

## Chapter 6

# Water Path: Accessing Data – XPATH

### IN THIS CHAPTER

- ◆ Retrieving Nested Values
- ◆ Finding a Specific Value
- ◆ Returning Multiple Matches
- ◆ Filtering by Key and Value
- ◆ Understanding Water Paths and URIs

THE PREVIOUS CHAPTERS COVERED how to create classes and instances. To effectively use those objects, you need a way to retrieve data from an object. Water Path is the standard method in Water used to flexibly retrieve any parts from an object. XPath is an alternative approach for accessing parts of an XML document, but Water Path is simpler and more flexible due to its consistent object-oriented approach.

## Retrieving Values

To show how Water Path can access XML data, I created a Water object to use in all the examples.

The variable named `a_starting_object` contains an instance of `thing`.

```
<set a_starting_object=
  <thing a_field=<thing foo="bar"/>
    b_field=100
    0=<vector "water" "ice"/>
    1=<thing x=5 y=10 foo="bar"/>
  />
/>
```

The `a_starting_object` field holds an instance of `thing`, which has four fields: `a_field`, `b_field`, and two vector fields, `0` and `1`. The `a_field` has value `<thing foo="bar"/>`, and the `b_field` has value `100`. The first vector field contains the

`vector <vector "water" "ice"/>` and the second vector field contains `<thing x=5 y=10 foo="bar"/>`. This object will be used throughout the examples in this chapter.

## Retrieving the Starting Object

In Water Path, the starting object can be retrieved by using the name of the field or local variable that holds the object. In the preceding example, the starting object has the name `a_starting_object`.

```
a_starting_object
```

### Output in ConciseXML:

```
<thing a_field=<thing foo="bar"/> b_field=100
  0=<vector "water" "ice"/> 1=<thing x=5 y=10 foo="bar"/>
/>
```

Executing the name of a field or variable will return the object contained in that field or variable.

## To Retrieve the Value of a Field

To retrieve the value of a field in the starting object, use the name of the starting object followed by a dot (`.`), and then followed by the key of the field to retrieve.

```
a_starting_object.a_field → <thing foo="bar"/>
```

The same technique can be used if the key of the field is an integer.

```
a_starting_object.0 → <vector "water" "ice"/>
```

## Quoting Fields

You can optionally use quotes around the field key. This is useful if the key of the field contains spaces or other special characters. This is also useful if the key is the string `"0"`, not the integer `0`.

```
a_starting_object."a_field" → <thing foo="bar"/>
```

The following code produces an error because `a_starting_object` does not have a field with key `"0"`; it only has a field with integer key `0`:

```
starting_object."0"
```

## Retrieving a Nested Field Value

For each nested object, add another dot and the key of the field.

```
a_starting_object.a_field.foo → "bar"
```

Long paths may be split over multiple lines. No special characters are required to continue the line, but the dot must end the last part on a line – not begin the part on the next line.

```
starting_object.  
  a_field.  
    foo
```

**Output in ConciseXML:**

```
"bar"
```

## Using a Variable for the Key of a Field

The field key can be calculated. The following example shows how the `get` method can be used to retrieve a value when the key of the field is in a variable:

```
<set the_key="a_field"/>  
starting_object.<get the_key/>
```

**Output in ConciseXML:**

```
<thing foo="bar"/>
```

The first line above sets a local variable named `the_key` to the value `"a_field"`. The second line calls `get` on `a_starting_object`. The argument to the call to `get` is the local variable `the_key` with value `"a_field"`. This call returns the value of the field with key `"a_field"` in object `a_starting_object`.

Calls to `get` can be made anywhere within a path.

```
starting_object.<get the_key/>.foo → "bar"
```

In this example, the call to `get` returns `<thing foo="bar"/>`. The next part in the path is `foo`, which will return `"bar"`, the value of the `foo` field.

Multiple calls to `get` can be made within a path.

```
starting_object.<get the_key/>.<get "foo"/> → "bar"
```

The ending call to `get` in the preceding path, `<get "foo"/>`, is similar to `.foo` in the previous example.

### Looking Up the Object Hierarchy

If the key is not found when looking up a key in an object, then the field is looked up in the next higher level in the object hierarchy, which is the `_parent` field of the object. Lookup continues until it reaches the top-level `thing` object. An error occurs if the field is not found.

If you do not want the lookup to continue up the object hierarchy, then use `get`. This is because `get`, by default, has `lookup=false`.

```
starting_object.<get "a_field" lookup=false/>
```

`lookup=false` is optional, but `lookup` is `false` by default.

