Perl XML::LibXML by Example Documentation

Release

Grant McLean

April 06, 2019

The XML::LibXML Perl module is a wrapper around the libxml2 parser library which is written in C. This tutorial uses example code to introduce the features of XML::LibXML and the ways in which you can use the module. The example scripts and XML documents are available as a `ZIP file download`.

Get started with a basic example or jump directly to a specific topic using the Table of Contents.

CHAPTER

ONE

A BASIC EXAMPLE

The first thing you'll need is an XML document. The example programs in this section will use the `playlist.xml` file shown below. This file contains details of five different movies:

```
<playlist>
  <movie id="tt0112384">
    <title>Apollo 13</title>
    <director>Ron Howard</director>
    <release-date>1995-06-30</release-date>
    <mpaa-rating>PG</mpaa-rating>
    <running-time>140</running-time>
    <genre>adventure</genre>
    <genre>drama</genre>
    <cast>
      <person name="Tom Hanks" role="Jim Lovell" />
      <person name="Bill Paxton" role="Fred Haise" />
      <person name="Kevin Bacon" role="Jack Swigert" />
      <person name="Gary Sinise" role="Ken Mattingly" />
      <person name="Ed Harris" role="Gene Kranz" />
    </cast>
    <imdb-info url="http://www.imdb.com/title/tt0112384/">
      <synopsis>
        NASA must devise a strategy to return Apollo 13 to Earth safely
        after the spacecraft undergoes massive internal damage putting
        the lives of the three astronauts on board in jeopardy.
      </synopsis>
      <score>7.6</score>
    </imdb-info>
  </movie>
  <movie id="tt0307479">
    <title>Solaris</title>
    <director>Steven Soderbergh</director>
    <release-date>2002-11-27</release-date>
    <mpaa-rating>PG-13</mpaa-rating>
    <running-time>99</running-time>
    <genre>drama</genre>
    <genre>mystery</genre>
    <genre>romance</genre>
    <cast>
      <person name="George Clooney" role="Chris Kelvin" />
      <person name="Natascha McElhone" role="Rheya" />
      <person name="Ulrich Tukur" role="Gibarian" />
    </cast>
    <imdb-info url="http://www.imdb.com/title/tt0307479/">
      <synopsis>
        A troubled psychologist is sent to investigate the crew of an
```

```
        isolated research station orbiting a bizarre planet.
      </synopsis>
      <score>6.2</score>
    </imdb-info>
  </movie>
  <movie id="tt1731141">
    <title>Ender's Game</title>
    <director>Gavin Hood</director>
    <release-date>2013-11-01</release-date>
    <mpaa-rating>PG-13</mpaa-rating>
    <running-time>114</running-time>
    <genre>action</genre>
    <genre>scifi</genre>
    <cast>
      <person name="Asa Butterfield" role="Ender Wiggin" />
      <person name="Harrison Ford" role="Colonel Graff" />
      <person name="Hailee Steinfeld" role="Petra Arkanian" />
    </cast>
    <imdb-info url="http://www.imdb.com/title/tt1731141/">
      <synopsis>
        Young Ender Wiggin is recruited by the International Military
        to lead the fight against the Formics, a genocidal alien race
        which nearly annihilated the human race in a previous invasion.
      </synopsis>
      <score>6.7</score>
    </imdb-info>
  </movie>
  <movie id="tt0816692">
    <title>Interstellar</title>
    <director>Christopher Nolan</director>
    <release-date>2014-11-07</release-date>
    <mpaa-rating>PG-13</mpaa-rating>
    <running-time>169</running-time>
    <genre>adventure</genre>
    <genre>drama</genre>
    <genre>scifi</genre>
    <cast>
      <person name="Matthew McConaughey" role="Cooper" />
      <person name="Anne Hathaway" role="Brand" />
      <person name="Jessica Chastain" role="Murph" />
      <person name="Michael Caine" role="Professor Brand" />
    </cast>
    <imdb-info url="http://www.imdb.com/title/tt0816692/">
      <synopsis>
        A team of explorers travel through a wormhole in space in an
        attempt to ensure humanity's survival.
      </synopsis>
      <score>8.6</score>
    </imdb-info>
  </movie>
  <movie id="tt3659388">
    <title>The Martian</title>
    <director>Ridley Scott</director>
    <release-date>2015-10-02</release-date>
    <mpaa-rating>PG-13</mpaa-rating>
    <running-time>144</running-time>
    <genre>adventure</genre>
    <genre>drama</genre>
```

```
    <genre>scifi</genre>
    <cast>
      <person name="Matt Damon" role="Mark Watney" />
      <person name="Jessica Chastain" role="Melissa Lewis" />
      <person name="Kristen Wiig" role="Annie Montrose" />
    </cast>
    <imdb-info url="http://www.imdb.com/title/tt3659388/">
      <synopsis>
        During a manned mission to Mars, Astronaut Mark Watney is
        presumed dead after a fierce storm and left behind by his crew.
        But Watney has survived and finds himself stranded and alone on
        the hostile planet. With only meager supplies, he must draw upon
        his ingenuity, wit and spirit to subsist and find a way to
        signal to Earth that he is alive.
      </synopsis>
      <score>8.1</score>
    </imdb-info>
  </movie>
</playlist>
```

Note: Although this XML document contains details which came from the fabulous IMDb.com web site, the file structure was created specifically for this example and does not represent an actual API for querying movie details.

Once you have the sample XML document, you can use `this script` to extract and print the title of each movie, in the order they appear in the XML:

```
#!/usr/bin/perl

use 5.010;
use strict;
use warnings;

use XML::LibXML;

my $filename = 'playlist.xml';

my $dom = XML::LibXML->load_xml(location => $filename);

foreach my $title ($dom->findnodes('/playlist/movie/title')) {
    say $title->to_literal();
}
```

and will produce the following output:

```
Apollo 13
Solaris
Ender's Game
Interstellar
The Martian
```

Is XML::LibXML installed?

If you try running this example script but you don't have the `XML::LibXML` module installed on your system, then you'll get an error like this:

```
Can't locate XML/LibXML.pm in @INC ... at ./010-list-titles.pl line 7.
```

If you do get this error, then refer to Installing XML::LibXML for help on installing `XML::LibXML`.

If we break the example down line-by-line we see that after a standard boilerplate section, the script loads the `XML::LibXML` module:

```
use XML::LibXML;
```

Next, the `load_xml()` class method is called to parse the XML file and return a document object:

```
my $dom = XML::LibXML->load_xml(location => $filename);
```

The `$dom` variable now contains an object representing all the elements of the XML document arranged in a tree structure known as a Document Object Model or 'DOM'.

Finally we get to the guts of the script where the `findnodes()` method is called to search the DOM for the elements we're interested in and a `foreach` loop is used to iterate through the matching elements:

```
foreach my $title ($dom->findnodes('/playlist/movie/title')) {
    say $title->to_literal();
}
```

The `findnodes()` method takes one argument - an **XPath expression**. This is a string describing the location and characteristics of the elements we want to find. XPath is a query language and the way we use it to select elements from the DOM is similar to the way we use SQL to select records from a relational database. The next section (XPath Expressions) will include examples of more complex queries.

The `findnodes()` method returns a list of objects from the DOM that match the XPath expression. Each time through the loop, `$title` will contain an object representing the next matching element. This object provides a number of properties and methods that you can use to access the element and its attributes, as well as any text content and 'child' elements.

Inside the loop, this example simply calls the `to_literal()` method to get the text content of the element. The string returned by `to_literal()` will not include any of the attributes but will include the text content of any child elements.

Other XML sources

The first example script called `XML::LibXML->load_xml()` with the `location` argument set to the name of a file. The `location` argument also accepts a URL:

```
$dom = XML::LibXML->load_xml(location => 'http://techcrunch.com/feed/');
```

Note: Not all versions of `libxml2` can retrieve documents over SSL/TLS. So if the URL is an 'https' URL (or if it redirects to one), you may need to use a module like LWP to retrieve the document and pass the response body to the XML parser as a string as shown below.

If you have the XML in a string, instead of `location`, use `string`:

```
$dom = XML::LibXML->load_xml(string => $xml_string);
```

Or, you can provide a Perl file handle to parse from an open file or socket, using `IO`:

```
$dom = XML::LibXML->load_xml(IO => $fh);
```

When providing a string or a file handle, it's crucial that you **do not** decode the bytes of the source data (for example by using `':utf8'` when opening a file). The underlying `libxml2` library is written in C to decode bytes and does not understand Perl's character strings. If you have assembled your XML document by concatenating Perl character strings, you will need to encode it to a byte string (for example using `Encode::encode_utf8()`) and then pass the byte string to the parser.

If you have enabled UTF-8 globally with something like this in your script:

```
use open ':encoding(utf8)';
```

Then you'll need to turn **off** the encoding IO layers for any file handle that you pass to XML::LibXML:

```
open my $fh, '<', $filename;
binmode $fh, ':raw';
$dom = XML::LibXML->load_xml(IO => $fh);
```

A more complex example

Now let's look at a slightly more complex example. `This script` takes the same XML input and extracts more details from each `<movie>` element:

```
#!/usr/bin/perl

use 5.010;
use strict;
use warnings;

use XML::LibXML;

my $filename = 'playlist.xml';

my $dom = XML::LibXML->load_xml(location => $filename);

foreach my $movie ($dom->findnodes('//movie')) {
    say 'Title:    ', $movie->findvalue('./title');
    say 'Director: ', $movie->findvalue('./director');
    say 'Rating:   ', $movie->findvalue('./mpaa-rating');
    say 'Duration: ', $movie->findvalue('./running-time'), " minutes";
    my $cast = join ', ', map {
        $_->to_literal();
    } $movie->findnodes('./cast/person/@name');
    say 'Starring: ', $cast;
    say "";
}
```

and will produce the following output:

```
Title:    Apollo 13
Director: Ron Howard
Rating:   PG
Duration: 140 minutes
Starring: Tom Hanks, Bill Paxton, Kevin Bacon, Gary Sinise, Ed Harris
```

```
Title:    Solaris
Director: Steven Soderbergh
Rating:   PG-13
Duration: 99 minutes
Starring: George Clooney, Natascha McElhone, Ulrich Tukur

Title:    Ender's Game
Director: Gavin Hood
Rating:   PG-13
Duration: 114 minutes
Starring: Asa Butterfield, Harrison Ford, Hailee Steinfeld

Title:    Interstellar
Director: Christopher Nolan
Rating:   PG-13
Duration: 169 minutes
Starring: Matthew McConaughey, Anne Hathaway, Jessica Chastain, Michael Caine

Title:    The Martian
Director: Ridley Scott
Rating:   PG-13
Duration: 144 minutes
Starring: Matt Damon, Jessica Chastain, Kristen Wiig
```

