# 5 Web Ontology Language: OWL

OWL (Web Ontology Language) is the latest recommendation of W3C [34], and is probably the most popular language for creating ontologies today. It is also the last technical component we need to familiarize ourselves with. The good news is that it is built on RDF schema; as you have already established a solid understanding of RDF schema, much of the material here is going to look familiar to you.

OWL = RDF schema + new constructs for expressiveness

Therefore, all the classes and properties provided by RDF schema can be used when creating an OWL document.

OWL and RDF schema have the same purpose: to define classes, properties, and their relationships. However, compared to RDF schema, OWL gives us the capability to express much more complex and richer relationships. The final result is that you can construct agents or tools with greatly enhanced reasoning ability.

Therefore, we often want to use OWL for the purpose of ontology development; RDF schema is still a valid choice, but its obvious limitations compared to OWL will always make it a second choice.

We developed a small camera ontology using RDF schema in Chapter 4 (see Figure 4.2). In this chapter, we are going to use OWL to rewrite the ontology. As we will be using OWL, new features will be added and, therefore, a new ontology with much more knowledge expressed is the result. However, the taxonomy (classes and class hierarchy) will still be the same.

## 5.1 USING OWL TO DEFINE CLASSES: LOCALIZE GLOBAL PROPERTIES

Our goal in this chapter is to understand OWL. We will accomplish this by rewriting our camera ontology. Let us start with class definitions.

In RDF schema, the root class of everything is `rdfs:resource`. More specifically, this root class has the following URI:

```
http://www.w3.org/2001/01/rdf-schema#resource
```

In the world of OWL, the `owl:Thing` class is the root of all classes; its URI is as follows:

```
http://www.w3.org/2002/07/owl#Thing
```

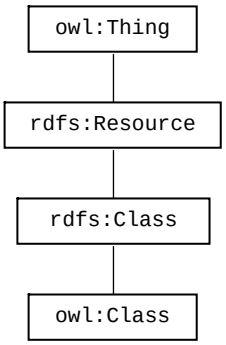


**FIGURE 5.1** OWL top class structure.

Clearly, owl represents the namespace for OWL; i.e.,

```
http://www.w3.org/2002/07/owl#
```

Also, OWL has created a new class called `owl:Class` to define classes in OWL documents; it is a subclass of `rdfs:Class`. The relationship between all these top classes is summarized in Figure 5.1.

To define one of our camera ontology top classes, such as `Camera`, you can do the following:

```
<owl:Class rdf:ID="Camera">
</owl:Class>
```

And the following is an equivalent format:

```
<owl:Class rdf:ID="Camera">
   <rdfs:subClassOf
         rdfs:resource="http://www.w3.org/2002/07/owl#Thing"/>
</owl:Class>
```

Therefore, to define all the classes in our camera ontology, List 5.1 will be good enough.

---

**LIST 5.1**
**Class Definitions in Camera Ontology Using OWL**

```
//
// Camera.owl
//
1: <?xml version="1.0"?>
2: <rdf:RDF
       xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
3:     xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
4:     xmlns:owl="http://www.w3.org/2002/07/owl#"
5:     xml:base="http://www.yuchen.net/photography/Camera.owl">
```

```
//
// classes definitions
//
6:    <owl:Class rdf:ID="Camera">
7:   </owl:Class>

8:   <owl:Class rdf:ID="Person">
9:   </owl:Class>

10:  <owl:Class rdf:ID="Digital">
11:     <rdfs:subClassOf rdf:resource="#Camera"/>
12:  </owl:Class>

13:  <owl:Class rdf:ID="Film">
14:     <rdfs:subClassOf rdf:resource="#Camera"/>
15:  </owl:Class>

16:  <owl:Class rdf:ID="SLR">
17:     <rdfs:subClassOf rdf:resource="#Digital"/>
18:  </owl:Class>

19:  <owl:Class rdf:ID="PointAndShoot">
20:     <rdfs:subClassOf rdf:resource="#Digital"/>
21:  </owl:Class>

22:  <owl:Class rdf:ID="Photographer">
23:     <rdfs:subClassOf rdf:resource="#Person"/>
24:  </owl:Class>

25:  <owl:Class rdf:ID="Specifications">
26:  </owl:Class>

//
// property definitions: coming up ...
//
```

It looks as if our job is done. We have just finished using OWL to define all the classes used in our camera ontology (note that we have changed the class `Point-And-Shoot` to a new name, `PointAndShoot` because some valuators do not like hyphens in the middle of class names).

True, but the preceding list defines a very simple class hierarchy. OWL offers much greater expressiveness when it comes to defining classes. Let us explore these features one by one. To show how these new features are used, we also have to change our camera ontology from time to time.

### 5.1.1 owl:allValuesFrom

In our current camera ontology, written using RDFS (List 4.14), we defined a property called `owned_by` and associated it with two classes, `SLR` and `Photographer`, to express the fact that `SLR` is `owned_by` `Photographer`.

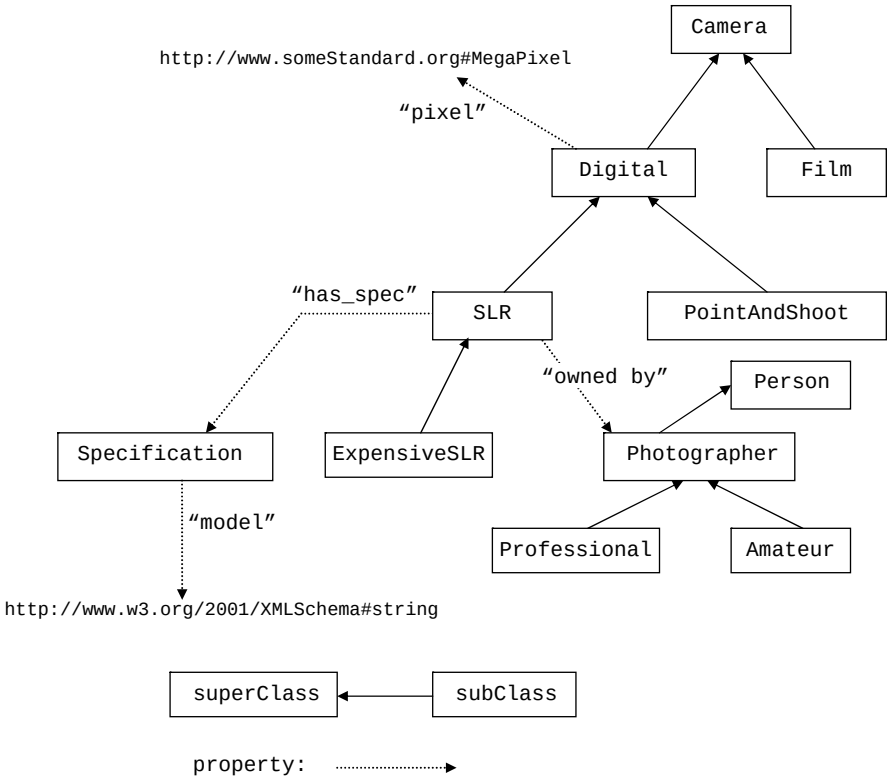


**FIGURE 5.2** A more complex camera ontology.

Compared to RDFS, OWL has much more powerful expressiveness. To see this, suppose we now want to express the following fact: `SLRs`, especially the expensive ones, are normally used by professional photographers (just the body of some high-end digital SLRs can cost as much as $5000). To further express this idea, we can create a subclass of `SLR`, called `ExpensiveSLR`, and two subclasses of photographer, `Professional` and `Amateur`. The current ontology is shown in Figure 5.2.

However, this does not solve our problem. Recall what we learned about RDF schema in Chapter 4. The property `owned_by` has `SLR` as its `rdfs:domain` and `Photographer` as its `rdfs:value`; given that `ExpensiveSLR` is a subclass of `SLR` and `Professional` and `Amateur` are both subclasses of `Photographer`, these new subclasses all inherit the `owned_by` property. Therefore, we did not exclude the following:

```
ExpensiveSLR owned_by Amateur
```

How do we modify the definition of `ExpensiveSLR` to ensure that it can be owned only by `Professional`? OWL uses `owl:allValuesFrom` to solve this problem, as shown in List 5.2.

**LIST 5.2**
**`owl:allValuesFrom` Example**

```
1: <owl:Class rdf:ID="ExpensiveSLR">
2:    <rdfs:subClassOf rdf:resource="#SLR"/>
3:      <rdfs:subClassOf>
4:        <owl:Restriction>
5:          <owl:onProperty rdf:resource="#owned_by"/>
6:          <owl:allValuesFrom rdf:resource="#Professional"/>
7:        </owl:Restriction>
8:    </rdfs:subClassOf>
9: </owl:Class>
```

We can interpret this definition as follows:

> Here is a definition of class `ExpensiveSLR`; it is a subclass of `SLR` and has a property named `owned_by`, and only an instance of class `Professional` can be the value of this property.

