Just Enough SAX

Chapter 10

In this chapter, we examine one of the most widely implemented APIs used to process XML documents. SAX (Simple API for XML) is a simple, event-oriented API that supports the event-driven programming style discussed in chapter 8.

10.1 History

The easiest route to an understanding of what SAX is and why it was created is via some background information about the very early days of XML and XML software development. The story begins with Peter Murray-Rust—pioneering XML software developer and creator of the xml-dev mailing list.[1]

[1] This mailing list has become something of an institution in the XML development world. To subscribe, send an e-mail to majordomo@scml.org with the message: subscribe xml-dev.

Peter was one of the first developers to create an XML-based application in the form of Jumbo—an XML browser based on the Java programming language. Jumbo began life as browser for XML documents conforming to the Chemical Markup Language DTD (CML). Since then, it has expanded in scope to become a general-purpose XML editing/viewing tool.

Originally, Peter developed his own XML parsing routines for Jumbo in the Java programming language. As development of XML progressed, powerful XML parsers began to appear, notably Ælfred by Dave Megginson, Lark by Tim Bray, and NXP by Norbert Mikula. Peter wanted to be able to configure Jumbo to use these different parsers. That is, he wanted to be able to swap XML parsers in and out of Jumbo without making code changes to Jumbo itself. The ability to swap XML parsers in an application is useful for a number of reasons:

- Parsers differ in their level of conformance to the XML specification.
- Parsers differ in the amount of memory they use.
- Parsers differ in the speed with which they parse XML.
- Nonvalidating parsers differ in their treatment of certain XML features such as external entities and defaulted attributes.
- Parsers differ in the quality of location information provided with error messages.

As things stood at the time, swapping parsers was not easy. It became evident to Peter Murray-Rust and others communicating on the xml-dev list that the emerging family of XML parsers had common core event-driven functionality for example, notifying applications when start-tags, end-tags, processing instructions, character data, and so on appear in the XML document.

However, each parser exposed this functionality *differently* in its API. Take the concept of notifying an application of the presence of character data as an example. One parser might provide a `characters` method, another might use the name `chardata`. One might provide an array of characters with a length indicator, another might provide a string object, and so on.

Peter christened the phenomenon of multiple incompatible APIs to XML parsers "YAXPAPI" (Yet Another XML Parser API). Discussion on the xml-dev mailing list led to the idea of a standardized API that would allow XML application developers to swap XML parsers in and out of their applications without code changes at the application level.

The name SAX was adopted for the standardization effort.[2] Work on SAX started in December 1997, with David Megginson doing the lion's share of the work. David made regular postings to xml-dev on SAX design questions, often with his own view of the pros and cons of each choice. Enthusiastic discussion would follow each new set of questions or design choices. David would assimilate all the debate, make a decision, and move on to the next set of design decisions.

This process continued at breakneck speed for a month or so, and the first draft of SAX appeared during January of 1998. The API was developed in the Java programming language but care was taken not to introduce language dependencies into it. Where functionality was considered useful but specific to the Java programming language, it was separated out into a language-specific section to make it easier for developers to port SAX to other languages.

Five months of implementation experience and fine tuning followed, culminating in the birth of SAX 1.0 in May 1998. Jon Bosak of Sun Microsystems, one of XML's founding fathers and chairman of the W3C's XML Activity, kindly offered use of his `xml.org` domain name for the Java version of the SAX package for the Java platform; that domain now bears the name `org.xml.sax`.

[2] At one point, the acronym JAX—Java API for XML— was considered instead of SAX. I pointed out that this is somewhat rude in my country (slang for, um, restroom) and SAX was used instead.

The decision to avoid Java programming language dependencies in SAX soon bore fruit. Python was the first to emerge with a full-featured SAX implementation developed by Lars Marius Garshol. There is a C++ version, and a Perl implementation is ongoing. Other languages are expected to follow suit as SAX support in the XML industry grows.

Although SAX has proved very popular among the XML development community, it has no official standing within the W3C unlike, for example, the DOM API, which was developed by a W3C working group. (The DOM API is the subject of chapter 11.)

Although SAX is not *wired* to the Java programming language or to object-oriented languages, it is fair to say that SAX maps more easily to languages that support an object-oriented approach. An understanding of OO concepts is very useful in understanding SAX. In particular, an understanding of the Java platform concept of an interface is important.

10.2 | The Concept of an "Interface"

Chapter 5 briefly mentioned the Java platform concept of an *interface* in the context of Python's type system. If you are comfortable with the material in section 5.3 from that chapter, then the Java platform concept of an interface should readily make sense to you. If you are less than comfortable with that material, this coverage might solidify it for you.

Interfaces are closely related to the object-oriented programming concept of *inheritance*. Inheritance in turn is closely related to the way our minds organize concepts into hierarchies. These hierarchies form a powerful and natural way to model concepts in software. For example, my mental hierarchy for airplanes is shown in figure 10–1.

This might be a good hierarchy with which to model aircraft in software and it might not. It all depends on the application under development. Another perfectly good airplane hierarchy is shown in figure 10–2.

Clearly, there are many ways to build these hierarchies. A key skill in object-oriented design is picking the best hierarchy given the task at hand and potential future applications of the models under consideration. Continuing with the aeronautical theme, how would you or-

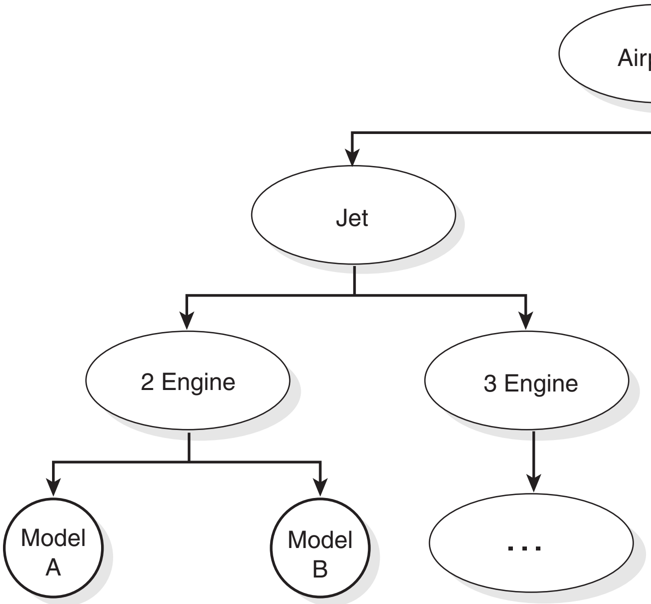


Figure 10–1 A hierarchy of airplane types.