`get` is similar to `dot`, but, by default, `get` will not error if the field is not found—it returns `null` by default. If you want an error to occur when a field is not found, then add `if_missing='error'` to the `get` call. Chapter 17 covers how to catch errors.

```
starting_object.<get "xxx" if_missing='error'/>
```

By default, `if_missing` has the value `'return'`. This means that if the field is missing, then the call will return the default value. The default value is `null` by default. To change the default value, pass in an argument named `default` to the `get` call that has the new default value. In the following example, that default value is `"field not found"`:

```
starting_object.<get "xxx" default="field not found"/> →  
"field not found"
```

## Retrieving an Object Having a Field with a Specific Value

In some cases, you may not know the exact path to retrieve an object, so Water Path can look for a nested object having a specific key-value pair.

```
a_starting_object.<get_with_value "foo" "bar"/> → <thing foo="bar"/>
```

The `starting_object` value is repeated as follows for reference:

```
<set a_starting_object=  
  <thing a_field=<thing foo="bar"/>  
    b_field=100  
    0=<vector "water" "ice"/>  
    1=<thing x=5 y=10 foo="bar"/>  
/>  
/>
```

Given any object, `get_with_value` will descend through all the values of the fields in the `starting_object` and return the first object found that has a field key that is equal to the first argument "foo" and a field value that is equal to the second argument "bar".

In the preceding example, the call returned the object `<thing foo="bar"/>`. `get_with_value` also works for hypertext objects. The following example defines a hypertext table object:

```
<set a_hypertext_object=
  <TABLE>
    <TR> <TD><input name="age" value="20"/></TD>
      <TD><input name="height" value="6"/></TD>
    </TR>
  </TABLE>
/>
```

The following example shows how to retrieve an object that has a `name` field with the value "age":

```
a_hypertext_object.<get_with_value "name" "age"/> → <input name="age"
value="20"/>
```

## Returning the Last Matching Object

By default, `get_with_value` returns the first matching object. The `returns` parameter of `get_with_value` can also be the value "last", which will return the last object whose field named "foo" has the value "bar".

```
a_starting_object.<get_with_value "foo" "bar" returns='last'/>
```

**Output in ConciseXML:**

```
<thing x=5 y=10 foo="bar"/>
```

## Returning All Matching Objects

`get_with_value` can take `returns='all'`, which will return all objects from the starting object whose field has that value.

```
starting_object.<get_with_value "foo" "bar" returns='all'/>
```

**Output in ConciseXML:**

```
<vector
  <thing foo="bar"/>
  <thing x=5 y=10 foo="bar"/>
/>
```



Water Path and the `get_with_value` method will also work with **circular data structures** where the same object is the value for multiple fields.

## Filtering

Water Path can look for a particular pattern in the fields and return a filtered collection of values. The `for_each` method is used to perform the filtering.



Chapter 17 on Water Flow covers the `for_each` method in more detail.

## Filtering by Field Keys

Typically, you want to return the value of a field, which is an object. You already have seen how to return the value of a specific field of an object by using the path notation:

```
starting_object.a_field.foo
```

The `a_starting_object` value is repeated as follows for reference:

```
<set a_starting_object=
  <thing a_field=<thing foo="bar"/>
    b_field=100
    0=<vector "water" "ice"/>
    1=<thing x=5 y=10 foo="bar"/>
  />
/>
```

In some cases, you may need to return the values from a subset of fields in the object instead of the entire object. For example, you might want to return just the values in the vector fields.

Vector fields have integer keys starting at 0.

In the following example, `for_each` is used to return only the vector fields of `a_starting_object` – the values in fields 0 and 1:

```
starting_object.<for_each include_key=vector_key returns='all'> value </for_each>
```

**Output in ConciseXML:**

```
<vector
  <vector "water" "ice"/>
  <thing x=5 y=10 foo="bar"/>
/>
```

The call to `for_each` iterates through the fields of `a_starting_object`. A field will be included only if it has a `vector_key`. A vector will be returned where the values of the vector are the values of each of the included fields.

The `returns='all'` argument is required to return all the values of all the matching fields in the object. `for_each` is described in detail in Chapter 17.

