**OWL**

1. **TransitiveProperty**

If a property, P, is specified as transitive then for any x, y, and z:

P(x,y) and P(y,z) implies P(x,z)

The property locatedIn is transitive.

<owl:ObjectProperty rdf:ID="locatedIn">

<rdf:type rdf:resource="&owl;TransitiveProperty" />

<rdfs:domain rdf:resource="&owl;Thing" />

<rdfs:range rdf:resource="#Region" />

</owl:ObjectProperty>

<Region rdf:ID="SantaCruzMountainsRegion">

<locatedIn rdf:resource="#CaliforniaRegion" />

</Region>

<Region rdf:ID="CaliforniaRegion">

<locatedIn rdf:resource="#USRegion" />

</Region>

Because the SantaCruzMountainsRegion is locatedIn the CaliforniaRegion, then it must also be locatedIn the USRegion, since locatedIn is transitive.

### 2) SymmetricProperty

If a property, P, is tagged as symmetric then for any x and y:

P(x,y) iff P(y,x)

The property adjacentRegion is symmetric, while locatedIn is not. To be more precise, locatedIn is not intended to be symmetric. Nothing in the wine ontology at present prevents it from being symmetric.

<owl:ObjectProperty rdf:ID="adjacentRegion">

<rdf:type rdf:resource="&owl;SymmetricProperty" />

<rdfs:domain rdf:resource="#Region" />

<rdfs:range rdf:resource="#Region" />

</owl:ObjectProperty>

<Region rdf:ID="MendocinoRegion">

<locatedIn rdf:resource="#CaliforniaRegion" />

<adjacentRegion rdf:resource="#SonomaRegion" />

</Region>

The MendocinoRegion is adjacent to the SonomaRegion and vice-versa. The MendocinoRegion is located in the CaliforniaRegion but not vice versa.

### 3) FunctionalProperty

If a property, P, is tagged as functional then for all x, y, and z:

P(x,y) and P(x,z) implies y = z

In our wine ontology, hasVintageYear is functional. A wine has a unique vintage year. That is, a given individual Vintage can only be associated with a single year using the hasVintageYear property. It is not a requirement of a owl:FunctionalProperty that all elements of the domain have values. See the discussion of [Vintage cardinality](http://www.w3.org/TR/2004/REC-owl-guide-20040210/#simpleCardinality).

<owl:Class rdf:ID="VintageYear" />

<owl:ObjectProperty rdf:ID="hasVintageYear">

<rdf:type rdf:resource="&owl;FunctionalProperty" />

<rdfs:domain rdf:resource="#Vintage" />

<rdfs:range rdf:resource="#VintageYear" />

</owl:ObjectProperty>

### 4) inverseOf

If a property, P1, is tagged as the owl:inverseOf P2, then for all x and y:

P1(x,y) iff P2(y,x)

Note that the syntax for owl:inverseOf takes a property name as an argument. A iff B means (A implies B) and (B implies A).

<owl:ObjectProperty rdf:ID="hasMaker">

<rdf:type rdf:resource="&owl;FunctionalProperty" />

</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="producesWine">

<owl:inverseOf rdf:resource="#hasMaker" />

</owl:ObjectProperty>

Wines have makers, which in the definition of Wine are restricted to Winerys. Then each Winery produces the set of wines that identify it as maker.

### 5) InverseFunctionalProperty

If a property, P, is tagged as InverseFunctional then for all x, y and z:

P(y,x) and P(z,x) implies y = z

Notice that producesWine in the preceding section is inverse functional. The reason is that the inverse of a functional property must be inverse functional. We could have defined hasMaker and producesWine as follows and achieved the identical effect as the preceding example.

<owl:ObjectProperty rdf:ID="hasMaker" />

<owl:ObjectProperty rdf:ID="producesWine">

<rdf:type rdf:resource="&owl;InverseFunctionalProperty" />

<owl:inverseOf rdf:resource="#hasMaker" />

</owl:ObjectProperty>  ¬

Think of the elements of the range in an inverse functional property as defining a unique key in the database sense. owl:InverseFunctional implies that the elements of the range provide a unique identifier for each element of the domain.

In OWL Full, we can tag a DatatypeProperty as inverseFunctional. This permits us to identify a string as a unique key. In OWL DL literals are disjoint from owl:Thing, which is why OWL DL does not permit InverseFunctional to be applied to DatatypeProperty

### 6) Cardinality

We have already seen examples of cardinality constraints. To date, they have been assertions about minimum cardinality. Even more straight-forward is owl:cardinality, which permits the specification of exactly the number of elements in a relation. For example, we specify Vintage to be a class with exactly one VintageYear.

