The **Web Ontology Language** (**OWL**) is a family of [knowledge representation](https://en.wikipedia.org/wiki/Knowledge_representation_and_reasoning) languages for authoring [ontologies](https://en.wikipedia.org/wiki/Ontology_(information_science)). Ontologies are a formal way to describe taxonomies and classification networks, essentially defining the structure of knowledge for various domains: the nouns representing classes of objects and the verbs representing relations between the objects.

Look these two…video lecture

<https://www.youtube.com/watch?v=CXw-P7H2rOQ>

<https://www.youtube.com/watch?v=u8AXayzKuSg>



## 4 Classes, Properties, and Individuals – And Basic Modeling With Them

After these general considerations, we now engage in the details of modeling with OWL 2. In the subsequent sections, we introduce the essential modeling features that OWL 2 offers, provide examples and give some general comments on how to use them. We proceed from basic features, which are essentially available in any modeling language, to more advanced constructs.

Thereby we will represent information about a particular family. Note that we do not intend this example to be representative of the sorts of domains OWL should be used for, or as a canonical example of good modeling with OWL, or a correct representation of the rather complex, shifting, and culturally dependent domain of families. Instead, we intend it to be a rather simple exhibition of various features of OWL.

### 4.1 Classes and Instances

We start by introducing the persons we are talking about. This can be done as follows:

**Functional-Style Syntax**

ClassAssertion( :Person :Mary )

This statement talks about an individual named Mary and states that this individual is a person. More technically, *being a person* is expressed by stating that Mary belongs to (or “is a member of” or, even more technically, “is an instance of”) the *class* of all persons. In general classes are used to group individuals that have something in common in order to refer to them. Hence, classes essentially represent sets of individuals. In modeling, classes are often used to denote the set of objects comprised by a concept of human thinking, like the concept *person* or the concept *woman*. Consequently, we can use the same type of statement to indicate that Mary is a woman by expressing that she is an instance of the class of women:

**Functional-Style Syntax**

ClassAssertion( :Woman :Mary )

Hereby it also becomes clear that class membership is not exclusive: as there may be diverse criteria to group individuals (like gender, age, shoe size, etc.), one individual may well belong to several classes simultaneously.

### 4.2 Class Hierarchies

In the previous section, we were talking about two classes: the class of all persons and that of all women. To the human reader it is clear that these two classes are in a special relationship: Person is more general than Woman, meaning that whenever we know some individual to be a woman, that individual must be a person. However, this correspondence cannot be derived from the labels “Person” and “Woman” but is part of the human background knowledge about the world and our usage of those terms. Therefore, in order to enable a system to draw the desired conclusions, it has to be informed about this correspondence. In OWL 2, this is done by a so-called subclass axiom:

**Functional-Style Syntax**

SubClassOf( :Woman :Person )

The presence of this axiom in an ontology enables reasoners to infer for every individual which is specified as an instance of the class Woman, that it is an instance of the class Person as well. As a rule of thumb, a subclass relationship between two classes A and B can be specified, if the phrase “every A is a B” makes sense and is correct.

It is common in ontological modeling to use subclass statements not only for sporadically declaring such interdependencies, but to model whole *class hierarchies* by specifying the generalization relationships of all classes in the domain of interest. Suppose we also want to state that all mothers are women:

**Functional-Style Syntax**

SubClassOf( :Mother :Woman )

Then a reasoner could not only derive for every single individual that is classified as mother, that it is also a woman (and consequently a person), but also that Mother must be a subclass of Person – coinciding with our intuition. Technically, this means that the subclass relationship between classes is *transitive*. Besides this, it is also *reflexive*, meaning that every class is its own subclass – this is intuitive as well since clearly, every person is a person etc.

Classes in our vocabulary may effectively refer to the same sets, and OWL provides a mechanism by which they are considered to be semantically equivalent. For example, we use the term Person and Human interchangeably, meaning that every instance of the class Person is also an instance of class Human, and vice versa. Two classes are considered equivalent if they contain exactly the same individuals. The following example states that the class Person is equivalent to the class Human.

**Functional-Style Syntax**

EquivalentClasses( :Person :Human )

Stating that Person and Human are equivalent amounts exactly to the same as stating that both Person is a subclass of Human and Human is a subclass of Person.

### 4.3 Class Disjointness

In Section 4.1, we stated that an individual can be an instance of several classes. However, in some cases membership in one class specifically excludes membership in another. For example, if we consider the classes Man and Woman, we know that no individual can be an instance of both classes (for the sake of the example, we disregard biological borderline cases). This “incompatibility relationship” between classes is referred to as *(class) disjointness*. Again, the information that two classes are disjoint is part of our background knowledge and has to be explicitly stated for a reasoning system to make use of it. This is done as follows:

**Functional-Style Syntax**

DisjointClasses( :Woman :Man )

In practice, disjointness statements are often forgotten or neglected. The arguable reason for this could be that intuitively, classes are considered disjoint unless there is other evidence. By omitting disjointness statements, many potentially useful consequences can get lost. Note that in our example, the disjointness axiom is needed to deduce that Mary is not a man. Moreover, given the above axioms, a reasoner can infer the disjointness of the classes Mother and Man.

