

Chapter 8

Water Import: Converting Data to XML

IN THIS CHAPTER

- ◆ Converting HTML into XML Objects
- ◆ Converting CSV into XML Objects
- ◆ Converting Packed Strings into XML Objects
- ◆ Converting XML 1.0 into XML Objects

THE VAST MAJORITY OF DATA is not in XML. Water Import can convert many data sources into XML objects that can be directly manipulated by Water. Each type of data format has a conversion method that takes a string and returns an XML Object. XML 1.0 has no consistent format for encoding data—there are multiple ways to encode the same information. ConciseXML provides a single, consistent way to encode data, and Water Import provides the tools for converting multiple XML 1.0 formats into ConciseXML representation.

Importing non-XML Formatted Data

A lot of data files are stored in simple-structured text formats such as CSV (Comma Separated Values). These files typically hold the same type of data that a relational database tables holds—a collection of records. In Water, a collection is called a `vector` object and a record is called an `instance` object. Water Import uses the `to_objects` method for creating a vector of instances.

Converting Delimited Strings to XML Objects

The following example sets a variable named `data_string` to a string with two non-empty lines containing some data for two people.

```
<set data_string=  
  <string>  
Mike,Plusch,Wellesley,MA  
Christopher,Fry,Lexington,MA
```

```

    </string>
  />
data_string.<to_objects/>

```

Output in ConciseXML:

```

<vector
  <vector "Mike" "Plusch" "Wellesley" "MA"/>
  <vector "Christopher" "Fry" "Lexington" "MA"/>
/>

```

By default, `to_objects` assumes that each row (instance) is separated by a line break and each field within an instance is separated by a comma. `to_objects` returns a vector containing a vector for each row. The vector for each row contains a value for each field in the row.

After the string has been converted to a Water object, `Water Path` can be used to retrieve data. The following example shows how to retrieve a vector containing the first character of the last name for every record.

```
data_string.<to_objects/>.<for_each returns='all'> value.1.0 </>
```

Output in ConciseXML:

```
<vector <char "P"/> <char "F"/>/>
```

You can change the default separators for `to_objects`. Structured data text files typically have one kind of separator between fields of an object, and another kind of separator between instances in the vector.

```
"Mike|Plusch|Wellesley|MA**Christopher|Fry|Lexington|MA".
  <to_objects object_separator="**" field_separator="|"/>
```

Output in ConciseXML:

```

<vector
  <vector "Mike" "Plusch" "Wellesley" "MA"/>
  <vector "Christopher" "Fry" "Lexington" "MA"/>
/>

```

Creating Instances from a string

In most cases, a row of a data file represents an instance of some class. `data_string`, for example, looks to be a list of people with address information. Returning a vector of `person` instances would be much easier to understand and manipulate compared to a two-dimensional vector. One advantage of returning instances rather than a vector is that data from each instance can be retrieved by using the named keys.

```

<set data_string=
  <string>

```

```

Mike,Plusch,Wellesley,MA
Christopher,Fry,Lexington,MA
  </string>
/>
<defclass person first_name last_name/>
data_string.<to_objects maker=person/>

```

Output in ConciseXML:

```

<vector
  <person first_name="Mike" last_name="Plusch" 0="Wellesley" 1="MA" />
  <person first_name="Christopher" last_name="Fry" 0="Lexington" 1="MA" />
/>

```

The next example is the same as a previous example, except that named fields can be used to search objects.

```

data_string.<to_objects/>.<for_each returns='all'> value.last_name.0 </>

```

Importing CSV Files

Converting CSV strings into Water objects is not quite as simple as splitting rows by newlines and splitting fields by commas. Quotes may or may not surround a field value, and commas and newlines may exist in a quoted value. Because CSV files are so common, Water Import has a special `csv_to_objects` method that is just like `to_objects`, but understands all the special quoting rules of CSV files. For example, strings are quoted if they contain a quote or a newline character and any quote characters within the data are escaped.

```

<set data_string=
  <string>
  "Mike",Plusch,"Welles
ley,MA"
,Fry,"Lexington,MA"
  </string>
/>
data_string.<csv_to_objects/>

```

Output in ConciseXML:

```

<vector
  <vector "Mike" "Plusch" "Welles\nley,MA" />
  <vector "" "Fry" "Lexington,MA" />
/>

```

Importing Packed String Format

Fixed column (also known as packed strings or SDF) are very common file formats. The `to_objects` method from Water Import has standard support for this format.