Let's compare the main loop of the first script:

```
foreach my $title ($dom->findnodes('/playlist/movie/title')) {
    say $title->to_literal();
}
```

with the main loop of the second script:

```
foreach my $movie ($dom->findnodes('//movie')) {
    say 'Title:    ', $movie->findvalue('./title');
    say 'Director: ', $movie->findvalue('./director');
    say 'Rating:   ', $movie->findvalue('./mpaa-rating');
    say 'Duration: ', $movie->findvalue('./running-time'), " minutes";
    my $cast = join ', ', map {
        $_->to_literal();
    } $movie->findnodes('./cast/person/@name');
    say 'Starring: ', $cast;
    say "";
}
```

The structure of the main loop is very similar but the XPath expression passed to `findnodes()` is different in each case:

'/playlist/movie/title'

> Will match every <title> element which is the child of ...
> a <movie> element which is the child of ...
> a <playlist> element which is ...
> the top-level element in the document.
>
> Or, to phrase it a different way, the search will start at the top of the document and look for a <playlist> element; if one is found, the search will continue for child <movie> elements; and for each one that is found the search will continue for child <title> elements.

'//movie' Will match every <movie> element at any level of nesting.

In both cases, the XPath expression starts with a '/' which means the search will start at the the top of the document.

Inside the second script's loop are a number of calls to `findvalue()`. This is a handy shortcut method that is typically used when you expect the XPath expression to match *exactly one node*. It combines the functionality of `findnodes()` and `to_literal()` into a single method. So this code:

```
$movie->findvalue('./title');
```

is equivalent to:

```
$movie->findnodes('./title')->to_literal();
```

There are a couple of other interesting differences with the XPath searches in the loop compared to previous examples. Firstly, the `findvalue()` method is being called on `$movie` (which represents one `<movie>` element) rather than on `$dom` (which represents the whole document). This means that the `$movie` element is the **context element**. Secondly, the XPath expression starts with a '.' which means: start the search at the context element rather than at the top of the document.

This second script illustrates a common pattern when working with `XML::LibXML`:

1. find 'interesting' elements using an XPath query starting with '/' or '//'
2. iterate through those elements in a `foreach` loop
3. get additional data from child elements using XPath queries starting with '.'

Accessing attributes

When listing cast members in the main loop of the script above, this code ...

```
    my $cast = join ', ', map {
        $_->to_literal();
    } $movie->findnodes('./cast/person/@name');
    say 'Starring: ', $cast;
```

is used to transform this XML ...

```
<cast>
  <person name="Matt Damon" role="Mark Watney" />
  <person name="Jessica Chastain" role="Melissa Lewis" />
  <person name="Kristen Wiig" role="Annie Montrose" />
</cast>
```

into this output:

```
Starring: Matt Damon, Jessica Chastain, Kristen Wiig
```

In an XPath expression, a name that starts with `@` will match an attribute rather than an element, so `'person/@name'` refers to an attribute called `name` on a `<person>` element. In this case, the call to `findnodes('./cast/person/@name')` will return three DOM nodes representing attribute values which are then transformed into plain strings using `to_literal()`, as we've seen for element nodes, inside a map block.

Another approach is to select the *element* with XPath and then call a DOM method on the element node to get the attribute value:

```
    my $cast = join ', ', map {
        $_->getAttribute('name');
    } $movie->findnodes('./cast/person');
    say 'Starring: ', $cast;
```

Attributes via tied hash

There's a shortcut syntax you can use to make this even easier, simply treat the element node as a hashref:

```
    my $cast = join ', ', map {
        $_->{name};
    } $movie->findnodes('./cast/person');
    say 'Starring: ', $cast;
```

You might be a bit wary of poking around directly inside the element object, rather than using accessor methods. But don't worry, that's **not** what this shortcut syntax is doing. Instead, every XML::LibXML::Element object returned from the XPath query has been 'tied' using XML::LibXML::AttributeHash so that hash lookups 'inside' the object actually get proxied to `getAttribute()` method calls.

This technique is less efficient than calling `getAttribute()` directly but it is very convenient when you want to access more than one attribute of an element or when you want to interpolate an attribute value into a string:

```
    my $cast = join "\n", map {
        " * $_->{name} (as $_->{role})";
    } $movie->findnodes('./cast/person');
    say "\nStarring:\n", $cast;
```

Which will produce this output:

```
Starring:
 * Matt Damon (as Mark Watney)
 * Jessica Chastain (as Melissa Lewis)
 * Kristen Wiig (as Annie Montrose)
```

Note: Overloading 'Element' nodes to support tied hash access to attribute values was added in version 1.91 of XML::LibXML. If the examples above don't work for you then it may be because you have a very old version installed.

Parsing Errors

One of the advantages of XML is that it has a few strict rules that every document must comply with to be considered "well-formed". If a document is not well-formed, it should be rejected in its entirety and no part of the XML document content should be used. Examples of things that would cause a document to be not well-formed include:

- missing or mismatched closing tag
- missing or mismatched quotes around attribute values
- whitespace before the initial XML declaration section
- byte sequences that do not match the document's declared character encoding
- any non-whitespace characters after the closing tag for the first top-level element

Like pretty much all XML parser modules, `libxml` will throw an exception if it encounters any violations of these rules. Since the whole of the XML document is processed when `load_xml` is called, an error at any point in the document will cause an exception to be raised.

If you wish to handle exceptions gracefully use must use an `eval` block or one of the "try/catch" syntax extension modules to catch the error. For example, this document contains an error:

```
<?xml version='1.0' encoding='UTF-8' standalone="yes" ?>
<book edition="2">
  <title>Training Your Pet Ferret</title>
  <authors>
    <author>Gerry Bucsis</author>
    <author>Barbara Somerville</author>
  </authors>
  <isbn>9780764142239</isbn>
  <dimensions width="162.56mm" height="195.58mm" depth="10.16mm" pages="96" />
</book>
```

This script will attempt to parse the bad input:

```
#my $filename = 'book.xml';
my $filename = 'book-borkened.xml';

my $dom = eval {
    XML::LibXML->load_xml(location => $filename);
};
if($@) {
    # Log failure and exit
    print "Error parsing '$filename':\n$@";
    exit 0;
}

foreach my $author ($dom->findnodes('//author')) {
    say $author->to_literal();
```

and will instead produce this output:

```
Error parsing 'book-borkened.xml':
book-borkened.xml:8: parser error : Opening and ending tag mismatch: isbn line 8 and isnb
  <isbn>9780764142239</isnb>
                            ^
```

Note that although the script is only looking for `<author>` elements and the error in the `<isbn>` element comes *after* all the `<author>` elements, an exception is still raised by the `load_xml` call inside the eval block, before the DOM has been fully constructed.

That's it for the basic examples. The next topic will look more closely at XPath expressions.

CHAPTER

TWO

XPATH EXPRESSIONS

As you saw in the basic examples section, the `findnodes()` method takes an XPath expression and finds nodes in the DOM that match the expression. There are two ways to call calling the `findnodes()` method:

- on the object representing the whole document, or
- on an element from the DOM - the element on which you call the method is called the context element

If your XPath expression starts with a '/' then the search will start at top-most element in the document - even if you call `findnodes()` on a different context element.

Start your XPath expression with '.' to search down through the children of the context element.

The remainder of this section simply includes examples of XPath expressions and descriptions of what they match.

Note: You can try out different XPath expressions in the XPath sandbox. The sandbox doesn't actually use Perl or libxml, it simply uses Javascript to access the XPath engine built into your browser. However, the expression matching should work just as it would in your Perl scripts.

```
/playlist
```

Match the top-most element of the document if (and *only if*) it is a `<playlist>` element.

```
//title
```

Match every `<title>` element in the document.

```
//movie/title
```

Match every `<title>` element that is the direct child of a `<movie>` element.

`./title`

Match every `<title>` element that is the direct child of the context element, e.g.:

```
foreach my $movie ($dom->findnodes('//movie')) {
    say 'Title: ', $movie->findvalue('./title');
}
```

```
//title/..
```

Match any element which is the parent of a `<title>` element.

```
/*
```

Match the top-most element of the document regardless of the element name.

```
//person/@role
```

Match the attribute named `role` on every `<person>` element.

```
//person/@*
```

Match every attribute on every `<person>` element.

```
//person[@role]
```

Match every `<person>` element *that has an attribute* named `role`.

```
//*[@url]
```

Match every element that has an attribute named `url`.

```
//*[@*]
```

Match every element that has an attribute of any name.

```
/playlist//*[not(@*)]
```

Match every element that is a descendant of the top-level `<playlist>` element and which does not have any attributes.

```
//movie[@id="tt0307479"]
```

Match every `<movie>` element that has an attribute named `id` with the value `tt0307479`.

```
//movie[not(@id="tt0307479")]
```

Match every `<movie>` element that does not have an attribute named `id` with the value `tt0307479` (including elements that do not have an `id` attribute at all).

```
//*[@id="tt0307479"]
```

Match every element that has an attribute named `id` with the value `tt0307479`.

```
//movie[@id="tt0307479"]//synopsis
```

Match every `synopsis` element within every `<movie>` element that has an attribute named `id` with the value `tt0307479`.

```
//person[position()=2]
```

Match the second `<person>` element in each sequence of adjacent `<person>` elements. Note that the first element in a sequence is at position 1 not 0.

```
//person[2]
```

This is simply a shorthand form of the `position()=2` expression above.

```
//person[position()<3]
```

Match the first two `<person>` elements in each sequence of adjacent `<person>` elements.

```
//person[last()]
```

Match the last `<person>` element in each sequence of adjacent `<person>` elements.

```
//cast[count(person)=3]
```

Match every <cast> element which contains exactly 3 <person> elements.

```
//*[name()='genre']
```

