedded text</child>more text</parent>
```

Forgetting that SAX parses sequentially, making callbacks as it sees elements and data, and forgetting that the XML is viewed as hierarchical, you might make the assumption that the output here would be something like:

```
startElement: parent has no associated namespace
characters: This is more text
startElement: child has no associated namespace
characters: embedded text
endElement: child
endElement: parent
```

This would seem logical, as the parent element completely “owns” the child element, right? Wrong. What actually occurs is that a callback is made at each SAX event-point, resulting in the following event-firing chain:

```
startElement: parent has no associated namespace
characters: This is
startElement: child has no associated namespace
characters: embedded text
endElement: child
characters: more text
endElement: parent
```

SAX does not do any reading ahead, so the result here is exactly what you would expect if you viewed the XML document as sequential data, without all the human assumptions that we tend to make. This is an important point to remember.

Finally, whitespace is often reported by the `characters()` method. This introduces additional confusion, as another SAX callback, `ignorableWhitespace()`,

also reports whitespace. In our example, we are not validating our XML document; however, we may still be using a validating (capable) parser. This subtle detail is very important, as the way in which whitespace is reported is defined by whether the parser being used is a validating one or not. Validating parsers will report all whitespace through the `ignoreableWhitespace()` method, due to some validation issues we will address in the next two chapters. Non-validating parsers can report whitespace either through the `ignoreableWhitespace()` method or the `characters()` method. To determine the difference, you will need to consult your parser's documentation to determine if you are using a validating parser or not. Remember, just because you are not requesting validation of your document does not mean that your parser is non-validating; a parser that is *capable* of validating, even if not actively doing so, is a validating parser.

To add to this confusion, many parsers are actually made up of dual parser implementations: one for validation and one for parsing without validation. At runtime, the correct class is loaded dynamically, as a non-validating parser often performs much better than a validating one, even if validation is not occurring, due to the extra data structures that must be implemented to allow validation to be used. This is exactly the case with the Apache Xerces parser; our example will utilize an instance of a non-validating parser, although if a DTD or schema was specified and validation was requested, a different parser class would be loaded and validation could occur.

The best way to avoid this confusion altogether is to not make any assumptions at all about whitespace. You should rarely, if ever, be using whitespace as data within your XML document. If you are forced to use whitespace, such as several spaces, non-space data, and then several more spaces, and the number of spaces in this data is relevant to an application, a CDATA section should be used. This ensures that your space-specific data will not be parsed at all; instead, it will be handed to the XML wrapper application as a large “chunk” of character data. Other than that special case, whitespace should be avoided as a data representation, and assumptions about which document callback will report whitespace should not be made.

Whitespace, Just the Whitespace

We have already addressed most of the issues with whitespace. We simply need to add this last SAX callback to our `MyContentHandler` class. The `ignoreableWhitespace()` method takes parameters in the exact same format as the `characters()` method, and should use the starting and ending indexes provided to read from the character array supplied:

```
/**
 * <p>
 * This will report whitespace that can be ignored in the
```

```

*   originating document. This is typically only invoked when
*   validation is occurring in the parsing process.
* </p>
*
* @param ch <code>char[]</code> character array with character data
* @param start <code>int</code> index in array where data starts.
* @param end <code>int</code> index in array where data ends.
* @throws <code>SAXException</code> when things go wrong
*/
public void ignorableWhitespace(char[] ch, int start, int end)
    throws SAXException {

    String s = new String(ch, start, end);
    System.out.println("ignorableWhitespace: [" + s + "]);
}

```

Of course, our sample will not print out any visible content, as the `String` created from the character array will be made up completely of whitespace, so we enclose the output within brackets. Whitespace is reported in the same manner as character data; it can be handled with one callback, or a SAX parser may break up the whitespace and report it over several method invocations. In either case, the precautions we have already discussed about not making assumptions or counting on whitespace as textual data should be closely adhered to in order to avoid troublesome bugs in your applications.

Skipped Entities

As you recall, we had one entity reference in our document, the `OReillyCopyright` entity. When parsed and resolved, this results in another file being loaded, either from the local filesystem or some other URI. However, we are not requesting that validation occur in our document. An often overlooked facet of non-validating parsers is that they are not required to resolve entity references, and instead may skip them. This has caused some headaches before, as parser results may simply not include entity references that were expected. SAX 2.0 nicely accounts for this with a callback that is issued when an entity is skipped by a non-validating parser. The callback gives the name of the entity, which we will include in our output (although Apache Xerces does not exhibit this behavior, your parser may):