It takes a while to get used to the structure between lines 3 and 8. Line 3 (together with line 8) states that `ExpensiveSLR` is a subclass of class `X`; class `X` is defined by the `owl:Restriction` structure in lines 4 to 7. In the world of OWL, `owl:Restriction` is frequently used to specify an anonymous class. In our case, lines 4 to 7 define a class that has `owned_by` as a property, and only an instance of `Professional` can be its value.

### 5.1.2 Enhanced Reasoning Power 1

Let us always ask the following question from now on: What kind of inferencing power does OWL give us through `owl:allValuesFrom`? The following is the answer:

***OWL inferencing power 1***

The agent sees this:

```
<ExpensiveSLR rdf:ID="Nikon D200">
  <owned_by rdf:resource="http://www.yuchen.net/people#Liyang"/>
  <owned_by rdf:resource="http://www.yuchen.net/people#Jin"/>
</ExpensiveSLR>
```

The agent understands:

> Both `Liyang` and `Jin` are `Professionals` (not `Photographers` or `Amateurs`).

In the later sections, we will always summarize the enhanced reasoning power using the preceding format.

### 5.1.3 `owl:someValuesFrom` and `owl:hasValue`

In the preceding section, we used `owl:allValuesFrom` to ensure that only `Professionals` can own `ExpensiveSLRs`. Now, let us loosen this restriction by allowing

some Amateurs as well to buy and own ExpensiveSLRs. However, we still require that for a given expensive SLR, at least one of its owners has to be a Professional.

In other words, ExpensiveSLR can be owned by either Professionals or Amateurs, but it has to be owned by at least one Professional. OWL uses owl:some-ValuesFrom to express this idea, as shown in List 5.3.

---

**LIST 5.3**
**owl:someValuesFrom Example**

```
1: <owl:Class rdf:ID="ExpensiveSLR">
2:   <rdfs:subClassOf rdf:resource="#SLR"/>
3:     <rdfs:subClassOf>
4:       <owl:Restriction>
5:         <owl:onProperty rdf:resource="#owned_by"/>
6:         <owl:someValuesFrom rdf:resource="#Professional"/>
7:       </owl:Restriction>
8:     </rdfs:subClassOf>
9: </owl:Class>
```

---

This can be interpreted as follows:

> A class called ExpensiveSLR is defined. It is a subclass of SLR, and it has a property called owned_by. Furthermore, at least one value of owned_by property is an instance of Professional.

Another way in which OWL localizes a global property in the context of a given class is to use owl:hasValue. Consider the following scenario, in which owl:has-Value will be needed. We have created a class called ExpensiveSLR to express the knowledge that there are expensive digital cameras. We then enhanced this idea by saying that these expensive cameras are mainly owned by professional photographers. This is a good way of expressing this knowledge, but we can use a more direct way to accomplish the same result, by creating a property called expensiveOrNot, like this:

```
<owl:DatatypeProperty rdf:ID="expensiveOrNot">
   <rdfs:domain rdf:resource="#Digital"/>
   <rdfs:range
         rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
</owl:DatatypeProperty>
```

As we are not defining properties in this section, let us not worry about the syntax for now; just understand that this defines a property called expensiveOrNot, which is used to describe Digital. Its value will be a string of your choice; for instance, you can assign expensive or inexpensive as its value.

Clearly, SLR, PointAndShoot, and ExpensiveSLR are all subclasses of Digital; therefore, they can all use the expensiveOrNot property however they want. In other words, expensiveOrNot, as a property, is global. Now, in order to directly

express the knowledge that an ExpensiveSLR is expensive, we can constrain the value of expensiveOrNot to be always expensive when used with ExpensiveSLRs. We can use owl:hasValue to implement this idea, as shown in List 5.4. This defines the ExpensiveSLR as follows:

> A class called ExpensiveSLR is defined. It is a subclass of SLR, and every instance of ExpensiveSLR has an expensiveOrNot property whose value is expensive.

---

**LIST 5.4**
**owl:hasValue Example**

```
1: <owl:Class rdf:ID="ExpensiveSLR">
2:    <rdfs:subClassOf rdf:resource="#SLR"/>
3:    <rdfs:subClassOf>
4:       <owl:Restriction>
5:          <owl:onProperty rdf:resource="#expensiveOrNot"/>
6:          <owl:hasValue rdf:datatype="http://www.w3.org/2001/
                 XMLSchema#string">expensive
            </owl:hasValue>
7:       </owl:Restriction>
8:    </rdfs:subClassOf>
9: </owl:Class>
```

---

On the other hand, instances of SLR or PointAndShoot can take whatever expensiveOrNot value they want (i.e., expensive or inexpensive), indicating that they can be either expensive or inexpensive. This is exactly what we want.

It is now a good time to take a look at the differences between these three properties. Whenever we use owl:allValuesFrom, it is equivalent to declaring that "all the values of this property must be of this type, but it is all right if there are no values at all." Therefore, the property instance does not even have to appear. On the other hand, using owl:someValuesFrom is equivalent to saying that "there must be some values for this property, and at least one of these values has to be of this type. It is okay if there are other values of other types." Clearly, imposing an owl:someValuesFrom restriction on a property implies that this property has to appear at least once, whereas an owl:allValuesFrom restriction does not require the property to show up at all.

Finally, owl:hasValue says, "regardless of how many values a class has for a particular property, at least one of them must be equal to the value that you specify." It is therefore very much the same as owl:someValuesFrom except that it is more specific, because it requires a particular instance instead of a class.

### 5.1.4 Enhanced Reasoning Power 2

As owl:hasValue and owl:someValuesFrom are quite similar, let us assume ExpensiveSLR is defined by using owl:someValuesFrom. Then, what additional reasoning power does this give us?

***OWL inferencing power 2***

The agent sees this:

```
<ExpensiveSLR rdf:ID="Nikon D200">
  <owned_by rdf:resource="http://www.yuchen.net/people#Liyang"/>
  <owned_by rdf:resource="http://www.yuchen.net/people#Jin"/>
</ExpensiveSLR>
```

The agent understands the following:

Either `Liyang` or `Jin` (or both) is `Professional`.

In this section, we have discussed several ways to define classes by imposing constraints on a global property. As a result, the expressiveness of our camera ontology has been greatly enhanced, and more reasoning power is gained by this enhancement.

### 5.1.5 Cardinality Constraints

Another way to define a class by imposing restrictions on properties is through cardinality considerations. For example, we can say we want exactly one person to own `ExpensiveSLR`, as shown in List 5.5. Note that you need to specify that the literal "`1`" is to be interpreted as a nonnegative integer, using the `rdf:datatype` property.

**LIST 5.5**
**`owl:cardinality` Example**

```
1:  <owl:Class rdf:ID="ExpensiveSLR">
2:     <rdfs:subClassOf rdf:resource="#SLR"/>
3:     <rdfs:subClassOf>
4:        <owl:Restriction>
5:           <owl:onProperty rdf:resource="#owned_by"/>
6:           <owl:cardinality
       rdf:datatype="http://www.w3.org/2001/XMLSchema
           #nonNegativeInteger">
7:           1
8:           </owl:cardinality>
9:        </owl:Restriction>
10:    </rdfs:subClassOf>
11: </owl:Class>
```

Similarly, if we want to express the idea that at least one person should own `ExpensiveSLR`, we can follow List 5.6.

**LIST 5.6**
**`owl:minCardinality` Example**

```
1:  <owl:Class rdf:ID="ExpensiveSLR">
2:     <rdfs:subClassOf rdf:resource="#SLR"/>
```

```
3:     <rdfs:subClassOf>
4:        <owl:Restriction>
5:           <owl:onProperty rdf:resource="#owned_by"/>
6:           <owl:minCardinality
       rdf:datatype="http://www.w3.org/2001/XMLSchema
           #nonNegativeInteger">
7:           1
8:           </owl:minCardinality>
9:        </owl:Restriction>
10:    </rdfs:subClassOf>
11: </owl:Class>
```

You can certainly use `owl:maxCardinality` to specify the maximum number of people who can own the same camera. It is also possible to use `owl:minCardinality` and `owl:maxCardinality` at the same time to specify a range, as shown in List 5.7. Clearly, this asserts that at least one person, and at most two people, can own the camera. As you can see, the expressiveness in OWL is indeed greatly enhanced.