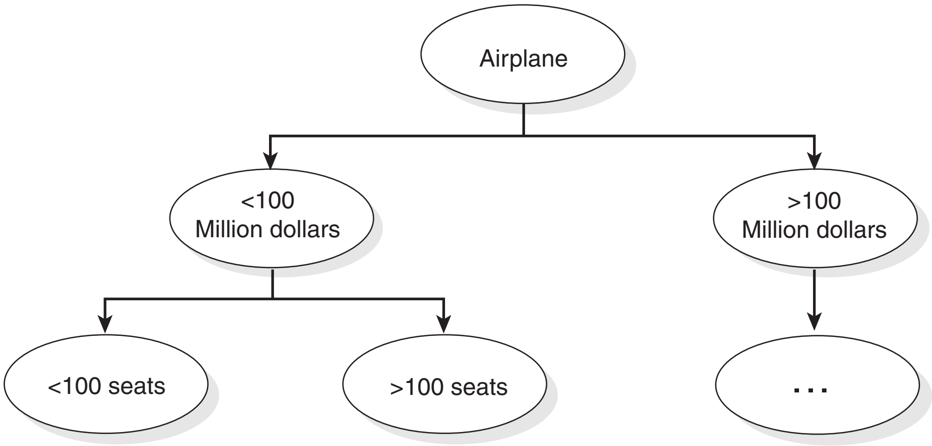


Figure 10–2 An alternative airplane type hierarchy.

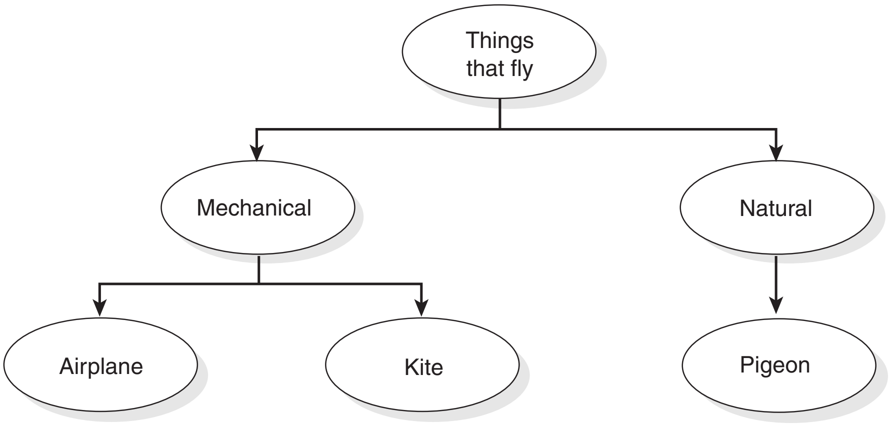


Figure 10–3 A hierarchy incorporating kites, pigeons, and airplanes.

ganize the concepts of airplane, kite, and pigeon into a hierarchy? One way to do it is shown in figure 10–3.

Now, airplanes and pigeons are very different things. There is a good chance that real-world software concerned with airplanes and pigeons would not use a hierarchy like that illustrated in figure 10–3. That said, airplanes and pigeons are clearly *related* by the fact that they both can fly.

As a software writer, you would expect to be able to ask software models of airplanes and pigeons to fly—even if they belong to completely different class hierarchies. This idea is illustrated in figure 10–4.

In figure 10–4, there are two classification hierarchies: Fixed Assets and Animals. Although airplanes and pigeons are in different hierarchies, we would like to be able to ask both of them to fly. In object-oriented design parlance, we would like to be able to send them both the *fly* message.

Relating this to XML processing, we might have classes in application areas as diverse as financial trading, hospital administration, and molecular biology, all requiring XML-processing capabilities. Each class is likely to be part of a natural hierarchy of classes to do with financial trading, hospital administration, etc.

One way to provide XML-processing capabilities would be to get each class to inherit from the XML-processing class in addition to its application area class. This technique is known as *multiple inheritance*.

