

Chapter 5

Water Contract: Specifying APIs – DTD and Schemas

IN THIS CHAPTER

- ◆ Building Class Contracts
- ◆ Constraining Fields using Types
- ◆ Building Method Contracts
- ◆ Constraining Parameters using Types
- ◆ Specifying the Return Type

A CONTRACT IS AN AGREEMENT THAT FORMALIZES a set of expectations between two or more parties. Water Contract takes the ideas behind a legal contract and applies them to software. Every Web service has an application programming interface (API) for calling that service. In the past, a service's API was described in a human-readable text document. With Web services, the API is described in a text document as well as in a machine-readable form. DTD and XML Schema are two examples of formal languages for describing the structure of an XML document.

Water Contract is a formal language for fully describing a data structure or API. It uses the ConciseXML syntax and Water Contract and is both easier to create and use than either DTD or XML Schema, as well as providing greater rigor and precision for describing an interface or data structure.

Defining a Contract with `defclass`

In Chapter 3, you created XML objects that were data objects. Most languages have some way to constrain the fields of an object as well as the values of any particular field. Languages use different terms to describe the object that defines these constraints. For example, SQL has table definitions, Java has classes, Cobol has record definitions, C has structs, and XML has schema languages such as W3C XML schema or DTD's (Document Type Definitions). Water Contract uses `defclass` to define constraints on data objects. A `defclass` represents a class, and instances created from that class must obey the constraints defined in `defclass`.

The following uses `defclass` to define a person class:

```
<defclass person name born_on/>
```

All instances of this class must supply values for the `name` and `born_on` fields, so both fields are required.

Required fields can be specified two ways:

- ◆ By just listing the name and not supplying any value.
- ◆ By explicitly putting `required` as the value of the field.

The following two lines are equivalent:

```
<defclass person name born_on/>  
<defclass person name=required born_on=required/>
```

Optional fields can be specified two ways:

- ◆ By giving a value to the field that is the field's default value.
- ◆ By using `optional` as the value of the field.

The following two classes both have two optional fields:

```
<defclass person name="" status="living"/>  
<defclass person name=optional status=optional/>
```

The first line has an optional `name` field with a default value of the empty string `""`. The `status` field has a default value of `"living"`. The second line uses `optional` for both fields to say that the fields are not required.

Constraining Field Values Using Types

The preceding section described how to specify the possible fields for an object by listing the fields in a call to `defclass`. `defclass` can also describe what values are permitted in specific fields. A constraint on a field uses Water Type, which was described in Chapter 4. You will now see how these types can be used in `defclass`.

The following example shows a person `defclass` with two fields: `fname` and `born_on`. `fname` has a required value and the value must be of type `string`. `born_on` has an optional value, but if a value is given, it must be of type `date`. Each field follows the `key=value=type` form as explained Chapter 4.

```
<defclass person
  fname=required=string
  born_on=optional=date
/>
```

If no type is specified for a field, it can hold any object; therefore, the default type is thing.

The following two lines have the identical meaning:

```
<defclass person fname=required      born_on=optional />
<defclass person fname=required=thing born_on=optional=thing />
```

The following example shows an instance of a book:

```
<book
  title="Water Book"
  a_author=<author fname="Mike" lname="Plusch"/>
  published_on=<date 2002 11 5/>
  chapters=<vector
    <chapter name="Introduction"/>
    <chapter name="Objects"/>
    <chapter name="Methods"/>
  />
/>
```

The following example uses Water Contract to describe the data structure of the previous book instance:

```
<defclass book
  title
  a_author
  published_on=required=date
  chapters=null=<vector_of chapter max=30 />
/>
<defclass author
  fname=required=string
  lname=required=string
/>
<defclass chapter
  name=required=<vector_of char 5 20/>
/>
```

As you can see, there are `defclass` calls for defining a book, author, and chapter. Each one represents a class of object that has one or more fields. Fields have positions for key, default value, and type. If a field has no default value, it is assumed to be

required. If a field has no type, it is assumed to be the most general type, `thing`. Notice that the `book` instance contained nested instances such as `author` and `chapter`, but the class definitions were not nested. That is because they do not logically inherit from each other. For example, `author` is not a type of `book`. If there was a `defclass` for `person`, `author` might inherit from it.



The types shown above were all described in Chapter 4 on Water Type.

Defining an inheritance hierarchy of classes is described in Chapter 22 on Water Object System.

Allowing Other Arguments

Water Contract supports the ability to accept other arguments that are not explicitly listed as fields or parameters. This flexibility is missing from most other languages and API specifications. The special `_promote` argument in `defclass` will assign unkeyed arguments to the vector fields of the object. Unkeyed arguments are arguments of a call that do not have an explicit key.

```
<defclass grocery_list _promote='other_unkeyed_args' />
<grocery_list
  "apple"
  "milk"
  "pears"
 />
```

The preceding code is similar to

```
<grocery_list
  0="apple"
  1="milk"
  2="pears"
 />
```

By convention, `_promote` is usually the last field listed in the `defclass`. The values can be any type of object. See the Water Contract reference on www.waterlang.org for descriptions of other values for `_promote` such as `'other_args'`, `'other_keyed_args'`, `'content'`, and `'content_and_other_args'`.