Match every element with the name genre - exactly equivalent to //genre.

```
//*[starts-with(name(), 'running')]
```

Match every element with a name starting with the word running.

```
//person[contains(@name, 'Matt')]
```

Match every <person> element that has an attribute named name which contains the text Matt anywhere in the attribute value.

```
//person[contains(@name, 'matt')]
```

Same as above except for the casing of the text to match. Matching is case-sensitive.

```
//person[not(contains(@name, 'e'))]
```

Match every <person> element that has an attribute named name which does not contain the letter e anywhere in the attribute value.

```
//person[starts-with(@name, 'K')]
```

Match every <person> element that has an attribute named name with a value that starts with the letter K.

```
//director/text()
```

Match every text node which is a direct child of a <director> element.

```
//cast/text()
```

Match every text node which is a direct child of a <cast> element. You might imagine that this would not match anything, since in the sample document the <cast> elements contain only <person> elements. But if you look carefully, you'll see that in between each <person> element there is some whitespace - a newline after the preceding element and then some spaces at the start of the next line. This whitespace is text and is therefore matched.

```
//person[contains(@name,'Matt')]/parent::*
```

Match the parent of every <person> element which contains Matt in the name attribute. (You could also use /.. for the parent). The syntax parent::* means any element on the parent axis.

```
//person[contains(@name,'Matt')]/ancestor::movie
```

Match every <movie> element which is an ancestor of a <person> element which contains Matt in the name attribute. The syntax ancestor::* means any element on the ancestor axis.

```
//genre[text()='drama']/following-sibling::*
```

Match every element of any name, which is a sibling of a <genre> element whose complete text content is drama and which follows that element in document order.

```
//genre[text()='drama']/following-sibling::genre
```

Match every <genre> element, which is a sibling of a <genre> element whose complete text content is drama and which follows that element in document order.

```
//genre[text()='drama']/preceding-sibling::genre
```

Match every <genre> element, which is a sibling of a <genre> element whose complete text content is drama and which comes before that element in document order.

```
//movie[@id="tt0112384"]/following::title
```

Match every <title> element, which comes after a <movie> element with tt0112384 as the value of the id attribute. Note that 'after' means after the closing tag so a <title> element *inside* the matching <movie> would not be included.

```
//movie[.//score/text() < 7.5]
```

Match every <movie> element which contains a <score> element with text content numerically less than 7.5.

```
//movie[.//score/text() > 8.0]//synopsis
```

Match every <synopsis> element in every <movie> element which contains a <score> element with text content numerically greater than 8.0.

```
//director or //genre
```

Match every element which is a <director> or a <genre>.

```
//person[contains(@name, 'Bill') and contains(@role, 'Fred')]
```

Match every <person> element which contains Bill in the name attribute **and** contains Fred in the role attribute.

```
//person[@name='Kevin Bacon']/../person[@name!='Kevin Bacon']
```

Find every person who has played alongside Kevin Bacon. First find every <person> element with a name attribute equal to Kevin Bacon. Then find the parent of each matching element and look for its child <person> elements with a name attribute which is not equal to Kevin Bacon.

XPath Functions

Some of the examples above used XPath functions. It's worth noting that the underlying libxml2 library only supports XPath version 1.0 and there are no plans to support 2.0.

XPath 1.0 does not include the lower-case() or upper-case() functions, so nasty workarounds like this are required if you need case-insensitive matching:

```
my $query = q{
    //person[
        contains(
            translate(
                @name,
                'ABCDEFGHIJKLMNOPQRSTUVWXZY',
                'abcdefghijklmnopqrstuvwxyz'
            ),
            'matt'
        )
    ]
};

foreach my $person ($dom->findnodes($query)) {
    say "Person: $person->{name}";
}
```

Alternatively, you can use the Perl API to register custom XPath functions.

CHAPTER

THREE

THE DOCUMENT OBJECT MODEL

The basic examples section introduced the `findnodes()` method and XPath expressions for extracting parts of an XML document. For most applications, that's pretty much all you need, but sometimes it's necessary to use lower-level methods and to understand the relationships between different parts of the document.

The XML::LibXML module implements Perl bindings for the W3C Document Object Model. The W3C DOM defines object classes, properties and methods for querying and manipulating the different parts of an XML (or HTML) document. In the Perl implementation, object properties are exposed via accessor methods.

Let's start our exploration of the DOM with a simple XML document which describes a book - `book.xml`

```
<?xml version='1.0' encoding='UTF-8' standalone="yes" ?>
<book edition="2">
  <title>Training Your Pet Ferret</title>
  <authors>
    <author>Gerry Bucsis</author>
    <author>Barbara Somerville</author>
  </authors>
  <isbn>9780764142239</isbn>
  <dimensions width="162.56mm" height="195.58mm" depth="10.16mm" pages="96" />
</book>
```

When you ask XML::LibXML to parse the document, it creates an object to represent each part of the document and assembles those objects into a hierarchy as shown here:

The source XML document has a `<book>` element which contains four other elements: `<title>`, `<authors>`, `<isbn>` and `<dimensions>`. The `<authors>` element in turn contains two `<author>` elements.

The hierarchy in the picture shows us that `<book>` has four "child" elements. Similarly, `<authors>` has two child elements and one "parent" element (`<book>`). Five of the elements have no child elements but four of them do contain text content and one has some attributes.

The 'Document' object

When you parse a document with XML::LibXML the parser returns a 'Document' object - represented in yellow in the picture above. The reference documentation for the XML::LibXML::Document class lists methods you can use to interact with the document. The 'Document' class inherits from the 'Node' class so you'll also need to refer to the docs for XML::LibXML::Node as well.

```
my $dom = XML::LibXML->load_xml(location => 'book.xml');

say '$dom is a ', ref($dom);
say '$dom->nodeName is: ', $dom->nodeName;
```

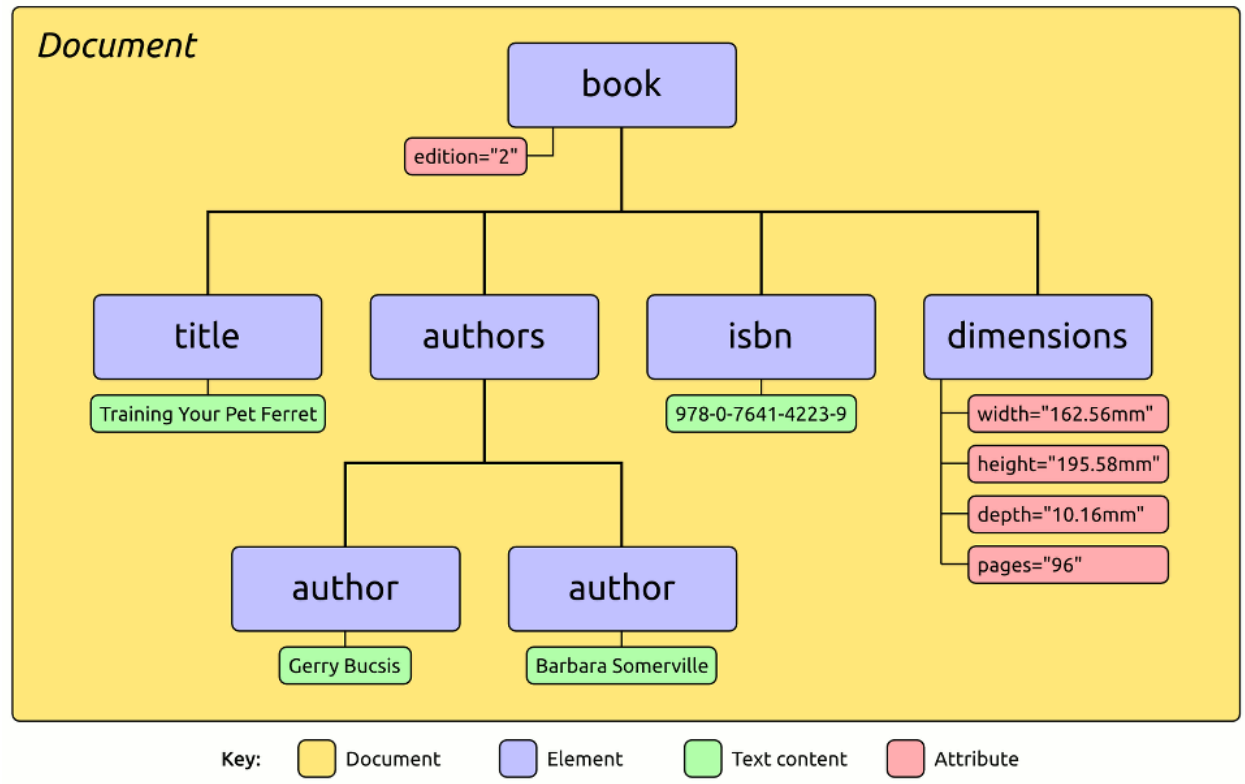

Fig. 3.1: A simplified representation of the Document Object Model.

Output:

```
$dom is a XML::LibXML::Document
$dom->nodeName is: #document
```

The document object also provides methods you can use to extract information from the XML declaration section - the very first line of the source XML, which precedes the `<book>` element:

```
say 'XML Version is: ', $dom->version;
say 'Document encoding is: ', $dom->encoding;
my $is_or_not = $dom->standalone ? 'is' : 'is not';
say "Document $is_or_not standalone";
```

Output:

```
XML Version is: 1.0
Document encoding is: UTF-8
Document is standalone
```

You can serialise a whole DOM back out to XML by calling the `toString()` method on the document object:

```
say "DOM as XML:\n", $dom->toString;
```

The document class also overrides the stringification operator, so if you simply treat the object as a string and print it out you'll also get the serialised XML:

```
say "DOM as a string:\n", $dom;
```

'Element' objects

The blue boxes in the picture represent 'Element' nodes. The reference documentation for the XML::LibXML::Element class lists a number of methods, but like the 'Document' class, many more methods are inherited from XML::LibXML::Node.

