

Chapter 4

Water Type: Creating Custom Types – XSD

IN THIS CHAPTER

- ◆ Understanding Water's primitive types and classes
- ◆ Using a type for specifying constraints on a field
- ◆ Creating new custom types
- ◆ Understanding the difference between a type and a class

A *TYPE* IS A DESCRIPTION FOR CATEGORIZING VALUES. Water Type defines a set of core classes and types that represent common values. A type is an object which can be used to describe the possible values in a field. Water Type also defines the way in which new types can be created.

Introducing Water Type

Water Type defines primitive classes and objects as several non-primitive classes. The three primitive objects are `true`, `false`, and `null`. The four classes that have instances that are primitive objects are `number`, `boolean`, `char`, and `string`. Primitive objects form the base for constructing all other objects. Primitive objects have a `_parent` field and no other fields, while non-primitive objects can have other fields. In every other respect, primitive objects share the same properties as every other object.

Water Type defines several non-primitive objects including `hypertext`, `vector`, `thing`, `date`, `time`, `datetime`, `datetime_interval`, and `duration`. These types are covered in other chapters. The chapter on Water Hypertext and XHTML describes `hypertext`; the chapter on Water Object and Object Oriented XML describes `vector` and `thing`; and the chapter on Water Date and Time describes `date`, `time`, `datetime`, `datetime_interval`, and `duration`.

number

Numbers have unlimited precision and can be positive or negative. A number can be an integer such as `-42` or a decimal number such as `3.14159` or `0.125`.

Decimals must always have at least one digit to the left of the decimal point (.). For example, 0.125 and -0.125 are decimal numbers, while .125 or -.125 are not valid representations for decimal numbers.

string

You can create strings by surrounding text with double quotes (" ") or single quotes (' '). To include special characters such as the end-of-line character (`\n`), use a back slash. Strings can span multiple lines. Any new lines within the string are treated as newline characters in the string.

Strings may also be created using `<string>Water</>` or `<string>Water</string>`. Strings with the same characters are actually the same object. `"water".<is "water"/>` returns `true`.

Officially, a string is not a true primitive object because it could be represented as a vector of characters.

char

Characters are usually parts of strings and are not commonly used as individual objects. Characters are created by using `char`.

```
<char "a"/>
<char 'A'/>
<char '\n'/> <!-- newline -->
<char "\t"/> <!-- tab -->
<char " "/> <!-- space character -->
<char "'"/> <!-- single quote character -->
<char '"'/> <!-- double quote character -->
```

null

`null` is an object that is commonly used for a field's value to indicate that the field is empty.

Specifying Field Types

The following line shows how the `integer` type can be used to define the type of the `x` field in an instance of `thing`:

```
<thing x=5=integer/>
```

Water uses the `key=value` notation for defining fields. Water extends that notation with a third position for the type of field. Specifying a type for a field is

optional, but when the type is given, it should appear in the third position of a field where the first and second positions are the key and value for the field. Every field follows the form of `key=value=type`.



A *type* is a classification for a value. A type can constrain the permissible objects that can be put into the value of a field.

The following code shows a `thing` with three fields. The first field has a value of "Mike" and a field type of `string`. The value of the `first_name` field must always be a type of `string`. The `start_date` field has a field type of `date` and a value of `<date 2002 10 15/>`. The value of the `over_18` field must be either `true` or `false` because it has a field type of `boolean`.

```
<thing
  first_name="Mike"=string
  start_date=<date 2002 10 15/>=date
  over_18=true=boolean
/>
```

Each field can have a key, value, and type. Each one is separated by an equal sign (=). The first field position before the first equal sign is the key of the field, the second field position after the first equal sign is the value of the field, and the third field position is the type of the field.

You can define a class with `defclass` and then use that class for specifying the type of a field.

```
<defclass author/>
<thing
  a_author=<author name="Mike"/>=author
/>
```

The preceding example defines a class `author`, which plays the role of a class as well the role of a type.

Creating Custom Types

Every object can be used in multiple roles. For example, one object can be used as a class, instance, type, key, and value. This is a very powerful concept that is described in detail in Chapter 22.



Chapter 22 describes Water's Multi-role Object System.

The list of types at the beginning of this chapter is also a list of built-in Water classes. Every Water class object can be used as a type. For example, `hypertext` is a class object for subclasses such as `BODY`, `FONT`, `INPUT`, and all other XHTML tags.