```

/**
* <p>
* This will report an entity that is skipped by the parser. This
* should only occur for non-validating parsers, and then is still
* implementation-dependent behavior.
* </p>
*
* @param name <code>String</code> name of entity being skipped

```

```
    * @throws <code>SAXException</code> when things go wrong
    */
    public void skippedEntity(String name) throws SAXException {
        System.out.println("Skipping entity " + name);
    }
}
```

Before you go trying to recreate this behavior, you should note that most established parsers will not skip entities, even if they are not validating. Apache Xerces, for example, will never invoke this callback; instead, the entity reference will be expanded and the result will be included in the data available after parsing. In other words, this is there for parsers to use, but you will be hard-pressed to find a case where it crops up! If you do have a parser that exhibits this behavior, be aware that the parameter passed does not include the leading ampersand and trailing semicolon in the entity reference. For `&OReillyCopyright;`, only the name of the entity, `OReillyCopyright`, is passed to `skippedEntity()`.

The Results

Finally, we need to register our handler with the `XMLReader` we have instantiated. This is done with `setContentHandler()`, which takes a `ContentHandler` implementation as its single argument. Add the following lines to the `demo()` method of your parser example program:

```
/**
 * <p>
 * This parses the file, using registered SAX handlers, and outputs
 * the events in the parsing process cycle.
 * </p>
 *
 * @param uri <code>String</code> URI of file to parse.
 */
public void performDemo(String uri) {
    System.out.println("Parsing XML File: " + uri + "\n\n");

    // Get instances of our handlers
    ContentHandler contentHandler = new MyContentHandler();

    try {
        // Instantiate a parser
        XMLReader parser =
            new SAXParser();

        // Register the content handler
        parser.setContentHandler(contentHandler);

        // Parse the document
        parser.parse(uri);
    }
}
```

```

    } catch (IOException e) {
        System.out.println("Error reading URI: " + e.getMessage());
    } catch (SAXException e) {
        System.out.println("Error in parsing: " + e.getMessage());
    }
}

```

If you have entered in all of the document callbacks as we have gone along, you should be able to compile the `MyContentHandler` class and the enclosing `SAXParserDemo` file. Once done, you may run the SAX parser demonstration on our XML sample file created earlier. The complete Java command should read:

```
D:\prod\JavaXML> java SAXParserDemo D:\prod\JavaXML\contents\contents.xml
```

This should result in a fairly long and verbose output. If you are on a Windows machine, you may need to increase the buffer size of your DOS window so you may scroll and view the complete command output. The output should look similar to that in Example 3-2.*

Example 3-2. SAXParserDemo Output

```
D:\prod\JavaXML>java SAXParserDemo D:\prod\JavaXML\contents.xml
Parsing XML File: D:\prod\JavaXML\contents.xml
```

```

    * setDocumentLocator() called
Parsing begins...
Mapping starts for prefix JavaXML mapped to URI
    http://www.oreilly.com/catalog/javaxml/
startElement: Book in namespace
    http://www.oreilly.com/catalog/javaxml/ (JavaXML:Book)
characters:

startElement: Title in namespace
    http://www.oreilly.com/catalog/javaxml/ (JavaXML:Title)
characters: Java and XML
endElement: Title

characters:

startElement: Contents in namespace
    http://www.oreilly.com/catalog/javaxml/ (JavaXML:Contents)
characters:

```

* In this and other output examples, note that carriage returns may have been inserted to ensure that the output is formatted correctly on the printed page. As long as the actual content is the same, you have got everything working correctly!

Example 3-2. SAXParserDemo Output (continued)