Every XML document has one single top-level element known as the "document element" that encloses all the other elements - in our example it's the <book> element. You can retrieve this element by calling the `documentElement()` method on the document object and you can determine what type of element it is by calling `nodeName()`:

```
my $book = $dom->documentElement;
say '$book is a ', ref($book);
say '$book->nodeName is: ', $book->nodeName;
```

Output:

```
$book is a XML::LibXML::Element
$book->nodeName is: book
```

The <book> element has four child elements. You can use `getChildrenByTagName()` to get a list of all the child elements with a specific element name (this is not a recursive search, it only looks through elements which are direct children):

```
my($isbn) = $book->getChildrenByTagName('isbn');
say '$isbn is a ', ref($isbn);
say '$isbn->nodeName is: ', $isbn->nodeName;
say '$isbn->to_literal returns: ', $isbn->to_literal;
say '$isbn stringifies to: ', $isbn;
```

Output:

```
$isbn is a XML::LibXML::Element
$isbn->nodeName is: isbn
$isbn->to_literal returns: 9780764142239
$isbn stringifies to: <isbn>9780764142239</isbn>
```

If you're not looking for one specific type of element, you can get all the children with `childNodes()`:

```
my @children = $book->childNodes;
my $count = @children;
say "\$book has $count child nodes:";
my $i = 0;
foreach my $child (@children) {
    say $i++, ": is a ", ref($child), ', name = ', $child->nodeName;
}
```

We already know that <book> contains four child elements, so you may be a little surprised to see `childNodes()` returns a list of nine nodes:

```
$book has 9 child nodes:
0: is a XML::LibXML::Text, name = #text
1: is a XML::LibXML::Element, name = title
2: is a XML::LibXML::Text, name = #text
3: is a XML::LibXML::Element, name = authors
4: is a XML::LibXML::Text, name = #text
5: is a XML::LibXML::Element, name = isbn
6: is a XML::LibXML::Text, name = #text
```

```
7: is a XML::LibXML::Element, name = dimensions
8: is a XML::LibXML::Text, name = #text
```

If you refer back to the source XML document, you can see that after the <book> tag and before the <title> tag there is some whitespace: a line-feed character followed by two spaces at the start of the next line:

```
<book edition="2">
  <title>Training Your Pet Ferret</title>
```

These strings of whitespace are represented in the DOM by 'Text' nodes, which are children of the parent element. So a more accurate DOM diagram would look like this:

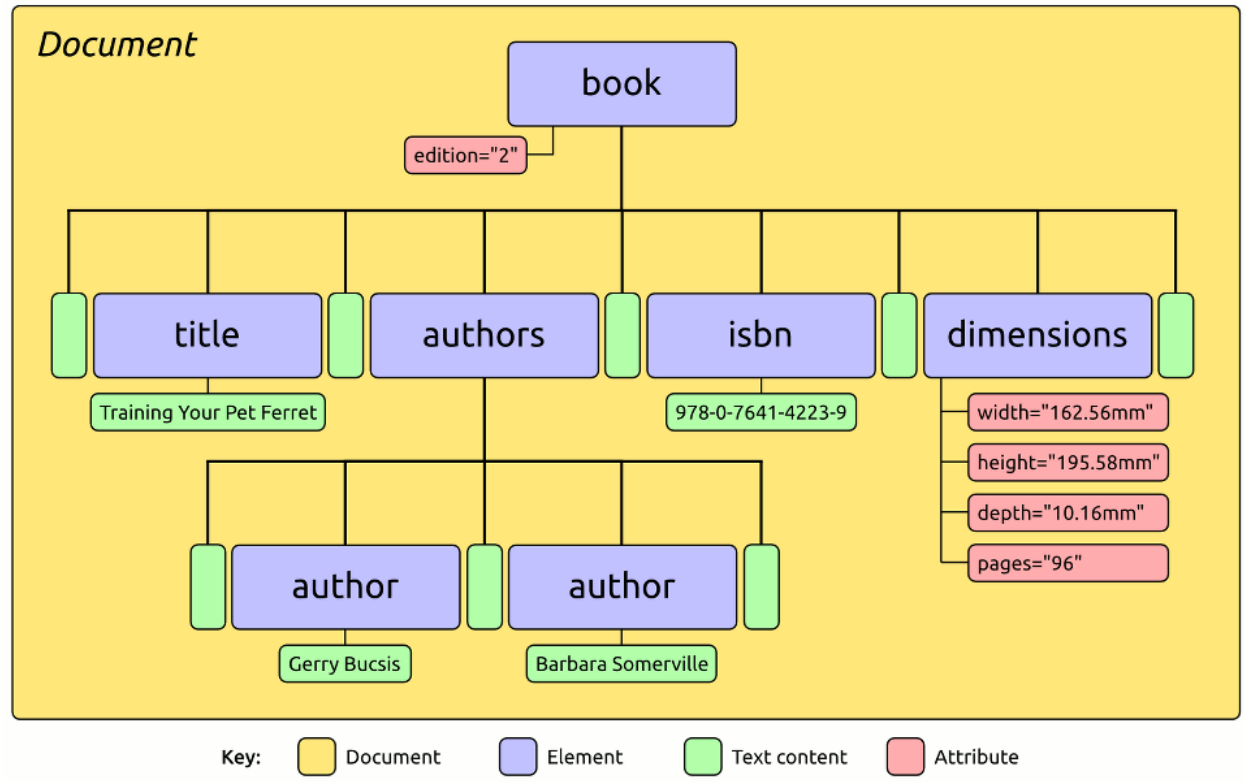

Fig. 3.2: Document Object Model including whitespace-only text nodes

If you want to filter child nodes by type, XML::LibXML provides a number of constants which you can import when you load the module:

```
use XML::LibXML qw(:libxml);
```

And then you can compare $node->nodeType to these constants:

```
my @elements = grep { $_->nodeType == XML_ELEMENT_NODE } $book->childNodes;
$count = @elements;
say "\$book has $count child elements:";
$i = 0;
foreach my $child (@elements) {
    say $i++, ": is a ", ref($child), ', name = ', $child->nodeName;
}
```

Output:

```
$book has 4 child elements:
0: is a XML::LibXML::Element, name = title
1: is a XML::LibXML::Element, name = authors
2: is a XML::LibXML::Element, name = isbn
3: is a XML::LibXML::Element, name = dimensions
```

That technique is useful for the general case of filtering child nodes by type, but if you simply want to exclude text nodes that contain only whitespace, you can do that by specifying the `no_blanks` option when parsing the source document. This causes `libxml` to discard those 'blank' text nodes rather than adding them into the DOM:

```
my $dom = XML::LibXML->load_xml(location => 'book.xml', no_blanks => 1);
```

Output:

```
$book has 4 child nodes:
0: is a XML::LibXML::Element, name = title
1: is a XML::LibXML::Element, name = authors
2: is a XML::LibXML::Element, name = isbn
3: is a XML::LibXML::Element, name = dimensions
```

Blank text nodes are really only a problem if you use the low-level DOM methods for walking through child nodes. You'll generally find that it's much easier to just use `findnodes()` and XPath Expressions to select exactly the elements or other nodes you want. If the blank nodes don't match your selector then they won't be returned in the result set.

'Text' objects

The green boxes in the picture represent 'Text' nodes. The reference documentation for the XML::LibXML::Text class lists a small number of methods and many more are inherited from the Node class.

There are numerous ways to get the text string out of a Text object but it's important to be clear on whether you want the text as it appears in the XML (including any entity escaping) or whether you want the plain text that the source represents. Consider this tiny source document:

```
<item>Fish &amp; Chips</item>
```

And these different methods for accessing the text:

```
my $item = $dom->documentElement;
my($text) = $item->childNodes();

say '$text is a ', ref($text);
say '$text->data = ', $text->data;
say '$text->nodeValue = ', $text->nodeValue;
say '$text->to_literal = ', $text->to_literal;
say '$text->toString = ', $text->toString;
say '$text as a string: ', $text;
```

Producing this output:

```
$text is a XML::LibXML::Text
$text->data = Fish & Chips
$text->nodeValue = Fish & Chips
$text->to_literal = Fish & Chips
$text->toString = Fish &amp; Chips
$text as a string: Fish &amp; Chips
```

The `data()` and `nodeValue()` methods are essentially aliases. The `to_literal()` method produces the same output via a more complex route, but has the advantage that you can call it on any object in the DOM.

The `toString()` method is really only useful for serialising a whole DOM or a DOM fragment out to XML. Stringification is particularly handy when you just want to print an object out for debugging purposes.

'Attr' objects

The red boxes in the picture represent attributes. You're unlikely to ever need to deal with attribute **objects** since it's easier to get and set attribute values by calling methods on an Element object and passing in plain string values. An even easier approach is to use the *tied hash interface* that allows you to treat each element as if it were a hashref and access attribute values via hash keys:

```
my $book = $dom->documentElement;
my($dim) = $book->getChildrenByTagName('dimensions');

say '$dim->getAttribute("width") = ', $dim->getAttribute("width");
say "\$dim->{width} = $dim->{width}";
```

Output:

```
$dim->getAttribute("width") = 162.56mm
$dim->{width} = 162.56mm
```

The class name for the attribute objects is 'Attr' - the unfortunate truncation of the class name derives from the W3C DOM spec. The reference documentation is at: XML::LibXML::Attr. Some additional methods are inherited from the Node class but not all the Node methods work with Attr objects (once again due to behaviour specified by the W3C DOM).

```
# You probably don't need this object interface for attributes at all.
# The previous example showed how to access attributes directly via
# the Element object.

my $book = $dom->documentElement;
my($dim) = $book->getChildrenByTagName('dimensions');
my($width_attr) = $dim->getAttributeNode('width');

say '$width_attr is a ', ref($width_attr);
say '$width_attr->nodeName: ', $width_attr->nodeName;
say '$width_attr->value: ', $width_attr->value;
say '$width_attr as a string: ', $width_attr;
```

Output:

```
$width_attr is a XML::LibXML::Attr
$width_attr->nodeName: width
$width_attr->value: 162.56mm
$width_attr as a string:  width="162.56mm"
```

'NodeList' objects

The 'NodeList' object is a part of the DOM that makes sense in DOM implementations for other languages (e.g.: Java) but doesn't make much sense in Perl. Methods such as `childNodes()` or `findnodes()` that may need to return multiple nodes, return a 'NodeList' object which contains the matching nodes and allows the caller to iterate through the result set:

```
my $result = $book->childNodes;
say '$result is a ', ref($result);
my $i = 1;
foreach my $i (1..$result->size) {
    my $node = $result->get_node($i);
    say $node->nodeName if $node->nodeType == XML_ELEMENT_NODE;
}
```

Output:

```
$result is a XML::LibXML::NodeList
title
authors
isbn
dimensions
```

But things don't need to be that complicated in Perl - if a method needs to return a list of values then it can just return a list of values. So the Perl bindings for DOM methods that would return a NodeList check the calling context. If called in a scalar context, they return a NodeList object (as above) but in a list context they just return the list of values - much simpler:

```
foreach my $node ($book->childNodes) {
    say $node->nodeName if $node->nodeType == XML_ELEMENT_NODE;
}
```

When you execute a search that you expect should match exactly one node, take care to still use list context:

```
my($dim) = $book->findnodes('./dimensions');
say '$dim is a ', ref($dim);
say 'Page count: ', $dim->{pages};
```

Output:

```
$dim is a XML::LibXML::Element
Page count: 96
```

In this example, the assignment `my($dim) = ...` uses parentheses to force list context, so `findnodes()` will return a list of Element nodes and the first will be assigned to `$dim`. Without the parentheses, a NodeList would be assigned to `$dim`.