The following example shows an instance of `thing` with a field named `description`, a field value of `<P>hello</P>`, and a field type of `hypertext`. The value of the `description` field must also be of type `hypertext`. `P` is the paragraph tag for HTML, so it is of type `hypertext`. `INPUT` is also of type `hypertext`.

```
<thing description=<P>hello</P>=hypertext />
<thing description=<INPUT value="hello"/>=hypertext />
```

one_of

Water has methods for easily creating new types as well. For example, one common type is named `one_of` and it represents a choice between multiple values.

```
type.<one_of "red" "green" "blue"/>
```

The preceding code creates a new type that specifies that a value must be "red", "green", or "blue". The `one_of` type has a similar use as an enumeration in other languages.

```
<thing color="red"=type.<one_of "red" "green" "blue"/> />
```

The preceding example has an instance of the `one_of` as the type of the `color` field. The value of the `color` field must be a primary color— "red", "green", or "blue".

Note that a `one_of` type can be created by using either `type.one_of` or just `one_of`. They are identical because the `type.one_of` object is promoted by Water so that it is available without using the `type` prefix.

```
type.<one_of "red" "green" "blue"/>
<one_of "red" "green" "blue"/>
```

To reuse a type, name it with a variable or field.

```
<set primary_color=<one_of "red" "green" "blue"/> />
<thing color="red"=primary_color/>
```

The first line sets a variable named `primary_color` whose value is the type `<one_of "red" "green" "blue"/>`. Now, the type named `primary_color` can be used anywhere a type is required. The second line sets the value of the `color` field to "red" and the type of the field to `primary_color`. This specifies that the value of the `primary_color` field must be a primary color.

In all the previous examples, `one_of` had only string objects as values, but any object can be used. For example, to create a type that accepted any integer or the string "infinity", do the following:

```
<one_of integer "infinity"/>
```

The following type represents a value that is either a string or a hypertext object.

```
<one_of string hypertext/>
```

vector_of

In addition to `one_of`, Water has the type `vector_of` to specify the permissible values in a vector.

```
<thing primes=<vector 3 7 9/>=type.<vector_of integer/> />
```

The preceding example has a field `primes` that has a value of `<vector 3 7 9/>`. The type of the field is `<type.vector_of integer/>`. This instance of type says that all values in the vector must be of type `integer`.

`vector_of` can also take fields of `min` and `max` that specify the minimum and maximum number of values in the vector.

The following is the type for having a maximum of four values in a vector:

```
type.<vector_of max=4/>
```

To have a minimum of three values in the vector:

```
type.<vector_of min=3/>
```

To have three to five values in the vector:

```
type.<vector_of min=3 max=5/>
```

`vector_of` can be combined with `one_of`:

```
<set up_to_four_primary_colors=
  type.<vector_of <one_of "red" "green" "blue"/> max=4/>
/>
```

These objects satisfy the constraints of type `up_to_four_primary_colors`:

```
<vector "red" "red" "blue" "blue"/>
<vector "red" "red"/>
```

These objects do not satisfy the constraints of type `up_to_four_primary_colors`:

```
<vector "red" "red" "blue" "blue" "blue"/>
<vector "purple"/>
```

The first line has five values, when `max` is set to 4. The second line has "purple" as a value, but "purple" is not one of the primary colors ("red", "green", or "blue").

The type `vector_of` can also be used to define the constraints on the length and permissible characters in a string. In business applications, it is very common to have a minimum and maximum length for a field (such as the last name of a person), as well as a limit on the set of possible characters.

```
<set uppercase=type.<one_of <char "A"/> <char "B"/> <char "C"/> /> />
<thing last_name="AC"=type.<vector_of uppercase min=2 max=15/> /> />
```

range_of

The type `range_of` specifies that a number must be in a given numerical range. The `range_of` can be given a minimum value and/or a maximum value.

