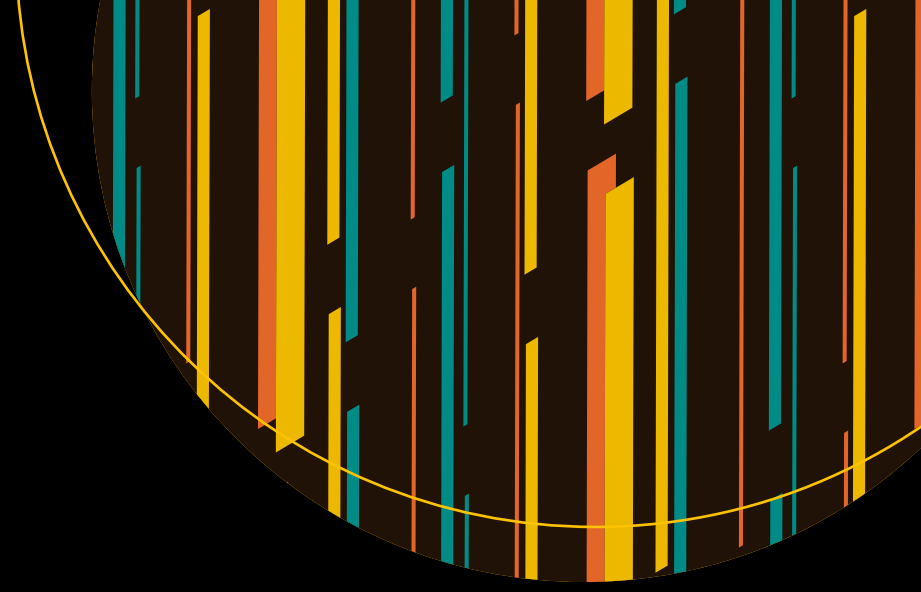

Java XML and JSON

Document Processing for Java SE

Second Edition

Jeff Friesen

Apress®

Java XML and JSON

Document Processing for Java SE

Second Edition

Jeff Friesen

Apress®

Java XML and JSON: Document Processing for Java SE

Jeff Friesen
Dauphin, MB, Canada

ISBN-13 (pbk): 978-1-4842-4329-9 ISBN-13 (electronic): 978-1-4842-4330-5
https://doi.org/10.1007/978-1-4842-4330-5

Library of Congress Control Number: 2018968598

Copyright © 2019 by Jeff Friesen

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

Trademarked names, logos, and images may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, logo, or image we use the names, logos, and images only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Jonathan Gennick
Development Editor: Laura Berendson
Coordinating Editor: Jill Balzano

Cover image designed by Freepik (www.freepik.com)

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a Delaware corporation.

For information on translations, please e-mail rights@apress.com, or visit http://www.apress.com/rights-permissions.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at http://www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/978-1-4842-4329-9. For more detailed information, please visit http://www.apress.com/source-code.

Printed on acid-free paper

To my parents.

Table of Contents

About the Author

Jeff Friesen is a freelance teacher and software developer with an emphasis on Java. In addition to authoring *Java I/O, NIO and NIO.2* (Apress), *Java Threads and the Concurrency Utilities* (Apress), and the first edition of this book, Jeff has written numerous articles on Java and other technologies (such as Android) for JavaWorld (`JavaWorld.com`), informIT (`InformIT.com`), `Java.net`, SitePoint (`SitePoint.com`), and other web sites. Jeff can be contacted via his web site at `JavaJeff.ca` or via his LinkedIn (`LinkedIn.com`) profile (`www.linkedin.com/in/javajeff`).

About the Technical Reviewer

Massimo Nardone has more than 24 years of experiences in Security, web/mobile development, Cloud, and IT architecture. His true IT passions are Security and Android.

He has been programming and teaching how to program with Android, Perl, PHP, Java, VB, Python, C/C++, and MySQL for more than 20 years.

He holds a Master of Science degree in Computing Science from the University of Salerno, Italy.

He has worked as a Project Manager, Software Engineer, Research Engineer, Chief Security Architect, Information Security Manager, PCI/SCADA Auditor, and Senior Lead IT Security/Cloud/SCADA Architect for many years.

His technical skills include Security, Android, Cloud, Java, MySQL, Drupal, Cobol, Perl, web and mobile development, MongoDB, D3, Joomla, Couchbase, C/C++, WebGL, Python, Pro Rails, Django CMS, Jekyll, Scratch, etc.