If for some reason you find yourself with a NodeList object you can extract the contents as a simple list with `$result->get_nodelist`.

The NodeList object does implement the `to_literal()` method, which returns the text content of all the nodes, concatenated together as a single string. If you need a list of individual string values, you can use `$result->to_literal_list()`:

```
say 'Authors: ', join ', ', $book->findnodes('.//author')->to_literal_list;
```

Output:

```
Authors: Gerry Bucsis, Barbara Somerville
```

Modifying the DOM

If you wish to modify the DOM, you can create new nodes and add them into the node hierarchy in the appropriate place. You can also modify, move and delete existing nodes. Let's start with a simple XML document:

```
my $xml = q{
<record>
  <event>Men's 100m</event>
</record>
};
my $dom = XML::LibXML->load_xml(string => $xml);
```

Navigate to the <event> element; change its text content; add an attribute and print out the resulting XML:

```
my $record = $dom->documentElement;
my($event) = $record->getChildrenByTagName('event');
my $text = $event->firstChild;
$text->setData("Men's 100 metres");
$event->{type} = 'sprint';
say $dom->toString;
```

Output:

```
<?xml version="1.0"?>
<record>
  <event type="sprint">Men's 100 metres</event>
</record>
```

You can use $dom->createElement to create a new element and then add it to an existing node's list of child nodes. You can append it to the end of the list of children or insert it before/after a specific existing child:

```
my $country = $dom->createElement('country');
$country->appendText('Jamaica');
$record->appendChild($country);

my $athlete = $dom->createElement('athlete');
$athlete->appendText('Usain Bolt');
$record->insertBefore($athlete, $country);

say $dom->toString;
```

Output:

```
<?xml version="1.0"?>
<record>
  <event type="sprint">Men's 100 metres</event>
<athlete>Usain Bolt</athlete><country>Jamaica</country></record>
```

Unfortunately that output is probably messier than you were expecting. To get nicely indented XML output, you'd need to create text nodes containing a newline and the appropriate number of spaces for indentation; and then add those text nodes in before each new element. Or, an easier way would be to pass the numeric value 1 to the toString() method as a flag indicating that you'd like the output auto-indented:

```
say $dom->toString(1);
```

Output:

```
<?xml version="1.0"?>
<record>
  <event type="sprint">Men's 100 metres</event>
<athlete>Usain Bolt</athlete><country>Jamaica</country></record>
```

But sadly that didn't seem to work. The libxml library won't add indentation to *'mixed content'* - an element whose list of child nodes contains a mixture of both Element nodes and Text nodes. In this case the <record> element

contains mixed content (there's a whitespace text node before the <event> and another after it) so libxml does not try to indent its contents.

If we strip out those extra text nodes then libxml will add indenting:

```
foreach my $node ($record->childNodes()) {
    $record->removeChild($node) if $node->nodeType != XML_ELEMENT_NODE;
}
```

Output:

```
<?xml version="1.0"?>
<record>
  <event type="sprint">Men's 100 metres</event>
  <athlete>Usain Bolt</athlete>
  <country>Jamaica</country>
</record>
```

While that did work, it required some rather specific knowledge of the document structure. We were relying on knowing that all the text children of the <record> element were whitespace-only and could be discarded. Here's a more generic approach which searches recursively through the document and deletes every text node that contains only whitespace:

```
foreach ($dom->findnodes('//text()')) {
    $_->parentNode->removeChild($_) unless /\S/;
}
```

That code is a little tricky so some explanation is probably in order:

- The loop does not declare a loop variable, so $_ is used implicitly.
- The trailing unless clause runs a regex comparison against $_ which implicitly calls toString() on the Text node.
- unless /\S/ is a double negative which means *"unless the text contains a non-whitespace character"*.
- the removeChild() method needs to be called on the *parent* of the node we're removing, so if the Text node is whitespace-only then we need to use parentNode().

Of course an even simpler solution in this case would have been to turn on the no_blanks option (described earlier) when parsing the initial XML document.

Another handy method for adding to the DOM is appendWellBalancedChunk(). This method takes a string containing a fragment of XML. It must be well balanced - each opening tag must have a matching closing tag and elements must be properly nested. The XML fragment is parsed to create a XML::LibXML::DocumentFragment which is then appended to the target element:

```
$record->appendWellBalancedChunk(
    '<time>9.58s</time><date>2009-08-16</date><location>Berlin, Germany</location>'
);
```

Output:

```
<?xml version="1.0"?>
<record>
  <event type="sprint">Men's 100 metres</event>
  <athlete>Usain Bolt</athlete>
  <country>Jamaica</country>
  <time>9.58s</time>
  <date>2009-08-16</date>
  <location>Berlin, Germany</location>
</record>
```

One 'gotcha' with the `appendWellBalancedChunk()` method is that the XML parsing phase expects a string of bytes. So if you have a Perl string that might contain non-ASCII characters, you first need to encode the character string to a byte string in UTF-8 and then pass the byte string to `appendWellBalancedChunk()`:

```
my $byte_string = Encode::encode_utf8($perl_string);
$record->appendWellBalancedChunk($byte_string, 'UTF-8');
```

Creating a new Document

You can create a document from scratch by calling `XML::LibXML::Document->new()` rather than parsing from an existing document. Then use the methods discussed above to add elements and text content:

```
#!/usr/bin/perl

use 5.010;
use strict;
use warnings;
use autodie;

use XML::LibXML;

my $dom = XML::LibXML::Document->new('1.0', 'UTF-8');
my $title = $dom->createElement('title');
$title->appendText("Caf\x{e9} lunch: \x{20ac}12.50");
$dom->setDocumentElement($title);

my $filename = 'temp-utf8.xml';
open my $out, '>:raw', $filename;
```

In this example, the document encoding was declared as UTF-8 when the Document object was created. Text content was added by calling `appendText()` and passing it a normal Perl character string - which happened to contain some non-ASCII characters. When opening the file for output it is not necessary to use an encoding layer since the output from `libxml` will already be encoded as utf-8 bytes.

The file contents look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<title>Café lunch: €12.50</title>
```

If we hex-dump the file we can see the e-acute character was written out as the 2-byte UTF-8 sequence `C3 A9` and the euro symbol was written as a 3-byte UTF-8 sequence: `E2 82 AC`:

```
00000000: 3c3f 786d 6c20 7665 7273 696f 6e3d 2231  <?xml version="1
00000010: 2e30 2220 656e 636f 6469 6e67 3d22 5554  .0" encoding="UT
00000020: 462d 3822 3f3e 0a3c 7469 746c 653e 4361  F-8"?>.<title>Ca
00000030: 66c3 a920 6c75 6e63 683a 20e2 82ac 3132  f.. lunch: ...12
00000040: 2e35 303c 2f74 6974 6c65 3e0a            .50</title>.
```

To output the document in a different encoding all you need to do is change the second parameter passed to `new()` when creating the Document object. No other code changes are required:

```
my $dom = XML::LibXML::Document->new('1.0', 'ISO8859-1');
```

This time when hex-dumping the file we can see the e-acute character was written out as the single byte `E9` and the euro symbol which cannot be represented directly in Latin-1 was written in numeric character entity form `&#8364;`:

```
00000000: 3c3f 786d 6c20 7665 7273 696f 6e3d 2231  <?xml version="1
00000010: 2e30 2220 656e 636f 6469 6e67 3d22 4953  .0" encoding="IS
00000020: 4f38 3835 392d 3122 3f3e 0a3c 7469 746c  O8859-1"?>.<titl
00000030: 653e 4361 66e9 206c 756e 6368 3a20 2623  e>Caf. lunch: &#
00000040: 3833 3634 3b31 322e 3530 3c2f 7469 746c  8364;12.50</titl
00000050: 653e 0a                                 e>.
```

If you're generating XML from scratch then creating and assembling DOM nodes is very fiddly and `XML::LibXML` might not be the best tool for the job. XML::Generator is an excellent module for generating XML - especially if you need to deal with namespaces.

CHAPTER

FOUR

WORKING WITH XML NAMESPACES

Using the `findnodes()` method as described in the basic examples section doesn't work when the XML document uses 'namespaces'. This section describes the extra steps you need to take to work with namespaces in XML.

XML 'namespaces' allow you to build documents using elements from more than one vocabulary. For example one XML document might include both SVG elements to describe a drawing, as well as Dublin Core elements to define metadata *about* the drawing. The two different vocabularies are defined by separate bodies - the W3C and the DCMI respectively. Associating each element in your document with a namespace allows a processor to distinguish elements that use the same element names.

The scripts in this section will use the SVG document: `xml-libxml.svg`. Which starts like this:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<svg
   xmlns="http://www.w3.org/2000/svg"
   xmlns:svg="http://www.w3.org/2000/svg"
   xmlns:dc="http://purl.org/dc/elements/1.1/"
   xmlns:cc="http://creativecommons.org/ns#"
   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
   xmlns:xlink="http://www.w3.org/1999/xlink"
   xmlns:sodipodi="http://sodipodi.sourceforge.net/DTD/sodipodi-0.dtd"
   xmlns:inkscape="http://www.inkscape.org/namespaces/inkscape"
   width="1031.3961"
   height="278.02112"
   id="svg2"
   sodipodi:version="0.32"
   inkscape:version="0.48.4 r9939"
   sodipodi:docname="xml-libxml.svg"
   inkscape:output_extension="org.inkscape.output.svg.inkscape"
   version="1.0"
   inkscape:export-filename="/home/grant/Desktop/xml-libxml.png"
   inkscape:export-xdpi="79.860001"
   inkscape:export-ydpi="79.860001">
```

Because the top-level `<svg>` element uses `xmlns="http://www.w3.org/2000/svg"` to declare a **default namespace** , every other element will be in that namespace unless the element name includes a prefix for a different namespace, or unless an element declares a different default namespace for itself and its children.

