Chapter 5
OWL: Web Ontology Language

This chapter is a natural extension of Chap. 4. As a key technical component in the world of the Semantic Web, the Web Ontology Language OWL is the most popular language to use when creating ontologies. In this chapter, we cover OWL in great detail, and after finishing this chapter, you will be quite comfortable when it comes to defining ontologies using OWL.

5.1 OWL Overview

5.1.1 OWL in Plain English

OWL is currently the most popular language to use when creating ontologies. Since we have already established a solid understanding about RDF Schema, understanding OWL becomes much easier.

The purpose of OWL is exactly the same as RDF Schema: to define ontologies that include classes, properties and their relationships for a specific application domain. When anyone wants to describe any resource, these terms can be used in the published RDF documents; therefore, everything we say, we have a reason to say it. And furthermore, a given application can implement reasoning processes to discover implicit or unknown facts with the help of the ontologies.

However, compared to RDF schema, OWL provides us with the capability to express much more complex and richer relationships. Therefore, we can construct applications with a much stronger reasoning ability. For this reason, we often want to use OWL for the purpose of ontology development. RDF Schema is still a valid choice, but it's obvious limitation compared to OWL will always make it a second choice.

In plain English, we can define OWL as follows:

L. Yu, *A Developer's Guide to the Semantic Web*,
DOI 10.1007/978-3-662-43796-4_5

$$\text{OWL} = \text{RDF Schema} + \text{new constructs for better expressiveness}$$

And remember, since OWL is built upon RDF Schema, all the terms contained in RDFS vocabulary can still be used when creating OWL documents.

Before we move on to the official definition of OWL, let us spend a few lines on its interesting acronym. Clearly, the natural acronym for Web Ontology Language would be WOL instead of OWL. The story dates back to December 2001, the days when the OWL group was working on OWL. Prof. Tim Finin, in an e-mail dated December 27, 2001,[1] suggested the name OWL based on these considerations: OWL has just one obvious pronunciation that is also easy on the ear, it yields good logos, it suggests wisdom and it can be used to honor the *One World Language* project, an Artificial Intelligence project at MIT in the mid 1970s. The name OWL since then has been accepted as it formal name.

5.1.2 OWL in Official Language: OWL 1 and OWL 2

Behind the development of OWL, there is actually quite a long history that dates back to the 1990s. Back then, a number of research efforts were set up to explore how the idea of *knowledge representationSee* KR (KR) from the area of Artificial Intelligence*See* AI (AI) could be used on the Web to make machines understand its content. These efforts resulted in a variety of languages. Among them, noticeably two languages called Simple HTML Ontology Extensions (SHOE*See* SHOE) and Ontology Inference Layer (OIL*See* OIL) later on become part of the foundation of OWL.

Meanwhile, another project named the DARPA Agent Markup Language*See* DAML, (DAML, where *DARPA* represents US Defense Advanced Research Projects Agency) was started in late 1990 with the goal of creating a machine-readable representation for the Web. The main outcome of the DAML project was the DAML language, an agent markup language based on RDF.

Based on DAML, SHOE and OIL, a new Web ontology language named DAML +OIL was developed by a group called "US/UK ad hoc Joint Working Group on Agent Markup Languages". This group was jointly funded by DARPA under the DAML program and the European Union's Information Society Technologies (IST) funding project. DAML+OIL since then has become the whole foundation of OWL language.

The OWL language started as a research-based revision of the DAML+OIL web ontology language. W3C created the Web Ontology Working Group[2] in November 2001, and the first working drafts of the abstract syntax, reference and synopsis were published in July 2002. The OWL documents became a formal W3C

[1] http://lists.w3.org/Archives/Public/www-webont-wg/2001Dec/0169.html

[2] http://www.w3.org/2007/OWL/wiki/OWL_Working_Group

recommendation on February 10, 2004. The recommendation includes the following documents[3]:

- OWL Web Ontology Language Overview
- OWL Web Ontology Language Guide
- OWL Web Ontology Language Reference
- OWL Web Ontology Language Semantics and Abstract Syntax
- OWL Web Ontology Language Test Cases
- OWL Web Ontology Language Use Cases and Requirements

The standardization of OWL has since then sparked the development of OWL ontologies in a number of fields including medicine, biology, geography, astronomy, defense and the aerospace industry. For example, in the life sciences community, OWL is extensively used and has become a de facto standard for ontology development and data exchange.

On the other hand, the numerous contexts in which OWL language has been applied have also revealed its deficiencies from a user's point of view. For instance, ontology engineers have identified some major limitations of its expressiveness, which is obviously needed for real development work. Also, OWL tool designers have come up with their list of some practical limitations of the OWL language as well.

In response to these comments and requests from the real users, it was decided that an incremental revision of OWL was needed, and was provisionally called OWL 1.1. Accordingly, the initial version of OWL language is referred to as OWL 1.

The 2005 OWL Experiences and Directions Workshop[4] created a list of new features to be provided by OWL 1.1. The actual development of these new features was then undertaken by an informal group of language users and developers. The deliverables of their work were submitted to W3C as a member submission, and as the same time, a new W3C OWL Working Group[5] was officially formed in September 2007.

Under the auspices of the Working Group, the original member submission has evolved significantly. In April 2008, the Working Group decided to call the new language OWL 2, and the initial 2004 OWL standard continues to be called OWL 1.