He worked as visiting lecturer and supervisor for exercises at the Networking Laboratory of the Helsinki University of Technology (Aalto University). He holds four international patents (PKI, SIP, SAML, and Proxy areas).

He currently works as Chief Information Security Officer (CISO) for Cargotec Oyj, and he is member of ISACA Finland Chapter Board.

Massimo has been reviewing more than 45 IT books for different publishing companies, and he is the coauthor of *Pro Android Games* (Apress, 2015), *Pro JPA 2 in Java EE 8* (APress 2018), and *Beginning EJB in Java EE 8* (Apress, 2018).

Acknowledgments

I thank Apress Acquisition Editor Jonathan Gennick and the Apress Editorial Board for giving me the opportunity to create this second edition. I also thank Editor Jill Balzano for guiding me through the book development process. Finally, I thank my technical reviewer and copy editor for catching mistakes and making the book look great.

Introduction

XML and (the more popular) JSON let you organize data in textual formats. This book introduces you to these technologies along with Java APIs for integrating them into your Java code. This book introduces you to XML and JSON as of Java 11.

Chapter 1 introduces XML, where you learn about basic language features (such as the XML declaration, elements and attributes, and namespaces). You also learn about well-formed XML documents and how to validate them via the Document Type Definition and XML Schema grammar languages.

Chapter 2 focuses on Java's SAX API for parsing XML documents. You learn how to obtain a SAX 2 parser; you then tour `XMLReader` methods along with handler and entity resolver interfaces. Finally, you explore a demonstration of this API and learn how to create a custom entity resolver.

Chapter 3 addresses Java's DOM API for parsing and creating XML documents. After discovering the various nodes that form a DOM document tree, you explore the DOM API, where you learn how to obtain a DOM parser/document builder and how to parse and create XML documents. You then explore the Java DOM APIs related to the Load and Save, and Traversal and Range specifications.

Chapter 4 places the spotlight on Java's StAX API for parsing and creating XML documents. You learn how to use StAX to parse XML documents with stream-based and event-based readers and to create XML documents with stream-based and event-based writers.

Moving on, Chapter 5 presents Java's XPath API for simplifying access to a DOM tree's nodes. You receive a primer on the XPath language, learning about location path expressions and general expressions. You also explore advanced features starting with namespace contexts.

Chapter 6 completes my coverage of XML by targetting Java's XSLT API. You learn about transformer factories and transformers, and much more. You also go beyond the XSLT 1.0 and XPath 1.0 APIs supported by Java.

Chapter 7 switches gears to JSON. You receive an introduction to JSON, take a tour of its syntax, explore a demonstration of JSON in a JavaScript context (because Java doesn't yet officially support JSON), and learn how to validate JSON objects in the context of JSON Schema.

You'll need to work with third-party libraries to parse and create JSON documents. Chapter 8 introduces you to the mJson library. After learning how to obtain and use mJson, you explore the `Json` class, which is the entry point for working with mJSon.

Google has released an even more powerful library for parsing and creating JSON documents. The Gson library is the focus of Chapter 9. In this chapter, you learn how to parse JSON objects through deserialization, how to create JSON objects through serialization, and much more.

Chapter 10 focuses on the JsonPath API for performing XPath-like operations on JSON documents.

Chapter 11 introduces you to Jackson, a popular suite of APIs for parsing and creating JSON documents.

Chapter 12 introduces you to JSON-P, an Oracle API that was planned for inclusion in Java SE, but was made available to Java EE instead.

Each chapter ends with assorted exercises that are designed to help you master the content. Along with long answers and true/false questions, you are often confronted with programming exercises. Appendix A provides the answers and solutions.

Thanks for purchasing this book. I hope you find it helpful in understanding XML and JSON in a Java context.

Jeff Friesen (October 2018)

Note You can download this book's source code by pointing your web browser to `www.apress.com/9781484243299` and clicking the Source Code tab followed by the Download Now link.

PART I

Exploring XML

CHAPTER 1

Introducing XML

Applications commonly use XML documents to store and exchange data. XML defines rules for encoding documents in a format that is both human-readable and machine-readable. Chapter 1 introduces XML, tours the XML language features, and discusses well-formed and valid documents.

What Is XML?

XML (eXtensible Markup Language) is a *meta-language* (a language used to describe other languages) for defining *vocabularies* (custom markup languages), which is the key to XML's importance and popularity. XML-based vocabularies (such as XHTML) let you describe documents in a meaningful way.