The first child element in the document is a `<title>` element with no namespace prefix, so it is associated with the default namespace URI: `http://www.w3.org/2000/svg`.

```
  <title id="title5798">Example SVG File</title>
```

A later section of the document includes a `<title>` element with the *dc:* namespace prefix, so it is associated with the URI: `http://purl.org/dc/elements/1.1/`.

```
        <dc:title>XML::LibXML Logo</dc:title>
```

You can confirm using the XPath sandbox that the XPath expression `//title` does not match either of the `<title>` elements in the test document:

```
//title
```

You can also use the following Perl code to confirm that `findnodes()` does not return any matches for the XPath expression `//title`:

```
my $match_count = $dom->findnodes('//title')->size;
say "XPath: //title  Matching node count: $match_count";
```

Output:

```
XPath: //title  Matching node count: 0
```

When an element in a document is associated with a namespace URI it will only match an XPath expression that includes a prefix that is also associated with the same namespace URI. However it's important to stress that it's not the prefix that is being matched, but the URI associated with the prefix.

Using the XPath sandbox, you can confirm that if we register the 'Dublin Core' namespace URI with the prefix `dc`, the XPath expression `//dc:title` will match the `<title>` element in the `<metadata>` section:

```
//dc:title
```

However if we register the same URI with the prefix `dublin` instead then we can match the same element using the `dublin` prefix in our XPath:

```
//dublin:title
```

In order to associate namespace prefixes in XPath expressions with namespace URIs, we need to use an XML::LibXML::XPathContext object. This is a multi-step process:

1. create an XPathContext object associated with the document you want to search
2. register a prefix and associated URI for each namespace you want to include in your query
3. call the `findnodes()` method on the XPathContext object rather than directly on the DOM object

```
use XML::LibXML;
use XML::LibXML::XPathContext;

my $filename = 'xml-libxml.svg';
my $dom = XML::LibXML->load_xml(location => $filename);

my $xpc = XML::LibXML::XPathContext->new($dom);
$xpc->registerNs('vg',  'http://www.w3.org/2000/svg');
$xpc->registerNs('dub', 'http://purl.org/dc/elements/1.1/');

my($match1) = $xpc->findnodes('//vg:title');
say 'XPath: //vg:title   Matched: ', $match1;

my($match2) = $xpc->findnodes('//dub:title');
say 'XPath: //dub:title  Matched: ', $match2;
```

Output:

```
XPath: //vg:title   Matched: <title id="title5798">Example SVG File</title>
XPath: //dub:title  Matched: <dc:title>XML::LibXML Logo</dc:title>
```

You'll recall from earlier examples that you can search within a node by calling `findnodes()` on the element node (rather than the document) and using an XPath expression like `./child` where the dot refers to the *context* node. However when you're dealing with namespaces that won't work, because you need to call `findnodes()` on the XPathContext object. The solution is to pass `findnodes()` a second argument, after the XPath expression. The additional argument is the element to use as a context node:

```
use XML::LibXML;
use XML::LibXML::XPathContext;

my $filename = 'xml-libxml.svg';
my $dom = XML::LibXML->load_xml(location => $filename, no_blanks => 1);

my $xpc = XML::LibXML::XPathContext->new($dom);
$xpc->registerNs('svg', 'http://www.w3.org/2000/svg');
$xpc->registerNs('dc',  'http://purl.org/dc/elements/1.1/');

my($metadata) = $xpc->findnodes('//svg:metadata') or die "No metadata";

foreach my $el ($xpc->findnodes('.//dc:*', $metadata)) {
    my $name  = $el->localname;
    my $value = $el->to_literal or next;
    say "$name=$value";
}
```

Output:

```
format=image/svg+xml
title=XML::LibXML Logo
creator=Grant McLean
date=2016-03-26
subject=perlxmllibxml
description=An SVG file created as an example for parsing XML with namespaces.
```

One small feature of that script which is worth noting is the use of `$el->localname` to get the name of the element *without* the namespace prefix. The more commonly used `$el->nodeName` method does include the namespace prefix as it appears in the document.

CHAPTER

FIVE

WORKING WITH LARGE DOCUMENTS

The examples so far have all started by creating a data structure called a Document Object Model to represent the whole XML document. Using XPath expressions to navigate the DOM can be both powerful and convenient, but the cost in memory consumption can be quite high. For example, parsing a 50MB XML file into a DOM might need 500MB of memory.

If you routinely work with very large XML documents, you might find that `XML::LibXML`'s DOM parser wants to consume more memory than your system has installed. In such cases, you can instead use the 'pull parser' API which is accessed via the `XML::LibXML::Reader` interface.

The Reader Loop

To gain a better understanding of how the reader API is used, let's start by seeing what happens when we parse this very simple XML document:

```
<country code="IE">
  <name>Ireland</name>
  <population>4761657</population>
</country>
```

This script loads the reader API and parses the XML file:

```
#!/usr/bin/perl

use 5.010;
use strict;
use warnings;

use XML::LibXML::Reader;

my $filename = 'country.xml';

my $reader = XML::LibXML::Reader->new(location => $filename)
    or die "cannot read file '$filename': $!\n";

while($reader->read) {
    printf(
        "Node type: %2u  Depth: %2u  Name: %s\n",
        $reader->nodeType,
        $reader->depth,
        $reader->name
    );
}
```

and produces the following output:

```
Node type:  1  Depth:  0  Name: country
Node type: 14  Depth:  1  Name: #text
Node type:  1  Depth:  1  Name: name
Node type:  3  Depth:  2  Name: #text
Node type: 15  Depth:  1  Name: name
Node type: 14  Depth:  1  Name: #text
Node type:  1  Depth:  1  Name: population
Node type:  3  Depth:  2  Name: #text
Node type: 15  Depth:  1  Name: population
Node type: 14  Depth:  1  Name: #text
Node type: 15  Depth:  0  Name: country
```

We can see from the output that the `while` loop executes 11 times. As the XML document is parsed, the `$reader` object acts as a cursor advancing through the document. Each time a 'node' has been parsed, the `read` method returns to allow the state of the parse and the current node to be interrogated.

To make sense of it we really need to turn those 'Node Type' numbers into something a bit more readable. The `XML::LibXML::Reader` module exports a set of constants for this purpose. Here's a modified version of the script:

```
#!/usr/bin/perl

use 5.010;
use strict;
use warnings;

use XML::LibXML::Reader;

my $filename = 'country.xml';

my $reader = XML::LibXML::Reader->new(location => $filename)
    or die "cannot read file '$filename': $!\n";

my %type_name = (
    &XML_READER_TYPE_ELEMENT                => 'ELEMENT',
    &XML_READER_TYPE_ATTRIBUTE              => 'ATTRIBUTE',
    &XML_READER_TYPE_TEXT                   => 'TEXT',
    &XML_READER_TYPE_CDATA                  => 'CDATA',
    &XML_READER_TYPE_ENTITY_REFERENCE       => 'ENTITY_REFERENCE',
    &XML_READER_TYPE_ENTITY                 => 'ENTITY',
    &XML_READER_TYPE_PROCESSING_INSTRUCTION => 'PROCESSING_INSTRUCTION',
    &XML_READER_TYPE_COMMENT                => 'COMMENT',
    &XML_READER_TYPE_DOCUMENT               => 'DOCUMENT',
    &XML_READER_TYPE_DOCUMENT_TYPE          => 'DOCUMENT_TYPE',
    &XML_READER_TYPE_DOCUMENT_FRAGMENT      => 'DOCUMENT_FRAGMENT',
    &XML_READER_TYPE_NOTATION               => 'NOTATION',
    &XML_READER_TYPE_WHITESPACE             => 'WHITESPACE',
    &XML_READER_TYPE_SIGNIFICANT_WHITESPACE => 'SIGNIFICANT_WHITESPACE',
    &XML_READER_TYPE_END_ELEMENT            => 'END_ELEMENT',
);

say " Step | Node Type                | Depth | Name";
say "------+--------------------------+-------+-------";

my $step = 1;
while($reader->read) {
    printf(
```

```
        " %3u  | %-22s  | %4u  | %s\n",
        $step++,
        $type_name{$reader->nodeType},
        $reader->depth,
        $reader->name
    );
}
```

that produces the following tidier output:

```
 Step | Node Type               | Depth | Name
------+-------------------------+-------+-------
   1  | ELEMENT                 |     0 | country
   2  | SIGNIFICANT_WHITESPACE  |     1 | #text
   3  | ELEMENT                 |     1 | name
   4  | TEXT                    |     2 | #text
   5  | END_ELEMENT             |     1 | name
   6  | SIGNIFICANT_WHITESPACE  |     1 | #text
   7  | ELEMENT                 |     1 | population
   8  | TEXT                    |     2 | #text
   9  | END_ELEMENT             |     1 | population
  10  | SIGNIFICANT_WHITESPACE  |     1 | #text
  11  | END_ELEMENT             |     0 | country
```

from the same XML :

```
<country code="IE">
  <name>Ireland</name>
  <population>4761657</population>
</country>
```

Some things to note:

- At step 1, when the `read` method returns for the first time, the cursor has advanced to the closing '>' of the `<country>` start tag. We could retrieve an attribute value by calling `$reader->getAttribute('code')` but we can't examine child elements or text nodes because the parser has not seen them yet.
- At step 2, the parser has processed a chunk of text and found that it contains only whitespace (side note: all whitespace is considered to be 'significant' unless a DTD is loaded and defines which whitespace is insignificant). Although we can get access to the text, the `$reader` object can no longer tell us that it is a child of a `<country>` element - the parser has discarded that information already.
- At step 3, the parser can tell us the current node is a `<name>` element, and the `depth` method can tell us that there is one ancestor element. However there is no way to determine the name of the parent element.
- At step 4 a text node has been identified and we can call `$reader->value` to get the text string `"Ireland"`, but the parser can no longer tell us the name of the element it belongs to.
- At step 5 we have reached the end of the `<name>` element, but we no longer have access to the text it contained.