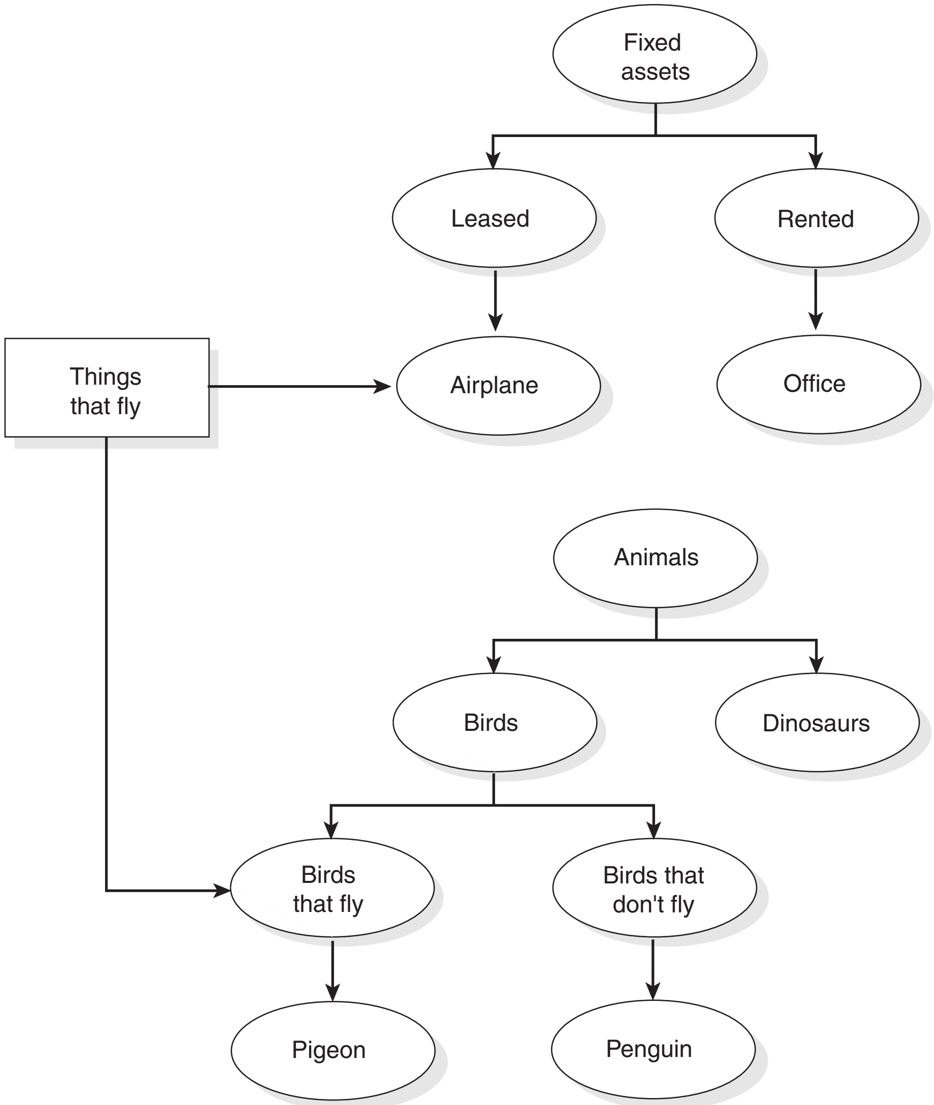


Figure 10–4 Combining hierarchies.

In the code fragment below, the class `myFinancialTradingSystem` is classified as both a `FinancialTradingSystem` and an `XMLProcessor`.

```
CD-ROM reference=10001.txt

class XMLProcessor:
      def StartElement(self,etn,attrs):
            pass

      def EndElement(self,etn,attrs):
            pass

      ...

class myFinancalTradingSystem (FinancialTradingSystem,
  XMLProcessor):
      def StartElement(self,etn,attrs):
            # Do something with an XML element and its
            # attributes

      def ShortSqueezeTheSilverMarket(self,Amount):
            # Perform a trade
```

Multiple inheritance is supported by a variety of object-oriented languages, including Python and C++. However, multiple inheritance can cause problems for language implementors and programmers alike. We will not go into the debate here. Suffice it to say that multiple inheritance is not universally acknowledged as a good idea.

The designers of the Java platform opted not to support multiple inheritance but to provide a facility to get the advantages of it without the perceived disadvantages. This is the Java concept of an *interface*.

Simply put, classes in the Java programming language are organized into single inheritance hierarchies. Any given class will have one and only one base class. However, any given class can support any number of interfaces. An interface specification is a collection of message names. Any class that provides implementations of the messages in the interface specification is said to *implement* the interface. By making the interface specification explicit in the language, Java compilers can check to make sure that classes that promise to implement interfaces do so properly.

Developers using SAX, for example, will state in their Java source code that the application implements one or more SAX interfaces.

The Java compiler can then check the source code to ensure that all the methods that need to be defined to implement the methods in the SAX interfaces have indeed been implemented.

Python does not currently provide any syntax for expressing interface specifications. As in Smalltalk before it, interfaces are more of a common convention than a language feature. It is up to the Python programmer to ensure that classes that should implement a particular interface actually do so.

10.3 | Overview of the SAX Specification

SAX consists of seven interfaces and three classes expressed in Java programming language syntax. The number of interfaces involved might sound somewhat daunting at first. In reality, only a subset of these are likely to be of interest to the majority of XML application developers because SAX caters to the needs of both XML parser developers and XML application writers.

The seven interfaces are:

- **`DocumentHandler`** Handling of start-tag, end-tags, character data, and so on.
- **`AttributeList`** Handling attributes
- **`Parser`** Parsing XML documents
- **`ErrorHandler`** Handling of parsing errors
- **`DTDHandler`** Handling of notation and entity declarations
- **`EntityResolver`** Handling of locating external entities
- **`Locator`** Supplying of location information from the parser to the application

The three classes are:

- **`HandlerBase`** A class that implements default document handling interfaces
- **`InputSource`** A class that provides all the information needed about an XML entity
- **`SAXException`** SAX exception handling

We will concentrate on one class and three interfaces:

- `HandlerBase` class
- `DocumentHandler` interface
- `AttributeList` interface
- `ErrorHandler` interface

10.4 | The HandlerBase Class

Four of the seven SAX interfaces are concerned with document handling:

- `DocumentHandler`
- `ErrorHandler`
- `DTDHandler`
- `EntityResolver`

We will take a close look at the first two of these later on. Together, these four interfaces comprise 14 different methods for handling aspects of document processing. The `HandlerBase` class implements