XML vocabulary documents are like HTML (see http://en.wikipedia.org/wiki/HTML) documents in that they are text-based and consist of *markup* (encoded descriptions of a document's logical structure) and *content* (document text not interpreted as markup). Markup is evidenced via *tags* (angle bracket–delimited syntactic constructs), and each tag has a name. Furthermore, some tags have *attributes* (name/value pairs).

Note XML and HTML are descendants of *Standard Generalized Markup Language* (*SGML*), which is the original meta-language for creating vocabularies—XML is essentially a restricted form of SGML, while HTML is an *application* of SGML. The key difference between XML and HTML is that XML invites you to create your own vocabularies with their own tags and rules, whereas HTML gives you a single pre-created vocabulary with its own fixed set of tags and rules. XHTML and other XML-based vocabularies are *XML applications*. XHTML was created to be a cleaner implementation of HTML.

© Jeff Friesen 2019
J. Friesen, *Java XML and JSON*, https://doi.org/10.1007/978-1-4842-4330-5_1

If you haven't previously encountered XML, you might be surprised by its simplicity and how closely its vocabularies resemble HTML. You don't need to be a rocket scientist to learn how to create an XML document. To prove this to yourself, check out Listing 1-1.

Listing 1-1. XML-Based Recipe for a Grilled Cheese Sandwich

```
<recipe>
   <title>
      Grilled Cheese Sandwich
   </title>
   <ingredients>
      <ingredient qty="2">
         bread slice
      </ingredient>
      <ingredient>
         cheese slice
      </ingredient>
      <ingredient qty="2">
         margarine pat
      </ingredient>
   </ingredients>
   <instructions>
      Place frying pan on element and select medium heat.
      For each bread slice, smear one pat of margarine on
      one side of bread slice. Place cheese slice between
      bread slices with margarine-smeared sides away from
      the cheese. Place sandwich in frying pan with one
      margarine-smeared side in contact with pan. Fry for
      a couple of minutes and flip. Fry other side for a
      minute and serve.
   </instructions>
</recipe>
```

Listing 1-1 presents an XML document that describes a recipe for making a grilled cheese sandwich. This document is reminiscent of an HTML document in that it consists of tags, attributes, and content. However, that's where the similarity ends. Instead of presenting HTML tags such as `<html>`, `<head>`, `<img>`, and `<p>`, this informal recipe language presents its own `<recipe>`, `<ingredients>`, and other tags.

Note Although Listing 1-1's `<title>` and `</title>` tags are also found in HTML, they differ from their HTML counterparts. Web browsers typically display the content between these tags in their title bars or tab headers. In contrast, the content between Listing 1-1's `<title>` and `</title>` tags might be displayed as a recipe header, spoken aloud, or presented in some other way, depending on the application that parses this document.

Language Features Tour

XML provides several language features for use in defining custom markup languages: XML declaration, elements and attributes, character references and CDATA sections, namespaces, and comments and processing instructions. You will learn about these language features in this section.

XML Declaration

An XML document usually begins with the *XML declaration*, special markup telling an XML parser that the document is XML. The absence of the XML declaration in Listing 1-1 reveals that this special markup isn't mandatory. When the XML declaration is present, nothing can appear before it.

The XML declaration minimally looks like `<?xml version="1.0"?>` in which the nonoptional `version` attribute identifies the version of the XML specification to which the document conforms. The initial version of this specification (1.0) was introduced in 1998 and is widely implemented.

Note The World Wide Web Consortium (W3C), which maintains XML, released version 1.1 in 2004. This version mainly supports the use of line-ending characters used on EBCDIC platforms (see http://en.wikipedia.org/wiki/EBCDIC) and the use of scripts and characters that are absent from Unicode (see http://en.wikipedia.org/wiki/Unicode) 3.2. Unlike XML 1.0, XML 1.1 isn't widely implemented and should be used only when its unique features are needed.

XML supports Unicode, which means that XML documents consist entirely of characters taken from the Unicode character set. The document's characters are encoded into bytes for storage or transmission, and the encoding is specified via the XML declaration's optional `encoding` attribute. One common encoding is *UTF-8* (see http://en.wikipedia.org/wiki/UTF-8), which is a variable-length encoding of the Unicode character set. UTF-8 is a strict superset of ASCII (see http://en.wikipedia.org/wiki/ASCII), which means that pure ASCII text files are also UTF-8 documents.

