Chapter 28

Working with XML

This chapter introduces Scala's support for XML. After discussing semi-structured data in general, it shows the essential functionality in Scala for manipulating XML: how to make nodes with XML literals, how to save and load XML to files, and how to take apart XML nodes using query methods and pattern matching. This chapter is just a brief introduction to what is possible with XML, but it shows enough to get you started.

28.1 SEMI-STRUCTURED DATA

XML is a form of *semi-structured data*. It is more structured than plain strings, because it organizes the contents of the data into a tree. Plain XML is less structured than the objects of a programming language, though, as it admits free-form text between tags and it lacks a type system.[1]

Semi-structured data is very helpful any time you need to serialize program data for saving in a file or shipping across a network. Instead of converting structured data all the way down to bytes, you convert it to and from semi-structured data. You then use pre-existing library routines to convert between semi-structured data and binary data, saving your time for more important problems.

There are many forms of semi-structured data, but XML is the most widely used on the Internet. There are XML tools on most operating systems, and most programming languages have XML libraries available. Its popularity is self-reinforcing. The more tools and libraries are developed in response to XML's popularity, the more likely software engineers are to choose XML as part of their formats. If you write software that communicates over the Internet, then sooner or later you will need to interact with some service that speaks XML.

For all of these reasons, Scala includes special support for processing XML. This chapter shows you Scala's support for constructing XML, processing it with regular methods, and processing it with Scala's pattern matching. In addition to these nuts and bolts, the chapter shows along the way several common idioms for using XML in Scala.

28.2 XML OVERVIEW

XML is built out of two basic elements, text and tags.[2] Text is, as usual, any sequence of characters. Tags, written like <pod>, consist of a less-than sign, an alphanumeric label, and a greater than sign. Tags can be *start* or *end* tags. An end tag looks just like a start tag except that it has a slash just before the tag's label, like this: </pod>.

Start and end tags must match each other, just like parentheses. Any start tag must eventually be followed by an end tag with the same label. Thus the following is illegal:

```
// Illegal XML
One <pod>, two <pod>, three <pod> zoo
```

Further, the contents of any two matching tags must itself be valid XML. You cannot have two pairs of matching tags overlap each other:

```
// Also illegal
<pod>Three <peas> in the </pod></peas>
```

You could, however, write it like this:

```
<pod>Three <peas></peas> in the </pod>
```

Since tags are required to match in this way, XML is structured as nested *elements*. Each pair of matching start and end tags forms an element, and elements may be nested within each other. In the above example, the entirety of <pod>Three <peas></peas> in the </pod> is an element, and <peas></peas> is an element nested within it.

Those are the basics. Two other things you should know are, first, there is a shorthand notation for a start tag followed immediately by its matching end tag. Simply write one tag with a slash put after the tag's label. Such a tag comprises an *empty element*. Using an empty element, the previous example could just as well be written as follows:

```
<pod>Three <peas/> in the </pod>
```

Second, start tags can have *attributes* attached to them. An attribute is a name-value pair written with an equals sign in the middle. The attribute name itself is plain, unstructured text, and the value is surrounded by either double quotes ("") or single quotes ('). Attributes look like this:

```
<pod peas="3" strings="true"/>
```

28.3 XML LITERALS

Scala lets you type in XML as a literal anywhere that an expression is valid. Simply type a start tag and then continue writing XML content. The compiler will go into an XML-input mode and will read content as XML until it sees the end tag matching the start tag you began with:

```
scala> <a>
         This is some XML.
         Here is a tag: <atag/>
       </a>
res0: scala.xml.Elem =
<a>
  This is some XML.
  Here is a tag: <atag/>
</a>
```

The result of this expression is of type Elem, meaning it is an XML element with a label ("a") and children ("This is some XML...," *etc.*). Some other important XML classes are:

- Class Node is the abstract superclass of all XML node classes.
- Class Text is a node holding just text. For example, the "stuff" part of <a>stuff is of classText.

- Class NodeSeq holds a sequence of nodes. Many methods in the XML library processNodeSeqs in places you might expect them to process individual Nodes. You can still use such methods with individual nodes, however, since Node extends from NodeSeq. This may sound weird, but it works out well for XML. You can think of an individual Node as a one-elementNodeSeq.

You are not restricted to writing out the exact XML you want, character for character. You can evaluate Scala code in the middle of an XML literal by using curly braces ({}) as an escape. Here is a simple example:

```
scala> <a> {"hello" + ", world"} </a>
res1: scala.xml.Elem = <a> hello, world </a>
```

A braces escape can include arbitrary Scala content, including further XML literals. Thus, as the nesting level increases, your code can switch back and forth between XML and ordinary Scala code. Here's an example:

```
scala> val yearMade = 1955
yearMade: Int = 1955

scala>  <a> { if (yearMade < 2000) <old>{yearMade}</old>
              else xml.NodeSeq.Empty }
        </a>
res2: scala.xml.Elem =
<a> <old>1955</old>
 </a>
```

If the code inside the curly braces evaluates to either an XML node or a sequence of XML nodes, those nodes are inserted directly as is. In the above example, if yearMade is less than 2000, it is wrapped in <old> tags and added to the <a> element. Otherwise, nothing is added. Note in the above example that "nothing" as an XML node is denoted with xml.NodeSeq.Empty.