```
startElement: Chapter in namespace
  http://www.oreilly.com/catalog/javaxml/ (JavaXML:Chapter)
  Attribute: focus=XML
characters:

startElement: Heading in namespace
  http://www.oreilly.com/catalog/javaxml/ (JavaXML:Heading)
characters: Introduction
endElement: Heading

characters:

startElement: Topic in namespace
  http://www.oreilly.com/catalog/javaxml/ (JavaXML:Topic)
  Attribute: subSections=7
characters: What Is It?
endElement: Topic

characters:

startElement: Topic in namespace
  http://www.oreilly.com/catalog/javaxml/ (JavaXML:Topic)
  Attribute: subSections=3
characters: How Do I Use It?
endElement: Topic

characters:

startElement: Topic in namespace
  http://www.oreilly.com/catalog/javaxml/ (JavaXML:Topic)
  Attribute: subSections=4
characters: Why Should I Use It?
endElement: Topic

characters:

startElement: Topic in namespace
  http://www.oreilly.com/catalog/javaxml/ (JavaXML:Topic)
  Attribute: subSections=0
characters: What's Next?
endElement: Topic
...
```

This output should go on quite a while, as the XML document being parsed has a number of elements within it. You can clearly see exactly how the parser sequentially handles each element, the element's attributes, any data within the element, nested elements, and the element's end tag. This process repeats for each element within the document. In our example, a non-validating instance of the

Xerces parser was used (remember our rather confusing discussion on this?), so whitespace is being reported with the `characters()` callback; in the next two chapters we will discuss validation and see how this reporting changes.

You have now seen how a SAX-compliant parser handles a well-formed XML document. You should also be getting an understanding of the document callbacks that occur within the parsing process and how an application can use these callbacks to get information about an XML document as it is parsed. In the next two chapters, we will spend time looking at validating an XML document by using additional SAX classes designed for handling DTDs. Before moving on, though, we want to address the issue of what happens when your XML document is not valid, and the errors that can result from this condition.

Error Handlers

In addition to providing the `ContentHandler` interface for handling parsing events, SAX provides an `ErrorHandler` interface that can be implemented to treat various error conditions that may arise during parsing. This class works in the same manner as the document handler we have already constructed, but only defines three callback methods. Through these three methods, all possible error conditions are handled and reported by SAX parsers.

Each method receives information about the error or warning that has occurred through a `SAXParseException`. This object holds the line number that trouble was encountered on, the URI of the document being treated, which could be the parsed document or an external reference within that document, and normal exception details such as a message and a printable stack trace. In addition, each method can throw a `SAXException`. This may seem a bit odd at first; an exception handler that throws an exception? Keep in mind that what each handler receives is a parsing exception. This can be a warning that should not cause the parsing process to stop or an error that needs to be resolved for parsing to continue; however, the callback may need to perform system I/O or another operation that can throw an exception, and it needs to be able to bubble this exception up the application chain. It can do this through the `SAXException` the method is allowed to throw.

For example, consider an error handler that receives error notifications and writes those errors to an error log. This method needs to be able to either append to or create an error log on the local filesystem. If a warning were to occur within the process of parsing an XML document, the warning would be reported to this method. The intent of the warning would be to give information to the callback and then continue parsing the document. However, if the error handler could not write to the log file, it might need to notify the parser and application that all parsing should stop. This can be done by catching any I/O exceptions and re-throwing

these to the calling application, thus causing any further document parsing to stop. This common scenario is why error handlers must be able to throw exceptions (see Example 3-3).

Example 3-3. Error Handler That May Throw a SAXException

```
public void warning(SAXParseException exception)
    throws SAXException {

    try {
        FileWriter fw = new FileWriter("error.log");
        BufferedWriter bw = new BufferedWriter(fw);
        bw.write("Warning: " + exception.getMessage() + "\n");
        bw.flush();
        bw.close();
        fw.close();
    } catch (Exception e) {
        throw new SAXException("Could not write to log file", e);
    }
}
```

We can now define the skeleton of our error handler and register it with our parser in the same way we registered our document handler. First we need to add the `SAXParseException` class and `ErrorHandler` interface to our import statements:

```
import java.io.IOException;
import org.xml.sax.Attributes;
import org.xml.sax.ContentHandler;
import org.xml.sax.ErrorHandler;
import org.xml.sax.Locator;
import org.xml.sax.SAXException;
import org.xml.sax.SAXParseException;
import org.xml.sax.XMLReader;
```

We should now create a class within the same Java file (again at the bottom, after the `MyContentHandler` class) to implement the `ErrorHandler` interface defined by SAX. Like our discussion of `ContentHandler`, empty implementations are provided here that we fill in the next section:

```
/**
 * <b><code>MyErrorHandler</code></b> implements the SAX
 * <code>ErrorHandler</code> interface and defines callback
 * behavior for the SAX callbacks associated with an XML
 * document's errors.
 */
class MyErrorHandler implements ErrorHandler {