Note In the absence of the XML declaration or when the XML declaration's `encoding` attribute isn't present, an XML parser typically looks for a special character sequence at the start of a document to determine the document's encoding. This character sequence is known as the *byte-order-mark* (*BOM*) and is created by an editor program (such as Microsoft Windows Notepad) when it saves the document according to UTF-8 or some other encoding. For example, the hexadecimal sequence `EF BB BF` signifies UTF-8 as the encoding. Similarly, `FE FF` signifies UTF-16 (see http://en.wikipedia.org/wiki/UTF-16) big endian, `FF FE` signifies UTF-16 little endian, `00 00 FE FF` signifies UTF-32 (see http://en.wikipedia.org/wiki/UTF-32) big endian, and `FF FE 00 00` signifies UTF-32 little endian. UTF-8 is assumed when no BOM is present.

If you'll never use characters apart from the ASCII character set, you can probably forget about the `encoding` attribute. However, when your native language isn't English or when you're called to create XML documents that include non-ASCII characters, you need to properly specify `encoding`. For example, when your document contains ASCII plus characters from a non-English Western European language (such as ç, the cedilla

used in French, Portuguese, and other languages), you might want to choose `ISO-8859-1` as the `encoding` attribute's value—the document will probably have a smaller size when encoded in this manner than when encoded with UTF-8. Listing 1-2 shows you the resulting XML declaration.

Listing 1-2. An Encoded Document Containing Non-ASCII Characters

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<movie>
   <name>Le Fabuleux Destin d'Amélie Poulain</name>
   <language>français</language>
</movie>
```

The final attribute that can appear in the XML declaration is `standalone`. This optional attribute, which is only relevant with DTDs (discussed later), determines whether or not there are external markup declarations that affect the information passed from an *XML processor* (a parser) to the application. Its value defaults to `no`, implying that there are or may be such declarations. A `yes` value indicates that there are no such declarations. For more information, check out "The standalone pseudo-attribute is only relevant if a DTD is used" (`www.xmlplease.com/xml/standalone/`).

Elements and Attributes

Following the XML declaration is a *hierarchical* (tree) structure of elements, where an *element* is a portion of the document delimited by a *start tag* (such as `<name>`) and an *end tag* (such as `</name>`), or is an *empty-element tag* (a standalone tag whose name ends with a forward slash [/], such as `<break/>`). Start tags and end tags surround content and possibly other markup, whereas empty-element tags don't surround anything. Figure 1-1 reveals Listing 1-1's XML document tree structure.

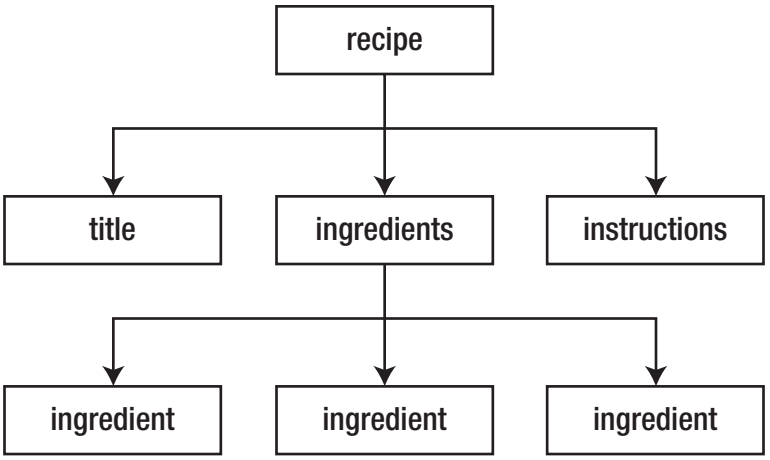

Figure 1-1. *Listing 1-1's tree structure is rooted in the* `recipe` *element*

As with HTML document structure, the structure of an XML document is anchored in a *root element* (the topmost element). In HTML, the root element is `html` (the `<html>` and `</html>` tag pair). Unlike in HTML, you can choose the root element for your XML documents. Figure 1-1 shows the root element to be `recipe`.

Unlike the other elements, which have parent elements, `recipe` has no parent. Also, `recipe` and `ingredients` have child elements: `recipe`'s children are `title`, `ingredients`, and `instructions`; and `ingredients`' children are three instances of `ingredient`. The `title`, `instructions`, and `ingredient` elements don't have child elements.

Elements can contain child elements, content, or *mixed content* (a combination of child elements and content). Listing 1-2 reveals that the `movie` element contains `name` and `language` child elements and also reveals that each of these child elements contains content (e.g., `language` contains `français`). Listing 1-3 presents another example that demonstrates mixed content along with child elements and content.

Listing 1-3. An `Abstract` Element Containing Mixed Content