**LIST 5.7**
**Using `owl:maxCardinality` and `owl:minCardinality` to Specify a Range**

```
1:  <owl:Class rdf:ID="ExpensiveSLR">
2:     <rdfs:subClassOf rdf:resource="#SLR"/>
3:     <rdfs:subClassOf>
4:        <owl:Restriction>
5:           <owl:onProperty rdf:resource="#owned_by"/>
6:           <owl:minCardinality
       rdf:datatype="http://www.w3.org/2001/XMLSchema
           #nonNegativeInteger">
7:           1
8:           </owl:minCardinality>
9:           <owl:maxCardinality
       rdf:datatype="http://www.w3.org/2001/XMLSchema
           #nonNegativeInteger">
10:          2
11:          </owl:maxCardinality>
12:       </owl:Restriction>
10:    </rdfs:subClassOf>
11: </owl:Class>
```

### 5.1.6 Enhanced Reasoning Power 3

To see how the preceding cardinality constraints can help reasoning, let us assume that the `ExpensiveSLR` class is defined by using the `owl:cardinality` (exact number) property:

***OWL inferencing power 3***

The agent sees this:

```
<ExpensiveSLR rdf:ID="Nikon D200">
  <owned_by rdf:resource="http://www.yuchen.net/people#Liyang"/>
  <owned_by rdf:resource="http://www.yuchen.net/people#Jin"/>
</ExpensiveSLR>
```

The agent understands the following:

`Liyang` and `Jin` must be the same person (because `owl:cardinality` is 1).

### 5.1.7 Updating Our `Camera` Ontology

Before we move on, as we have created several new classes, i.e., `Professional`, `Amateur`, and `ExpensiveSLR` (let us not worry about properties for now), we can update our camera ontology as shown in List 5.8. Note that for experimental reasons we have imposed many different constraints on the `owned_by` property when defining the `ExpensiveSLR` class, but in our final camera ontology, we will create the `ExpensiveSLR` class by stating that only `Professionals` can own it.

---

**LIST 5.8**
**Updated `Camera` Ontology**

```
//
// Camera.owl
//  all the classes definitions are final!!
//
1: <?xml version="1.0"?>
2: <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
3:          xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
4:          xmlns:owl="http://www.w3.org/2002/07/owl#"
5:          xml:base="http://www.yuchen.net/photography/Camera.owl">

//
// classes definitions
//
6:    <owl:Class rdf:ID="Camera">
7:   </owl:Class>

8:   <owl:Class rdf:ID="Person">
9:   </owl:Class>

10:  <owl:Class rdf:ID="Digital">
11:     <rdfs:subClassOf rdf:resource="#Camera"/>
12:  </owl:Class>
```

```
13:  <owl:Class rdf:ID="Film">
14:     <rdfs:subClassOf rdf:resource="#Camera"/>
15:  </owl:Class>

16:  <owl:Class rdf:ID="SLR">
17:     <rdfs:subClassOf rdf:resource="#Digital"/>
18:  </owl:Class>

19:  <owl:Class rdf:ID="PointAndShoot">
20:     <rdfs:subClassOf rdf:resource="#Digital"/>
21:  </owl:Class>

22:  <owl:Class rdf:ID="Photographer">
23:     <rdfs:subClassOf rdf:resource="#Person"/>
24:  </owl:Class>

25:  <owl:Class rdf:ID="Specifications">
26:  </owl:Class>

27:  <owl:Class rdf:ID="Professional">
28:     <rdfs:subClassOf rdf:resource="#Photographer"/>
29:  </owl:Class>

30:  <owl:Class rdf:ID="Amateur">
31:     <rdfs:subClassOf rdf:resource="#Photographer"/>
32:  </owl:Class>

33:  <owl:Class rdf:ID="ExpensiveSLR">
34:    <rdfs:subClassOf rdf:resource="#SLR"/>
35:    <rdfs:subClassOf>
36:       <owl:Restriction>
37:          <owl:onProperty rdf:resource="#owned_by"/>
38:          <owl:someValuesFrom rdf:resource="#Professional"/>
39:       </owl:Restriction>
40:    </rdfs:subClassOf>
41: </owl:Class>

//
// property definitions: coming up...
//
```

Up to this point, we have covered the following OWL vocabulary: `owl:Thing`, `owl:Class`, `owl:Restriction`, `owl:allValuesFrom`, `owl:hasValuesFrom`, `owl:hasValue`, `owl:cardinality`, `owl:minCardinality`, and `owl:maxCardinality`.

## 5.2 USING OWL TO DEFINE CLASS: SET OPERATORS AND ENUMERATION

### 5.2.1 Set Operators

The goal of this chapter is to give you an understanding of OWL and to demonstrate its enhanced reasoning power, instead of giving you a full OWL tutorial. Therefore, we are not going to dive into the details about set operations in OWL; based on what you have already learned about OWL, these operations should be quite intuitive and straightforward. As a summary, OWL includes the following set operations, based on which you can define new classes:

- `owl:intersectionOf`
- `owl:unionOf`
- `owl:complementOf`

### 5.2.2 Enumerations

Enumeration is another brand-new feature that has been added by OWL, and it could be quite useful in many cases. To see this, let us recall how we defined the `ExpensiveSLR` class. So far we have been defining the class `ExpensiveSLR` by stating that it has to be owned by a professional photographer, or its `expensiveOrNot` property has to take the value `expensive`, etc. However, this is just a *descriptive* way to define `ExpensiveSLR`; in other words, it asserts that as long as an instance satisfies all these conditions, it is a member of the `ExpensiveSLR` class.

A large number of instances could still qualify. In some cases, it will be more useful to explicitly enumerate the qualified members; this will result in more accurate semantics for many applications. OWL provides the `owl:oneOf` property for this (see List 5.9).

---

**LIST 5.9**
**`owl:oneOf` Example**

```
1:  <owl:Class rdf:ID="ExpensiveSLR">
2:     <rdfs:subClassOf rdf:resource="#SLR"/>
3:     <owl:oneOf rdf:parseType="Collection">
4:       <SLR rdf:about="http://www.someNikonSite.com/digital/#D70"/>
5:       <SLR rdf:about="http://www.someNikonSite.com/digital/#D200"/>
6:       <SLR rdf:about="http://www.someCanonSite.com/digital/#20D"/>
7:       ... // other instances you might have
8:     </owl:oneOf>
9:  </owl:Class>
```

---

Note the syntax; you need to use `owl:oneOf` together with `rdf:parseType` to tell the parser that you are enumerating all the members of the class you are defining.

Up to this point, we have covered the following OWL vocabulary: `owl:Thing`, `owl:Class`, `owl:Restriction`, `owl:allValuesFrom`, `owl:hasValuesFrom`, `owl:hasValue`, `owl:cardinality`, `owl:minCardinality`, `owl:maxCardinality`, `owl:intersectionOf`, `owl:unionOf`, `owl:complementOf`, and `owl:oneOf`.

## 5.3 USING OWL TO DEFINE PROPERTIES: A RICHER SYNTAX FOR MORE REASONING POWER

We have finished defining the necessary classes for our project of rewriting the camera ontology using OWL. It is now time to define all the necessary properties.

Recall that when creating ontologies using RDF schema, we have the following three ways to describe a property:

- `rdfs:domain`
- `rdfs:range`
- `rdfs:subPropertyOf`

With just these three methods, however, a smart agent already shows impressive reasoning power, and more importantly, most of this power comes from the agent's understanding of properties (see the last section in Chapter 4).

This shows a simple yet important fact: richer semantics embedded into the properties will directly result in greater reasoning capabilities. This is why OWL, besides continuing to use these three methods, has also greatly increased the number of ways of characterizing a property, as we will see in this section.

The first point to note is that defining properties using OWL is quite different from defining properties using RDF schema. The general procedure is to first define the property and then use it to connect one resource with either another resource or with a typed or untyped value. Recall that in the world of RDF and RDF schema, `rdf:Property` is used for both connections. However, OWL uses two different classes to implement these two different connections, as shown:

- `owl:ObjectProperty` is used to connect a resource to another resource.
- `owl:DatatypePropery` is used to connect a resource to an `rdfs:Literal` (untyped) or an XML schema built-in data type (typed) value.

Also, `owl:ObjectProperty` and `owl:DatatypeProperty` are both subclasses of `rdf:Property`. For example, List 5.10 defines `owned_by` and `expensiveOrNot` properties by using RDF schema. In OWL, these definitions are as shown in List 5.11.

---

**LIST 5.10**
**Using RDFS to Define Properties**

```
1:  <rdf:Property rdf:ID="owned_by">
2:    <rdfs:domain rdf:resource="#SLR"/>
```

```
3:    <rdfs:range rdf:resource="#Photographer"/>
4:  </rdf:Property>

5:  <rdf:Property rdf:ID="expensiveOrNot">
6:    <rdfs:domain rdf:resource="#Digital"/>
7:    <rdfs:range
            rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
8:  </rdf:Property>
```

**LIST 5.11**
**Using OWL to Define Properties**

```
1:  <owl:ObjectProperty rdf:ID="owned_by">
2:    <rdfs:domain rdf:resource="#SLR"/>
3:    <rdfs:range rdf:resource="#Photographer"/>
4:  </owl:ObjectProperty>

5:  <owl:DatatypeProperty rdf:ID="expensiveOrNot">
6:    <rdfs:domain rdf:resource="#Digital"/>
7:    <rdfs:range
            rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
8:  </owl:DatatypeProperty>
```

We can now see that except for `owl:ObjectProperty` and `owl:DatatypeProperty`, the basic syntax of defining properties in both RDF schema and OWL is quite similar. In fact, we can now go ahead and define all the properties that appear in Figure 5.2; after defining these properties, we get an entire camera ontology written in OWL. (Note that we added a new property, `expensiveOrNot`, into the ontology; this property is not shown in Figure 5.2.) Our current (completed) camera ontology is given in List 5.12.