On October 27, 2009, OWL 2 became a W3C standard,[6] with the following core specifications:

- OWL 2 Web Ontology Language Structural Specification and Functional-Style Syntax
- OWL 2 Web Ontology Language Mapping to RDF Graphs
- OWL 2 Web Ontology Language Direct Semantics

[3] http://www.w3.org/2004/OWL/#specs

[4] http://www.mindswap.org/2005/OWLWorkshop/

[5] http://www.w3.org/2007/OWL/wiki/OWL_Working_Group

[6] http://www.w3.org/TR/2009/REC-owl2-overview-20091027/

- OWL 2 Web Ontology Language RDF-Based Semantics
- OWL 2 Web Ontology Language Conformance
- OWL 2 Web Ontology Language Profiles

These core specifications are part of the W3C OWL 2 Recommendations, and they are mainly useful for ontology tool designers. For example, in order to implement an OWL 2 validator or a reasoner that understands OWL 2, one has to be familiar with these specifications. For developers like us, this chapter will help you to learn how to use OWL 2 to develop your own ontology documents.

With the understanding of the history behind OWL, let us take a look at its official definition. W3C's OWL 2 Primer[7] has given a good definition of OWL:

> The W3C OWL 2 Web Ontology LanguageSee OWL (OWL) is a Semantic Web language designed to represent rich and complex knowledge about things, groups of things, and relations between things. OWL is a computational logic-based language such that knowledge expressed in OWL can be reasoned with by computer programs either to verify the consistency of that knowledge or to make implicit knowledge explicit.

If you don't fully understand this definition at this point, rest assured that it will gradually shape up during the course of this chapter. In this chapter, we will cover the details of OWL language, and you will be able to define ontologies on your own, and understand existing ontologies that are written by using OWL.

5.1.3 From OWL 1 to OWL 2

With the discussion of OWL history in place, we understand OWL 2 is the latest standard from W3C. In fact, OWL 1 can be considered as a subset of OWL 2, and all the ontologies that are created by using OWL 1 will be recognized and understood by any application that can understand OWL 2.

However, OWL 1 still plays a special role, largely for some historical reasons. More specifically, up to this point, most practical-scale and well-known ontologies are written in OWL 1, and most ontology engineering tools, including development environments for the Semantic Web, are equipped with the ability to understand OWL 1 ontologies only. Therefore, in this chapter, we will make a clear distinction between OWL 1 and OWL 2: the language constructs of OWL 1 will be covered first, followed by the language constructs of OWL 2. Once you have finished the part about OWL 1, you should be able to understand most of the ontologies in the real Semantic Web world. With a separate section covering OWL 2, you can get a clear picture about what has been improved since OWL 1.

Also notice that for the rest of this chapter, we will use the names OWL and OWL 2 interchangeably. We will always explicitly state OWL 1 if necessary.

[7] http://www.w3.org/TR/2009/REC-owl2-primer-20091027/

5.2 OWL 1 and OWL 2: The Big Picture

Form this point through the rest of this chapter, we will cover the syntax of OWL, together with examples. Our goal is to rewrite our camera vocabulary developed in Chap. 4, and by doing so, we will cover most of the OWL language features.

Similar to RDFS, OWL can be viewed as a collection of terms we can use to define classes and properties for a specific application domain. These predefined OWL terms all have the following URI as their leading string (applicable to both OWL 1 and OWL 2),

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

and by convention, this URI prefix string is associated with namespace prefix `owl:`, and is typically used in RDF/XML documents with the prefix `owl`.

For the rest of this section, we will discuss several important concepts related to OWL, so we will be ready for the rest of this chapter.

5.2.1 Basic Notions: Axiom, Entity, Expression and IRI Names

An *axiom* is a basic statement that an OWL ontology has. It represents a basic piece of knowledge. For example, a statement like "the `Digital` camera class is a subclass of the `Camera` class" is an axiom. Clearly, any given OWL ontology can be viewed as a collection of axioms. Furthermore, this ontology asserts that all its axioms are true.

Clearly, each axiom, as a statement, will have to involve some class, some property and sometimes, some individual. For example, one axiom can claim that a Nikon D300 camera is an individual of class `Digital` camera, and another axiom can state that a `Photographer` individual can `own` a given camera. These classes, properties and individuals can be viewed as the atomic constituents of axioms, and these atomic constituents are also called *entities*. Sometimes, in OWL, individual entity is also called object, and class entity is called category and property entity is called relation.

As we will see in this chapter, a key feature of OWL is to combine different class entities and/or property entities to create new class entities and property entities. The combinations of entities to form complex descriptions about new entities are called *expressions*. In fact, expressions are a main reason why we claim OWL language has a much more enhanced expressiveness compared to ontology language such as RDFS.

The last concept we would like to mention here is the *IRI* names, which you will encounter when reading OWL 2-related literature.

As we know at this point, URIs are the standard mechanism for identifying resources on the Web. For the vision of the Semantic Web, we have been using

URIs to represent classes, properties and individuals, as shown in Chaps. 2 and 4. This URI system fits well into the Semantic Web for the following two main reasons:

1. It provides a mechanism to uniquely identify a given resource.
2. It specifies a uniform way to retrieve machine-readable descriptions about the resource being identified by the URI.

The first point here should be fairly clear (refer to Chap. 2 for details), and the second point will be covered in detail in Chap. 9.

Internationalized Resource Identifiers (IRIs)*See* IRI are just like URIs except that they can make use of the whole range of Unicode characters. As a comparison, URIs are limited to the ASCII subset of the characters, which only has 127 characters. In addition, the ASCII subset itself is based on the needs of English-speaking users, which presents some difficulty for non-English users. And these considerations have been the motivation of IRIs.

There are standard ways to convert IRIs to URIs, and vice versa. Therefore, an IRI can be coded into a URI, which is quite helpful when we need to use the IRI in a protocol that accepts only URIs (such as the HTTP protocol).

For our immediate purpose in this chapter, IRIs are interchangeable with URIs, and there is not much need to make a distinction between these two. However, understanding IRIs is helpful, especially if you are doing development using a language other than English.

5.2.2 Basic Syntax Forms: Functional-Style, RDF/XML Syntax, Manchester Syntax and XML Syntax

OWL specifications provide various syntaxes for persisting, sharing and editing ontologies. These syntaxes could be confusing for someone new to the language. In this section, we give a brief description of each syntax form so you understand which one will work the best for your needs.