But now you surely get the idea - the `XML::LibXML::Reader` API is able to keep its memory requirements low by discarding data from one parse step before proceeding to the next. The vastly lowered memory demands come at the cost of significantly lowered convenience for the programmer. However, as we'll see in the next section, there is a middle ground that can provide the convenience of the DOM API combined with the reduced memory usage of the Reader API.

Bring Back the DOM

Huge XML documents usually contain a long list of similar elements. For example Wikipedia make XML 'dumps' available for download.

At the time of writing, the `enwiki-latest-abstract1.xml.gz` file was about 100MB in size - about 800MB uncompressed. However it contained information summarising over half a million Wikipedia articles. So whilst the file is very large, the `<doc>` elements describing each article are, on average, less than 1.5KB. The following extract is reformatted for clarity to illustrate the file structure:

```
<feed>
  <doc>
    <title>Wikipedia: Anarchism</title>
    <url>https://en.wikipedia.org/wiki/Anarchism</url>
    <abstract>Anarchism is a political philosophy that advocates
    self-governed societies based on voluntary institutions.
    These are often described as stateless societies ...</abstract>
    <links>
      <sublink linktype="nav">
        <anchor>History</anchor>
        <link>https://en.wikipedia.org/wiki/Anarchism#History</link>
      </sublink>
      <sublink linktype="nav">
        <anchor>Origins</anchor>
        <link>https://en.wikipedia.org/wiki/Anarchism#Origins</link>
      </sublink>
      <!-- more sublink elements -->
    </links>
  </doc>
  <doc>
    <title>Wikipedia: Autism</title>
    <url>https://en.wikipedia.org/wiki/Autism</url>
    <abstract>...</abstract>
    <links>
      <!-- sublink elements -->
    </links>
  </doc>
  <!-- (many) more doc elements -->
</feed>
```

To process this file, we can use the Reader API to locate each `<doc>` element and then parse that element *and all its children* into a DOM fragment. We can then use the familiar and convenient XPath tools and DOM methods to process each fragment.

Another useful technique when working with large files is to leave the files in their compressed form and use a Perl IO layer to decompress them on the fly. You can achieve this using the PerlIO::gzip module from CPAN.

To illustrate these techniques, the following script uses the Reader API to pick out each `<doc>` element and slurp it into a DOM fragment. Then XPath queries are used to examine the child nodes and determine if the `<doc>` is 'interesting' - does it have a sub-heading that contains variant of the word "controversy"? Uninteresting elements are skipped, interesting elements are reported in summary form: article title, interesting subheading, URL.

```
#!/usr/bin/perl

use 5.010;
use strict;
use warnings;
use autodie;
```

```
use PerlIO::gzip;
use XML::LibXML::Reader;

binmode(STDOUT, ':utf8');

my $filename = 'enwiki-latest-abstract1-abridged.xml.gz';
open my $fh, '<:gzip', $filename;

my $reader = XML::LibXML::Reader->new(IO => $fh);

my $controversy_xpath = q{./links/sublink[contains(./anchor, 'Controvers')]};

while($reader->read) {
    next unless $reader->nodeType == XML_READER_TYPE_ELEMENT;
    next unless $reader->name eq 'doc';
    my $doc = $reader->copyCurrentNode(1);
    if(my($target) = $doc->findnodes($controversy_xpath)) {
        say 'Title: ', $doc->findvalue('./title');
        say '  ', $target->findvalue('./anchor');
        say '  ', $target->findvalue('./link');
        say '';
    }
    $reader->next;
}
```

In the script above, `$doc` is a DOM fragment that can be queried and manipulated using the DOM methods described in earlier chapters.

At the start of the `while` loop, a couple of conditional `next` statements allow skipping quickly to the start of the next `<doc>` element. Depending on the document you're dealing with, you might need to also use the `depth` method to avoid deeply nested elements that also happened to be named "doc".

The call to `$reader->copyCurrentNode(1)` creates a DOM fragment from the current element. The `1` passed as an argument is a boolean flag that causes all child elements to be included.

In order to build the DOM fragment, the `$reader` has to process all content up to the matching `XML_READER_TYPE_END_ELEMENT` node. You may be surprised to learn that this does not advance the cursor. So the next call to `$reader->read` will advance to the first child node of the current `<doc>`. In our case, that would be a waste of time - there is no need to use the Reader API to re-process the child nodes that we already processed with the DOM API. Therefore after processing a `<doc>`, we call `$reader->next` to skip directly to the node following the matching `</doc>` end tag. When this script was used to process the full-sized file, adding this call to `next` reduced the run time by almost 50%.

When processing files with millions of elements, a small optimisation in the main loop can make a noticeable difference to the run time. For example, building the DOM fragment is a relatively expensive operation. The call to `$reader->copyCurrentNode(1)` is equivalent to:

```
my $xml = $reader->readOuterXml;
my $doc = XML::LibXML->load_xml(string => $xml);
```

As an optimisation, we can avoid the step of building the DOM fragment if a quick regex check of the source XML tells us that it doesn't contain the word we're going to look for with the XPath query. This rewritten main loop shave about 20% off the run time:

```
my $controversy_xpath = q{/doc/links/sublink[contains(./anchor, 'Controvers')]};

while($reader->read) {
    next unless $reader->nodeType == XML_READER_TYPE_ELEMENT;
    next unless $reader->name eq 'doc';
```

```
    my $xml = $reader->readOuterXml;
    if($xml =~ /Controvers/) {
        my $doc = XML::LibXML->load_xml(string => $xml);
        if(my($target) = $doc->findnodes($controversy_xpath)) {
            say 'Title: ', $doc->findvalue('/doc/title');
            say '  ', $target->findvalue('./anchor');
            say '  ', $target->findvalue('./link');
            say '';
        }
    }
    $reader->next;
}
```

Error Handling

Error handling is a little different with the Reader API vs the DOM API. The DOM API will parse the whole document and throw an exception immediately if it encounters and error in the XML. So if there's an error you won't get a DOM.

The Reader API on the other hand will start returning nodes to your script via `$reader->read` as soon as the parsing starts [1]. If there is an error in your document, you won't know until your parser reaches the error - then you'll get the exception.

You need to bear this in mind when parsing with the Reader API. For example if you were reading elements to populate records in a database, you might want to wrap all the database INSERT statement in a transaction so that you can roll them all back if you encounter a parse error.

Another useful technique is to parse the document twice, once to check the XML is well-formed and once to actually process it. The `finish` method provides a quick way to parse from the current position to the end of the document:

```
    my $reader = XML::LibXML::Reader->new(IO => $fh);
    $reader->finish;
```

You'll then need to reopen the file and create a new Reader object for the second parse.

In some applications you might scan through the file looking for a specific section. Once the target has been located and the required information extracted, you might not need to look at any more elements. However as we've seen, you should call `finish` to ensure there are no errors in the rest of the XML.

Working With Patterns

Our sample script is identifying elements at the top of the main loop by examining the node type and the node name:

```
while($reader->read) {
    next unless $reader->nodeType == XML_READER_TYPE_ELEMENT;
    next unless $reader->name eq 'doc';
```

Although these are simple checks, they do still involve two method calls and passing scalar values across the XS boundary between `libxml` and the Perl runtime. An alternative approach is to compile a 'pattern' (essentially a simplified subset of XPath) using XML::LibXML::Pattern and run a complex set of checks with a single method call:

```
my $doc_pattern = XML::LibXML::Pattern->new('/feed/doc');
while($reader->read) {
    next unless $reader->matchesPattern($doc_pattern);
```

[1] In practice, the Reader API will read the XML in chunks and check each chunk is well-formed before it starts delivering node events. This means that a short document with an error may trigger an exception before any nodes have been delivered.

In our example, the <doc> elements that we're interested in are all adjacent, so when we finish processing one, the very next element is another <doc>. If your document is not structured this way, you might find it useful to skip over large sections of document to find the next element that matches a pattern, like this:

```
$reader->nextPatternMatch($pattern);
```

You can also use patterns with the preservePattern method to create a DOM subset of a larger document. For example:

```
my $filename = 'enwiki-latest-abstract1-structure.xml';

my $reader = XML::LibXML::Reader->new(location => $filename);
$reader->preservePattern('/feed/doc/title');
$reader->finish;

say $reader->document->toString(1);
```

Which will produce this output:

```
<?xml version="1.0"?>
<feed>
  <doc>
    <title>Wikipedia: Anarchism</title>
  </doc>
  <doc>
    <title>Wikipedia: Autism</title>
  </doc>
</feed>
```

Note, this technique does construct the DOM in memory and then serialise it at the end, so if you have a huge document and many nodes match the pattern then you will consume a large amount of memory.

CHAPTER

SIX

WORKING WITH HTML

If you ever need to extract text and data from HTML documents, the `libxml` parser and DOM provide very useful tools. You might imagine that `libxml` would only work with XHTML and even then only strictly well-formed documents. In fact, the parser has an HTML mode that handles unclosed tags like `<img>` and `<br>` and is even able to recover from parse errors caused by poorly formed HTML.