**LIST 5.12**
**Complete Camera Ontology**

```
//
// Camera.owl
// adding the initial definition for the properties
//
1: <?xml version="1.0"?>
2: <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
3:          xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
4:          xmlns:owl="http://www.w3.org/2002/07/owl#"
5:          xml:base="http://www.yuchen.net/photography/Camera.owl">

//
// classes definitions
//
```

```
6:    <owl:Class rdf:ID="Camera">
7:   </owl:Class>

8:   <owl:Class rdf:ID="Person">
9:   </owl:Class>

10:  <owl:Class rdf:ID="Digital">
11:     <rdfs:subClassOf rdf:resource="#Camera"/>
12:  </owl:Class>

13:  <owl:Class rdf:ID="Film">
14:     <rdfs:subClassOf rdf:resource="#Camera"/>
15:  </owl:Class>

16:  <owl:Class rdf:ID="SLR">
17:     <rdfs:subClassOf rdf:resource="#Digital"/>
18:  </owl:Class>

19:  <owl:Class rdf:ID="PointAndShoot">
20:     <rdfs:subClassOf rdf:resource="#Digital"/>
21:  </owl:Class>

22:  <owl:Class rdf:ID="Photographer">
23:     <rdfs:subClassOf rdf:resource="#Person"/>
24:  </owl:Class>

25:  <owl:Class rdf:ID="Specifications">
26:  </owl:Class>

27:  <owl:Class rdf:ID="Professional">
28:     <rdfs:subClassOf rdf:resource="#Photographer"/>
29:  </owl:Class>

30:  <owl:Class rdf:ID="Amateur">
31:     <rdfs:subClassOf rdf:resource="#Photographer"/>
32:  </owl:Class>

33:  <owl:Class rdf:ID="ExpensiveSLR">
34:    <rdfs:subClassOf rdf:resource="#SLR"/>
35:    <rdfs:subClassOf>
36:       <owl:Restriction>
37:          <owl:onProperty rdf:resource="#owned_by"/>
38:          <owl:someValuesFrom rdf:resource="#Professional"/>
39:       </owl:Restriction>
40:    </rdfs:subClassOf>
41:  </owl:Class>

//
// property definitions
//
42:  <owl:DatatypeProperty rdf:ID="expensiveOrNot">
43:    <rdfs:domain rdf:resource="#Digital"/>
```

```
44:    <rdfs:range
             rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
45:  </owl:DatatypeProperty>
46:  <rdfs:datatype
           rdf:about="http://www.w3.org/2001/XMLSchema#string"/>

47:  <owl:DatatypeProperty rdf:ID="model">
48:     <rdfs:domain rdf:resource="#Specifications"/>
49:     <rdfs:range
              rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
50:  </owl:DatatypeProperty>
51:  <rdfs:datatype
           rdf:about="http://www.w3.org/2001/XMLSchema#string"/>

52:  <owl:DatatypeProperty rdf:ID="pixel">
53:    <rdfs:domain rdf:resource="#Digital"/>
54:    <rdfs:range
             rdf:resource="http://www.someStandard.org#MegaPixel"/>
55:  </owl:DatatypeProperty>
56:  <rdfs:datatype rdf:about="http://www.someStandard.org#MegaPixel">
57:     <rdfs:subClassOf
              rdf:resource="http://www.w3.org/2001/XMLSchema#decimal"/>
58:  </rdfs:datatype>

59:  <owl:ObjectProperty rdf:ID="has_spec">
60:     <rdfs:domain rdf:resource="#SLR"/>
61:     <rdfs:range rdf:resource="#Specifications"/>
62:  </owl:ObjectProperty>

63:  <owl:ObjectProperty rdf:ID="owned_by">
64:     <rdfs:domain rdf:resource="#SLR"/>
65:     <rdfs:range rdf:resource="#Photographer"/>
66:  </owl:ObjectProperty>

67: </rdf:RDF>
```

At this point, we have just finished rewriting the ontology using OWL by adding the property definitions. OWL provides much richer features related to property definitions than we have employed thus far. We will discuss these features in detail in the next several sections, but here is a quick look at them:

- Property can be symmetric.
- Property can be transitive.
- Property can be functional.
- Property can be inverse functional.
- Property can be the inverse of another property.

## 5.4 USING OWL TO DEFINE PROPERTIES: PROPERTY CHARACTERISTICS

### 5.4.1 Symmetric Properties

A symmetric property describes the situation in which, if resource R1 is connected to resource R2 by property P, then resource R2 is also connected to resource R1 by the same property. For instance, we can define a property friend_with (for Person class), and if person A is friend_with person B, then person B is certainly friend_with person A. This is shown in List 5.13.

**LIST 5.13**
**Example of Symmetric Property**

```
1: <owl:ObjectProperty rdf:ID="friend_with">
2:    <rdf:type
          rdf:resource="http://www.w3.org/2002/07/owl
              #SymmetricProperty"/>
3:    <rdfs:domain rdf:resource="#Person"/>
4:    <rdfs:range rdf:resource="#Person"/>
5: </owl:ObjectProperty>
```

It is line 2 that indicates this property is a symmetric property. Adding this line means this: friend_with is a property. In our case, it is used to describe instances of class Person; its values are also instances of Person; and it is a symmetric property.

### 5.4.2 Enhanced Reasoning Power 4

***OWL inferencing power 4***

The agent sees this:

```
<Photographer rdf:ID=" http://www.yuchen.net/people#Liyang">
  <friend_with rdf:resource="http://www.yuchen.net/people#Jin"/>
</Photographer>
```

The agent understands:

Since Liyang is friend_with Jin, Jin must be friend_with Liyang.

Note that the value of the rdfs:domain property used when defining friend_with is Person; given that Photographer is a subclass of Person, it inherits the friend_with property. Therefore, when you describe a resource whose type is Photographer, you can use the friend_with property.

### 5.4.3 Transitive Properties

A transitive property describes the situation in which, if a resource `R1` is connected to resource `R2` by property `P`, and resource `R2` is connected to resource `R3` by the same property, then resource `R1` is also connected to resource `R3` by property `P`.

This can be a very useful feature in some cases. For our camera ontology, we created a class called `ExpensiveSLR` and another property called `expensiveOrNot` because photography is a very expensive hobby for many people. Given this, a better rule to decide which camera to buy is to get the one that offers a better ratio of quality to price. Consider an expensive camera having very superior quality and performance; the ratio could be high. On the other hand, a `PointAndShoot` camera, with a very appealing price, may not offer you much room to discover your creative side. We would like to capture this part of our knowledge in this specific domain by using a property that should be able to provide a way to compare two different cameras.

Let us define another new property called `betterQualityPriceRatio`; we will also declare it to be a transitive property: if camera `A` has `betterQualityPriceRatio` than camera `B`, and camera `B` has `betterQualityPriceRatio` than camera `C`, it should be true that camera `A` has `betterQualityPriceRatio` than camera `C`. List 5.14 shows the syntax we use in OWL to define such a property.

---

**LIST 5.14**
**Example of Transitive Property**

```
1: <owl:ObjectProperty rdf:ID="betterQualityPriceRatio">
2:    <rdf:type
          rdf:resource="http://www.w3.org/2002/07/owl
              #TransitiveProperty"/>
3:    <rdfs:domain rdf:resource="#Camera"/>
4:    <rdfs:range rdf:resource="#Camera"/>
5: </owl:ObjectProperty>
```

---

This syntax should look familiar to you; it is just like the symmetric property. All you need to do is to indicate that this property is a transitive property. You should be able to express the definition easily now.

### 5.4.4 Enhanced Reasoning Power 5

***OWL inferencing power 5***

The agent sees the following in one RDF instance file:

```
<SLR rdf:ID="NikonD70">
  <betterQualityPriceRation rdf:resource="NikonD50"/>
</SLR>
```

and in another RDF instance file, the agent find this:

```
<SLR
  rdf:about="http://www.yuchen.net/photography/myCameraInstance
      .rdf#NikonD50"
```

```
    xmlns="http://www.yuchen.et/photography/Camera.owl#">
     <betterQualityPriceRatio>
        <SLR rdf:about="http://www.someSite.net/otherCameraInstance
                 #Canon20D">
     </betterQualityPriceRatio>
  </SLR>
```

The agent understands the following:

`betterQualityPriceRatio` is a transitive property, and therefore:

`http://www.yuchen.net/photography/myCameraInstance.rdf#NikonD50` must have a `betterQualityPriceRatio` than `http://www.someSite.net/otherCameraInstance#Canon20D`.

The agent collected the information from two different instance files; yet it was able to draw the conclusions based on our camera ontology. In other words, the distributed information over the Internet was integrated and intelligent reasoning was done by the machine because we had expressed the knowledge in our ontology.