An expression inside a brace escape does not have to evaluate to an XML node. It can evaluate to any Scala value. In such a case, the result is converted to a string and inserted as a text node:

```
scala> <a> {3 + 4} </a>
res3: scala.xml.Elem = <a> 7 </a>
```

Any <, >, and & characters in the text will be escaped if you print the node back out:

```
scala> <a> {"</a>potential security hole<a>"} </a>
res4: scala.xml.Elem = <a> &lt;/a&gt;potential security
hole&lt;a&gt; </a>
```

To contrast, if you create XML with low-level string operations, you will run into traps such as the following:

```
scala> "<a>" + "</a>potential security hole<a>" + "</a>"
res5: String = <a></a>potential security hole<a></a>
```

What happens here is that a user-supplied string has included XML tags of its own, in this case and <a>. This behavior can allow some nasty surprises for the original programmer, because

it allows the user to affect the resulting XML tree outside of the space provided for the user inside the <a> element. You can prevent this entire class of problems by always constructing XML using XML literals, not string appends.

28.4 SERIALIZATION

You have now seen enough of Scala's XML support to write the first part of a serializer: conversion from internal data structures to XML. All you need for this are XML literals and their brace escapes.

As an example, suppose you are implementing a database to keep track of your extensive collection of vintage Coca-Cola thermometers. You might make the following internal class to hold entries in the catalog:

```
abstract class CCTherm {
  val description: String
  val yearMade: Int
  val dateObtained: String
  val bookPrice: Int      // in US cents
  val purchasePrice: Int  // in US cents
  val condition: Int      // 1 to 10

  override def toString = description
}
```

This is a straightforward, data-heavy class that holds various pieces of information such as when the thermometer was made, when you got it, and how much you paid for it.

To convert instances of this class to XML, simply add a toXML method that uses XML literals and brace escapes, like this:

```
abstract class CCTherm {
  ...
  def toXML =
    <cctherm>
      <description>{description}</description>
      <yearMade>{yearMade}</yearMade>
      <dateObtained>{dateObtained}</dateObtained>
      <bookPrice>{bookPrice}</bookPrice>
      <purchasePrice>{purchasePrice}</purchasePrice>
      <condition>{condition}</condition>
    </cctherm>
}
```

Here is the method in action:

```
scala> val therm = new CCTherm {
         val description = "hot dog #5"
         val yearMade = 1952
         val dateObtained = "March 14, 2006"
         val bookPrice = 2199
         val purchasePrice = 500
         val condition = 9
       }
therm: CCTherm = hot dog #5
```

```
scala> therm.toXML
res6: scala.xml.Elem =
<cctherm>
               <description>hot dog #5</description>
               <yearMade>1952</yearMade>
               <dateObtained>March 14, 2006</dateObtained>
               <bookPrice>2199</bookPrice>
               <purchasePrice>500</purchasePrice>
               <condition>9</condition>
             </cctherm>
```

Note

The "new CCTherm" expression in the previous example works even thoughCCTherm is an abstract class, because this syntax actually instantiates an anonymous subclass of CCTherm. Anonymous classes were described inSection 20.5.

By the way, if you want to include a curly brace (`{' or `}') as XML text, as opposed to using them to escape to Scala code, simply write two curly braces in a row:

```
scala> <a> {{{{brace yourself!}}}} </a>
res7: scala.xml.Elem = <a> {{brace yourself!}} </a>
```

28.5 TAKING XML APART

Among the many methods available for the XML classes, there are three in particular that you should be aware of. They allow you to take apart XML without thinking too much about the precise way XML is represented in Scala. These methods are based on the XPath language for processing XML. As is common in Scala, you can write them directly in Scala code instead of needing to invoke an external tool.