### 4.4 Object Properties

In the preceding sections we were concerned with describing single individuals, their class memberships, and how classes can relate to each other based on their instances. But more often than not, an ontology is also meant to specify how the individuals relate to other individuals. These relationships are central when describing a family. We start by indicating that Mary is John's wife.

**Functional-Style Syntax**

ObjectPropertyAssertion( :hasWife :John :Mary )

Hereby, the entities describing in which way the individuals are related – like hasWife in our case, are called *properties.*

Note that the order in which the individuals are written is important. While “Mary is John's wife” might be true, “John is Mary's wife” certainly isn't. Indeed, this is a common source of modeling errors that can be avoided by using property names which allow only one unique intuitive reading. In case of nouns (like “wife”), such unambiguous names might be constructions with “of” or with “has” (wifeOf or hasWife). For verbs (like “to love”) an inflected form (loves) or a passive version with “by” (lovedBy) would prevent unintended readings.

We can also state that two individuals are *not* connected by a property. The following, for example, states that Mary is not Bill's wife.

**Functional-Style Syntax**

NegativeObjectPropertyAssertion( :hasWife :Bill :Mary )

Negative property assertions provide a unique opportunity to make statements where we know something that is not true. This kind of information is particularly important in OWL where the default stance is that anything is possible until you say otherwise.

### 4.5 Property Hierarchies

In Section 4.2 we argued that it is useful to specify that one class membership implies another one. Essentially the same situation can occur for properties: whenever B is known to be A's wife, it is also known to be A's spouse (note, that this is not true the other way round). OWL allows to specify this statement as follows:

**Functional-Style Syntax**

SubObjectPropertyOf( :hasWife :hasSpouse )

There is also a syntactic shortcut for property equivalence, which is similar to class equivalence.

### 5.1 Complex Classes

By means of the language elements described so far, simple ontologies can be modeled. In order to express more complex knowledge, OWL provides logical class constructors. In particular, OWL provides language elements for logical and, or, and not. The corresponding OWL terms are borrowed from set theory: *(class) intersection*, *union* and *complement*. These constructors combine atomic classes – i.e. classes with names – to complex classes.

The *intersection* of two classes consists of exactly those individuals which are instances of both classes. The following example states that the class Mother consists of exactly those objects which are instances of both Woman and Parent:

**Functional-Style Syntax**

EquivalentClasses(

 :Mother

ObjectIntersectionOf( :Woman :Parent )

)

An example for an inference which can be drawn from this is that all instances of the class Mother are also in the class Parent.

The *union* of two classes contains every individual which is contained in at least one of these classes. Therefore we could characterize the class of all parents as the union of the classes Mother and Father:

**Functional-Style Syntax**

EquivalentClasses(

 :Parent

ObjectUnionOf( :Mother :Father )

)

The *complement* of a class corresponds to logical negation: it consists of exactly those objects which are not members of the class itself. The following definition of childless persons uses the class complement and also demonstrates that class constructors can be nested:

**Functional-Style Syntax**

EquivalentClasses(

 :ChildlessPerson

ObjectIntersectionOf(

 :Person

ObjectComplementOf( :Parent )

)

)

All the above examples demonstrate the usage of class constructors in order to *define* new classes as combination of others. But, of course, it is also possible to use class constructors together with a subclass statement in order to indicate necessary, but not sufficient, conditions for a class. The following statement indicates that every Grandfather is both a man and a parent (whereas the converse is not necessarily true):

**Functional-Style Syntax**

SubClassOf(

 :Grandfather

ObjectIntersectionOf( :Man :Parent )

)

In general, complex classes can be used in every place where named classes can occur, hence also in class assertions. This is demonstrated by the following example which asserts that Jack is a person but not a parent.

**Functional-Style Syntax**

ClassAssertion(

ObjectIntersectionOf(

 :Person

ObjectComplementOf( :Parent )

)

 :Jack

)

### Property Restrictions

Property restrictions provide another type of logic-based constructors for complex classes. As the name suggests, property restrictions use constructors involving properties.

One property restriction called *existential quantification* defines a class as the set of all individuals that are connected via a particular property to another individual which is an instance of a certain class. This is best explained by an example, like the following which defines the class of parents as the class of individuals that are linked to a person by the hasChild property.

**Functional-Style Syntax**

EquivalentClasses(

 :Parent

ObjectSomeValuesFrom( :hasChild :Person )

)

This means that there is an expectation that for every instance of Parent, there exists at least one child, and that child is a member of the class Person. This is useful to capture *incomplete* knowledge*. For example, Sally tells us that Bob is a parent,* and therefore we can infer that he has at least one child even if we don't know their name. Natural language indicators for the usage of existential quantification are words like “some,” or “one.”

Another property restriction, called *universal quantification* is used to describe a class of individuals for which all related individuals must be instances of a given class. We can use the following statement to indicate that somebody is a happy person exactly if all their children are happy persons.

**Functional-Style Syntax**

EquivalentClasses(

 :HappyPerson

ObjectAllValuesFrom( :hasChild :HappyPerson )

)