- Functional-Style syntax

It is important to realize the fact that the OWL language is not defined by using a particular concrete syntax, but rather it is defined in a high-level structural specification that is then mapped into different concrete syntaxes. By doing so, it is possible to clearly describe the essential language features without getting into the technical details of exchange formats.

Once the structural specification is complete, it is necessary to move toward some concrete syntaxes. The first step of doing so is the Functional-Style syntax. This syntax is designed for translating the structural specification to various other syntaxes, and it is often used by OWL tool designers. In general, it is not intended to

be used as an exchange syntax, and as OWL language users, we will not be seeing or using this syntax often.

- RDF/XML syntax

This is the syntax we are familiar with, and it is also the sterilization format we have been using throughout the book. Most well-known ontologies written in OWL 1 use this syntax as well. In addition, this is the only syntax that is mandatory to be supported by all OWL tools. Therefore, as a developer, you should be familiar with this syntax. We will be using this syntax for the rest of this chapter as well.

- Manchester syntax

The Manchester syntax provides a textual-based representation of OWL ontologies that is easy to read and write. The motivation behind Manchester syntax was to design a syntax that could be used for editing class expressions in tools such as Protégé and the like. Since it is quite successful in these tools, it has been extended to represent a complete ontology.

Manchester syntax is fairly easy to learn, and it has a compact format that is easy to read and write as well. We will not be using this format in this book. With what you will learn from using the RDF/XML format, understanding Manchester syntax will not present too much of a challenge at all.

- OWL/XML

Although RDF/XML syntax is the normative format specified by the W3C OWL standard, it is not easy to work with. More specifically, it is difficult to use existing XML tools for tasks other than parsing and rendering it. Even standard XML tools such as Xpath and XSLT will not work well with RDF/XML representations of ontologies. In order to take advantage of existing XML tools, a more regular and simple XML format is needed. OWL/XML is such a format for representing OWL ontologies. Its main advantage is that it conforms to an XML schema, and therefore it is possible to use existing XML tools such as Xpath and XSLT for processing and querying tasks. In addition, parsing can be done more easily than in RDF/XML syntax.

In this book, we will not use this format. Again, once you are familiar with the RDF/XML format, understanding OWL/XML syntax will not be too hard.

5.3 OWL 1 Web Ontology Language

In this section, we concentrate on the language constructs offered by OWL 1. Again, all these constructs are now part of OWL 2, and any ontology created by using OWL 1 will continue to be recognized and understood by any application that understands OWL 2.

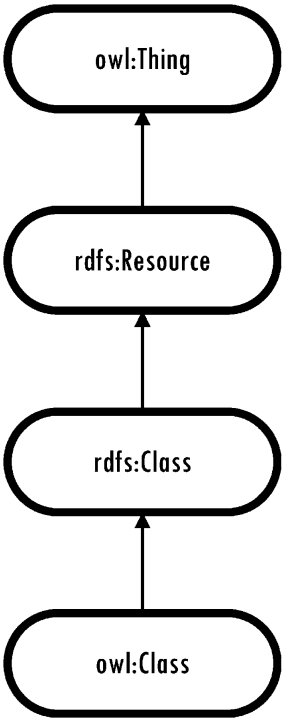

Fig. 5.1 Relationship between top classes

5.3.1 *Defining Classes: The Basics*

Recall in RDF Schema, the root class of everything is `rdfs:Resource`. In the world of OWL 1, `owl:Thing` is the root of all classes, it is also the base class of `rdfs:Resource`. Furthermore, `owl:Class` is defined by OWL 1 so that we can use it to define classes in OWL 1 ontologies, and `owl:Class` is a subclass of `rdfs:Class`. The relationship between all these top classes can therefore be summarized in Fig. 5.1:

Now, to declare one of our camera ontology's top classes using OWL 1, such as `Camera`, we can do the following:

```
<rdf:Description
      rdf:about="http://www.liyangyu.com/camera#Camera">
    <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#Class"/>
</rdf:Description>
```

And the following is an equivalent format:

```
<owl:Class rdf:about="http://www.liyangyu.com/camera#Camera">
</owl:Class>
```

To define all the classes in our camera ontology, List 5.1 will be good enough:

List 5.1 Class definitions for our camera ontology using OWL 1

```
1: <?xml version="1.0"?>
2: <rdf:RDF
2a:     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:      xmlns:myCamera="http://www.liyangyu.com/camera#"
6:      xml:base="http://www.liyangyu.com/camera#">
7:
8:  <owl:Class rdf:about="http://www.liyangyu.com/camera#Camera">
9:  </owl:Class>
10:
11: <owl:Class rdf:about="http://www.liyangyu.com/camera#Lens">
12: </owl:Class>
13:
14: <owl:Class rdf:about="http://www.liyangyu.com/camera#Body">
15: </owl:Class>
16:
17: <owl:Class
17a:     rdf:about="http://www.liyangyu.com/camera#ValueRange">
18: </owl:Class>
19:
20: <owl:Class
20a:     rdf:about="http://www.liyangyu.com/camera#Digital">
21:   <rdfs:subClassOf rdf:resource="#Camera"/>
22: </owl:Class>
23:
24: <owl:Class rdf:about="http://www.liyangyu.com/camera#Film">
25:   <rdfs:subClassOf rdf:resource="#Camera"/>
26: </owl:Class>
27:
28: <owl:Class rdf:about="http://www.liyangyu.com/camera#DSLR">
29:   <rdfs:subClassOf rdf:resource="#Digital"/>
30: </owl:Class>
31:
32: <owl:Class
32a:    rdf:about="http://www.liyangyu.com/camera#PointAndShoot">
33:   <rdfs:subClassOf rdf:resource="#Digital"/>
34: </owl:Class>
35:
36: <owl:Class
36a:    rdf:about="http://www.liyangyu.com/camera#Photographer">
37:   <rdfs:subClassOf
37a:        rdf:resource="http://xmlns.com/foaf/0.1/Person"/>
38: </owl:Class>
39:
40: </rdf:RDF>
```

Looks like we are done: we have just finished using OWL 1 terms to define all the classes used in our camera ontology.