    /**
     * <p>
     * This will report a warning that has occurred; this indicates
```

```

    * that while no XML rules were broken, something appears
    * to be incorrect or missing.
    * </p>
    *
    * @param exception <code>SAXParseException</code> that occurred.
    * @throws <code>SAXException</code> when things go wrong
    */
    public void warning(SAXParseException exception)
        throws SAXException {
    }

    /**
    * <p>
    * This will report an error that has occurred; this indicates
    * that a rule was broken, typically in validation, but that
    * parsing can reasonably continue.
    * </p>
    *
    * @param exception <code>SAXParseException</code> that occurred.
    * @throws <code>SAXException</code> when things go wrong
    */
    public void error(SAXParseException exception)
        throws SAXException {
    }

    /**
    * <p>
    * This will report a fatal error that has occurred; this indicates
    * that a rule has been broken that makes continued parsing either
    * impossible or an almost certain waste of time.
    * </p>
    *
    * @param exception <code>SAXParseException</code> that occurred.
    * @throws <code>SAXException</code> when things go wrong
    */
    public void fatalError(SAXParseException exception)
        throws SAXException {
    }
}

```

Finally, in preparation to use our custom error handler, we need to register this error handler with our SAX parser. This is done with the `setErrorHandler()` method of the `XMLReader` interface, and occurs in our example's `demo()` method. This method takes the `ErrorHandler` interface or an implementation of that interface as the single parameter:

```

// Get instances of our handlers
ContentHandler contentHandler = new MyContentHandler();
ErrorHandler errorHandler = new MyErrorHandler();

```

```
try {
    // Instantiate a parser
    XMLReader parser =
        new SAXParser();

    // Register the content handler
    parser.setContentHandler(contentHandler);

    // Register the error handler
    parser.setErrorHandler(errorHandler);

    // Parse the document
    parser.parse(uri);

} catch (IOException e) {
    System.out.println("Error reaaading URI: " + e.getMessage());
} catch (SAXException e) {
    System.out.println("Error in parsing: " + e.getMessage());
}
...

```

Now let's take a look at making these methods give us some feedback when they are invoked.

Warnings

Any time a warning (as defined by the XML 1.0 specification) occurs, this method is invoked in the registered error handler. There are several conditions that can generate a warning; however, all of them are related to the DTD and validity of a document, and we will discuss them in the next two chapters rather than here. For now, we need to define a simple method that prints out the line number, URI, and warning message when a warning occurs. Because we want any warnings to stop parsing, we throw a `SAXException` and let the wrapping application exit gracefully, cleaning up any used resources:

```
/**
 * <p>
 * This will report a warning that has occurred; this indicates
 * that while no XML rules were "broken", something appears
 * to be incorrect or missing.
 * </p>
 *
 * @param exception <code>SAXParseException</code> that occurred.
 * @throws <code>SAXException</code> when things go wrong
 */
public void warning(SAXParseException exception)
    throws SAXException {

```

```

        System.out.println("***Parsing Warning**\n" +
            " Line:    " +
                exception.getLineNumber() + "\n" +
            " URI:      " +
                exception.getSystemId() + "\n" +
            " Message: " +
                exception.getMessage());
        throw new SAXException("Warning encountered");
    }

```

Non-Fatal Errors

Errors that occur within parsing that can be recovered from, but constitute a violation of some portion of the XML specification, are considered non-fatal errors. An error handler should always at least log these, as they are typically serious enough to merit informing the user or administrator of an application, if not so critical as to cause parsing to cease. Like warnings, most non-fatal errors are concerned with validation, and will be covered in the relevant chapters. Also like warnings, we want our simple error handler to report information sent to the callback method and exit the parsing process:

```

/**
 * <p>
 * This will report an error that has occurred; this indicates
 * that a rule was broken, typically in validation, but that
 * parsing can reasonably continue.
 * </p>
 *
 * @param exception <code>SAXParseException</code> that occurred.
 * @throws <code>SAXException</code> when things go wrong
 */
public void error(SAXParseException exception)
    throws SAXException {