```
<?xml version="1.0"?>
<article title="The Rebirth of JavaFX" lang="en">
   <abstract>
      JavaFX 2 marks a significant milestone in the history
      of JavaFX. Now that Sun Microsystems has passed the
      torch to Oracle, JavaFX Script is gone and
      JavaFX-oriented Java APIS (such as
```

```
      <code>javafx.application.Application</code>) have
      emerged for interacting with this technology. This
      article introduces you to this refactored JavaFX,
      where you learn about JavaFX 2 architecture and key
      APIs.
   </abstract>
   <body>
   </body>
</article>
```

This document's root element is `article`, which contains `abstract` and `body` child elements. The `abstract` element mixes content with a `code` element, which contains content. In contrast, the `body` element is empty.

Note As with Listings 1-1 and 1-2, Listing 1-3 also contains *whitespace* (invisible characters such as spaces, tabs, carriage returns, and line feeds). The XML specification permits whitespace to be added to a document. Whitespace appearing within content (such as spaces between words) is considered part of the content. In contrast, the parser typically ignores whitespace appearing between an end tag and the next start tag. Such whitespace isn't considered part of the content.

An XML element's start tag can contain one or more attributes. For example, Listing 1-1's `<ingredient>` tag has a `qty` (quantity) attribute, and Listing 1-3's `<article>` tag has `title` and `lang` attributes. Attributes provide additional details about elements. For example, `qty` identifies the amount of an ingredient that can be added, `title` identifies an article's title, and `lang` identifies the language in which the article is written (`en` for English). Attributes can be optional. For example, when `qty` isn't specified, a default value of `1` is assumed.

Note Element and attribute names may contain any alphanumeric character from English or another language and may also include the underscore (`_`), hyphen (`-`), period (`.`), and colon (`:`) punctuation characters. The colon should only be used with namespaces (discussed later in this chapter), and **names cannot contain whitespace**.

Character References and CDATA Sections

Certain characters cannot appear literally in the content that appears between a start tag and an end tag or within an attribute value. For example, you cannot place a literal < character between a start tag and an end tag because doing so would confuse an XML parser into thinking that it had encountered another tag.

One solution to this problem is to replace the literal character with a *character reference*, which is a code that represents the character. Character references are classified as numeric character references or character entity references:

- A *numeric character reference* refers to a character via its Unicode code point and adheres to the format `&#`*nnnn*`;` (not restricted to four positions) or `&#x`*hhhh*`;` (not restricted to four positions), where *nnnn* provides a decimal representation of the code point and *hhhh* provides a hexadecimal representation. For example, `&#0931;` and `&#x03A3;` represent the Greek capital letter sigma. Although XML mandates that the `x` in `&#x`*hhhh*`;` be lowercase, it's flexible in that the leading zero is optional in either format and in allowing you to specify an uppercase or lowercase letter for each *h*. As a result, `&#931;`, `&#x3A3;`, and `&#x03a3;` are also valid representations of the Greek capital letter sigma.
- A *character entity reference* refers to a character via the name of an *entity* (aliased data) that specifies the desired character as its replacement text. Character entity references are predefined by XML and have the format `&`*name*`;`, in which *name* is the entity's name. XML predefines five character entity references: `<` (<), `>` (>), `&` (&), `&apos;` ('), and `&quot;` (").

Consider `<expression>6 < 4</expression>`. You could replace the < with numeric reference `&#60;`, yielding `<expression>6 &#60; 4</expression>`, or better yet with `<`, yielding `<expression>6 < 4</expression>`. The second choice is clearer and easier to remember.

Suppose you want to embed an HTML or XML document within an element. To make the embedded document acceptable to an XML parser, you would need to replace each literal < (start of tag) and & (start of entity) character with its < and & predefined character entity reference, a tedious and possibly error-prone undertaking—you might forget to replace one of these characters. To save you from tedium and potential errors, XML provides an alternative in the form of a CDATA (character data) section.

A *CDATA section* is a section of literal HTML or XML markup and content surrounded by the `<![CDATA[` prefix and the `]]>` suffix. You don't need to specify predefined character entity references within a CDATA section, as demonstrated in Listing 1-4.

Listing 1-4. Embedding an XML Document in Another Document's CDATA Section