Notice that List 5.1 only contains a very simple class hierarchy. OWL 1 offers much greater expressiveness than we have just utilized. Let us explore these features one by one, and in order to show how these new features are used, we will also change our camera ontology from time to time.

5.3.2 *Defining Classes: Localizing Global Properties*

In Chap. 4, we defined properties by using RDFS terms. For example, recall the definition of `owned_by` property,

```
<rdf:Property
      rdf:about="http://www.liyangyu.com/camera#owned_by">
  <rdfs:domain rdf:resource="#DSLR"/>
  <rdfs:range rdf:resource="#Photographer"/>
</rdf:Property>
```

Notice that `rdfs:range` imposes a global restriction on `owned_by` property, i.e., the `rdfs:range` value applies to `Photographer` class and all subclasses of `Photographer` class.

However, there will be cases where we actually would like to localize this global restriction on a given property. Clearly, RDFS terms will not be able to help us to implement this. OWL 1, on the other hand, provides ways to localize a global property by defining new classes, as we show in this section.

5.3.2.1 Value Constraints: `owl:allValuesFrom`

Let us go back to our definition of `owned_by` property. More specifically, we have associated this property with two classes, `DSLR` and `Photographer`, in order to express the knowledge "`DSLR` is `owned_by` `Photographer`".

Let us say we now want to express the following fact: `DSLR`, especially an expensive one, are normally used by professional photographers. For example, the body alone of some high-end digital SLRs can cost as much as $5,000.00.

To accomplish this, we decide to define a new class called `ExpensiveDSLR`, as a subclass of `DSLR`. We would also like to define two more classes, `Professional` and `Amateur`, as subclasses of `Photographer`. These two classes represent professional and amateur photographers, respectively. List 5.2 shows the definitions of these two new classes.

List 5.2 New class definitions are added: `Professional` and `Amateur`

```
1: <?xml version="1.0"?>
2: <rdf:RDF
2a:     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:      xmlns:myCamera="http://www.liyangyu.com/camera#"
6:      xml:base="http://www.liyangyu.com/camera#">
7:
8:
   ... same as List 5.1

38:
39:
40: <owl:Class
40a:     rdf:about="http://www.liyangyu.com/camera#Professional">
41:    <rdfs:subClassOf rdf:resource="#Photographer"/>
42: </owl:Class>
43:
44: <owl:Class
44a:     rdf:about="http://www.liyangyu.com/camera#Amateur">
45:    <rdfs:subClassOf rdf:resource="#Photographer"/>
46: </owl:Class>
47:
48: </rdf:RDF>
```

Does this ontology successfully express our idea? Not really. Since `owned_by` has `DSLR` as its `rdfs:domain` and `Photographer` as its `rdfs:value`, and given the fact that `ExpensiveDSLR` is a subclass of `DSLR`, and `Professional` and `Amateur` are both subclasses of `Photographer`, these new subclasses all inherit the `owned_by` property. Therefore, we can indeed say something like this:

`ExpensiveDSLR owned_by Professional`

which is what we wanted. However, we cannot exclude the following statement either:

`ExpensiveDSLR owned_by Amateur`

How do we modify the definition of `ExpensiveDSLR` to make sure it can be owned *only* by `Professional`? OWL 1 uses `owl:allValuesFrom` to solve this problem, as shown in List 5.3.

List 5.3 Use `owl:allValuesFrom` to define `ExpensiveDSLR` class

```
1: <owl:Class
1a:     rdf:about="http://www.liyangyu.com/camera#ExpensiveDSLR">
2:   <rdfs:subClassOf rdf:resource="#DSLR"/>
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>
```

To understand how List 5.3 defines `ExpensiveDSLR` class, we need to understand `owl:Restriction` first.

`owl:Restriction` is an OWL 1 term used to describe an anonymous class, which is defined by adding some restriction on some property. Furthermore, all the instances of this anonymous class have to satisfy this restriction; hence the term `owl:Restriction`.

The restriction itself has two parts. The first part is about to which property this restriction is applied to, and this is specified by using `owl:onProperty` property. The second part is about the property constraint itself, or, exactly what is the constraint. Two kinds of property restrictions are allowed in OWL: *value constraints* and *cardinality constraints*. A value constraint puts constraints on the range of the property, while a cardinality constraint puts constraints on the number of values a property can take. We will see value constraints in this section and cardinality constraints in the coming sections.

One way to specify a value constraint is to use the built-in OWL 1 property called `owl:allValuesFrom`. When this property is used, the value of the restricted property has to all come from the specified class or data range.

With all this said, List 5.3 should be easier to understand. Lines 4–7 use `owl:Restriction` to define an anonymous class; the constraint is applied on `owned_by` property (this is specified by using `owl:onProperty` property on line 5), and the values for `owned_by` property must all come from instances of class `Professional` (this is specified by using `owl:allValuesFrom` property on line 6). Therefore, lines 4–7 can be read as follows:

> lines 4–7 have defined an anonymous class which has a property `owned_by`, and all values for `owned_by` property must be instances of `Professional`.

With this, List 5.3 can be read as follows:

> Here is a definition of class `ExpensiveDSLR`: it is a subclass of `DSLR`, and a subclass of an anonymous class which has a property `owned_by` and all values for this property must be instances of `Professional`.

It does take a while to get used to this way of defining new classes. Once you are used to it, you can simply read List 5.3 like this:

> Here is a definition of class `ExpensiveDSLR`: it is a subclass of `DSLR` and it has a property named `owned_by`, and only instances of class `Professional` can be the value for this property.