<owl:Class rdf:ID="Vintage">

<rdfs:subClassOf>

<owl:Restriction>

<owl:onProperty rdf:resource="#hasVintageYear"/>

<owl:cardinality rdf:datatype="&xsd;nonNegativeInteger">1</owl:cardinality>

</owl:Restriction>

</rdfs:subClassOf>

</owl:Class>

We specified hasVintageYear to be a functional property, which is the same as saying that every Vintage has at most one VintageYear. This application of that property to Vintage using the cardinality restriction asserts something stronger, that every Vintage has exactly one VintageYear.

Cardinality expressions with values limited to 0 or 1 are part of OWL Lite. This permits the user to indicate 'at least one', 'no more than one', and 'exactly one'. Positive integer values other than 0 and 1 are permitted in OWL DL. owl:maxCardinality can be used to specify an upper bound. owl:minCardinality can be used to specify a lower bound. In combination, the two can be used to limit the property's cardinality to a numeric interval.

### 7) Union [[OWL DL]](http://www.w3.org/TR/2004/REC-owl-guide-20040210/#OWL_DL_tag)

The following example demonstrates the use of the unionOf construct. It is used exactly like the intersectionOf construct:

<owl:Class rdf:ID="Fruit">

<owl:unionOf rdf:parseType="Collection">

<owl:Class rdf:about="#SweetFruit" />

<owl:Class rdf:about="#NonSweetFruit" />

</owl:unionOf>

</owl:Class>

The class Fruit includes both the extension of SweetFruit and the extension of NonSweetFruit.

Note how completely different this union type construct is from the following.

<owl:Class rdf:ID="Fruit">

<rdfs:subClassOf rdf:resource="#SweetFruit" />

<rdfs:subClassOf rdf:resource="#NonSweetFruit" />

</owl:Class>  ¬

This says that the instances of Fruit are a subset of the intersection of sweet and non-sweet fruit, which we would expect to be the empty set

### 8) Intersection [[some uses of OWL DL]](http://www.w3.org/TR/2004/REC-owl-guide-20040210/#OWL_DL_tag)

The following examples demonstrate the use of the intersectionOf construct.

<owl:Class rdf:ID="WhiteWine">

<owl:intersectionOf rdf:parseType="Collection">

<owl:Class rdf:about="#Wine" />

<owl:Restriction>

<owl:onProperty rdf:resource="#hasColor" />

<owl:hasValue rdf:resource="#White" />

</owl:Restriction>

</owl:intersectionOf>

</owl:Class>

Classes constructed using the set operations are more like definitions than anything we have seen to date. The members of the class are completely specified by the set operation. The construction above states that WhiteWine is exactly the intersection of the class Wine and the set of things that are white in color. This means that if something is white and a wine, then it is an instance of WhiteWine. Without such a definition we can know that white wines are wines and white, but not vice-versa. This is an important tool for categorizing individuals. (Note that 'rdf:parseType="Collection"' is a required syntactic element.)

<owl:Class rdf:about="#Burgundy">

<owl:intersectionOf rdf:parseType="Collection">

<owl:Class rdf:about="#Wine" />

<owl:Restriction>

<owl:onProperty rdf:resource="#locatedIn" />

<owl:hasValue rdf:resource="#BourgogneRegion" />

</owl:Restriction>

</owl:intersectionOf>

</owl:Class>

Here we define Burgundy to include exactly those wines that have at least one locatedIn relation to the Bourgogne Region. We could have declared a new class ThingsFromBourgogneRegion and used it as a class in the owl:intersectionOf construct. Since we do not have any other use for ThingsFromBourgogneRegion, the declaration above is shorter, clearer and doesn't require the creation of a contrived name.

<owl:Class rdf:ID="WhiteBurgundy">

<owl:intersectionOf rdf:parseType="Collection">

<owl:Class rdf:about="#Burgundy" />

<owl:Class rdf:about="#WhiteWine" />

</owl:intersectionOf>

</owl:Class>

Finally, the class WhiteBurgundy is exactly the intersection of white wines and Burgundies. Burgundies in turn are grown in the French region of Bourgogne and are dry wines. Accordingly all individual wines that meet these criteria are part of the class extension of WhiteBurgundy

## 9) Disjoint Classes disjointWith [[OWL DL]](http://www.w3.org/TR/2004/REC-owl-guide-20040210/#OWL_DL_tag)

The disjointness of a set of classes can be expressed using the owl:disjointWith constructor. It guarantees that an individual that is a member of one class cannot simultaneously be an instance of a specified other class.