The following example defines two classes: `grocery_list` and `grocery_item`. `grocery_list` has a required `store_name` and required vector fields because of the `items` field. `grocery_item` has a required `name` and an optional `quantity` field with a default value of 1. The `'other_unkeyed_args'` value in the `items` argument

is a special construct that specifies that any other unkeyed arguments will be available in a field named `items`. The value of that field will always be a vector.

```
<defclass grocery_list store_name=required
                        items='other_unkeyed_args'
/>
<defclass grocery_item name=required quantity=1/>
```

The following code is an XML object that satisfies the previous two Water Contracts:

```
<grocery_list
  store_name="Acme Foods"
  "apple"
  <grocery_item name="milk"/>
  <grocery_item name="pears" quantity=3/>
/>
```

Defining Method Contracts

Water Contract uses `defmethod` for defining a method's contract or interface. Water methods can be called as a Web service.

A method is a Water object that has a contract and an implementation. A `defmethod` creates a method definition. The method definition specifies the contract and may also have an implementation. The contract is defined within the open tag of a call and the implementation is defined in the content argument in the call to `defmethod`.

```
<defmethod foo param1> "implementation" </defmethod>
```

In the preceding example, `foo` is the name of the method, `param1` is the method's contract, and `"implementation"` is the method's implementation.



A *method contract* is a specification or interface that unambiguously defines the inputs and outputs of a method. A fully specified contract will describe the inputs, the output, the assumptions about available resources used by the method (preconditions), and any effects that the method has external to it (postconditions).

Specifying Method Parameters

The input specification for a method is defined in the attributes of a `defmethod` call. The inputs to a method are called *parameters*.

The format for the inputs follows the exact format for fields of a `defclass`. Here is a `defclass`.

```
<defclass person
  fname=required=string
  born_on=optional=date
/>
```

If you converted the `defclass` to a `defmethod`, it would be

```
<defmethod person
  fname=required=string
  born_on=optional=date
/>
```

Notice that only the name `defclass` changed to `defmethod`. Everything else remained exactly the same except for the terminology. For example, the `fname` and `born_on` fields in `defclass` are fields of the `defclass`, but the `fname` and `born_on` fields in the `defmethod` are the input parameters to the method. Parameters are the fields of the `defmethod` object.

Everything that was described for typing the fields of a class using `defclass` is applicable for describing the types of the input parameters.

Allowing Other Keyed Arguments

A special argument value named `other_keyed_args` specifies that the method accepts any number of named arguments. For example, the second line in the following example shows a call to `foo` that passes in an argument `yak="yum"`. The contract for `foo`, described by the `defmethod` in line one, does not explicitly specify a `yak` parameter—it uses `other_keyed_args` to accept any argument that has an explicit key. The arguments will be available in the field that has `other_keyed_args`. Only one argument in a Water Contract can have `other_keyed_args`.

```
<defmethod foo misc=optional=other_keyed_args/>
<foo yak="yum"/>.misc
```

Output in ConciseXML:

```
<thing yak="yum"/>
```

Other keyed arguments may be promoted using the special `_promote` parameter. This means that each argument is a field of the instance, rather than one level down.

```
<defmethod foo _promote=optional=other_keyed_args/>
<foo yak="yum"/>
```

Output in ConciseXML:

```
<foo yak="yum"/>
```

For `defclass`, the `other_keyed_args` is promoted by default. This means that you can create instances and specify any fields for the instance – even if they are not specified in the `defclass`. By default, `defmethod` does not accept any arguments that do not have a corresponding parameter.

Specifying a Return Type

So far, I've described using Water Contract to specify input parameters for a method or object. Water Contract also supports the specification of the output or return value. `_return_type` is a special parameter for `defmethod` and `defclass`. The type of the object to be returned must be the value of `_return_type`.

```
<defmethod some_formatter _return_type=string/>
```

The preceding example describes a method `some_formatter` that is expected to return a value of type `string`. `_return_type` has no default value, so the type is placed after the first equal sign.

Preconditions and Postconditions

Water Contract also supports executable preconditions and postconditions. A precondition describes any expectations about the calling environment that must hold true before a call is made. Typed input parameters is one form of a precondition, but other examples might be the existence of a resource such as a database, or perhaps some relationship between arguments that must hold true. A precondition that is not specified in the parameter list should put in the value of a `_precond` parameter. A postcondition that is not described by the `return_type` should be put in the value of a `_postcond` parameter.

Continued

Preconditions and Postconditions (*Continued*)

```
<defmethod delete_my_database
  _precond=<defmethod> thing.<has_key my_database/> </>
  _postcond=<defmethod> thing.<has_key my_database/>.<not/> </>
/>
<defmethod increment_global_count_field
  _precond=<defmethod>
    thing.<has_key "count"/>.<and args.count.<more 0/>
  />
  </defmethod>
  _postcond=<defmethod>
    _args.count.<is count.<minus 1/> />
  </defmethod>
/>
```

If the precondition or postcondition is static text and is not an executable method, then the doc object can be used to store the value of the precond or postcond parameters.

```
<defmethod add_customer/>.
<doc
  precondition="customer has never had an account"
  postcond="a single account exists for that customer"
/>
```

Summary

Water Contract describes data structures and method interfaces in an executable representation. Every call to a method can be checked to verify that it obeys the method's contract. Every change to an object can be checked against the class' contract. Water Contract is very concise and much simpler than either DTD or XML Schema, while being more flexible and rigorous.