```
<?xml version="1.0"?>
<svg-examples>
   <example>
      The following Scalable Vector Graphics document
      describes a blue-filled and black-stroked
      rectangle.
      <![CDATA[<svg width="100%" height="100%"
           version="1.1"
           xmlns="http://www.w3.org/2000/svg">
         <rect width="300" height="100"
              style="fill:rgb(0,0,255);stroke-width:1;
                     stroke:rgb(0,0,0)"/>
      </svg>]]>
   </example>
</svg-examples>
```

Listing 1-4 embeds a Scalable Vector Graphics (SVG) [see http://en.wikipedia.org/wiki/Scalable_Vector_Graphics] XML document within the `example` element of an SVG examples document. The SVG document is placed in a `CDATA` section, obviating the need to replace all < characters with < predefined character entity references.

Namespaces

It's common to create XML documents that combine features from different XML languages. Namespaces are used to prevent name conflicts when elements and other XML language features appear. Without namespaces, an XML parser couldn't distinguish between same-named elements or other language features that mean different things, for example, two same-named `title` elements from two different languages.

Note Namespaces aren't part of XML 1.0. They arrived about a year after this specification was released. To ensure backward compatibility with XML 1.0, namespaces take advantage of colon characters, which are legal characters in XML names. Parsers that don't recognize namespaces return names that include colons.

A *namespace* is a Uniform Resource Identifier (URI)-based container that helps differentiate XML vocabularies by providing a unique context for its contained identifiers. The namespace URI is associated with a *namespace prefix* (an alias for the URI) by specifying, typically on an XML document's root element, either the `xmlns` attribute by itself (which signifies the default namespace) or the `xmlns:`*prefix* attribute (which signifies the namespace identified as *prefix*), and assigning the URI to this attribute.

Note A namespace's scope starts at the element where it's declared and applies to all of the element's content unless overridden by another namespace declaration with the same prefix name.

When *prefix* is specified, the prefix and a colon character are prepended to the name of each element tag that belongs to that namespace—see Listing 1-5.

Listing 1-5. Introducing a Pair of Namespaces

```
<?xml version="1.0"?>
<h:html xmlns:h="http://www.w3.org/1999/xhtml"
        xmlns:r="http://www.javajeff.ca/">
   <h:head>
      <h:title>
```

```
      Recipe
   </h:title>
</h:head>
<h:body>
<r:recipe>
   <r:title>
      Grilled Cheese Sandwich
   </r:title>
   <r:ingredients>
      <h:ul>
      <h:li>
      <r:ingredient qty="2">
         bread slice
      </r:ingredient>
      </h:li>
      <h:li>
      <r:ingredient>
         cheese slice
      </r:ingredient>
      </h:li>
      <h:li>
      <r:ingredient qty="2">
         margarine pat
      </r:ingredient>
      </h:li>
      </h:ul>
   </r:ingredients>
   <h:p>
   <r:instructions>
      Place frying pan on element and select medium
      heat. For each bread slice, smear one pat of
      margarine on one side of bread slice. Place
      cheese slice between bread slices with
      margarine-smeared sides away from the cheese.
      Place sandwich in frying pan with one
```