        System.out.println("***Parsing Error**\n" +
            " Line:    " +
                exception.getLineNumber() + "\n" +
            " URI:      " +
                exception.getSystemId() + "\n" +
            " Message: " +
                exception.getMessage());
        throw new SAXException("Error encountered");
    }

```

Fatal Errors

Fatal errors are those that necessitate stopping the parser. These are typically related to a document not being well-formed, and make further parsing either a complete waste of time or technically impossible. An error handler should almost

always notify the user or application administrator when a fatal error occurs; without intervention, these can bring an application to a shuddering halt. For our example, we want to emulate the behavior of the other two callback methods and stop parsing and write an error message to the screen when a fatal error is encountered:

```
/**
 * <p>
 * This will report a fatal error that has occurred; this indicates
 * that a rule has been broken that makes continued parsing either
 * impossible or an almost certain waste of time.
 * </p>
 *
 * @param exception <code>SAXParseException</code> that occurred.
 * @throws <code>SAXException</code> when things go wrong
 */
public void fatalError(SAXParseException exception)
    throws SAXException {

    System.out.println("***Parsing Fatal Error**\n" +
        "   Line:   " +
        exception.getLineNumber() + "\n" +
        "   URI:    " +
        exception.getSystemId() + "\n" +
        "   Message: " +
        exception.getMessage());
    throw new SAXException("Fatal Error encountered");
}
```

With this third error handler coded, you should be able to compile the example source file successfully, and run it on our XML file once again. Your output should not be any different than it was earlier, as there are no reportable errors, in the XML. We will next demonstrate some errors in non-validated XML documents.

Breaking the Data

Now that we have some error handlers in place, it is possible to view some of these handlers in action. As mentioned several times, most warnings and non-fatal errors are associated with document validity issues, which we will address in the next few chapters. However, there is one non-fatal error that can result from a non-validated XML document. This involves the version of XML that a document reports. To view this error, make the following change to the XML table of contents example:

```
<?xml version="1.2"?>

<!-- We don't need these yet
<?xml-stylesheet href="XSL\JavaXML.html.xsl" type="text/xsl"?>
<?xml-stylesheet href="XSL\JavaXML.wml.xsl" type="text/xsl"
    media="wap"?>
```

```

<?cocoon-process type="xslt"?>
<!DOCTYPE JavaXML:Book SYSTEM "DTD\JavaXML.dtd">
-->

```

You should now attempt to run the Java parser example program on the modified XML document. Your output should be similar to that in Example 3-4.

Example 3-4. SAXParserDemo Output Issuing an Error

```

D:\prod\JavaXML>java SAXParserDemo D:\prod\JavaXML\contents.xml
Parsing XML File: D:\prod\JavaXML\contents.xml

```

```

**Parsing Error**
Line:    1
URI:     file:/e:/prod/JavaXML/contents.xml
Message: XML version "1.2" is not supported.

```

When an XML parser is operating upon a document that reports a version of XML greater than that supported by the parser, a non-fatal error is reported, in accordance with the XML 1.0 Specification. This allows an application to know that newer features expected to be utilized by the document may not be available within the parser and the version that it supports. Because parsing is still able to continue, this is a non-fatal error. However, because it signifies a major impact on the document (such as newer syntax possibly generating subsequent errors), it is considered more important than a warning. This is why our `error()` method is invoked and triggers the error message and parsing halt in the example program.

All other meaningful warnings and non-fatal errors will be discussed in the next two chapters; still, there are a variety of fatal errors that a non-validated XML document may have. These are related to an XML document not being well-formed. There is no logic built into XML parsers to try to resolve or estimate fixes to malformed XML, so an error in syntax results in the parsing process halting. The easiest way to demonstrate one of these errors is to introduce problems within our XML document. Reset the XML declaration to specify XML Version 1.0, and make the following change to the XML document:

```

<?xml version="1.0"?>

<!-- We don't need these yet
<?xml-stylesheet href="XSL\JavaXML.html.xsl" type="text/xsl"?>
<?xml-stylesheet href="XSL\JavaXML.wml.xsl" type="text/xsl"
      media="wap"?>
<?cocoon-process type="xslt"?>
<!DOCTYPE JavaXML:Book SYSTEM "DTD\JavaXML.dtd">
-->

```

```

<!-- Java and XML -->
<JavaXML:Book xmlns:JavaXML="http://www.oreilly.com/catalog/javaxml/">
  </JavaXML:Title>Java and XML</JavaXML:Title>
  <!-- Note the incorrect slash before the JavaXML:Title element -->