Let's start with this mess of HTML tag soup:

```
<html><head><title>Example (Untidy) HTML Doc</title></head>
<body><p>Here's a paragraph with <i>poorly <b>nested</i></b>
tags.  Followed by a list of items &mdash; with unclosed tags</p>
<ul><li>red</li><li>orange<li>yellow</ul></body></html>
```

To read the file in, you'd use the `load_html()` method rather than `load_xml()`. You'll almost certainly want to use the `recover => 1` option to tell the parser to try to recover from parse errors and carry on to produce a DOM.

```
#!/usr/bin/perl

use 5.010;
use strict;
use warnings;

use XML::LibXML;

my $filename = 'untidy.html';

my $dom = XML::LibXML->load_html(
    location  => $filename,
    recover   => 1,
);

say $dom->toStringHTML();
```

When the DOM is serialised with `toStringHTML()`, some rudimentary formatting is applied automatically. Unfortunately there is no option to add indenting to the HTML output:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN" "http://www.w3.org/TR/REC-html40/loose
<html>
<head><title>Example (Untidy) HTML Doc</title></head>
<body>
<p>Here's a paragraph with <i>poorly <b>nested</b></i>
tags.  Followed by a list of items &mdash; with unclosed tags</p>
<ul>
<li>red</li>
<li>orange</li>
<li>yellow</li>
```

```
</ul>
</body>
</html>
```

While the document is being parsed, you'll see messages like this on STDERR:

```
untidy.html:2: HTML parser error : Opening and ending tag mismatch: i and b
<body><p>Here's a paragraph with <i>poorly <b>nested</i></b>
                                                    ^
untidy.html:2: HTML parser error : Unexpected end tag : b
<body><p>Here's a paragraph with <i>poorly <b>nested</i></b>
                                                        ^
```

You can turn off the error output with the `suppress_errors` option:

```
my $dom = XML::LibXML->load_html(
    location        => $filename,
    recover         => 1,
    suppress_errors => 1,
);
```

That option doesn't seem to work with all versions of `XML::LibXML` so you may want to use a routine like this that sends STDERR to `/dev/null` during parsing, but still allows other output to STDERR when the parse function returns:

```
use File::Spec;

sub parse_html_file {
    my($filename) = @_;

    local(*STDERR);
    open STDERR, '>>', File::Spec->devnull();
    return XML::LibXML->load_html(
        location        => $filename,
        recover         => 1,
        suppress_errors => 1,
    );
};
```

Querying HTML with XPath

The main tool you'll use for extracting data from HTML is the `findnodes()` method that was introduced in A Basic Example and XPath Expressions. For these examples, the source HTML comes from the CSS Zen Garden Project and is in the file `css-zen-garden.html`.

This script locates every `<h3>` element inside the `<div>` with an `id` attribute value of `"zen-supporting"`:

```
my $filename = 'css-zen-garden.html';

my $dom = XML::LibXML->load_html(
    location        => $filename,
    recover         => 1,
    suppress_errors => 1,
);
```

```
my $xpath = '//div[@id="zen-supporting"]//h3';
say "$_" foreach $dom->findnodes($xpath)->to_literal_list;
```

Output:

```
So What is This About?
Participation
Benefits
Requirements
```

For a more complex example, the next script iterates through each <li> in the "Select a Design" section and extracts three items of information for each: the name of the design, the name of the designer, and a link to view the design. Once the information has been collected, it is dumped out in JSON format:

```
use XML::LibXML;
use URI::URL;
use JSON qw(to_json);

my $base_url = 'http://csszengarden.com/';
my $filename = 'css-zen-garden.html';

my $dom = XML::LibXML->load_html(
    location        => $filename,
    recover         => 1,
    suppress_errors => 1,
);

my @designs;
my $xpath = '//div[@id="design-selection"]//li';
foreach my $design ($dom->findnodes($xpath)) {
    my($name, $designer) = $design->findnodes('./a')->to_literal_list;
    my($url) = $design->findnodes('./a/@href')->to_literal_list;
    $url = URI::URL->new($url, $base_url)->abs;
    push @designs, {
        name      => $name,
        designer  => $designer,
        url       => "$url",
    };
}

say to_json(\@designs, {pretty => 1});
```

Output:

```
[
   {
      "designer" : "Andrew Lohman",
      "url" : "http://csszengarden.com/221/",
      "name" : "Mid Century Modern"
   },
   {
      "name" : "Garments",
      "url" : "http://csszengarden.com/220/",
      "designer" : "Dan Mall"
   },
   {
      "name" : "Steel",
      "designer" : "Steffen Knoeller",
      "url" : "http://csszengarden.com/219/"
```

```
    },
    {
       "designer" : "Trent Walton",
       "url" : "http://csszengarden.com/218/",
       "name" : "Apothecary"
    },
    {
       "name" : "Screen Filler",
       "designer" : "Elliot Jay Stocks",
       "url" : "http://csszengarden.com/217/"
    },
    {
       "name" : "Fountain Kiss",
       "designer" : "Jeremy Carlson",
       "url" : "http://csszengarden.com/216/"
    },
    {
       "name" : "A Robot Named Jimmy",
       "designer" : "meltmedia",
       "url" : "http://csszengarden.com/215/"
    },
    {
       "name" : "Verde Moderna",
       "designer" : "Dave Shea",
       "url" : "http://csszengarden.com/214/"
    }
]
```

In both these examples we were fortunate to be dealing with 'semantic markup' – where sections of the document could be readily identified using `id` attributes. If there were no `id` attributes, we could change the XPath expression to select using element text content instead:

```
my $xpath = '//h3[contains(.,"Select a Design")]/..//li';
```

This XPath expression first looks for an `<h3>` element that contains the text `'Select a Design'`. It then uses `/..` to find that element's parent (a `<div>` in the example document) and then uses `//li` to find all `<li>` elements contained within the parent.

Another common problem is finding that although your XPath expressions do match the content you want, they also match content you don't want – for example from a block of navigation links. In these cases you might identify a block of uninteresting content using `findnodes()` and then use `removeChild()` to remove that whole section from the DOM before running your main XPath query. Because you're only removing the nodes from the in-memory copy of the document, the original source remains unchanged. This technique is used in the `spell-check script` used to find typos in this document.

Matching class names

An HTML element can have multiple classes applied to it by using a space-separated list in the `class` attribute. Some care is needed to ensure your XPath expressions always match one whole class name from the list. For example, if you were trying to match `<li>` elements with the class `member`, you might try something like:

```
$xpath = '//li[contains(@class, "member")]';
```

which will match an element like this:

```
    <li class="member">Catherine Trenton</li>
```

but it will also match an element like this:

```
    <li class="non-member">Daniel Ifflehirst</li>
```

The most common way to solve the problem is to add an extra space to the beginning and the end of the `class` attribute value like this: `concat(" ", @class, " ")` and then add spaces around the classname we're looking for: `' member '`. Giving a expression like this:

```
$xpath = '//li[contains(concat(" ", @class, " "), " member ")]';
```

Using CSS-style selectors

The XPath expression in the last example is an effective way to select elements by class name, but the syntax is very unwieldy compared to CSS selectors. For example, the CSS selector to match elements with the class name `member` would simply be: `.member`

Wouldn't it be great if there was a way to provide a CSS selector and have it converted into an XPath expression that you could pass to `findnodes()`? Well it turns out that's exactly what the HTML::Selector::XPath module does:

```
use HTML::Selector::XPath qw(selector_to_xpath);

sub find_by_css {
    my($dom, $selector) = @_;
    my $xpath = selector_to_xpath($selector);
    return $dom->findnodes($xpath);
}
```

Some example inputs ("Selector") and outputs ("XPath"):

```
Selector: #zen-supporting h3
XPath:    //*[@id='zen-supporting']//h3

Selector: .designer-name
XPath:    //*[contains(concat(' ', normalize-space(@class), ' '), ' designer-name ')]

Selector: .preamble abbr
XPath:    //*[contains(concat(' ', normalize-space(@class), ' '), ' preamble ')]//abbr

Selector: .preamble h3, .requirements h3
XPath:    //*[contains(concat(' ', normalize-space(@class), ' '), ' preamble ')]//h3 | //*[contains(
```

CHAPTER

SEVEN

INSTALLING XML::LIBXML

You *can* install the XML::LibXML module using standard tools like cpanm, but there are a couple of factors to consider first. Because the module wraps a C library, to install this way you must have a C compiler installed and you must have already installed the `libxml2` library along with its development header files.

Note: Since version 2.0200, the XML::LibXML distribution uses a dependency on Alien::Libxml2 to install the `libxml2` library if your system does not already have it. So if the easier install options listed below are not suitable for your use case, you may be able to just use the normal CPAN install process:

```
cpan install XML::LibXML
```

There may be easier install options for your platform.

Installing on Windows

Strawberry Perl

The most popular Perl distribution for Windows is Strawberry Perl, which happens to include XML::LibXML in the base Perl installer. So if you have Strawberry Perl, you already have XML::LibXML.

ActivePerl

Another popular Perl distribution for Windows is ActivePerl from ActiveState (who also package Perl for Mac OS X, Linux and Solaris). ActivePerl includes a tool called PPM (Perl Package Manager) for installing pre-built Perl modules. You can use the PPM graphical user interface to search for the XML::LibXML package then click to select and install it. A command-line interface is also available:

```
ppm install XML-LibXML
```

Installing on Linux

If you are using the system Perl binary, you can install a pre-compiled version of XML::LibXML and the underlying libxml2 library from your distribution's package archive.

On systems using dpkg/apt (Debian, Ubuntu, Mint, etc.):

```
sudo apt-get install libxml-libxml-perl
```

On systems using rpm/yum (RedHat, CentOS, Fedora, etc.):

```
sudo yum install "perl(XML::LibXML)"
```

Manual installation

If for some reason you want to compile and install a version of XML::LibXML directly from CPAN, you must first install both the `libxml2` library and the header files for linking against the library. The easiest way to do this is to use your distribution's packages. For example on Debian:

```
sudo apt-get install libxml2 libxml2-dev
```

You can test that the library is correctly installed and your PATH is set up correctly with this command:

```
xml2-config --version
```

For more information about manual builds, refer to the README file in the XML::LibXML distribution.

Installing on Mac OS X

You can install the `libxml2` library using homebrew:

```
brew install libxml2
```

If you do not have Homebrew, you can install it at the homebrew website.

Once you have the `libxml2` library installed, you can install the XML::LibXML Perl module using standard tools such as `cpan` or `cpanm`.

CHAPTER

EIGHT

ALTERNATE FORMATS

The primary target for this project is the set of HTML pages. Alternate formats are available but may be missing some elements or features which are present in the HTML:

- `.pdf version`
- `.epub version`

CHAPTER

NINE

CORRECTIONS AND UPDATES

If you spot errors in the text of this document, please raise an issue on GitHub. You are also welcome to fork the project, commit a fix and raise a pull request.

If you find this document useful please link to it from your blogs, tweets, Stack Overflow answers etc. The canonical URL for linking is http://grantm.github.io/perl-libxml-by-example/.

CHAPTER

TEN

CONTRIBUTORS

In alphabetical order:

- Brandon Youngdale
- Grant McLean