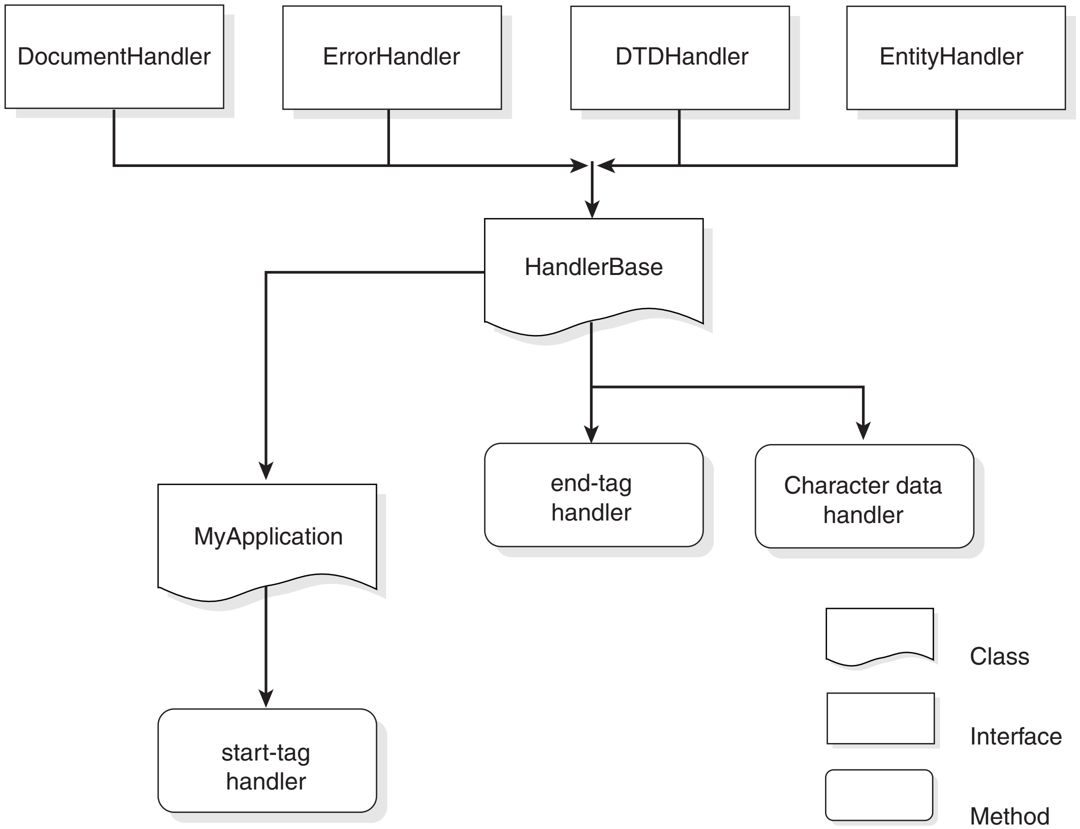


Figure 10–5 Implementing an interface.

default methods for all four interfaces. That is, all 14 methods required by the interfaces are defined, but they do not do anything.

Why is this useful? It is useful because it makes writing SAX applications very easy—simply make your class inherit from `HandlerBase` and you have implemented all four interfaces. You can simply override the methods that you actually need rather than provide stub implementions of those you do not need.

This concept is illustrated in figure 10–5.

The `HandlerBase` class lives in the file `saxlib.py`. It looks like this.

```
CD-ROM reference=10002.txt
class HandlerBase(EntityResolver,
                          DTDHandler,
                          DocumentHandler,
                          ErrorHandler):

    def __init__(self):
        pass
```

The handlers for the methods that make up each of the four interfaces are found in their respective classes. The `HandlerBase` class inherits the implementation of these methods through Python's inheritance mechanism.

Here is a fragment of the `DocumentHandler` class that shows some of the stub method implementations.

```
CD-ROM reference=10003.txt
class DocumentHandler:
    def startElement(self, name, atts):
        "Handle an event for the beginning of an element."
        pass

    def characters(self, ch, start, length):
        "Handle a character data event."
        pass

       ...
```

For example, imagine an application that is interested in nothing except start-tags. All that is necessary is a class derived from `HandlerBase` that implements the `startElement` method.

```
CD-ROM reference=10004.txt
from xml.sax import saxexts, saxlib, saxutils

class myHandler (saxlib.HandlerBase):
      def startElement (self, ElementTypeName, Attributes):
            print "Start-tag for ", ElementTypeName
```

10.5 | The DocumentHandler Interface

DocumentHandler is the main interface in SAX from a SAX application development perspective. It establishes handlers for start-tags, end-tags, processing instructions, character data, and so on. There are eight methods in the interface. The full stub interface looks like this.

CD-ROM reference=10005.txt

```
class DocumentHandler:
    def characters(self, ch, start, length):
        "Handle a character data event."
        pass

    def endDocument(self):
        "Handle an event for the end of a document."
        pass

    def endElement(self, name):
        "Handle an event for the end of an element."
        pass

    def ignorableWhitespace(self, ch, start, length):
        "Handle an event for ignorable whitespace in element
          content."
        pass

    def processingInstruction(self, target, data):
        "Handle a processing instruction event."
        pass

    def setDocumentLocator(self, locator):
        "Receive an object for locating the origin of SAX
          document events."
        pass

    def startDocument(self):
        "Handle an event for the beginning of a document."
        pass

    def startElement(self, name, atts):
        "Handle an event for the beginning of an element."
        pass
```

We take a closer look at each method and provide a code example of each one in the sections below.

10.5.1 *The startDocument Method*

This method is called at the very start of a document parse. SAX guarantees that this method is called before any other methods in the interface with the exception of `setDocumentLocator`. This makes it very useful for housing initialization code such as opening files, allocating data structures, and so on.

```
CD-ROM reference=10006.txt
from xml.sax import saxlib

class MySAXApplication (saxlib.HandlerBase):
      def startDocument(self):
            print "Start of Document"
```

10.5.2 *The endDocument Method*

This method is called at the very end of a document parse. It is a handy place to put cleanup code such as closing files, deleting data structures, and so on.

```
CD-ROM reference=10007.txt
from xml.sax import saxlib

class MySAXApplication (saxlib.HandlerBase):
      def endDocument(self):
            print "End of Document"
```

10.5.3 *The startElement Method*

This method is called when a start-tag and associated attributes are recognized in the XML document. The `Attributes` parameter implements another SAX interface called AttributeList, which is discussed later.