Also important is the correct use of the namespaces; it is another reason why the aforesaid reasoning is possible. Remember Rule #3 — you can talk about anything you want over the Internet, but you need to use the right URI; otherwise, you will be talking about something else.

In our example, the URI

```
http://www.yuchen.net/photography/myCameraInstance.rdf#NikonD50
```

represents a camera resource that we have described. Someone else, in his own instance file, added some extra information about the same camera instance by using the same URI. Also, he used another URI to represent another camera, namely:

```
http://www.someSite.net/otherCameraInstance#Canon20D
```

The agent is then able to draw conclusions, as shown earlier. By now, you must see the magic: yes, the information is distributed all over the place, but the URI connects them all.

An important conclusion:

> Reuse URIs as much as you can, especially when the following is true: if you know you are talking about some resource that has already been described by someone else using the vocabulary defined in some ontology files, and if you agree with the semantics expressed in the ontology files, then reuse the URI when you describe that resource; do not invent your own.

As an aside, the URI you have invented or are going to invent might be used again and again by people all over the world; so try to come up with a good namespace for it.

### 5.4.5 Functional Properties

A functional property describes the situation in which, for any given instance, there is at most one value for that property. In other words, it defines a many-to-one situation: there is at most one unique value for each instance.

Recall that in our camera ontology, we had defined a `pixel` property. Clearly, each digital camera has only one `pixel` value. Another example in this ontology is the `model` property: for each camera, we should be using only one `model` string to describe its model, for instance, "Nikon D70." On the other hand, there are many cameras whose model is "Nikon-D70."

Let us change the `model` definition to include this requirement, as shown in List 5.15.

**LIST 5.15**
**Example of `FunctionalProperty`**

```
1: <owl:DatatypeProperty rdf:ID="model">
2:   <rdf:type
         rdf:resource="http://www.w3.org/2002/07/owl
            #FunctionalProperty"/>
3 :  <rdfs:domain rdf:resource="#Specifications"/>
4:   <rdfs:range
           rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
5: </owl:DatatypeProperty>
```

### 5.4.6 Enhanced Reasoning Power 6

***OWL inferencing power 6***

The agent sees this in one RDF instance file:

```
<SLR
  rdf:about="http://www.yuchen.net/photography/myCameraInstance
      .rdf#NikonD70">
   <model rdf:datatype="http://www.w3.org/2001/XMLSchema#string>
   Nikon D70
   </model>
</SLR>
```

and in another RDF instance file, the agent finds this:

```
<SLR
  rdf:about="http://www.yuchen.net/photography/myCameraInstance
      .rdf#NikonD70"
  xmlns="http://www.yuchen.et/photography/Camera.owl#">
   <model rdf:datatype="http://www.w3.org/2001/XMLSchema#string>
   Nikon-D70
   </model>
</SLR>
```

The agent understands the following:

Because `model` is a functional property, and these two RDF statements describe the same resource, i.e., the URI for this resource is

```
http://www.yuchen.net/photography/myCameraInstance
.rdf#NikonD70
```

Therefore, `Nikon D70` and `Nikon-D70` have the same meaning.

These two descriptions collected by the agent are located in two different RDF documents. Without the functional property, the agent has no way of deciding that `Nikon D70` and `Nikon-D70` are in fact the same. You can even use `D70` or `D-70`, but they are all the same. This is not a big deal for humans, but for a machine it is a big achievement. If the agent happens to be a crawler of a search engine, you can imagine how helpful this ability will be.

### 5.4.7 Inverse Property

An inverse property describes the situation in which, if a resource `R1` is connected to resource `R2` by property `P`, then the inverse property of `P` will connect resource `R2` to resource `R1`.

A good example in our camera ontology is the property `owned_by`. Clearly, if a camera is `owned_by` a `Photographer`, then we can define an inverse property of `owned_by`, say, `own`, to indicate that the `Photographer owns` the camera. This example is given in List 5.16.

---

**LIST 5.16**
**Example of `owl:inverseOf` Property**

```
1: <owl:ObjectProperty rdf:ID="owned_by">
2:    <rdfs:domain rdf:resource="#SLR"/>
3:    <rdfs:range rdf:resource="#Photographer"/>
4: </owl:ObjectProperty>

5: <owl:ObjectProperty rdf:ID="own">
6:    <owl:inverseOf rdf:resource="#owned_by"/>
7:    <rdfs:domain rdf:resource="#Photographer"/>
8:    <rdfs:range rdf:resource="#SLR"/>
9: </owl:ObjectProperty>
```

---

### 5.4.8 Enhanced Reasoning Power 7

***OWL inferencing power 7***

The agent sees this in one RDF instance file:

```
<Photographer rdf:ID="Liyang">
  <own
  rdf:resource="http://www.yuchen.net/photography/myCameraInstance
      .rdf#NikonD70"/>
</Photographer>
```

It will add the following into its "knowledge base":

Subject: `#Liyang`
Predicate: `#own`
Object: `#NikonD70`

Once the agent realizes own is an inverse property of owned_by, it will add the following into the database, without you doing anything:

Subject: #NikonD70
Predicate: #owned_by
Object: #Liyang

### 5.4.9 Inverse Functional Property

Recall the functional property. It states that for a given rdfs:domain value, there is a unique rdfs:range value. For instance, for a given camera, there is only one model value. An inverse functional property, as its name suggests, is just the opposite of functional property; for a given rdfs:range value, the rdfs:domain value must be unique.

We can modify the own property to make it an inverse functional property (see List 5.17). This states that for a given SLR, there is a unique Photographer who owns it.

---

**LIST 5.17**
**Example of `InverseFunctionalProperty` Property**

```
1: <owl:ObjectProperty rdf:ID="owned_by">
2:   <rdfs:domain rdf:resource="#SLR"/>
3:   <rdfs:range rdf:resource="#Photographer"/>
4: </owl:ObjectProperty>

5: <owl:ObjectProperty rdf:ID="own">
6:   <rdf:type
  rdf:resource="http://www.w3.org/2002/07/owl#InverseFunctional
      Property"/>
7:    <owl:inverseOf rdf:resource="#owned_by"/>
8:    <rdfs:domain rdf:resource="#Photographer"/>
9:    <rdfs:range rdf:resource="#SLR"/>
10: </owl:ObjectProperty>
```

---

### 5.4.10 Enhanced Reasoning Power 8

***OWL inferencing power 8***

The agent sees this in one RDF instance file:

```
<Photographer rdf:ID="LiyangJin">
   <own
   rdf:resource="http://www.yuchen.net/photography/myCamera
       Instance.rdf#NikonD70"/>
</Photographer>
```

and in another RDF instance file, the agent finds this:

```
<Photographer rdf:ID="JinLiyang">
   <own
   rdf:resource="http://www.yuchen.net/photography/myCamera
       Instance.rdf#NikonD70"/>
</Photographer>
```

Because own is defined as an inverse functional property, the agent understands the following:

```
someNamespace1:LiyangJin = someNamespace2:JinLiyang
```

Now, at this point, you can understand how important this is. For example, the life of a search engine will be much easier. I will leave this to you as an exercise.

> Up to this point, we have covered the following OWL vocabulary: owl:Thing, owl:Class, owl:Restriction, owl:allValuesFrom, owl:hasValuesFrom, owl:hasValue, owl:cardinality, owl:minCardinality, owl:maxCardinality, owl:intersectionOf, owl:unionOf, owl:complementOf, owl:oneOf, owl:ObjectProperty, owl:DatatypeProperty, owl:SymmetricProperty, owl:TransitiveProperty, owl:FunctionalProperty, and owl:InverseFunctionalProperty.

### 5.4.11 Summary and Comparison

Let us summarize what we have learned about property classes in OWL. In fact, we can make a hierarchy structure of these property classes, as shown in Figure 5.3.

Based on Figure 5.3, we need to remember several things:

1. owl:SymmetricProperty and owl:TransitiveProperty are subclasses of owl:OjbectProperty; therefore, they can only be used to connect resources to resources.
2. owl:FunctionalProperty and owl:InverseFunctionalProperty can be used to connect resources to resources, or resources to an untyped literal (such as RDF schema literal) or an typed value (such as an XMLSchema data type).
3. owl:inverseOf is not included in Figure 5.3 because it is an OWL property, not a property class.

Up to this point we have covered all the major language features of OWL. To build a solid background about OWL so that you will be very well equipped to explore the world of the Semantic Web on your own, we still have to cover other OWL-related issues. These will be the topics of the next few sections.