  <JavaXML:Contents>

```

This is no longer a well-formed document. To see the fatal error that parsing this document generates, run the `SAXParserDemo` on this modified file (the output is shown in Example 3-5).

Example 3-5. SAXParserDemo Output Issuing a Fatal Error

```

D:\prod\JavaXML>java SAXParserDemo D:\prod\JavaXML\contents.xml
Parsing XML File: D:\prod\JavaXML\contents.xml

```

```

* setDocumentLocator() called
Parsing begins...
startElement: Book in namespace
    http://www.oreilly.com/catalog/javaxml/ (JavaXML:Book)
  Attribute: xmlns:JavaXML=http://www.oreilly.com/catalog/javaxml/
characters:

**Parsing Fatal Error**
Line:    12
URI:     file:/e:/prod/xml-book/contents.xml
Message: The element type "JavaXML:Book" must be terminated by the
         matching end-tag "</JavaXML:Book>".

```

The parser reports an incorrect ending to the `JavaXML:Book` element. This fatal error is exactly as we expected; parsing could not continue beyond this error. To understand the error message, you should realize that the parser sees the slash character before the `JavaXML:Title` element, and makes the assumption that the element that must be closed is the `JavaXML:Book` element, the current “open” element. When it finds a closing tag for the `JavaXML:Title` element, it reports that the tag is incorrect for the closing of the open element, `JavaXML:Book`.

With our error handler, we have begun to understand what can go wrong within the parsing process, as well as how to handle those events. In Chapter 5, we will revisit our error handlers and look at the problems that can be reported by the validating parser.

A Better Way to Load a Parser

Although we now have a successful demonstration of SAX parsing, there is a glaring problem with our code. Let’s take a look again at how we obtain an instance of `XMLReader`:

```

try {
    // Instantiate a parser
    XMLReader parser =
        new SAXParser();

    // Register the content handler
    parser.setContentHandler(contentHandler);

    // Register the error handler
    parser.setErrorHandler(errorHandler);

    // Parse the document
    parser.parse(uri);

} catch (IOException e) {
    System.out.println("Error reading URI: " + e.getMessage());
} catch (SAXException e) {
    System.out.println("Error in parsing: " + e.getMessage());
}

```

Do you see anything that rubs you wrong? Let's look at another line of our code that may give you a hint:

```

// Import your vendor's XMLReader implementation here
import org.apache.xerces.parsers.SAXParser;

```

We have to explicitly import our vendor's `XMLReader` implementation, and then instantiate that implementation directly. The problem here is not the difficulty of this task, but that we have broken one of Java's biggest tenets: portability. Our code cannot run or even be compiled on a platform that does not use the Apache Xerces parser. In fact, it is conceivable that an updated version of Xerces might even change the name of the class used here! Our "portable" Java code is no longer very portable.

What is preferred is to request an instance of a class by the name of the implementation class. This allows a simple `String` parameter to be changed in your source code. Luckily, this facility is available in SAX 2.0. The `org.xml.sax.helpers.XMLReaderFactory` class provides the method you should be looking for:

```

/**
 * Attempt to create an XML reader from a class name.
 *
 * <p>Given a class name, this method attempts to load
 * and instantiate the class as an XML reader.</p>
 *
 * @return A new XML reader.
 * @exception org.xml.sax.SAXException If the class cannot be
 *         loaded, instantiated, and cast to XMLReader.
 * @see #createXMLReader()
 */

```

```
public static XMLReader createXMLReader (String className)
    throws SAXException {

    // Implementation
}
```

We can use this method in our code like this:

```
try {
    // Instantiate a parser
    XMLReader parser =
        XMLReaderFactory.createXMLReader(
            "org.apache.xerces.parsers.SAXParser");

    // Register the content handler
    parser.setContentHandler(contentHandler);

    // Register the error handler
    parser.setErrorHandler(errorHandler);

    // Parse the document
    parser.parse(uri);

} catch (IOException e) {
    System.out.println("Error reading URI: " + e.getMessage());
} catch (SAXException e) {
    System.out.println("Error in parsing: " + e.getMessage());
}
```

This static method takes in the name of the parser class to load and returns an instantiated version of the class, cast to the `XMLReader` interface (assuming that it actually does implement `XMLReader`). If any problems occur, they are all handled and then wrapped in a `SAXException` that is thrown to the calling program. Add in the additional import statement, remove the vendor-specific parser reference, make the changes noted above, and you should be able to recompile your source file:

