# The AQuA Domain Ontology

AQuA domain ontologies are used to provide a defined structure and template for information within specific domains of knowledge. Each domain ontology consists of one or more class definitions. Each class is intended to represent some entity whose characteristics (represented by the class properties) might be evaluated in an AQuA statement.

Classes consist of one or more properties. Each property has a name and a data type and is intended to represent a specific characteristic of the entity the class represents. Properties may be defined as classes themselves and thus contain sub-properties. The intent is that the properties of a class collectively cover the important characteristics (i.e., the characteristics that an AQuA statement might wish to evaluate) of the entity corresponding to the class.

## Data Types

AQuA domain ontologies specify a data type for all defined properties. AQuA statements are strongly typed, so the data types of properties dictate where that property can be used and how it can be processed within an AQuA statement. The following sections describe the datatypes in AQuA.

### Primitive Data Types

Primitive data types in AQuA are data types that are “atomic” in that they do not have sub-properties. The primitive data types in AQuA are:

INTEGER – a signed or unsigned whole number

FLOAT – a signed or unsigned decimal number

STRING – a collection of Unicode characters

BOOLEAN – either TRUE or FALSE

DATETIME – an RFC 3339-conformant timestamp

All data types have an associated list of operations, comparisons, and functions that can be used with that type within an AQuA statement. See the relevant sections of the AQuA syntax and semantics (section 6) for details. In addition, AQuA supports a limited set of type conversion functions, described in section 6.6.2.7. Note that success of a conversion operation depends not only on having the correct data type as an input, but that the input value must meet certain syntactic criteria as well. The type conversion functions descriptions identify relevant criteria.

### Class Types

AQuA domain ontologies define a set of classes. Each defined class, in effect, creates its own data type. The name of each of these data types is the name of the class. This name can be used within AQuA domain ontologies to indicate that the properties of one class represent instances of a different class. (This was seen in the simple example given in section 4.1.)

AQuA defines no functions or operations over class types and the only comparison operations permitted are equals (=) or not-equals (!=). The latter are only considered syntactically valid when comparisons are made between instances of the same class type.

## AQuA Domain Ontology Syntax and Semantics

The grammar of ontologies is standardized in AQuA to allow parties to create new ontologies that enable AQuA statements over new subjects. AQuA uses a JSON syntax to define ontology structures. It uses this instead of more standardized ontology languages (like OWL or JSON-LD) because AQuA domain ontologies have an extremely simple structure and do not need most of the capabilities of full ontology languages. Instead, AQuA uses a very simple structure that is sufficient for AQuA’s needs and is much easier to read.

### Class Names

Class names can be any Unicode string of printable characters with the exception of whitespace characters, which are not permitted in class names. Class names are sensitive to the Unicode values used rather than the rendering of those Unicode characters. As a result, it is possible for two class names to look identical in a file but be treated as distinct. For example, Unicode values of 0x41 and 0x0391 both render as “A”, but would be treated as different values when comparing class names. Ontology authors should be careful to use the most common character rendering within their expected area of use. The use of Unicode character values also means that class names are case sensitive.

Class names MUST not duplicate reserved words in AQuA or the names of AQuA functions. Each class name MUST be globally unique across all domain ontologies employed by a given AQuA interpreter. An interpreter will return an error if it encounters two ontology class definitions that use the same class name. Authors may wish to include information about the author within the class name (for example, the author’s email address or the company’s DNS name) to reduce the likelihood of name collisions and ensure that the class used by an AQuA interpreter is the specific class an AQuA statement author intended. Since a given class’s name only appears once at the very beginning of an AQuA statement, a somewhat long class name is not a significant burden to AQuA statement authors.

### Property Names

Like class names, property names can be any Unicode string of printable characters, excluding whitespace. Also like class names, property names are Unicode-value sensitive and case sensitive.

A class MUST NOT have duplicate property names in its definition. However, the same property name can be used in different classes. This is true even when classes that duplicate property names reference each other. For example, consider Classes A and B, both of which include a property named “ID”. Assume further that Class A has another property (foo) that is of type Class B. This is allowed within the AQuA domain ontology definition syntax and will not lead to ambiguity because, within an AQuA statement, it is always clear which class is the parent of any referenced property. As a result, it will always be clear whether a given reference to property ID is referring to a property of Class A or of Class B. How AQuA statements reference class properties is described in more detail in section 6.6.2.3.

### Property Types

All properties in an AQuA domain ontology are assigned a data type. A property can be assigned either a primitive data type or a class type. In the case that a property has a primitive data type, the type of the property MUST be represented by one of the following strings:

* STRING
* INTEGER
* FLOAT
* BOOLEAN
* DATETIME

These strings are case sensitive.

A property whose data type is a class MUST explicitly identify the appropriate class type. There is no “root class” in AQuA that represents “any class”. (E.g., there is nothing like the Java Object class in AQuA.)

The property’s class type can either be identified by reference or through an in-line class definition. When identifying a class type by reference, the name of the appropriate class is given where the property’s type is named. This class type does not need to be defined in the same domain ontology file that references it, but the referenced class needs to be available to the AQuA interpreter processing the AQuA statement.

AQuA does not prohibit cycles of class references. (E.g., Class A has a property of type Class B, and Class B has a property of type Class A.)

Alternately, a property with a non-primitive data type MAY have its class defined inline. Inline classes do not have a name. They are only instantiated within the context of their parent class. Mapping functions need to know how to handle inline sub-classes when populating class instances from an enterprise data source.

### The Domain Ontology File

AQuA domain ontologies are defined in a text file. A file containing AQuA domain ontologies MUST consist of a single JSON array. This array MUST consist of one or more JSON objects. Each of these objects represents one AQuA class.