Therefore, by adding constraints on a given property, we have defined a new class that satisfies our needs. In fact, this new way of defining classes is frequently used in OWL ontologies, so make sure you understand it and feel comfortable with it as well.

5.3.2.2 Enhanced Reasoning Power 1

In this chapter, we are going to talk about OWL's reasoning power in more detail, so you will see more sections like this one coming up frequently. The following should be clear before we move on.

First off, when we say an application can understand a given ontology, we mean that the application can parse the ontology and create a list of axioms based on the ontology, and all the facts are expressed as RDF statements. You will see how this is accomplished in later chapters, but for now, just assume this can be easily done.

Second, when we say an application can make inferences, we refer to the fact that the application can add new RDF statements into the existing collection of statements. The newly added statements are not mentioned anywhere in the original ontology or original instance document.

Finally, when we say instance document, we refer to an RDF document that is created by using the terms presented in the given ontology. Also, when we present this instance document, we will only show the part that is relevant to that specific reasoning capability being discussed. The rest of the instance file that is not related to this specific reasoning capability will not be included.

With all this being said, we can now move on to take a look at the reasoning power provided by `owl:allValuesFrom` construct. Let us say our application sees the following instance document:

```
<myCamera:ExpensiveDSLR
          rdf:about="http://dbpedia.org/resource/Canon_EOS-1D">
   <myCamera:owned_by
          rdf:resource="http://www.liyangyu.com/people#Liyang"/>
   <myCamera:owned_by
          rdf:resource="http://www.liyangyu.com/people#Connie"/>
</myCamera:ExpensiveDSLR>
```

The application will be able to add the following facts (in Turtle format):

```
<http://www.liyangyu.com/people#Liyang> rdf:type
myCamera:Professional.
<http://www.liyangyu.com/people#Connie> rdf:type
myCamera:Professional.
```

Notice that it is certainly true that our application will also be able to add quite a few other facts, such as http://www.liyangyu.com/people#Liyang must also be myCamera:Photographer, and also http://dbpedia.org/resource/Canon_EOS-1D must be a myCamera:DSLR, just to name a few. Here, we are not going to list all these added facts; instead, we will only concentrate on the new facts that are related to the OWL 1 language feature that is being discussed.

5.3.2.3 Value Constraints: owl:someValuesFrom

In last section, we used owl:allValuesFrom to make sure that ExpensiveDSLRs are those cameras that can only be owned by Professionals. Now, let us loosen up this restriction by allowing some Amateurs to buy and own ExpensiveDSLRs as well. However, we still require that at least one of the owners has to be a Professional. OWL 1 uses owl:someValuesFrom to express this idea, as shown in List 5.4.

List 5.4 Use owl:someValuesFrom to define ExpensiveDSLR class

```
1: <owl:Class
1a:     rdf:about="http://www.liyangyu.com/camera#ExpensiveDSLR">
2:   <rdfs:subClassOf rdf:resource="#DSLR"/>
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 read like this:

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

With what we have learned from the previous section, this does not require much explanation.

5.3.2.4 Enhanced Reasoning Power 2

Our application sees the following instance document:

```
<myCamera:ExpensiveDSLR
          rdf:about="http://dbpedia.org/resource/Canon_EOS-1D">
  <myCamera:owned_by
           rdf:resource="http://www.liyangyu.com/people#Liyang"/>
</myCamera:ExpensiveDSLR>
```

The application will be able to add the following facts:

```
<http://www.liyangyu.com/people#Liyang> rdf:type
myCamera:Professional.
```

If our application sees the following instance document:

```
<myCamera:ExpensiveDSLR
          rdf:about="http://dbpedia.org/resource/Canon_EOS-1D">
   <myCamera:owned_by
          rdf:resource="http://www.liyangyu.com/people#Liyang"/>
   <myCamera:owned_by
          rdf:resource="http://www.liyangyu.com/people#Connie"/>
</myCamera:ExpensiveDSLR>
```

then *at least* one of the following two statements is true (it could be that both are true):

```
<http://www.liyangyu.com/people#Liyang> rdf:type
myCamera:Professional.
<http://www.liyangyu.com/people#Connie> rdf:type
myCamera:Professional.
```

It is important to understand the difference between `owl:allValuesFrom` and `owl:someValuesFrom`. Think about it on your own, and we will summarize the difference in a later section.

5.3.2.5 Value Constraints: `owl:hasValue`

Another way OWL 1 uses to localize a global property in the context of a given class is to use `owl:hasValue`. So far, we have defined `ExpensiveDSLR` as being a `DSLR` that is owned by a professional photographer (List 5.3), or owned by at least one professional photographer (List 5.4). These definitions are fine, but they are not

straightforward. In fact, we can use a more direct approach to define what it means to be an expensive DSLR.

Let us first define a property called `cost` like this:

```
<owl:DatatypeProperty
      rdf:about="http://www.liyangyu.com/camera#cost">
    <rdfs:domain rdf:resource="#Digital"/>
    <rdfs:range
          rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
</owl:DatatypeProperty>
```

Since we have not yet reached the section about defining properties, let us not worry about the syntax here. For now, understand this defines a property called `cost`, which is used to describe `Digital` and its value will be a string of your choice. For instance, you can take `expensive` or `inexpensive` as its value.

Clearly, `DSLR` and `PointAndShoot` are all subclasses of `Digital`; therefore they can all use property `cost` in the way they want. In other words, `cost` as a property is global. Now in order to directly express the knowledge that "an `ExpensiveDSLR` is `expensive`", we can specify the fact that the value of `cost`, when used with `ExpensiveDSLR`s, should always be `expensive`. We can use `owl:hasValue` to implement this idea, as shown in List 5.5.

List 5.5 Use `owl:hasValue` to define `ExpensiveDSLR` class