Note that the AttributeList object is *transient*. It is only valid for the life of the call to startElement. To keep a permanent copy of an attribute list, you must make a copy of it during this method.

```
CD-ROM reference=10008.txt
from xml.sax import saxlib

class MySAXApplication (saxlib.HandlerBase):
      def startElement(self,ElementTypeName, Attributes):
            LocalAttributes = []
            print "Start of Element", ElementTypeName
            for i in range (0,Attributes.getLength()):
                  # Copy out attribute name and value into
                  # local list.
                  LocalAttributes.append (
                        Attributes.getName(i),
                        Attributes.getValue(i))
```

10.5.4 *The endElement Method*

This method is called whenever an end-tag is recognized in the XML document.

```
CD-ROM reference=10009.txt
from xml.sax import saxlib

class MySAXApplication (saxlib.HandlerBase):
      def endElement(self,ElementTypeName):
            print "End of Element ", ElementTypeName
```

Note that empty elements also trigger endElement events. For empty elements, the endElement event happens directly after the startElement event.

10.5.5 *The characters Method*

This method is called when character data is recognized in the XML document. The method takes three parameters: the first is a string, the second is an offset into the string at which the new chunk of character data starts, and the third parameter gives the length of the chunk.

```
CD-ROM reference=10010.txt
from xml.sax import saxlib

class MySAXApplication (saxlib.HandlerBase):
      def characters(self,chars,startOffset,Length):
            print "Character data", chars[startOffset:
              startOffset+Length]
```

An important aspect of XML parsers in general is that you cannot predict how much data will arrive in each call to the character handler. An XML parser is free to hand back chunks of character data in whatever sizes it likes. For example, consider this XML document:

```
CD-ROM reference=10011.txt
<test>
Hello World
</test>
```

A SAX application will receive one call to `startElement`, one call to `endElement`, but one or more calls to the `characters` method.

A second important point is that if the parser is nonvalidating, all character data—including white space—is routed through this method. If the parser is a validating XML parser, some white space (known in XML as ignorable white space) will be routed to the `ignorableWhitespace` method) discussed below.

10.5.6 *The ignorableWhitespace Method*

This method is called when ignorable white space is recognized in an XML document. Validating XML parsers are required to differentiate between normal and ignorable white space. In theory, nonvalidating parsers can do this if they have parsed and understood the element content models, but they are not required to differentiate by the XML 1.0 recommendation.

```
CD-ROM reference=10012.txt
from xml.sax import saxlib

class MySAXApplication(saxlib.HandlerBase)
      def ignorableWhitespace(self,chars,startOffset,Length):
            print "Ignorable White Space",
            print chars[startOffset:startOffset:Length]
```

10.5.7 *The processingInstruction Method*

This method is called when a processing instruction is recognized in the XML document.

```
CD-ROM reference=10013.txt
from xml.sax import saxlib

class MySAXApplication(saxlib.HandlerBase):
      def processingInstruction(self,target,data):
            print "Processing Instruction Target",target
            print align print "Processing Instruction Data",data
```

10.5.8 *The setDocumentLocator Method*

The SAX specification strongly urges, but does not require, implementors to provide a means of tracking locations in XML documents via the `Locator` interface. The `setDocumentLocator` method will be called by the parser with a `Locator` object that can be used to find out about where the XML parser is in the source XML document.

The parser calls this method once at the very start of the parsing process—that is, before the `startDocument` event. In the example below, a SAX application prints out the line number for each start-tag event.

```
CD-ROM reference=10014.txt
from xml.sax import saxlib

class MySAXApplication (saxlib.HandlerBase):
      def setDocumentLocator (self,L):
            self.MyLocatorObject = L
```

```
        def startElement (self,ElementTypeName,Attributes):
            print "start-tag for ", ElementTypeName, "on line"
            print self.MyLocatorObject.getLineNumber()
```

The intent of the `Locator` interface is to provide the application with the location of the first piece of data after the data that caused the event. For example, in the document below, the line number associated with the `test` start-tag event is 4.

```
CD-ROM reference=10015.txt
<test
x = "1"
y = "2"
>Hello World
</test>
```

10.6 | The AttributeList Interface

As we have seen, the `startElement` handler in the `DocumentHandler` interface receives attributes in an object that implements the `AttributeList` interface.

The Python implementation of the `AttributeList` interface is in the file `saxlib.py` and looks like this.

```
CD-ROM reference=10016.txt
class AttributeList:
    def getLength(self):
        "Return the number of attributes in list."
        pass

    def getName(self, i):
        "Return the name of an attribute in the list."
        pass

    def getType(self, i):
        """Return the type of an attribute in the list.
        (Parameter can be either integer index or attribute
        name.)"""
        pass

    def getValue(self, i):
        """Return the value of an attribute in the list.
        (Parameter can be either integer index or attribute
        name.)"""
        pass
```

We take a closer look at each method and provide a code example of each one in the sections below.

10.6.1 *The getLength Method*

This method retrieves the number of attributes associated with a start-tag.

```
CD-ROM reference=10017.txt
from xml.sax import saxlib

class MySAXApplication (saxlib.HandlerBase):
      def startElement (self,ElementTypeName,a):
            print "Start-tag for element ", ElementTypeName
            print "Total Attributes = ", a.getLength()
```

10.6.2 *The getName Method*

This method retrieves the name of an attribute associated with an element start-tag. It takes an integer parameter and returns the i'th parameter in the `AttributeList` structure. This method is primarily used in iterations in conjunction with the `getLength` method. In the example below, the `startElement` handler prints the names of the attributes associated with the start-tag.

```
CD-ROM reference=10018.txt
from xml.sax import saxlib
class MySAXApplication (saxlib.HandlerBase):

      def startElement (self,ElementTypeName, a):
            print "Start-tag for element ", ElementTypeName
            i = 0
            while i < a.getLength():
                  print "Attribute Name ",a.getName(i)
                  i = i + 1
```

Since attribute ordering is never significant in XML, you cannot know in what order the attributes will appear. That is, just because the attributes appear in a particular order in the source document does not mean they will appear in that order in the `AttributeList` structure.