```
import java.io.IOException;

import org.xml.sax.Attributes;
import org.xml.sax.ContentHandler;
import org.xml.sax.ErrorHandler;
import org.xml.sax.Locator;
import org.xml.sax.SAXException;
import org.xml.sax.SAXParseException;
import org.xml.sax.XMLReader;
import org.xml.sax.helpers.XMLReaderFactory;

// This goes away
// import org.apache.xerces.parsers.SAXParser;
```

Suddenly you are writing portable code again! To further portability, it makes sense to store the name of the parser class in a properties file. This allows easy loading of the class at runtime, and it means your code can be moved from platform to platform without recompilation based on the parser being used; instead, only the properties file could be changed. Although the code to read properties files is not provided here, take a look at some code that performs this behavior:

```
try {
    // Instantiate a parser
    XMLReader parser =
        XMLReaderFactory.createXMLReader(
            PropertiesReader().getInstance()
                .getProperty("parserClass"));

    // Register the content handler
    parser.setContentHandler(contentHandler);

    // Register the error handler
    parser.setErrorHandler(errorHandler);

    // Parse the document
    parser.parse(uri);

} catch (IOException e) {
    System.out.println("Error reading URI: " + e.getMessage());
} catch (SAXException e) {
    System.out.println("Error in parsing: " + e.getMessage());
}
```

Here a utility class, `PropertiesReader`, is being used to read a properties file and return the value for the key “`parserClass`”, which would contain the parser class name to use on the specific platform the code was being used for. In our examples, this would be our old friend `org.apache.xerces.parsers.SAXParser`. Of course, Java system properties could also be used, but they are not as appropriate for web-centric distributed applications like the ones we focus on in this book, as they must be specified on a command line. Often, distributed applications are started up in whole, rather than individually, making specification of a system property to one particular component difficult.

“Gotcha!”

Before leaving our introduction to parsing XML documents, there are a few pitfalls to make you aware of. These “gotchas” will help you avoid common programming mistakes when using SAX, and we will discuss more of these for other APIs in the appropriate sections.

My Parser Doesn't Support SAX 2.0: What Can I Do?

For those of you who are unlucky enough not to have a parser with SAX 2.0 support, don't despair. First, you always have the option of changing parsers; keeping current on SAX standards is an important part of an XML parser's responsibility, and if your vendor is not doing this, you may have other concerns to address with them as well. However, there are certainly cases where you are forced to use a parser because of legacy code or applications; in these situations, you are still not “left out in the cold.”

SAX 2.0 includes a helper class, `org.xml.sax.helpers.ParserAdapter`, which can actually cause a SAX 1.0 Parser implementation to behave like a SAX 2.0 `XMLReader` implementation. This handy class takes in a 1.0 Parser implementation as an input parameter and then can be used in the stead of that implementation. It allows a `ContentHandler` to be set, and handles all namespace callbacks properly. The only feature loss you will see is that skipped entities will not be reported, as this capability was not available in a 1.0 implementation in any form, and cannot be emulated by a 2.0 adapter class. The sample class would be used as shown in Example 3-6.

Example 3-6. Using a SAX 1.0 Parser as a 2.0 XMLReader

```
try {
    // Register a parser with SAX
    Parser parser =
        ParserFactory.makeParser(
            "org.apache.xerces.parsers.SAXParser");

    ParserAdapter myParser = new ParserAdapter(parser);

    // Register the document handler
    myParser.setContentHandler(contentHandler);

    // Register the error handler
    myParser.setErrorHandler(errHandler);

    // Parse the document
    myParser.parse(uri);
} catch (ClassNotFoundException e) {
    System.out.println(
        "The parser class could not be found.");
} catch (IllegalAccessException e) {
    System.out.println(
        "Insufficient privileges to load the parser class.");
} catch (InstantiationException e) {
    System.out.println(
```

Example 3-6. Using a SAX 1.0 Parser as a 2.0 XMLReader (continued)