Extracting text. By calling the text method on any XML node you retrieve all of the text within that node, minus any element tags:

```
scala> <a>Sounds <tag/> good</a>.text
res8: String = Sounds  good
```

Any encoded characters are decoded automatically:

```
scala> <a> input ---&gt; output </a>.text
res9: String = " input ---> output "
```

Extracting sub-elements. If you want to find a sub-element by tag name, simply call \with the name of the tag:

```
scala> <a><b><c>hello</c></b></a> ~"b"
res10: scala.xml.NodeSeq = NodeSeq(<b><c>hello</c></b>)
```

You can do a "deep search" and look through sub-sub-elements, *etc.*, by using \\ instead of the \ operator:

```
scala>  <a><b><c>hello</c></b></a> ~"c"
```

```
res11: scala.xml.NodeSeq = NodeSeq()

scala>  <a><b><c>hello</c></b></a> \~"c"
res12: scala.xml.NodeSeq = NodeSeq(<c>hello</c>)

scala>  <a><b><c>hello</c></b></a> ~"a"
res13: scala.xml.NodeSeq = NodeSeq()

scala>  <a><b><c>hello</c></b></a> \~"a"
res14: scala.xml.NodeSeq =
NodeSeq(<a><b><c>hello</c></b></a>)
```

Note

Scala uses \ and \\ instead of XPath's / and //. The reason is that // starts a comment in Scala! Thus, some other symbol has to be used, and using the other kind of slashes works well.

Extracting attributes. You can extract tag attributes using the same \ and \\ methods.Simply put an at sign (@) before the attribute name:

```
scala> val joe = <employee
          name="Joe"
          rank="code monkey"
          serial="123"/>
joe: scala.xml.Elem = <employee name="Joe" rank="code monkey"
 serial="123"/>

scala>  joe ~"@name"
res15: scala.xml.NodeSeq = Joe

scala>  joe ~"@serial"
res16: scala.xml.NodeSeq = 123
```

28.6 DESERIALIZATION

Using the previous methods for taking XML apart, you can now write the dual of a serializer, a parser from XML back into your internal data structures. For example, you can parse back aCCTherm instance by using the following code:

```
def fromXML(node: scala.xml.Node): CCTherm =
  new CCTherm {
    val description   = (node ~"description").text
    val yearMade      = (node ~"yearMade").text.toInt
    val dateObtained  = (node ~"dateObtained").text
    val bookPrice     = (node ~"bookPrice").text.toInt
    val purchasePrice = (node ~"purchasePrice").text.toInt
    val condition     = (node ~"condition").text.toInt
  }
```

This code searches through an input XML node, named node, to find each of the six pieces of data needed to specify a CCTherm. The data that is text is extracted with .text and left as is. Here is this method in action:

```
scala> val node = therm.toXML
node: scala.xml.Elem =
```

```
<cctherm>
               <description>hot dog #5</description>
               <yearMade>1952</yearMade>
               <dateObtained>March 14, 2006</dateObtained>
               <bookPrice>2199</bookPrice>
               <purchasePrice>500</purchasePrice>
               <condition>9</condition>
             </cctherm>

scala> fromXML(node)
res17: CCTherm = hot dog #5
```

28.7 LOADING AND SAVING

There is one last part needed to write a data serializer: conversion between XML and streams of bytes. This last part is the easiest, because there are library routines that will do it all for you. You simply have to call the right routine on the right data.

To convert XML to a string, all you need is toString. The presence of a workable toString is why you can experiment with XML in the Scala shell. However, it is better to use a library routine and convert all the way to bytes. That way, the resulting XML can include a directive that specifies which character encoding was used. If you encode the string to bytes yourself, then the onus is on you to keep track of the character encoding.

To convert from XML to a file of bytes, you can use the XML.save command. You must specify a file name and a node to be saved:

```
scala.xml.XML.save("therm1.xml", node)
```

After running the above command, the resulting file therm1.xml looks like the following:

```
<?xml version='1.0' encoding='UTF-8'?>
<cctherm>
              <description>hot dog #5</description>
              <yearMade>1952</yearMade>
              <dateObtained>March 14, 2006</dateObtained>
              <bookPrice>2199</bookPrice>
              <purchasePrice>500</purchasePrice>
              <condition>9</condition>
            </cctherm>
```

Loading is simpler than saving, because the file includes everything the loader needs to know. Simply call XML.loadFile on a file name:

```
scala> val loadnode = xml.XML.loadFile("therm1.xml")
loadnode: scala.xml.Elem =
<cctherm>
      <description>hot dog #5</description>
      <yearMade>1952</yearMade>
      <dateObtained>March 14, 2006</dateObtained>
      <bookPrice>2199</bookPrice>
      <purchasePrice>500</purchasePrice>
      <condition>9</condition>
    </cctherm>
```

```
scala> fromXML(loadnode)
res14: CCTherm = hot dog #5
```

Those are the basic methods you need. There are many variations on these loading and saving methods, including methods for reading and writing to various kinds of readers, writers, input and output streams.

28.8 PATTERN MATCHING ON XML

So far you have seen how to dissect XML using text and the XPath-like methods, \ and \\.These are good when you know exactly what kind of XML structure you are taking apart. Sometimes, though, there are a few possible structures the XML could have. Maybe there are multiple kinds of records within the data, for example because you have extended your thermometer collection to include clocks and sandwich plates. Maybe you simply want to skip over any white space between tags. Whatever the reason, you can use the pattern matcher to sift through the possibilities.