```
        margarine-smeared side in contact with pan.
        Fry for a couple of minutes and flip. Fry
        other side for a minute and serve.
      </r:instructions>
      </h:p>
   </r:recipe>
   </h:body>
</h:html>
```

Listing 1-5 describes a document that combines elements from the XHTML (see http://en.wikipedia.org/wiki/XHTML) language with elements from the recipe language. All element tags that associate with XHTML are prefixed with `h:`, and all element tags that associate with the recipe language are prefixed with `r:`.

The `h:` prefix associates with the `www.w3.org/1999/xhtml` URI, and the `r:` prefix associates with the `www.javajeff.ca` URI. XML doesn't mandate that URIs point to document files. It only requires that they be unique to guarantee unique namespaces.

This document's separation of the recipe data from the XHTML elements makes it possible to preserve this data's structure while also allowing an XHTML-compliant web browser (such as Mozilla Firefox) to present the recipe via a web page (see Figure 1-2).

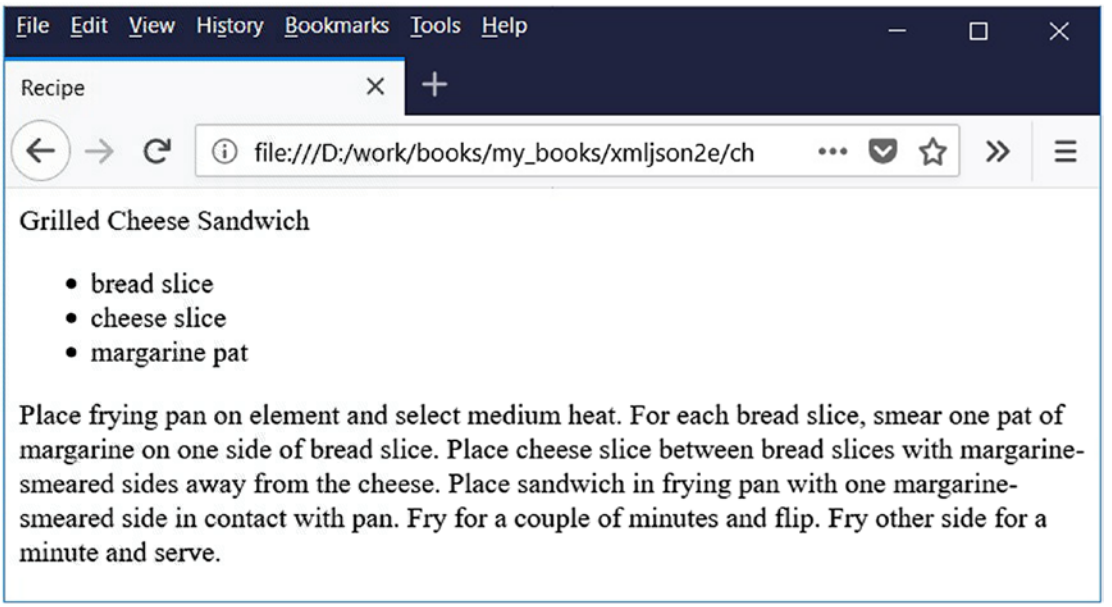

Figure 1-2. *Mozilla Firefox presents the recipe data via XHTML tags*

A tag's attributes don't need to be prefixed when those attributes belong to the element. For example, `qty` isn't prefixed in `<r:ingredient qty="2">`. However, a prefix is required for attributes belonging to other namespaces. For example, suppose you want to add an XHTML `style` attribute to the document's `<r:title>` tag to provide styling for the recipe title when displayed via an application. You can accomplish this task by inserting an XHTML attribute into the `title` tag, as follows:

```
<r:title h:style="font-family: sans-serif;">
```

The XHTML `style` attribute has been prefixed with `h:` because this attribute belongs to the XHTML language namespace and not to the recipe language namespace.

When multiple namespaces are involved, it can be convenient to specify one of these namespaces as the default namespace to reduce the tedium in entering namespace prefixes. Consider Listing 1-6.

Listing 1-6. Specifying a Default Namespace

```
<?xml version="1.0"?>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:r="http://www.javajeff.ca/">
   <head>
      <title>
         Recipe
      </title>
   </head>
   <body>
   <r:recipe>
      <r:title>
         Grilled Cheese Sandwich
      </r:title>
      <r:ingredients>
         <ul>
         <li>
         <r:ingredient qty="2">
            bread slice
         </r:ingredient>
         </li>
```

```
         <li>
         <r:ingredient>
            cheese slice
         </r:ingredient>
         </li>
         <li>
         <r:ingredient qty="2">
            margarine pat
         </r:ingredient>
         </li>
         </ul>
      </r:ingredients>
      <p>
      <r:instructions>
         Place frying pan on element and select medium
         heat. For each bread slice, smear one pat of
         margarine on one side of bread slice. Place
         cheese slice between bread slices with
         margarine-smeared sides away from the cheese.
         Place sandwich in frying pan with one
         margarine-smeared side in contact with pan.
         Fry for a couple of minutes and flip. Fry
         other side for a minute and serve.
      </r:instructions>
      </p>
   </r:recipe>
   </body>
</html>
```

Listing 1-6 specifies a default namespace for the XHTML language. No XHTML element tag needs to be prefixed with `h:`. However, recipe language element tags must still be prefixed with the `r:` prefix.