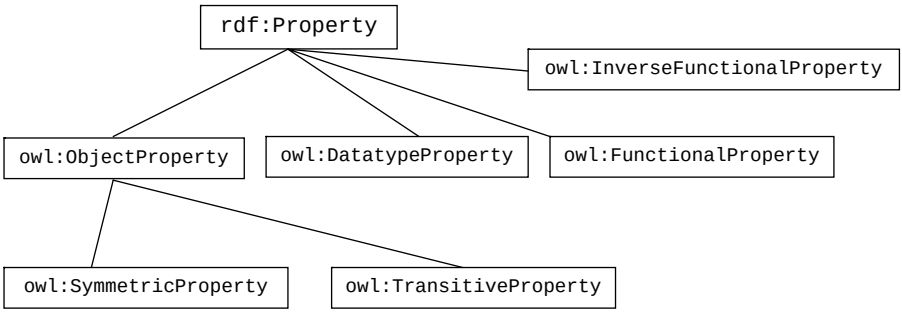


**FIGURE 5.3** Property class hierarchy.

## 5.5 ONTOLOGY MATCHING AND DISTRIBUTED INFORMATION

The integration of distributed information is always an exciting topic, and we know that the Semantic Web is going to help us handle the distributed information in a more efficient and global way. We have already had a foretaste of its capability in "OWL inferencing power 5," and we have summarized a simple rule: if appropriate, try to reuse URIs as much as possible to ensure your data or information will be nicely collected and understood, instead of always being distributed somewhere over the Internet.

However, what if you are simply unaware that a URI for the resource you are describing already exists and have invented your own URI?

When you describe the world using RDF statements, you are using a vocabulary written using RDF schema, or better, using OWL, and this vocabulary is the ontology you are using. Maybe someone has already built an ontology in the same domain, and you do not know of its existence; and later on, how are you going to indicate these two ontologies are somehow similar to each other?

As you can tell, these are very important topics for handling of distributed information, and the Semantic Web is all about distributed information. Fortunately, OWL provides some capabilities to solve these problems to some extent. We will examine these features next.

### 5.5.1 Defining Equivalent and Disjoint Classes

One way to make ontology matching easier (and enhance the automatic processing of distributed information) is to explicitly declare that two classes in two different ontologies are equivalent classes. OWL provides us a property called `owl:equivalentClass` to accomplish this.

Let us assume that after creating our camera ontology, we become aware that another ontology exists in the same domain, which among other things, has defined the following classes:

- `DigitalCamera`
- `SingleLensReflex`

After further examining the semantics in this ontology, it is clear that these two classes express the same meanings we intended to; therefore, we need to explicitly indicate that class `DigitalCamera` is equivalent to class `Digital`, and class `SingleLensReflex` is equivalent to class `SLR`, as shown in List 5.18.

**LIST 5.18**
**Example of Using a `owl:equivalentClass` Property**

```
1: <owl:Class rdf:ID="Digital">
2:   <rdfs:subClassOf rdf:resource="#Camera"/>
3:   <owl:equivalentClass
          rdf:resource="http://www.yetAnotherOne.com#DigitalCamera"/>
4: </owl:Class>

5: <owl:Class rdf:ID="SLR">
6:   <rdfs:subClassOf rdf:resource="#Digital"/>
7:   <owl:equivalentClass
          rdf:resource="http://www.yetAnotherOne.com#SingleLens
              Reflex"/>
8: </owl:Class>
```

Now, in any RDF document, if you have described an instance of type `SLR`, it is also an instance of type `SingleLensReflex`. It is not hard to imagine at this point that this declaration will greatly improve the accuracy or smartness of our agent when processing the distributed information over the Internet.

OWL also provides a way to define that two classes are not related in any way. For instance, in our camera ontology, we have defined `SLR` and `PointAndShoot` as subclasses of `Digital`. You might have noticed that an `SLR` camera in many cases can be simply used as a `PointAndShoot` camera. To avoid this confusion, we can define `SLR` to be disjoint from the `PointAndShoot` class, as shown in List 5.19.

**LIST 5.19**
**Example of Using `owl:disjointWith` Property**

```
1: <owl:Class rdf:ID="SLR">
2:   <rdfs:subClassOf rdf:resource="#Digital"/>
3:   <owl:equivalentClass
          rdf:resource="http://www.yetAnotherOne.com#SingleLens
              Reflex"/>
4:   <owl:disjointWith rdf:resource="#PointAndShoot"/>
5: </owl:Class>
```

Once the agent sees this definition, it will understand that any instance of `SLR` can never be an instance of the `PointAndShoot` camera at the same time. Also, note that `owl:disjointWith` by default is a symmetric property: if `SLR` is disjoint with `PointAndShoot`, then `PointAndShoot` is disjoint with `SLR`.

### 5.5.2 Distinguishing Instances in Different RDF documents

In fact, we can see that two instances are the same even when creating the instance files (RDF documents). For example, part of one instance document looks like this:

```
<SLR rdf:ID="NikonD70"
     xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
     xmlns="http://www.yuchen.net/photography/Camera.owl#">
  <owned_by rdf:resource="http://www.yuchen.net/people#Liyang"/>
```

In another RDF document, we find the following:

```
<SLR rdf:ID="Nikon-D70"
     xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
     xmlns="http://www.yuchen.net/photography/Camera.owl#">
  <owned_by rdf:resource="http://www.yuchen.net/people#Liyang"/>
  ... ...

</SLR>
```

The question the agent has is, are these two instances the same? We can use the `owl:sameIndividualAs` property to make it clear to the agent (we need to do this only in one document, assuming the URI of the preceding instance is `http://www.theURI.com#Nikon-D70`). This is shown in List 5.20.

---

**LIST 5.20**
**Example of Using `owl:sameIndividualAs` Property**

```
<SLR rdf:ID="Nikon-D70-Asian Version"
     xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
     xmlns:owl="http://www.w3.org/2002/07/owl#"
     xmlns="http://www.yuchen.net/photography/Camera.owl#">
  <owl:sameIndividualAs rdf:resource="http://www.theURI.com#Nikon-
D70"/>
  <owned_by rdf:resource="http://www.yuchen.net/people#Liyang"/>
  ... ...
</SLR>
```

---

This will make it clear to the agent that these two instances are the same. On the other hand, how do we indicate that the two instances are different? OWL provides another property to accomplish this, as shown in List 5.21.

---

**LIST 5.21**
**Example of Using `owl:differentFrom` Property**

```
<SLR rdf:ID="Nikon-D70-Asian Version"
     xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
     xmlns:owl="http://www.w3.org/2002/07/owl#"
     xmlns="http://www.yuchen.net/photography/Camera.owl#">
  <owl:differentFrom rdf:resource="http://www.theURI.com#Nikon-D70"/>
```

```
  <owned_by rdf:resource="http://www.yuchen.net/people#Liyang"/>
  ... ...
</SLR>
```

This is all good. But what if you forget to use the `owl:differentFrom` or the `owl:sameIndividualAs` property? A much easier way is to use `owl:AllDifferent` in the ontology file, as shown in List 5.22.

**LIST 5.22**
**Example of Using `owl:AllDifferent` Property**

```
... other definitions ...

<owl:AllDifferent>
  <owl:distinctMembers rdf:parseType="Collection">
    <SLR rdf:about="some-namespace-here#NikonD70"/>
    <SLR rdf:about="some-namespace-here#Nikon-D70-AsianVersion"/>
  </owl:distinctMembers>
</owl:AllDifferent>

... other definitions ...
```

This is clearly a good way to go and much easier to maintain. If you have more instances that are different from each other, you do not have to use `owl:differentFrom` in every single RDF document; you can just make the change in this one place.

> Up to this point, we have covered the following OWL vocabulary: `owl:Thing`, `owl:Class`, `owl:Restriction`, `owl:allValuesFrom`, `owl:hasValuesFrom`, `owl:hasValue`, `owl:cardinality`, `owl:minCardinality`, `owl:maxCardinality`, `owl:intersectionOf`, `owl:unionOf`, `owl:complementOf`, `owl:oneOf`, `owl:ObjectProperty`, `owl:DatatypeProperty`, `owl:SymmetricProperty`, `owl:TransitiveProperty`, `owl:FunctionalProperty`, `owl:InverseFunctionalProperty`, `owl:equivalentClass`, `owl:disjointWith`, `owl:sameIndividualAs`, `owl:differentFrom`, `owl:AllDifferent`, and `owl:distinctMembers`.

## 5.6 OWL ONTOLOGY HEADER

Before we end, we need to discuss the header of OWL documents. OWL documents are more often called OWL ontologies. They are also RDF documents, which is why the root element of an OWL ontology is always an `rdf:RDF` element.

A typical header part of an OWL ontology is shown in List 5.23. The `owl:Ontology` class is new here. Normally, an OWL ontology starts with a collection of assertions for housekeeping purposes and these statements are grouped under `owl:Ontology`, as shown in lines 6 to 12.