```
1: <owl:Class
1a:     rdf:about="http://www.liyangyu.com/camera#ExpensiveDSLR">
2:   <rdfs:subClassOf rdf:resource="#DSLR"/>
3:   <rdfs:subClassOf>
4:     <owl:Restriction>
5:       <owl:onProperty rdf:resource="#cost"/>
6:       <owl:hasValue
6a:       rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
7:         expensive
8:       </owl:hasValue>
9:     </owl:Restriction>
10:   </rdfs:subClassOf>
11: </owl:Class>
```

This defines class `ExpensiveDSLR` as follows:

A class called `ExpensiveDSLR` is defined. It is a subclass of `DSLR`, and every instance of `ExpensiveDSLR` has a `cost` property whose value is `expensive`.

Meanwhile, instances of DSLR or PointAndShoot can take whatever cost value they want (i.e., expensive or inexpensive), indicating that they can be 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. More specifically, whenever we decide to use owl:allValuesFrom, it is equivalent to declare 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 "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, using an owl:someValuesFrom restriction on a property implies 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 it is more specific because it requires a particular instance instead of a class.

5.3.2.6 Enhanced Reasoning Power 3

Our application sees the following instance document (notice the definition of ExpensiveDSLR is given in List 5.5):

```
<myCamera:DSLR
          rdf:about="http://dbpedia.org/resource/Canon_EOS-1D">
   <myCamera:cost
      rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
      expensive</myCamera:cost>
</myCamera:DSLR>
```

The application will be able to add the following facts:

```
< http://dbpedia.org/resource/Canon_EOS-1D > rdf:type
myCamera:ExpensiveDSLR.
```

Notice the original instance document only shows the class type as DSLR, and the application can assert the more accurate type would be ExpensiveDSLR.

5.3.2.7 Cardinality Constraints: owl:cardinality, owl:min (max)Cardinality

Another way to define class by adding restrictions on properties is to constrain the cardinality of a property based on the class on which it is intended to be used.

In this section, we will add cardinality constraints to some of our existing class definitions in our camera ontology. By doing so, not only will we learn how to use the cardinality constraints, our camera ontology will also become more accurate.

In our camera ontology, class `Digital` represents a digital camera, and property `effectivePixel` represents the picture resolution of a given digital camera, and this property can be used on instances of `Digital` class. Obviously, when defining `Digital` class, it would be useful to indicate that there can be only one `effectivePixel` value for any given digital camera. We cannot accomplish this by using RDFS vocabulary; however, OWL 1 does allow us to do so, as shown in List 5.6.

List 5.6 Definition of class `Digital` using `owl:cardinality` constraint

```
1: <owl:Class rdf:about="http://www.liyangyu.com/camera#Digital">
2:   <rdfs:subClassOf rdf:resource="#Camera"/>
3:   <rdfs:subClassOf>
4:     <owl:Restriction>
5:       <owl:onProperty rdf:resource="#effectivePixel"/>
6:       <owl:cardinality rdf:datatype=
6a:         "http://www.w3.org/2001/XMLSchema#nonNegativeInteger">
7:       1
8:       </owl:cardinality>
9:     </owl:Restriction>
10:   </rdfs:subClassOf>
11: </owl:Class>
```

This defines class `Digital` as follows:

> A class called `Digital` is defined. It is a subclass of `Camera`; it has a property called `effectivePixel`; there can be only one `effectivePixel` value for an instance of `Digital` class.

and notice that we need to specify that the literal "`1`" is to be interpreted as a non-negative integer using `rdf:datatype` property. Also, be aware that this does not place any restrictions on the number of occurrences of `effectivePixel` property in any instance document. In other words, a given instance of `Digital` class (or its subclass) can indeed have multiple `effectivePixel` values, however, when it does, these values must all be equal.

What about `model` property? Clearly, each camera should have at least one `model` value, but it can have multiple `model` values: as we have discussed, the exact

same camera, when sold in Asia or North America, can indeed have different `model` values. To take this constraint into account, we can modify the definition of `Camera` class as shown in List 5.7.

List 5.7 Definition of class `Camera` using `owl:minCardinality` constraint

```
1: <owl:Class rdf:about="http://www.liyangyu.com/camera#Camera">
2:   <rdfs:subClassOf>
3:     <owl:Restriction>
4:       <owl:onProperty rdf:resource="#model"/>
5:       <owl:minCardinality rdf:datatype=
5a:         "http://www.w3.org/2001/XMLSchema#nonNegativeInteger">
6:       1
7:       </owl:minCardinality>
8:     </owl:Restriction>
9:   </rdfs:subClassOf>
10: </owl:Class>
```

And you can use `owl:minCardinality` together with `owl:maxCardinality` to specify a range, as shown in List 5.8, which says that a camera should have a least one `model` value, but cannot have more than three.

List 5.8 Definition of class `Camera` using `owl:minCardinality` and `owl:maxCardinality` constraints

```
1: <owl:Class rdf:about="http://www.liyangyu.com/camera#Camera">
2:   <rdfs:subClassOf>
3:     <owl:Restriction>
4:       <owl:onProperty rdf:resource="#model"/>
5:       <owl:minCardinality rdf:datatype=
5a:         "http://www.w3.org/2001/XMLSchema#nonNegativeInteger">
6:       1
7:       </owl:minCardinality>
8:       <owl:maxCardinality rdf:datatype=
8a:         "http://www.w3.org/2001/XMLSchema#nonNegativeInteger">
9:       3
10:      </owl:maxCardinality>
11:    </owl:Restriction>
12:  </rdfs:subClassOf>
13: </owl:Class>
```

5.3.2.8 Enhanced Reasoning Power 4

Our application sees the following statement from one instance document (notice the definition of `Digital` is given in List 5.6):