```
        "The parser class could not be instantiated.");
    } catch (ClassCastException e) {
        System.out.println(
            "The parser does not implement org.xml.sax.Parser");
    } catch (IOException e) {
        System.out.println("Error readding URI: " + e.getMessage());
    } catch (SAXException e) {
        System.out.println("Error in parsing: " + e.getMessage());
    }
}
```

If SAX is new to you and this example doesn't make much sense, don't worry about it; you are using the latest and greatest version of SAX (2.0) and probably won't ever have to write code like this. Only in cases where a 1.0 parser must be used is this code helpful.

The SAX XMLReader: Reused and Reentrant

One of Java's nicest features is the ease of reuse of objects, and the memory advantages of this reuse. SAX parsers are no different. Once an `XMLReader` has been instantiated, it is possible to continue using that parser, parsing several or even hundreds of XML documents. Different documents or `InputSources` may be continually passed to a parser, allowing it to be used for a variety of different tasks. However, parsers are not reentrant. Once the parsing process has started, a parser may not be used until the parsing of the requested document or input has completed. For those of you who are prone to coding recursive methods, this is definitely a "gotcha!" The first time that you attempt to use a parser that is in the middle of processing another document, you will receive a rather nasty `SAXException` and all parsing will stop. What is the lesson learned? Parse one document at a time, or pay the price of instantiating multiple parser instances.

The Misplaced Locator

Another dangerous but seemingly innocuous feature of SAX events is the `Locator` instance that is made available through the `setDocumentLocator()` callback method. This gives the application the origin of a SAX event, and is useful for making decisions about the progress of parsing and how to react to events. However, this origin point is only valid for the duration of the life of the `ContentHandler` instance; once parsing is complete, the `Locator` is no longer valid, including in the case when another parse begins. A "gotcha" that many XML newcomers make is to hold a reference to the `Locator` object within a class member variable outside of the callback method:

```
public void setDocumentLocator(Locator locator) {
    // Saving the Locator to a class outside the ContentHandler
    myOtherClass.setLocator(locator);
}
```

```

}
...

public myOtherClassMethod() {
    // Trying to use this outside of the ContentHandler
    System.out.println(locator.getLineNumber());
}

```

This is an extremely bad idea, as this Locator becomes meaningless as soon as the scope of the ContentHandler implementation is left. Often, using the member variable resulting from this operation results in not only erroneous information being supplied to an application, but corruption of the XML document that was parsed. In other words, use this object locally, and not globally. In our ContentHandler implementation, we saved the supplied Locator to a member variable. It could then correctly be used (for example) to give you the line number of each element as it was encountered:

```

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

    System.out.print("startElement: " + localName +
        " at line " + locator.getLineNumber());

    if (!namespaceURI.equals("")) {
        System.out.println(" in namespace " + namespaceURI +
            " (" + rawName + ")");
    } else {
        System.out.println(" has no associated namespace");
    }

    for (int i=0; i<atts.getLength(); i++)
        System.out.println(" Attribute: " + atts.getLocalName(i) +
            "=" + atts.getValue(i));
}

```

Getting Ahead of the Data

The characters() callback method accepts a character array and start and end parameters to signify which index to start and end reading of that array from. This can cause some confusion; a common mistake is to include code like this example to read from the character array:

```

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

    for (int i=0; i<ch.length; i++)
        System.out.print(i);
}

```

The mistake here is in reading from the beginning to the end of the character array. This natural “gotcha” results from years of iterating through arrays, either in Java, C, or another language. However, in the case of a SAX event, this can cause quite a bug. SAX parsers are required to pass in starting and ending boundaries on the character array which any loop constructs should use to read from the array. This allows lower-level manipulation of textual data to occur to optimize parser performance, such as reading data ahead of the current location as well as array reuse. This is all legal behavior within SAX, as the expectation is that a wrapping application will not try to “get ahead” of the `end` parameter sent to the callback.

Mistakes as in the example shown can result in gibberish data being output to the screen or used within the wrapping application, and are almost always problematic for applications. The loop construct looks very normal and compiles without a hitch, so this “gotcha” can be a very tricky problem to track down.

What’s Next?

You should now have a solid understanding of the SAX interfaces and how they interact with an XML parser and the parsing process, with regard to a non-validated XML document. These interfaces are key to the rest of our discussions and Java code, as we will expand on our knowledge of SAX and add additional SAX classes to our example program. In the next chapter, we will look at how an XML document can be validated, and cover an XML document’s DTD and schema. These will teach you how to constrain an XML document, and then in the chapter after that, we will look at implementing validation in our example parsing code.