10.6.3 *The getValue Method*

This method retrieves the value associated with an attribute. The method comes in two flavors. It can be called with an integer parameter to return the value of the i'th attribute in the `AttributeList`. Alternatively, it can be called with a string parameter. The value of the attribute with the name matching the string parameter is returned.

In this example, the integer parameter version of `getValue` is used with the `getLength` method.

```
CD-ROM reference=10019.txt
from xml.sax import saxlib

class MySAXApplication (saxlib.HandlerBase):

      def startElement (self,ElementTypeName,a):
            print "Start-tag for element ",ElementTypeName
            AttributeCount = a.getLength()
            i = 0
            while i < a.getLength():
                  print "Attribute value ",a.getValue(i)
```

In the example below, the attribute name `quantity` is passed into `getValue` to retrieve the value of the attribute named `quantity`.

```
CD-ROM reference=10020.txt
from xml.sax import saxlib

class MySAXApplication (saxlib.HandlerBase):

      def startElement (self,ElementTypeName,a):
            print "Start-tag for element ",ElementTypeName
            Quantity = a.getValue("quantity")
```

10.6.4 *The getType Method*

This method retrieves the type associated with an attribute. Like getValue, the method comes in two flavors. It can be called with an integer parameter to return the type of the i'th attribute in the AttributeList. Alternatively, it can be called with a string parameter to return the type of the attribute with the matching name.

In this example, the integer parameter version of getType is used with the getLength method to print the types of all the attributes.

```
CD-ROM reference=10021.txt
from xml.sax import saxlib

class MySAXApplication (saxlib.HandlerBase):

      def startElement (self, ElementTypeName, a):
            print "Start-tag for element ",ElementTypeName
            AttributeCount = a.getLength()
            i = 0
            while i < a.getLength():
                  print "Attribute type ",a.getType(i)
                  i = i + 1
```

The attribute type is returned as a string. It can have the following values:

- CDATA
- ENTITY
- ENTITIES
- NOTATION
- ID
- IDREF
- IDREFS
- NMTOKEN
- NMTOKENS

Nonvalidating parsers are not required to process attribute declarations. As a result, they can return CDATA for all of the attribute types listed above.

10.7 | The ErrorHandler Interface

The ErrorHandler interface gives SAX application developers control over what happens when parsing errors are encountered. The XML 1.0 recommendation differentiates between two types of errors:

- **error—**This is a violation of the rules of the XML 1.0 specification. Conforming software is allowed to both detect and attempt to recover from this type of error, but it must report the error to the application. For example, it is an error for an element to have two attributes of type ID.
- **fatal error—**This is an error that conforming software must detect and report to the application. After a fatal error, a parser is allowed to continue processing in order to detect more errors, but it must not continue to parse the XML. In SAX terms, this means it must cease calling handlers in the DocumentHandler interface.

The ErrorHandler interface supports these two types of XML errors, using methods called, naturally enough, error and fatalError. The interface also supports a third type of error known as *warning*. This is intended for conditions that are neither errors nor fatal errors as defined in XML 1.0. For example, if a parser encounters multiple attribute declarations for the same element type, it may issue a warning. This is not an error condition and processing must continue unaffected.

The ErrorHandler interface class in saxlib.py looks like this.

```
CD-ROM reference=10022.txt
class ErrorHandler:
    def error(self, exception):
        "Handle a recoverable error."
        pass

    def fatalError(self, exception):
        "Handle a non-recoverable error."
        pass

    def warning(self, exception):
        "Handle a warning."
        pass
```

By default, SAX parsers ignore all errors except for fatal errors, for which they throw exceptions. To get the parser to stop throwing exceptions and call the warning/fatal/error methods of the `ErrorHandler` interface, the SAX application calls the `setErrorHandler` method of the `Parser` object, as shown below.

```
CD-ROM reference=10023.txt
from xml.sax import saxlib

class MySAXHandler (saxlib.HandlerBase):

      def error (self,ErrorInfo):
            print "Error %d %d" % (
                  ErrorInfo.getLineNumber(),
                  ErrorInfo.getColumnNumber()
                  }

      def warning (self,ErrorInfo)
            print "Warning %d %d" % (
                  ErrorInfo.getLineNumber(),
                  ErrorInfo.getColumnNumber())

      def fatalError (self,ErrorInfo)
            print "Fatal Error %d %d" % (
                  ErrorInfo.getLineNumber(),
                  ErrorInfo.getColumnNumber())
            sys.exit()
# Main line
# Create a new XML parser object.
Parser = XMLParser()

# Create an instance of the MySAXHandler class.
```

```
Handler = MySAXHandler()

# Tell the parser object where the DocumentHandler is.
Parser.setDocumentHandler (Handler)

# Tell the parser object where the errorHandler is.
Parser.setErrorHandler (Handler)

// Parse a document.
Parser.parse("greeting.xml")
```

10.8 | A SAX Inspection Application

In this section, we develop a useful application that converts a sequence of SAX events into an XML document. This application is a useful debugging tool because it allows you to see at a glance the precise sequence of events generated by any SAX-compliant parser.

Given an XML document like this

```
CD-ROM reference=10024.txt
<Greeting x = "y">Hello World</Greeting>
```

the application generates this:

```
CD-ROM reference=10025.txt
<?xml version="1.0"?>
<!DOCTYPE SAXShow SYSTEM "SAXShow.dtd">
<SAXShow>
<Document>
<Element name="Greeting">
<Attribute name = "x" type = "CDATA" value="y"/>
<chars>Hello World</chars>
</Element>
</Document>
</SAXShow>
```

We will start with the DTD for the event sequence.

```
CD-ROM reference=10026.txt
<!ELEMENT SAXShow (DocumentLocator?,Document)>
```

```
<!ELEMENT Document (PI*,(Element|WS|PI|chars)*)>
<!ELEMENT Element (Attribute*,(Element|WS|PI|chars)*)>
<!ATTLIST Element name CDATA #REQUIRED>
<!ELEMENT Attribute EMPTY>
<!ATTLIST Attribute
              name  CDATA #REQUIRED
              type  CDATA #REQUIRED
              value CDATA #REQUIRED>
<!ELEMENT PI EMPTY>
<!ATTLIST PI
              target CDATA #REQUIRED
              data CDATA #REQUIRED>
<!ELEMENT DocumentLocator EMPTY>
<!ELEMENT WS (#PCDATA)>
<!ELEMENT chars (#PCDATA)>
```