```
<myCamera:Digital
          rdf:about="http://www.liyangyu.com/camera#Nikon_D300">
   <myCamera:effectivePixel rdf:resource=
          "http://www.example.org/digitalCamera#pixelValue12.3"/>
</myCamera:Digital>
```

And it has also collected this statement from another instance document:

```
<myCamera:Digital
          http://www.liyangyu.com/camera#Nikon_D300">
   <myCamera:effectivePixel rdf:resource=
          "http://dbpedia.org/resource/Nikon_D300_Resolution"/>
</myCamera:Digital>
```

The application will be able to add the following fact:

```
<http://www.example.org/digitalCamera#pixelValue12.3>
owl:sameAs <http://dbpedia.org/resource/Nikon_D300_Resolution>.
```

Notice `owl:sameAs` means the two given resources are exactly the same. In other words, the following two URIs are URI aliases to each other:

```
http://www.example.org/digitalCamera#pixelValue12.3
http://dbpedia.org/resource/Nikon_D300_Resolution
```

Lists 5.7 and 5.8 yield similar reasoning power. As you can easily see them by yourself, we are not going to discuss them in much detail here.

5.3.3 *Defining Classes: Using Set Operators*

In the previous sections, we have defined classes by placing constraints on properties, including property value constraints and cardinality constraints. OWL 1 also gives us the ability to construct classes by using set operators. In this section, we briefly introduce these operators so you have more choices when it comes to defining classes.

5.3.3.1 Set Operators

The first operator is `owl:intersectionOf`. Recall the definition of `Expensive DSLR` presented in List 5.5; we can rewrite this definition as shown in List 5.9.

List 5.9 Definition of class `ExpensiveDSLR` using `owl:intersectionOf`

```
1: <owl:Class
1a:     rdf:about="http://www.liyangyu.com/camera#ExpensiveDSLR">
2:    <owl:intersectionOf rdf:parseType="Collection">
3:       <owl:Class rdf:about="#DSLR"/>
4:       <owl:Restriction>
5:         <owl:onProperty rdf:resource="#cost"/>
6:         <owl:hasValue rdf:datatype=
6a:              "http://www.w3.org/2001/XMLSchema#string">
7:           expensive
8:         </owl:hasValue>
9:       </owl:Restriction>
10:   </owl:intersectionOf>
11: </owl:Class>
```

Based on what we have learned so far, List 5.9 is quite straightforward: lines 4–9 define an anonymous class using `owl:Restriction` pattern. This class represents all the individuals that have `cost` property, and the value for this property is `expensive`. Line 3 includes class `DSLR` into the picture, which represents all the `DSLR` cameras. Line 2 then claims the new class, `ExpensiveDSLR`, represents all the individuals that are in the intersection of these two sets of individuals. Therefore, we can read List 5.9 as follows:

> A class called `ExpensiveDSLR` is defined. It is the intersection of `DSLR` class and an anonymous class that has a property called `cost`, and this property has the value `expensive`.

Or, we can simply read List 5.9 as this:

> A class called `ExpensiveDSLR` is defined. It is a `DSLR` for which `cost` is `expensive`.

So what is the difference between List 5.5 and List 5.9? Notice List 5.5 uses multiple `owl:subClassOf` terms, and in OWL 1, this means qualified individuals should all come from a *subset* of the final intersection of all the classes specified by the multiple `owl:subClassOf` terms. On the other hand, List 5.9 means that qualified individuals should all come from the final intersection of the classes

included in the class collection (line 2 of List 5.9). So there is indeed some subtle difference between these two definitions. However, as far as reasoning is concerned, these two definitions will produce the same inferred facts.

The second operator is the `owl:unionOf` operator. For example, although most photographers today mainly use digital cameras, still they may keep their film cameras around in case they do need them. Therefore if we define a class called `CameraCollection` to represent a photographer's camera collection, it could be defined as shown in List 5.10.

List 5.10 Definition of class `CameraCollection` using `owl:unionOf`

```
1: <owl:Class
1a:  rdf:about="http://www.liyangyu.com/camera#CameraCollection">
2:    <owl:unionOf rdf:parseType="Collection">
3:       <owl:Class rdf:about="#Digital"/>
4:       <owl:Class rdf:about="#Film"/>
5:   </owl:unionOf>
6: </owl:Class>
```

List 5.10 says `CameraCollection` should include both the extension of `Digital` and the extension of `Film`, and clearly, this is exactly what we want.

The last set operator is the `owl:complementOf` operator. A good example is the set of professional photographers and the set of amateur photographers; they are exactly the complement of each other. Therefore, we can rewrite the definition of `Amateur` photographer as shown in List 5.11.

List 5.11 Definition of class `Amateur` using `owl:complementOf`