The following is a type that represents any number from 0 up to, but not including 10.

```
type.<range_of max=10/>
```

The following is a type that represents an integer from 0 up to and including 9.

```
type.<range_of max=10 number_type=integer/>
```

The following is a type that represents any number that is equal to or greater than 10:

```
type.<range_of min=10/>
```

The following is a type that represents any number from 0 up to and including 10:

```
type.<range_of max=10 include_max=true/>
```

The following is an instance of `thing` that has two fields each with a `range_of` type.

```
<thing
  age=25=type.<range_of min=21 max=120/>
  height=6.2=type.<range_of min=3 max=8/>
/>
```

not_of

A type called `not_of` says that a value can't be a specific type.

```
<thing priority_level="high"=type.<not_of "na"/> />
```

The `priority_level` value can't be the string "na".

```
<thing priority_level="high"=<one_of string type.<not_of "na"/> /> />
```

Here, `priority_level` can be any string except "na".

Using a Method as a Type

A method can be used as a type in two ways. First, a method can specify the type of method that is allowed in a field. Second, a method can be called on any object and return a boolean value.

When a method is used as a type of a field, it says that the value in the field must be a method, and that method must have an interface or contract that is compatible with the contract type. A Water method acts as a type because the method can describe the contract for an interface. A method that is the type of another method means that any valid call to the method will also be a valid call to the method used as a type.

The second use of a method as a type is a method that returns a boolean value that can be used to define a type. The following example defines a method called `carry_on_luggage` that has some logic that returns a boolean value.

```
<defmethod carry_on_luggage>
  this.girth.<less 40/>.<and this.weight.<less 25/> />
</defmethod>
```

The following example creates a type based on the preceding method.

```
type.<call_method_of carry_on_luggage />
```

The new type can be used for the field's type.

```
<thing a_bag=<thing girth=20 weight=10/>=type.<call_method_of carry_on_luggage /> />
```

Remember, using the `key=value=type` form for a field, `a_bag` is the name of the field's key, `<thing girth=20 weight=10/>` is the value of the field, and the field's type is `type.<call_method_of carry_on_luggage/>.carry_on_luggage`, which is a method that will return either `true` or `false`.

Now only values that are of type `carry_on_luggage` will be permitted in the `a_bag` field. For methods used as a type, you could explicitly call the method on a value. If the call returns `true`, the value is of that type.

```
<thing girth=20 weight=10/>.<carry_on_luggage/>
```

In this example, the `<thing girth=20 weight=10/>` is the value to test. `<carry_on_luggage/>` is a call to the method `carry_on_luggage`, and it is called on the value `<thing girth=20 weight=10/>`. If this expression returns `true`, then the value satisfies the type `type.<call_method_of carry_on_luggage/>`. This is similar to what happens when a field with a value is checked against a type where the type is a method object.

When an object needs to type check its field values, the method `is_type_for` is called on the field type, and the value to check is given as the argument. Take a look at the following two examples:

```
boolean.<is_type_for true/> → true
type.<one_of "red" "blue"/>.<is_type_for "blue"/> → true
person.<is_type_for <person name="Mike"/> /> → true
```

The first line asks if `boolean` is-the-type-for `true`. The argument to `is_type_for` is the value to test, and `is_type_for` is called on the object that is playing the role of a type. Any object can play the role of a type because any object can have a method defined on it called `is_type_for`.

Discussing Type versus Class

A `Water` object can be used as both a type and a class. An object plays the role of a class when instances are created from it. An object plays the role of a type when it is used as the type for a field. If an instance belongs to some class, it means that the instance can inherit fields from that class. If a field is not found in an instance, the field will be looked up in its parent class. The meaning of class is based on the class inheritance hierarchy and the `is_a` method. The meaning of a type is based on a method that defines “type-ness” for any object. The `is_type_for` method can be defined on any object. By default, the `is_a` and `is_type_for` methods have similar uses.

```
100.<is_a number/> → true
number.<is_type_for 100/> → true
```

If a class is used as a type, an instance will be the type of that class if it follows the class's contract.

When you define a class, you can use it as a type by default because it has a `is_type_for` method. You can override the `is_type_for` method on a given class to change the type definition for that class. This is useful when you need the type system to be different from the class inheritance hierarchy.

Types do not inherit from other types – there is no inheritance among types. Classes, however, can inherit from other classes.

Summary

In this chapter, you learned how to specify the type of a field. You also saw how any Water class can be used as a type – both Water standard classes as well as user-defined classes. The types `one_of`, `vector_of`, `range_of`, and `not_of` were methods to create custom types. The next chapter on Water Contract uses types to specify parameters and fields in classes.