Note how DTD syntax crisply captures quite a lot of information about the time ordering of SAX-generated events. Here is the code for saxshow.py.

```
CD-ROM reference=10027.txt
C>type saxshow.py
"""
A utility to create an XML document describing the order in
which SAX events have been called in processing an XML docu-
ment
"""
from xml.sax import saxlib,saxexts

class myHandler (saxlib.HandlerBase):
      """
      SAX document handler to create an XML document conforming
      to the saxshow DTD
      """
    def characters(self, ch, start, length):
            "Character data handler"
            print "<chars>%s</chars>" % ch[start:start+length]

    def endDocument(self):
            "End of document handler"
            print "</Document>"

    def endElement(self, name):
            "End of element handler"
            print "</Element>"

    def ignorableWhitespace(self, ch, start, length):
            "Ignorable white space (validating XML parsers only)"
            print '<WS>%s</WS>"  % ch[start:start+length]

    def processingInstruction(self, target, data):
            "Processing instruction handler"
            print '<PI target="%s" data="%s"/>' % (target,data)

    def setDocumentLocator(self, locator):
            "Parser has provided a document locator object"
            print '<SetDocumentLocator/>'

    def startDocument(self):
            "Start of document handler"
            print '<Document>'

    def startElement(self, name, atts):
            "Start of element handler"
            print '<Element name="%s">' % name
            for i in range (0,atts.getLength()):
                  print '<Attribute name = "%s"' %
                    atts.getName(i)
                        'type = "%s"' % atts.getType(i)
                        'value="%s"/>' % atts.getValue(i)
def SAXShow(fo):
      # Given an XML file object (fo), parse the XML, generating
      # an output XML document conforming to the saxshow DTD.
      # Create a document event handler.
      h = myHandler()
      # Create a SAX parser.
      parser = saxexts.make_parser()
      # Tell the parser about the document handler.
      parser.setDocumentHandler(h)
      # Print header of output document.
      print '<?xml version="1.0"?>'
      print '<!DOCTYPE SAXShow SYSTEM "SAXShow.dtd">'
      # Root element is SAXShow.
      print '<SAXShow>'
      parser.parseFile (fo)
      print '</SAXShow>'

if __name__ == "__main__":
      import sys
      SAXShow (open(sys.argv[1],"r"))
```

The following simple XML file illustrates the SAXShow application in action.

```
CD-ROM reference=10028.txt
<names>
<name x = "y">
Mr. Sean Mc Grath
</name>
<name>
Mr. Stephen Murphy
</name>
<name>
Mr. Sandy Duffy
</name>
</names>
```

```
CD-ROM reference=10029.txt
C>python SAXShow.py test.xml
<?xml version="1.0"?>
<!DOCTYPE SAXShow SYSTEM "SAXShow.dtd">
<SAXShow>
<Document>
<Element name="names">
<chars>
</chars>
<Element name="name">
<Attribute name = "x" type = "CDATA" value="y"/>
<chars>
</chars>
<chars>Mr. Sean Mc Grath</chars>
<chars>
</chars>
</Element>
<chars>
</chars>
<Element name="name">
<chars>
</chars>
<chars>Mr. Stephen Murphy</chars>
<chars>
</chars>
</Element>
<chars>
</chars>
```

```
<Element name="name">
<chars>
</chars>
<chars>Mr. Sandy Duffy</chars>
<chars>
</chars>
</Element>
<chars>
</chars>
</Element>
</Document>
</SAXShow>
```

We can validate this XML document and generate PYX at the same time by piping the output through `xmlv`.

```
CD-ROM reference=10030.txt

C>python saxshow.py test.xml | xmlv
(SAXShow
-\n
(Document
-\n
(Element
Aname
-\n
...
```

We can validate the generated XML document without generating any PYX by redirecting standard output to the `null` device.

Windows

```
CD-ROM reference=10031.txt
C>python saxshow.py test.xml | xmlv >nul
```

No output appears on standard output, indicating that the generated XML document passed a validating XML parse.

Linux

```
CD-ROM reference=10032.txt
$python saxshow.py test.xml | xmlv /dev/null
```

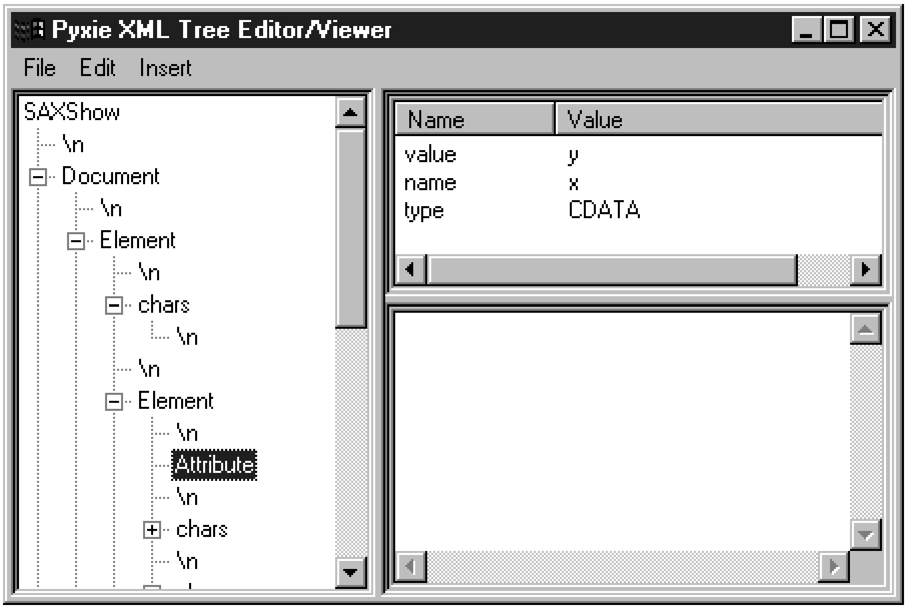
No output appears on standard output, indicating that the generated XML document passed a validating XML parse.