**LIST 5.23**
**OWL Ontology Header**

```
1: <?xml version="1.0"?>
2: <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
3:          xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
4:          xmlns:owl="http://www.w3.org/2002/07/owl#"
5:          xml:base="http://www.yuchen.net/photography/Camera.owl">

6:  <owl:Ontology
                rdf:about="http://www.yuchen.net/photography/
                     Camera.owl">
7:   <rdfs:comment>our camera ontology</rdfs:comment>
8:   <rdfs:label>Camera ontology</rdfs:label>
9:   <owl:priorVersion
          rdf:resource=http://www.yuchen.net/photography/Camera0.owl"/>
10:  <owl:versionInfo>Camera.owl 0.2</owl:versionInfo>
11:  <owo:imports
         rdf:resource="http://www.somedomain.org/someOnt.owl"/>
12: </owl:Ontology>
```

Among these lines, `rdfs:comment`, `rdfs:label`, `owl:priorVersion`, and `owl:versionInfo` are for humans; the only statement that means anything to the parser is the `owl:imports` statement. It includes other ontologies whose contents are assumed to be part of the current ontology; in other words, imported ontologies provide definitions that can be used directly.

> Up to this point, we have covered the following OWL vocabulary: `owl:Thing`, `owl:Class`, `owl:Restriction`, `owl:allValuesFrom`, `owl:hasValuesFrom`, `owl:hasValue`, `owl:cardinality`, `owl:minCardinality`, `owl:maxCardinality`, `owl:intersectionOf`, `owl:unionOf`, `owl:complementOf`, `owl:oneOf`, `owl:ObjectProperty`, `owl:DatatypeProperty`, `owl:SymmetricProperty`, `owl:TransitiveProperty`, `owl:FunctionalProperty`, `owl:InverseFunctionalProperty`, `owl:equivalentClass`, `owl:disjointWith`, `owl:sameIndividualAs`, `owl:differentFrom`, `owl:AllDifferent`, `owl:distinctMembers`, `owl:Ontology`, `owl:priorVersion`, `owl:versionInfo`, `owl:imports`, `rdfs:comment`, and `rdfs:label`.

## 5.7 FINAL `Camera` ONTOLOGY REWRITTEN IN OWL

Now that we have discussed all the language constructs of OWL, it is time to finally complete our project: rewrite the camera ontology using OWL.

### 5.7.1 `Camera` Ontology

In the previous sections, to discuss the details of different OWL language features, we made our camera ontology quite complex, especially the property definitions. In

our final product, though, we are not going to include all these features. The complete final camera ontology is given in List 5.24.

---

**LIST 5.24**
**Final `Camera` Ontology Written in OWL**

```
//
// Camera.owl
//
1:  <?xml version="1.0"?>
2:  <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
3:           xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
4:           xmlns:owl="http://www.w3.org/2002/07/owl#"
5:           xml:base="http://www.yuchen.net/photography/Camera.owl">

6:  <owl:Ontology
        rdf:about="http://www.yuchen.net/photography/Camera.owl">
7:    <rdfs:comment>our final camera ontology</rdfs:comment>
8:    <rdfs:label>Camera ontology</rdfs:label>
9:    <owl:versionInfo>Camera.owl 1.0</owl:versionInfo>
10: </owl:Ontology>

//
// classes definitions
//

11: <owl:Class rdf:ID="Camera">
12: </owl:Class>

13: <owl:Class rdf:ID="Person">
14: </owl:Class>

15: <owl:Class rdf:ID="Film">
16:   <rdfs:subClassOf rdf:resource="#Camera"/>
17: </owl:Class>

18: <owl:Class rdf:ID="Digital">
19:   <rdfs:subClassOf rdf:resource="#Camera"/>
20:   <owl:equivalentClass
          rdf:resource="http://www.yetAnotherOne.com#DigitalCamera"/>
21: </owl:Class>

22: <owl:Class rdf:ID="SLR">
23:   <rdfs:subClassOf rdf:resource="#Digital"/>
24:   <owl:equivalentClass
          rdf:resource="http://www.yetAnotherOne.com#SingleLens
              Reflex"/>
25:   <owl:disjointWith rdf:resource="#PointAndShoot"/>
26: </owl:Class>

27: <owl:Class rdf:ID="PointAndShoot">
```

```
28:   <rdfs:subClassOf rdf:resource="#Digital"/>
29: </owl:Class>

30: <owl:Class rdf:ID="Photographer">
31:   <rdfs:subClassOf rdf:resource="#Person"/>
32: </owl:Class>

33: <owl:Class rdf:ID="Specifications">
34: </owl:Class>

35: <owl:Class rdf:ID="Professional">
36:   <rdfs:subClassOf rdf:resource="#Photographer"/>
37:   <owl:disjointWith rdf:resource="#Amateur"/>
38: </owl:Class>

39: <owl:Class rdf:ID="Amateur">
40:   <rdfs:subClassOf rdf:resource="#Photographer"/>
41: </owl:Class>

42: <owl:Class rdf:ID="ExpensiveSLR">
43:   <rdfs:subClassOf rdf:resource="#SLR"/>
44:   <rdfs:subClassOf>
45:      <owl:Restriction>
46:         <owl:onProperty rdf:resource="#owned_by"/>
47:         <owl:someValuesFrom rdf:resource="#Professional"/>
48:      </owl:Restriction>
49:   </rdfs:subClassOf>
50:   <rdfs:subClassOf>
51:      <owl:Restriction>
52:         <owl:onProperty rdf:resource="#expensiveOrNot"/>
53:         <owl:hasValue
                rdf:datatype="http://www.w3.org/2001/XMLSchema
                    #string">
            expensive
            </owl:hasValue>
54:      </owl:Restriction>
55:   </rdfs:subClassOf>
56: </owl:Class>
57:

58: <owl:AllDifferent>
59:   <owl:distinctMembers rdf:parseType="Collection">
60:   </owl:distinctMembers>
61: </owl:AllDifferent>

//
// property definitions
//

62: <owl:DatatypeProperty rdf:ID="expensiveOrNot">
63:   <rdfs:domain rdf:resource="#Digital"/>
64:   <rdfs:range
```

```
            rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
65: </owl:DatatypeProperty>

66: <rdfs:datatype
          rdf:about="http://www.w3.org/2001/XMLSchema#string"/>

67: <owl:DatatypeProperty rdf:ID="model">
68:   <rdf:type
         rdf:resource="http://www.w3.org/2002/07/owl#Functional
             Property"/>
69:   <rdfs:domain rdf:resource="#Specifications"/>
70:   <rdfs:range
            rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
71: </owl:DatatypeProperty>

72: <owl:DatatypeProperty rdf:ID="pixel">
73:   <rdfs:domain rdf:resource="#Digital"/>
74:   <rdfs:range
            rdf:resource="http://www.someStandard.org#MegaPixel"/>
75: </owl:DatatypeProperty>

76: <rdfs:datatype rdf:about="http://www.someStandard.org#MegaPixel">
77:   <rdfs:subClassOf
            rdf:resource="http://www.w3.org/2001/XMLSchema#decimal"/>
78: </rdfs:datatype>

79: <owl:ObjectProperty rdf:ID="has_spec">
80:   <rdfs:domain rdf:resource="#SLR"/>
81:   <rdfs:range rdf:resource="#Specifications"/>
82: </owl:ObjectProperty>

83: <owl:ObjectProperty rdf:ID="owned_by">
84:   <rdfs:domain rdf:resource="#SLR"/>
85:   <rdfs:range rdf:resource="#Photographer"/>
86: </owl:ObjectProperty>

87: <owl:ObjectProperty rdf:ID="own">
88:   <owl:inverseOf rdf:resource="#owned_by"/>
89:   <rdfs:domain rdf:resource="#Photographer"/>
90:   <rdfs:range rdf:resource="#SLR"/>
91: </owl:ObjectProperty>

92: <owl:ObjectProperty rdf:ID="friend_with">
93:   <rdf:type
          rdf:resource="http://www.w3.org/2002/07/owl#Symmetric
              Property"/>
94:   <rdfs:domain rdf:resource="#Person"/>
95:   <rdfs:range rdf:resource="#Person"/>
96: </owl:ObjectProperty>

97: <owl:ObjectProperty rdf:ID="betterQualityPriceRatio">
98:   <rdf:type
```

```
        rdf:resource="http://www.w3.org/2002/07/owl#Transitive
            Property"/>
99:   <rdfs:domain rdf:resource="#Camera"/>
100:  <rdfs:range rdf:resource="#Camera"/>
101: </owl:ObjectProperty>

102: </rdf:RDF>
```

This is the camera ontology written in OWL, and it is indeed quite impressive. For instance, we used `owl:versionInfo` to nicely identify the version number of this ontology, and we have added several new properties and imposed constraints on some classes. Also, note that lines 58 to 61 leave some space for us to later on declare the instances that should be treated differently. But what does this ontology tell us? What is the semantics encoded in it? Let us take a closer look.