For each top-level object (i.e., AQuA class), that object MUST consist of a single string-value pair. The string is the name of the class. The corresponding value MUST be a JSON array. Each element in this array is a JSON object representing one property of the named class. The named fields of these properties describe the property. All property definitions MUST have a field named “Name” and a field named “Type”. All properties SHOULD have a field named “Description” that contains a description of the intended meaning of that property. The author of an AQuA domain ontology MAY add other fields to a property definition. However, interpreters MUST NOT require the presence of any property definition fields other than Name and Type. The fields in a property definition can appear in any order. Note that “Name” and “Type” are not reserved words and it is legal to use either of these as the name of a property.

The value of the Name field in the property definition MUST be a valid property name. The value of the Type field in the property definition MUST be the string corresponding to a primitive data type, a class name, or a JSON array of JSON objects. The latter is used to define an in-line class and the JSON objects in the JSON array MUST be valid property definitions as defined above.

## Suggestions for Designing an AQuA Domain Ontology

As a general rule, when designing an AQuA domain ontology, authors SHOULD avoid too many in-line class definitions for property types. AQuA statements do not have shortcuts for referencing properties deep within nested classes, which can result in very long variable property references. Hierarchical structures in these ontologies can be used to make them more easily readable by human beings, but otherwise should be kept to a minimum.

Properties also SHOULD avoid overlapping meanings to the extent possible. Overlapping meaning can lead to ambiguity for AL authors in how best to express a test and could increase inconsistency in results. For this reason, to the extent possible, properties ought to be disjoint in their meaning.

Properties SHOULD also be as atomic as possible. In this regard, the ontology seeks to minimize the amount of “unpacking” necessary within an AL statement in order to interpret a value aligned with that ontology property. For example, a computer configuration file might store certain configurations as a bit map, where bit 0 indicates whether feature A is turned on, but 1 indicates whether feature B is turned on, etc. This produces a very compact encoding, but it is not atomic. As such, it would be better to represent this information using multiple properties in a domain ontology, where feature A is captured in one Boolean property, feature B is captured in another Boolean property, etc. This makes the meaning in the ontology clearer, makes writing AQuA statements simpler (since the statement does not need to unpack the meaning of the property), and is also more agnostic to how the information might be represented in an underlying enterprise data source.

## Using AQuA Domain Ontologies

AQuA domain ontologies are used in two ways: by AQuA statement authors creating new AQuA statements, and by AQuA interpreters parsing AQuA statements.

AQuA statement authors can use the classes and properties defined in AQuA domain ontologies to represent information that their statements test. In addition to defining the names for these classes and properties, the types associated with properties dictate the operations, comparisons, and functions that can be used on the property values.

When an AQuA interpreter processes an AQuA statement, the domain ontologies become templates for the organization of information from its enterprise data source. AQuA interpreters need to have a definition for each of the classes named in an AQuA statement. They must also know how to map the elements of an enterprise data source into instances of each of the utilized class as comprehensively as possible. See section 8 for more details on the processing of AQuA statements.

## AQuA Ontologies

grammar AQuA\_Ontology;

aquaOntology: (aquaOntologyJSON) EOF;

aquaOntologyJSON: LEFT\_BRACKET aquaOntologyClass (COMMA aquaOntologyClass)\* RIGHT\_BRACKET;

aquaOntologyClass: LEFT\_CURLY classVarProperty COLON aquaOntologyProperties RIGHT\_CURLY;

aquaOntologyProperties: LEFT\_BRACKET aquaOntologyProperty (COMMA aquaOntologyProperty)\* RIGHT\_BRACKET;

aquaOntologyProperty: LEFT\_CURLY(aquaOntologyNameProperty COMMA aquaOntologyTypeProperty) (COMMA aquaOntologyDescriptionProperty)? RIGHT\_CURLY;

aquaOntologyNameProperty: NAME COLON LiteralValue;

aquaOntologyTypeProperty: TYPE COLON (quotedDatatypes | LiteralValue | aquaOntologyJSON);

aquaOntologyDescriptionProperty: DESCRIPTION COLON LiteralValue;

aquaOntologyDefinedProperty: LiteralValue COLON LiteralValue;

//////////////////////////////

// classes

classVarProperty: LiteralValue;

//////////////////////////////

// datatypes

quotedDatatypes: QUOTE Datatypes QUOTE;

Datatypes: (INTEGER | FLOAT | STRING | BOOLEAN | DATETIME);

INTEGER: 'INTEGER';

FLOAT: 'FLOAT';

STRING: 'STRING';

BOOLEAN : 'BOOLEAN';

DATETIME: 'DATETIME';

//////////////////////////////

// Reserved

RIGHT\_CURLY: '}' ;

LEFT\_CURLY: '{' ;

RIGHT\_BRACKET: ']' ;

LEFT\_BRACKET: '[' ;

NAME: QUOTE 'Name' QUOTE ;

TYPE: QUOTE 'Type' QUOTE ;

DESCRIPTION: QUOTE 'Description' QUOTE ;

//////////////////////////////

// Literals

COMMA: ',';

QUOTE: DOUBLEQUOTE;

DOUBLEQUOTE: '\"';

COLON: ':';

//////////////////////////////

// Generic values

LiteralValue: QUOTE (~["])\* QUOTE;

//////////////////////////////

// Whitespace and comments

WS: [ \n\r\t]+ -> skip;

COMMENT: '/\*' .\*? '\*/' -> skip;

LINE\_COMMENT: '//' ~[\r\n]\* -> skip;