We can get a graphical view of the SAX event sequence order by simply displaying the resultant document in the C3 viewer.

```
CD-ROM reference=10033.txt
C>python saxshow.py test.xml > res.xml
C>python c3.py res.xml
```

A screenshot of res.xml in the C3 viewer is shown in figure 10–6.

Note how making the output an XML file has led to some nice functionality for manipulating these files "for free." We can validate their structure with xmlv, we can view them in c3, we can use the XML-aware searching facilities of xgrep to locate particular SAX event sequences, and so on.



Name	Value
value	y
name	x
type	CDATA

Figure 10–6 A SAXShow document viewed with the C3 XML viewer.

10.9 | SAX as a Source of PYX

We now have covered all the material on SAX required to build a SAX application that generates PYX. This facility will be part of the Pyxie library later on, but a standalone version of the routine is shown here to illustrate a SAX application.

```
CD-ROM reference=10034.txt
#! /usr/bin/env python

"""
Utility to create PYX notation from the default SAX-compliant
parser

Sean Mc Grath
XML Processing with Python
"""

# Import the SAX modules.
from xml.sax import saxexts, saxlib, saxutils
import string

def Encode (s):
      """
      Convert line ends to the two character sequence
      '\' followed by 'n'.
      """
      return  string.replace (s,"\n","\\n")

class myHandler (saxlib.DocumentHandler):
      def startElement(self,Element,Attributes):
            print "(%s" % Element
            # PYX attributes must be in alphabetical order
            # by attribute name.
            # Assemble a list of (name,value) pairs and
            # sort them.
            ListOfAttributes = []
            for i in range (0,Attributes.getLength()):
                  ListOfAttributes.append
                    ((Attributes.getName(i),
Encode(Attributes.getValue(i))))
            ListOFAttributes.sort()
            for (a,v) in ListOfAttributes:
                  print "A%s %s" % (a,v)

      def characters(self,data,offset,length):
            print "-%s" % Encode(data[offset:offset+length])

      def processingInstruction (target, data):
            print "?target data"

      def endElement(self,Element):
            print ")%s"  % Element

      def error (self,exception):
            print "Parsing problem",exception
            sys.exit()

      def fatalErrror(self,exception):
            self.error(exception)

      def warning(Self,exception):
            self.error(exception)

def SAX2PYX(fo):
      # Create a SAX Document Handler.
      h = myHandler()
      # Create a SAX Parser.
      parser = saxexts.make_parser()
      # Tell the Parser where the document handler is.
      parser.setDocumentHandler(h)
      # Tell the Parser where the error handler is.
      parser.setErrorHandler(h)
      # Parse the file.
      parser.parseFile (fo)

if __name__ == "__main__":
      import sys
      SAX2PYX(open(sys.argv[1]))
```

We illustrate `SAX2PYX` in action with this XML document.

```
CD-ROM reference=10035.txt
C>type test.xml
```

```
<names>
<name x = "y">
Mr. Sean Mc Grath
</name>
<name>
Mr. Stephen Murphy
</name>
<name>
Mr. Sandy Duffy
</name>
</names>

C>python sax2pyx.py test.xml

(names
-\n
(name
Ax y
-\n
-Mr. Sean Mc Grath
-\n
)name
-\n
(name
-\n
-Mr. Stephen Murphy
-\n
)name
-\n
(name
-\n
-Mr. Sandy Duffy
-\n
)name
-\n
)names
```

10.10 | Switching SAX Parsers

As mentioned at the start of this chapter, one of the big advantages of basing an XML application on SAX is that you can switch from XML parser to XML parser without making code changes.

In the demonstration programs of this chapter, you have seen numerous calls to the make_parser function provided in the saxexts (SAX extensions) module.

```
CD-ROM reference=10036.txt
parser = saxexts.make_parser()
```

When invoked with no parameters, make_parser will return the first SAX-compliant parser it finds installed. You can specify a parser to use by giving its name as a parameter to the make_parser call. In the example below, James Clark's expat parser is selected.

```
CD-ROM reference=10037.txt
    parser = saxexts.make_parser("xml.sax.drivers.drv_pyexpat")
```

To see a full list of the SAX drivers installed on your system, look in the xml/sax/drivers subdirectory. This subdirectory also gives you the correct names to use when requesting specific parsers with the make_parser() function. Table 10.1 provides a partial list.

Table 10.1 Drivers for SAX-Compliant Parsers

Driver	*Description*
drv_htmllib.py	The HTML parsing library from the Python distribution.
drv_pyexpat.py	James Clark's expat non-validating XML parser. A good parser to use when parsing speed is of the essence.
drv_sgmllib.py	The SGML parsing library from the Python distribution. This parser is written in Python. It is a non-validating SGML parser. It makes no attempt to infer the presence of any tagging.
drv_sgmlop.py	The optimized SGML parsing library from the Python distribution by Fredrick Lundh. Use this as a plug-in replacement if you have previously used drv_sgmllib for a big speed increase.

(continued)

Table 10.1 Drivers for SAX-Compliant Parsers (continued)

Driver	*Description*
`drv_xmldc.py`	An XML scanner by Dan Connolly.
`drv_xmllib.py`	The XML parser from the Python distribution by Sjoerd Mullender. This is a non-validating XML parser written in Python.
`drv_xmlproc.py`	The non-validating version of `xmlproc` by Lars Marius Garshol. This is written in Python.
`drv_xmlproc_val.py`	The validating version of `xmlproc` by Lars Marius Garshol. This is the only validing XML parser in the Python XML package.
`drv_xmltoolkit.py`	The XML parser by David Scheres.