An XML pattern looks just like an XML literal. The main difference is that if you insert a {}escape, then the code inside the {} is not an expression but a pattern. A pattern embedded in{} can use the full Scala pattern language, including binding new variables, performing type tests, and ignoring content using the _ and _* patterns. Here is a simple example:

```
def proc(node: scala.xml.Node): String =
  node match {
    case <a>{contents}</a> => "It's an a: " + contents
    case <b>{contents}</b> => "It's a b: " + contents
    case _ => "It's something else."
  }
```

This function has a pattern match with three cases. The first case looks for an <a> element whose contents consist of a single sub-node. It binds those contents to a variable namedcontents and then evaluates the code to the right of the associated right arrow (=>). The second case does the same thing but looks for a <b> instead of an <a>, and the third case matches anything not matched by any other case. Here is the function in use:

```
scala> proc(<a>apple</a>)
res18: String = It's an a: apple
scala> proc(<b>banana</b>)
res19: String = It's a b: banana
scala> proc(<c>cherry</c>)
res20: String = It's something else.
```

Most likely this function is not exactly what you want, because it looks precisely for contents consisting of a single sub-node within the <a> or <b>. Thus it will fail to match in cases like the following:

```
scala> proc(<a>a <em>red</em> apple</a>)
res21: String = It's something else.
scala> proc(<a/>)
res22: String = It's something else.
```

If you want the function to match in cases like these, you can match against a sequence of nodes instead of a single one. The pattern for "any sequence" of XML nodes is written `_*'.Visually, this sequence looks like the wildcard pattern (_) followed by a regex-style Kleene star (*). Here is the updated function that matches a sequence of sub-elements instead of a single sub-element:

```
def proc(node: scala.xml.Node): String =
  node match {
    case <a>{contents @ _*}</a> => "It's an a: " + contents
    case <b>{contents @ _*}</b> => "It's a b: " + contents
    case _ => "It's something else."
  }
```

Notice that the result of the _* is bound to the contents variable by using the @ pattern described in Section 15.2. Here is the new version in action:

```
scala> proc(<a>a <em>red</em> apple</a>)
res23: String = It's an a: ArrayBuffer(a , <em>red</em>,
apple)
scala> proc(<a/>)
res24: String = It's an a: WrappedArray()
```

As a final tip, be aware that XML patterns work very nicely with for expressions as a way to iterate through some parts of an XML tree while ignoring other parts. For example, suppose you wish to skip over the white space between records in the following XML structure:

```
val catalog =
  <catalog>
    <cctherm>
      <description>hot dog #5</description>
      <yearMade>1952</yearMade>
      <dateObtained>March 14, 2006</dateObtained>
      <bookPrice>2199</bookPrice>
      <purchasePrice>500</purchasePrice>
      <condition>9</condition>
    </cctherm>
    <cctherm>
      <description>Sprite Boy</description>
      <yearMade>1964</yearMade>
      <dateObtained>April 28, 2003</dateObtained>
      <bookPrice>1695</bookPrice>
      <purchasePrice>595</purchasePrice>
      <condition>5</condition>
    </cctherm>
  </catalog>
```

Visually, it looks like there are two nodes inside the <catalog> element. Actually, though, there are five. There is white space before, after, and between the two elements! If you do not consider this white space, you might incorrectly process the thermometer records as follows:

```
catalog match {
  case <catalog>{therms @ _*}</catalog> =>
    for (therm <- therms)
      println("processing: " +
              (therm ~"description").text)
}
```

```
processing:
processing: hot dog #5
processing:
processing: Sprite Boy
processing:
```

Notice all of the lines that try to process white space as if it were a true thermometer record. What you would really like to do is ignore the white space and process only those sub-nodes that are inside a <cctherm> element. You can describe this subset using the pattern <cctherm>{_*}</cctherm>, and you can restrict the for expression to iterating over items that match that pattern:

```
catalog match {
  case <catalog>{therms @ _*}</catalog> =>
    for (therm @ <cctherm>{_*}</cctherm>  <-  therms)
      println("processing: " +
              (therm ~"description").text)
}

processing: hot dog #5
processing: Sprite Boy
```

28.9 CONCLUSION

This chapter has only scratched the surface of what you can do with XML. There are many other extensions, libraries, and tools you could learn about, some customized for Scala, some made for Java but usable in Scala, and some language-neutral. What you should walk away from this chapter with is how to use semi-structured data for interchange, and how to access semi-structured data via Scala's XML support.

Footnotes for Chapter 28:

[1] There are type systems for XML, such as XML Schemas, but they are beyond the scope of this book.

[2] The full story is more complicated, but this is enough to be effective with XML.