<owl:Class rdf:ID="Pasta">

<rdfs:subClassOf rdf:resource="#EdibleThing"/>

<owl:disjointWith rdf:resource="#Meat"/>

<owl:disjointWith rdf:resource="#Fowl"/>

<owl:disjointWith rdf:resource="#Seafood"/>

<owl:disjointWith rdf:resource="#Dessert"/>

<owl:disjointWith rdf:resource="#Fruit"/>

</owl:Class>

The Pasta example demonstrates multiple disjoint classes. Note that this only asserts that Pasta is disjoint from all of these other classes. It does not assert, for example, that Meat and Fruit are disjoint. In order to assert that a set of classes is mutually disjoint, there must be an owl:disjointWith assertion for every pair.

A common requirement is to define a class as the union of a set of mutually disjoint subclasses.

<owl:Class rdf:ID="SweetFruit">

<rdfs:subClassOf rdf:resource="#EdibleThing" />

</owl:Class>

<owl:Class rdf:ID="NonSweetFruit">

<rdfs:subClassOf rdf:resource="#EdibleThing" />

<owl:disjointWith rdf:resource="#SweetFruit" />

</owl:Class>

<owl:Class rdf:ID="Fruit">

<owl:unionOf rdf:parseType="Collection">

<owl:Class rdf:about="#SweetFruit" />

<owl:Class rdf:about="#NonSweetFruit" />

</owl:unionOf>

</owl:Class>

Here we define Fruit to be exactly the union of SweetFruit and NonSweetFruit. And we know that these subclasses exactly partition Fruit into two distinct subclasses because they are disjoint. As the number of mutually disjoint classes grows, the number of disjointness assertions grows proportionally to n2. However, in the use cases we have seen, n is typically small.

When n is large, alternate approaches can be used to avoid quadratic growth in the number of assertions. One such method is illustrated in the [OWL test suite](http://www.w3.org/TR/2004/REC-owl-test-20040210/byIssue#I5.21-002)

The illustrated method works as follows. We describe a parent class whose elements have a property with cardinality equal to one. That is, each instance must have one and only one value for this property. Then, for every subclass of the parent we require that its instances must have a particular unique value for the property. In which case none of the distinct subclasses can have members in common.

### 1.5 Differences between XForms 2.0 and XForms 1.1

|  |  |
| --- | --- |
| **Editorial note: Complete list** | 2011-08-31 |
| Complete list of differences below | |

* Custom functions
* Variable support
* repeat over sequences of atomic values and nodes (doesn't need to be in document order, same node can occur multiple times)
* Pluggable expression language (XPath 2.0 module)
* json, csv and other external instance data formats
* @nodeset is deprecated in favour of @ref
* p3ptype is deprecated
* repeat @indexref
* Multiple MIPs of the same property allowed on one node.
* @iterate
* model-based switch : @caseref
* Attribute Value Template support
* script action
* property child of dispatch element
* dialog
* support for typed values in XPath 2 expressions
* @label, @hint, @help, @alert
* @appearance on <alert> (erratum)

**How is metadata is implemented in HTML 5**

The collection of metadata in a [head](https://www.w3.org/TR/2014/CR-html5-20140429/document-metadata.html#the-head-element) element can be large or small. Here is an example of a very short one:

<!doctype html>

<html>

<head>

<title>A document with a short head</title>

</head>

<body>

...

Here is an example of a longer one:

<!DOCTYPE HTML>

<HTML>

<HEAD>

<**META CHARSET="UTF-8">**

<BASE HREF="http://www.example.com/">

<TITLE>An application with a long head</TITLE>

<LINK REL="STYLESHEET" HREF="default.css">

<LINK REL="STYLESHEET ALTERNATE" HREF="big.css" TITLE="Big Text">

<SCRIPT SRC="support.js"></SCRIPT>

<META NAME="APPLICATION-NAME" CONTENT="Long headed application">

</HEAD>

<BODY>

...

**What are Data\* attributes?**

The data-\* attributes is used to store custom data private to the page or application.

The data-\* attributes gives us the ability to embed custom data attributes on all HTML elements.

The stored (custom) data can then be used in the page's JavaScript to create a more engaging user experience (without any Ajax calls or server-side database queries).

The data-\* attributes consist of two parts:

1. The attribute name should not contain any uppercase letters, and must be at least one character long after the prefix "data-"
2. The attribute value can be any string

**Note:** Custom attributes prefixed with "data-" will be completely ignored by the user agent.

## **Differences Between HTML 4.01 and HTML5**

The data-\* attributes are new in HTML5.

## **Syntax**

<*element* data-\*="somevalue">

## **Attribute Values**

|  |  |
| --- | --- |
| **Value** | **Description** |
| somevalue | Specifies the value of the attribute (as a string) |