### 5.7.2 Semantics of the OWL Camera Ontology

1. Our camera ontology defines a set of concepts or classes in the domain of photography. It tells us the following by defining these classes:
   - `Camera` is a class, and `Person` is a class.
   - `Film` and `Digital` are subclasses of `Camera`. Therefore, they are special types of `Cameras`.
   - `SLR` and `PointAndShoot` are subclasses of `Digital`. Therefore, they are special types of `Digital` cameras and, certainly, they are also `Cameras`.
   - `ExpensiveSLR` is a subclass of `SLR`. Therefore, it is a special kind of `SLR`; it is a `Digital` camera; and it is a `Camera` in general.
   - `Photographer` is a subclass of `Person`. Therefore, it is a special kind of `Person`.
   - `Professional` and `Amateur` are subclasses of `Photographer`. Therefore, they are all `Photographers`, and they are also `Person` in general.
   - `Specifications` is another class or concept in our camera ontology.
2. Our camera ontology also defines some more details about these classes:
   - The `Digital` class in this ontology is the same as the `DigitalCamera` concept defined in another ontology.
   - The `SLR` class in this ontology is the same as the `SingleLensReflex` class defined in another ontology.
   - An instance of `SLR` cannot be an instance of `PointAndShoot` at the same time; these two classes have no overlap of any kind.
   - An instance of `Professional` cannot be an instance of `Amateur` at the same time; these two classes have no overlap of any kind.
3. Our camera ontology defines a set of properties, and these properties are used to relate class to class or class to values. This has added considerable semantics to our ontology:
   - A property called `owned_by` is defined. It is used to relate the classes `SLR` and `Photographer`, meaning that an instance of `SLR` is `owned_by` an instance of `Photographer`.

- A property called own is defined. It is the inverse property of owned_by, meaning that an instance of Photographer can own an instance of SLR.
- A property called friend_with is defined. It is used to relate Person class to itself, meaning that an instance of Person can be friend_with another instance of Person. It is also defined to be a symmetric property; therefore, for two given instances of Person, say, P1 and P2, if P1 is friend_with P2, then P2 is friend_with P1.
- A property called betterQualityPriceRatio is defined. It is used to relate Camera class to itself, meaning that one instance of Camera class can have betterQualityPriceRatio than another. It is also defined to be a transitive property; therefore, for three given instances of Camera classes, say, C1, C2, and C3, if C1 is betterQualityPriceRatio C2 and C2 is betterQualityPriceRatio C3, then C1 is betterQualityPriceRatio C3.
- A property called has_spec is defined. It is used to relate SLR class to Specifications class, meaning that an instance of SLR has an instance of Specifications class as its specification.
- A property called pixel is defined. It is used to relate Digital class to some typed value, meaning that an instance of Digital has some pixel value.
- A property called model is defined. It is used to relate Specifications class to some typed value, meaning that an instance of Specifications has some model value. Also, model is defined to be a functional property; i.e., for any instance of Specifications class, there can be at most one model value.
- A property called expensiveOrNot is defined. It is used to relate Digital class to some typed value, meaning that an instance of Digital class has some expensiveOrNot value.

4. Given all the defined classes and properties, our camera ontology further uses some properties to put constraints on some classes to express more complex knowledge, as shown:
   - For any instance of ExpensiveSLR, its owned_by property can have multiple values, but at least one of these values has to be an instance of Professional class.
   - For any instance of ExpensiveSLR, its expensiveOrNot property always has to be the following:

     http://www.yuchen.net/photography/Camera.owl#expensive.

That is it! Our camera ontology, with just over 100 lines, has expressed so much knowledge about a specific domain; this would certainly make the agent's work much easier. You should be able to understand at this point how all this knowledge can help the agent. Refresh your memory by reading the "enhanced inferencing power" sections. You will see the enhanced reasoning power in subsequent chapters.

## 5.8 THREE FACES OF OWL

Now that we have seen all the constructs in OWL and finished our project, it is time to see the three faces of OWL. Let us understand the need for three faces and what they are. We will answer all these questions in this section.

### 5.8.1 Why Do We Need This?

To make this clear, we need to review some history first. As we already know, the expressiveness of RDF and RDF schema is very limited. RDF is the instance document that contains RDF statements, and RDF schema provides the vocabulary needed for RDF documents. RDF schema is quite simple: it defines a class hierarchy and a property hierarchy with domain and range constraints on the properties.

After the release of RDF schema, however, the Web Ontology Working Group of W3C (`http://www.w3.org/2001/sw/WebOnt/`) soon identified a number of characteristic use cases for constructing ontologies that would indeed require much more expressiveness than the RDF schema. For example:

- There is no way to declare equivalent and disjoint classes. Equivalent classes are very useful when two or more ontologies are involved or compared. Disjointedness is also important; for instance, `Male` and `Female` classes have to be disjoint. In RDF schema, only subclass relationships can be stated.
- RDF schema does not allow the concepts of union, intersection, and complement. These concepts are useful when building classes not just by inheritance, but also by Boolean combinations of other classes, such as combining two classes by using union concept.
- RDF schema does not allow cardinality restrictions on properties. In many cases, it is important to be able to decide how many distinct values a given property may or must take. A common case would be an e-mail account, which should belong to exactly one `Person`.
- RDF schema does not provide any mechanism to localize the scope of a property; once an `rdfs:range` is defined, it has to be true for the class (and all its subclasses) defined in the `rdfs:domain` field. As we have seen in our camera ontology, some property of the `ExpensiveSLR` class should take some localized values instead of every possible value defined in the `rdfs:range`.
- RDF does not define special characteristics of properties. This is obvious in our camera ontology: we have defined symmetric, transitive, and functional properties to make it more powerful.

Realizing this need, several research groups in both America and Europe launched a joint effort to develop a more powerful ontology modeling language. The result is the DAML+OIL language [35]. A few words about this name: the American proposal is DAML-ONT [36] and the European is OIL [37]; the joint name is DAML+OIL.

DAML+OIL has been taken as the starting point for the W3C Web Ontology Working Group for development of the language we discussed here, OWL. OWL is intended to be the standardized ontology language of the Semantic Web.

An important issue when designing the ontology language is the trade-off between the expressiveness and the efficiency of the reasoning process. In other words, it is generally true that the richer the language, the more inefficient the reasoning; sometimes, the reasoning can become complex enough to be computationally impossible. The goal therefore is to design a language that is sufficiently expressive for large ontologies and also simple enough to be supported by reasonably efficient reasoning engines.

Unfortunately, in the case of OWL, though some of its constructs are very expressive, they lead to uncontrollable computational complexities. This trade-off between reasoning efficiency and expressiveness has led the W3C Working Group to the definitions of three different subsets of OWL, each of which is aimed at a different level of this trade-off.

Now that we understand why there are three definitions of OWL, we can take a look at each of them.

### 5.8.2 The Three Faces

#### 5.8.2.1 OWL Full

The entire OWL language we discussed in this chapter is called OWL Full. Every construct we covered in this chapter is available to the ontology developer. It also allows combining these constructs in arbitrary ways with RDF and RDF schema, including mixing the RDF schema definitions with OWL definitions. Any legal RDF document is a legal OWL Full document.

The advantage of OWL Full is obvious: you have everything at your disposal, and you enjoy very convenient expressiveness when developing your ontology. The disadvantage is that the ontology can become so powerful as to be computationally expensive to provide complete reasoning support; efficiency is another factor to consider.

#### 5.8.2.2 OWL DL

OWL DL is short for OWL Description Logic. It is a sublanguage of OWL Full and has restrictions about how the constructs from OWL and RDF can be used. More specifically, the following rules must be observed when building ontologies:

- No arbitrary combination is allowed: Any resource can be only a class, a data type, a data type property, an object property, an instance, or a data value, and not more than one of these. In other words, a class cannot be at the same time a member of another class.
- Restrictions on functional property and inverse functional property: These two properties are subclasses of `rdf:Property`; therefore, they can connect resource to resource or resource to value. However, in OWL DL,

they can only be used with the object property, and not with the datatype property.
- Restriction on transitive property: You cannot use `owl:cardinality` with the transitive property, or their subproperties; these subproperties are transitive properties by implication.
- Restriction on `owl:imports`: If you are developing an OWL DL ontology but are also using `owl:imports` to import an OWL Full ontology, your ontology will not be qualified as an OWL DL.

Clearly, the advantage is that OWL DL permits a quicker response from the reasoning engine and, also, the reasoning engine itself is easier to build. The disadvantage is that you do not have the expressiveness or the convenience provided by OWL Full.

#### 5.8.2.3 OWL Lite

OWL Lite is a further restricted subset of OWL DL:

- The following constructs are not allowed in OWL Lite: `owl:hasValue`, `owl:disjointWith`, `owl:unionOf`, `owl:complementOf`, and `owl:oneOf`.
- Cardinality constraints are more restricted: You cannot use `owl:minCardinality` or `owl:maxCardinality`. You can still use `owl:cardinality`, but the value is restricted to either 0 or 1.
- `owl:equivalentClass` statement can no longer be used to relate anonymous classes, but only to connect class identifiers.

The advantage is again efficiency on the reasoning side, both for the users and the tool builders. The disadvantage is, of course, the loss of even more expressive power.

Now, let us decide the face of our camera ontology. Well, it is not OWL Lite, because we did use `owl:hasValue`; it is also not OWL DL as we also used functional property on `owl:DatatypeProperty`. Therefore, our camera ontology is an OWL Full version ontology.