The `field_separator` parameter for `to_objects` takes a vector with a pair of start and end character positions for each field value.

```
<set fixed_column_data=
  <string>
Mike Plusch Wellesley MA
ChristopherFry Lexington MA
  </string>
/>
fixed_column_data.<to_objects field_separator=<vector 0 11 11 20 20
31 31 33/>
                                trim_fields=true
                                />
```

Output in ConciseXML:

```
<vector
  <vector "Mike" "Plusch" "Wellesley" "MA"/>
  <vector "Christopher" "Fry" "Lexington" "MA"/>
/>
```

A vector of character positions is the most general way to represent how to cut up a string. But if you have a long list of field widths, you could use that list to automatically calculate the start and end positions for every value. The following example is a method that takes a list of field widths and returns a vector of start and end positions.

```
<defmethod field_widths_to_start_end_pairs
  positions=required=other_unkeyed_args>
  <set P=0/> <!-- P=character position -->
  <set R=<vector/> /> <!-- R=result vector -->
  positions.<for_each>
    R.<insert P
      <do> <set P=P.<plus value/> />
        P
      </do>
    />
  </for_each>
  R
</defmethod>
<field_widths_to_start_end_pairs 11 9 11 2/>
```

Output in ConciseXML:

```
<vector 0 11 11 20 20 31 31 33/>
```

Now you can call the method to calculate the value for `field_separator`.

```
fixed_column_data.
<to_objects field_separator=<field_widths_to_start_end_pairs 11 9 11 2/>
    trim_fields=true
/>
```

Importing XML Formatted Data

A growing amount of data is available in XML syntax. Water Import provides methods for converting both ConciseXML and XML 1.0 formatted strings into Water objects.

Converting a ConciseXML String to a Water Object

The first example shows how Water's `execute` method can convert a string of ConciseXML into a Water object.

```
"<vector> 'water' </vector>".<execute/>
```

Output in ConciseXML:

```
<vector "water" />
```

The string represented a `vector` of one string value `'water'`. The call to `execute` parsed the string and executed it, which returned a `vector` containing a string `"water"`. The single and double quotes have exactly the same meaning.

The string could contain textual representations for any type of objects. The next example shows the conversion a decimal number (`10.0`), a string (`'water'`), a nested vector (`<vector 5/>`), and a character (`<char '%'/>`).

```
"<vector> 10.0 'water' <vector 5/> <char '%'/> </vector>".<execute/>
```

Output in ConciseXML:

```
<vector 10.0 "water" <vector 5/> <char '%'/> />
```

Many examples in this chapter often start with a string, but that string could come from the content of a file.

```
<file "C:/test.txt"/>.content → "<vector> 'water' </vector>"
```

The first example could be rewritten as

```
<file "C:/test.txt"/>.<execute/> → <vector> "water" </vector>
```

`execute` executes everything in the string, but, by default, returns only the last value.

```
"5 10 15".<execute/> → 15
```

`execute` can return a vector of all the values if `execute` is passed the argument `returns='all'`. This is useful for ConciseXML data files containing multiple top-level expressions.

```
"5 10 15".<execute returns='all' /> → <vector 5 10 15 />
```

An XML 1.0 file by definition can only contain a single top-level element, so `returns='all'` is not unnecessary.

Converting XHTML to XML Objects

Every HTML page represents a Water object because Water Hypertext has definitions for all the HTML tags. The following example shows that executing a string of XHTML will return a Water Hypertext object with exactly the same structure.

```
"<TABLE>
  <TR> <TD>one</TD>
    <TD>two</TD>
  </TR>
</TABLE>".<execute />
```

Output in ConciseXML:

```
<TABLE>
  <TR>
    <TD>one</TD>
    <TD>two</TD>
  </TR>
</TABLE>
```

Converting Poorly Structured HTML

Most HTML pages on the Web are not well-formed XML documents. Water has a conversion method named `html_to_xhtml` that converts a string of poorly formed HTML into a valid string of XHTML.

```
"<table><tr><td>one".<html_to_xhtml />
```

Output in ConciseXML:

```
"<TABLE>
  <TR>
    <TD>one</TD>
  </TR>
</TABLE>"
```

Converting HTML into Data

A `hypertext.TABLE` object can be converted into a vector of vectors by using `to_objects`. An HTML table that shows structured data usually represents a vector of objects where each row represents an instance. By default, `to_objects` on a `TABLE` object will return a two-dimension vector where each row is a vector.

```
<TABLE>
  <TR>
    <TD>row 0, field 0</TD> <TD>row 0, field 1</TD>
    <TD>row 1, field 0</TD> <TD>row 1, field 1</TD>
  </TR>
</TABLE>.<to_objects/>
```