The following example returns all four fields because the default value of the `include_key` parameter of `for_each` is `regular_key`. A `regular_key` is a key that is either an integer or a string.

```
a_starting_object.<for_each returns='all'> value </for_each>
```

To return the values in fields with string keys, set the value of the `include_key` argument to `string_key`.

```
starting_object.<for_each include_key=string_key returns='all'> value </for_each>
```

**Output in ConciseXML:**

```
<vector
  <thing foo="bar"/>
  100
/>
```

The following example returns a vector of the values for all fields in the object because `for_each` has `regular_key` for the `include_key` argument:

```
starting_object.<for_each include_key=regular_key returns='all'>
value </for_each>
```

**Output in ConciseXML:**

```
<vector <thing foo="bar"/>
  100
  <vector "water" "ice"/>
  <thing x=5 y=10 foo="bar"/>
/>
```

The following example uses the `include_value` argument to filter fields based on their value:

```
<defmethod foo_starts_with_ba>
  .foo.<starts_with "ba"/>
```

```

</defmethod>
starting_object.<for_each include_value=foo_starts_with_ba
returns='all'>
  value
</for_each>

```

**Output in ConciseXML:**

```

<vector <thing foo="bar"/>
  100
  <vector "water" "ice"/>
  <thing x=5 y=10 foo="bar"/>
/>

```

## Filtering by Explicitly Named Fields

To return the values for a set of specific fields, list the field keys in the `include_key` argument. This can be accomplished by making a vector containing the field keys.

```

starting_object.
  <for_each include_key=<vector "a_field" 1/> returns='all'> value </for_each>

```

**Output in ConciseXML:**

```

<vector
  <thing foo="bar"/>
  <thing x=5 y=10 foo="bar"/>
/>

```

In the preceding example, the values for the `a_field` and `1` fields are returned in a vector.

To skip specific fields, list the keys to skip in the `exclude_key` argument.

```

starting_object.
  <for_each exclude_key=<vector "a_field"/> returns='all'> value </for_each>

```

**Output in ConciseXML:**

```

<vector
  <thing x=5 y=10 foo="bar"/>
/>

```

In the preceding example, only the fields that are not listed in `exclude_key` will be returned.

You can define a method that filters fields based on any criteria. To use the filter, put the name of the filter method in the `include` argument.

```

<defmethod field_ends_with_field>
  _subject.<ends_with "field"/>
</defmethod>

```

```
a_starting_object.<for_each include=field_ends_with_field
returns='all'> value </for_each>
```

**Output in ConciseXML:**

```
<vector
  <thing foo="bar"/>
  100
/>
```

The method `field_ends_with_field` only matches field keys that end with the string "field". In the preceding example, it returns the values of `a_field` and `b_field` in a vector.

The `a_starting_object` value is repeated as follows for reference:

```
<set a_starting_object=
  <thing a_field=<thing foo="bar"/>
    b_field=100
    0=<vector "water" "ice"/>
    1=<thing x=5 y=10 foo="bar"/>
  />
/>
```

## Filtering by Field Value

To filter values based on some criteria, use the `include_value` argument in `for_each` and use an `if` expression in the content of `for_each` that will test the value on some criteria. The following example looks to see if the value has a length more than 1:

```
a_starting_object.<for_each include_value=<defmethod> .<is_a vector/> </>
returns='all'>
  value
</for_each>
```

**Output in ConciseXML:**

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

Instead of putting the filter within the path, you can define a method to perform the filtering, and then call the method from anywhere within the path. In the following example, the `vector_fields_with_vector_values` method is defined by using the code from the preceding example:

```
<defmethod vector_fields_with_vector_values>
  .<for_each include_value=<defmethod> .<is_a vector/> </> returns='all'>
    value
  </for_each>
</>
```

Now the method can be called from within a path.

```
a_starting_object.<vector_fields_with_vector_values/>
```

**Output in ConciseXML:**

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

## Water Paths and URIs

Water Paths can be converted to URIs by replacing the dots with slashes. The following example shows a Water Path and the equivalent URI path:

```
a_starting_object.a_field.foo
```

**Equivalent URI path:**

```
"a_starting_object/a_field/foo"
```

The following code is another example showing a Water Path containing numbers and the equivalent URI path:

```
a_starting_object.1.0
```

**Equivalent URI path:**

```
"a_starting_object/1/0"
```

Water Path expressions that call a method can also be converted to a URI.

```
a_starting_object.<vector_fields_with_vector_values/>
```

**Equivalent URI path:**

```
"a_starting_object/vector_fields_with_vector_values?"
```

This feature will be used in other chapters when a URI is typed into a Web browser to retrieve an XML object. Methods for converting a Water expression to a URI and vice-versa will be covered in Chapter 10.

## Summary

Water Path is very simple and elegant, yet provides a high degree of flexibility when retrieving data from an XML object. You learned multiple ways to retrieve nested data from an object including how to return the first match or every match. Water Path supports filtering by the key or value of a field, and every Water Path has a corresponding URI. The next chapter on Water Transform uses Water Paths throughout the examples.