Output in ConciseXML:

```
<vector
  <vector "row 0, field 0" "row 0, field 1"/>
  <vector "row 1, field 0" "row 1, field 1"/>
/>
```

If a table represents a vector of objects, `to_objects` can convert the table into a vector of objects. table below is converted into a vector of person by passing `class=person` to `to_objects`.

```
<defclass person first_name last_name/>
<TABLE>
  <TR>
    <TD>Mike</TD> <TD>Plusch</TD>
    <TD>Christopher</TD> <TD>Fry</TD>
  </TR>
</TABLE>.<to_objects class=person/>
```

Output in ConciseXML:

```
<vector
  <person first_name="Mike" last_name="Plusch"/>
  <person first_name="Christopher" last_name="Fry"/>
/>
```

Extracting data from HTML pages can become quite sophisticated. Clear Methods (www.clearmethods.com) sells Water-based screen-scraping frameworks that provide a very flexible and robust way to extract objects from HTML pages.[CT1]

Converting XML 1.0 to ConciseXML

Although XML is a standard syntax, XML 1.0 does not specify how objects and fields should be represented in XML. As a consequence, each standard that uses XML syntax may encode objects in different ways.

The following example shows a person class and an instance of person in ConciseXML:

```
<defclass person first zip/>
<person first="Mike" zip=22222/>
```

The following example shows one particular XML 1.0 format for the instance in the preceding example.

```
<person>
  <first>Mike</first>
  <zip>22222</zip>
</person>
```

Executing the preceding example will return an error because Water expects `first` and `zip` to be objects. The following example will convert the XML 1.0 fragment into a normalized Water object. First call `execute` on the string of XML 1.0 to parse it into a data object, then use the `to_object` method to convert it into a normalized Water object.

```
"<person>
  <first>Mike</first>
  <zip>22222</zip>
</person>".
<execute execution_kind="ek_data"/>.
<to_object/>
```

Output in ConciseXML:

```
<person first="Mike" zip=22222/>
```

The following example shows another possible XML 1.0 representation for the same instance.

```
<person>
  <attr name='zip'>22222</attr>
  <attr name='first'>Mike</attr>
</person>
```

Executing the preceding example will error because Water expects that `attr` is an object to call. To convert it into a Water object, call `execute` with an `execution_kind` of `'ek_data'`, then call `to_object` with an argument of an `xml_format` class that specifies the rules in which to convert the XML 1.0 string.

```
"<person>
  <attr name='zip'>22222</attr>
  <attr name='first'>Mike</attr>
```

```
</person>".  
<execute execution_kind="ek_data"/>.  
<to_object xml_format.attr_name/>
```

Output in ConciseXML:

```
<person first="Mike" zip=22222/>
```

See www.waterlang.org for how to modify and create custom `xml_format` classes for handling all variations of XML 1.0 formats. The following example is yet another XML 1.0 format for the same instance.

```
<person>  
  <person-param>  
    <param-name>first</param-name>  
    <param-value>Mike</param-value>  
  </person-param>  
  <person-param>  
    <param-name>zip</param-name>  
    <param-value>22222</param-value>  
  </person-param>  
</person>
```

If the preceding XML 1.0 was a string in the variable `xml_string`, then the following example would convert it into a Water object.

```
xml_string.<execute execution_kind="ekdata"/>.  
<to_object xml_format.type_param_name_value/>
```

Output in ConciseXML:

```
<person zip=22222 first="Mike"/>
```

Each representation may appear straightforward, but the notion that there are many representations for exactly the same instance causes severe problems when trying to interpret the exact meaning of the data. Objects, fields, field keys, field values, and field types are all encoded differently. This is one of the primary reasons why XML appears easy on the surface, but is actually quite complex when you try to use it for data. If you have been using XML outside of Water, you may have heard the phrase, “attributes versus elements.” This debate centers on whether data should use XML attributes or XML elements. There is no right answer and Water was designed to make the entire question irrelevant.

A single, unambiguous representation is required that can serve as the “normal” form. Water uses ConciseXML for that representation. The preceding examples showed how to take several different XML 1.0 formats and create Water objects.

Summary

Water Import is a powerful tool for converting XML and non-XML data into Water objects. Water Import is a crucial tool for normalizing the many XML 1.0 data formats currently being used in various standards. The Water objects can be manipulated in Water and then formatted in many ways using Water Export, described in Chapter 9.