```
1: <owl:Class rdf:about="http://www.liyangyu.com/camera#Amateur">
2:   <owl:intersectionOf rdf:parseType="Collection">
3:      <owl:Class rdf:about="http://xmlns.com/foaf/0.1/Person"/>
4:      <owl:Class>
5:        <owl:complementOf rdf:resource="#Professional"/>
6:      </owl:Class>
7:   </owl:intersectionOf>
8: </owl:Class>
```

This says that an `Amateur` is a `Person` who is not a `Professional`, and notice we have used `owl:intersectionOf` as well to make the definition correct.

5.3.3.2 Enhanced Reasoning Power 5

Like other OWL 1 language features, using set operators to define class also provides enhanced reasoning power. Since there are indeed quite a few related to using set operators, we will discuss the related ones without using concrete examples.

More specifically, the following are some of these enhanced reasoning conditions:

- If a class C_0 is the `owl:intersectionOf` a list of class C_1, C_2 and C_3, then C_0 is subclass of each one of C_1, C_2 and C_3.
- If a class `A` is the `owl:intersectionOf` a list of classes and class `B` is the `owl:intersectionOf` another list of classes, class `A` is a subclass of class `B` if every constituent class of `A` is a subclass of some constituent class of `B`.
- If a class C_0 is the `owl:unionOf` a list of class C_1, C_2 and C_3, then each one of C_1, C_2 and C_3, is a subclass of C_0.
- If a class `A` is the `owl:unionOf` a list of classes and class `B` is the `owl:unionOf` another list of classes, class `A` is a subclass of class `B` if every constituent class of `B` is a super-class of some constituent class of `A`.
- If a class `A` is `owl:complementOf` a class `B`, then all the subclasses of `A` will be `owl:disjointWith` class `B`.

Notice this is not a complete list, but the above will give you some idea about reasoning based on set operators.

5.3.4 Defining Classes: Using Enumeration, Equivalent and Disjoint

Besides all the methods we have learned so far about defining classes, OWL 1 still has more ways that we can use:

- construct a class by enumerating its instances;
- specify a class is equivalent to another class, and
- specify a class is disjoint from another class.

We discuss the details in this section.

5.3.4.1 Enumeration, Equivalent and Disjoint

Defining classes by enumeration could be quite useful for many cases. To see why, let us recall the methods we have used when defining the `ExpensiveDSLR` class. So far we have defined the class `ExpensiveDSLR` by saying that it has to be owned by a professional photographer, or, its `cost` property has to take the value `expensive`,

etc. All these methods are a *descriptive* way to define a class: as long as an instance satisfies all the conditions, it is a member of the defined class.

The drawback of this descriptive method is the fact that there could be a large number of instances qualified, and sometimes, it takes computing time to make the decision of qualification. In some cases, it will be more efficient and useful if we can explicitly enumerate which are the qualified members; which will simply present a more accurate semantics for many applications. The `owl:oneOf` property provided by OWL 1 can be used to accomplish this. List 5.12 shows how.

List 5.12 Definition of class `ExpensiveDSLR` using `owl:oneOf`

```
1: <owl:Class
1a:     rdf:about="http://www.liyangyu.com/camera#ExpensiveDSLR">
2:   <rdfs:subClassOf rdf:resource="#DSLR"/>
3:   <owl:oneOf rdf:parseType="Collection">
4:     <myCamera:DSLR
4a:            rdf:about="http://dbpedia.org/resource/Nikon_D3"/>
5:     <myCamera:DSLR
5a:        rdf:about="http://dbpedia.org/resource/Canon_EOS-1D"/>
6:   </owl:oneOf>
7: </owl:Class>
```

It is important to understand that no other individuals can be included in the extension of class `ExpensiveDSLR`, except for the instances listed on lines 4 and 5. Therefore, if you do decide to use enumeration to define this class, you might want to add more instances there. Also, notice that `Nikon_D3` (line 4) is not a typo, it is indeed a quite expensive DSLR, and this URI is taken from DBpedia project, similar to the URI on line 5.

Since each individual is referenced by its URI, it is fine not to use a specific type for it, and just use `owl:Thing` instead. Therefore, List 5.13 is equivalent to List 5.12.

List 5.13 Definition of class `ExpensiveDSLR` using `owl:oneOf`

```
1: <owl:Class
1a:     rdf:about="http://www.liyangyu.com/camera#ExpensiveDSLR">
2:   <rdfs:subClassOf rdf:resource="#DSLR"/>
3:   <owl:oneOf rdf:parseType="Collection">
4:     <owl:Thing
4a:        rdf:about="http://dbpedia.org/resource/Nikon_D3"/>
5:     <owl:Thing
5a:        rdf:about="http://dbpedia.org/resource/Canon_EOS-1D"/>
6:    </owl:oneOf>
7: </owl:Class>
```

The last thing to remember is the syntax: we need to use `owl:oneOf` together with `rdf:parseType` to tell the parser that we are in fact enumerating all the members of the class being defined.

We can also define a class by using `owl:equivalentClass` property, which indicates that two classes have precisely the same instances. For example, List 5.14 declares another class called `DigitalSLR`, and it is exactly the same as `DSLR` class:

List 5.14 Use `owl:equivalentClass` to define class `DigitalSLR`

```
1: <owl:Class
1a:     rdf:about="http://www.liyangyu.com/camera#DigitalSLR">
2:    <owl:equivalentClass rdf:resource="#DSLR"/>
3: </owl:Class>
```

More often, property `owl:equivalentClass` is used to explicitly declare that two classes in two different ontology documents are in fact equivalent classes. For example, if another ontology document defines a class called `DigitalSingleLensReflex`, and we would like to claim our class, `DSLR`, is equivalent to this class, we can accomplish this as shown in List 5.15.

List 5.15 Use `owl:equivalentClass` to specify two classes are equivalent

```
1: <owl:Class rdf:about="http://www.liyangyu.com/camera#DSLR">
2:   <rdfs:subClassOf rdf:resource="#Digital"/>
3:   <owl:equivalentClass rdf:resource=
3a:       "http://www.example.org#DigitalSingleLensReflex"/>
4: </owl:Class>
```

Now, in any RDF document, if we have described an instance that is of type `DSLR`, it is also an instance of type `DigitalSingleLensReflex`.

Finally, OWL 1 also provides a way to define the fact that two classes are not related in any way. For instance, in our camera ontology, we have defined `DSLR` and `PointAndShoot` as subclasses of `Digital`. To make things simpler and without worrying about the fact that a `DSLR` camera in many cases can be simply used as a `PointAndShoot` camera, we can define `DSLR` to be disjoint from the `PointAndShoot` class, as shown in List 5.16:

List 5.16 Use `owl:disjointWith` to specify two classes are disjoint

```
1: <owl:Class rdf:about="http://www.liyangyu.com/camera#DSLR">
2:   <rdfs:subClassOf rdf:resource="#Digital"/>
3:   <owl:equivalentClass rdf:resource=
3a:       "http://www.example.org#DigitalSingleLensReflex"/>
4:   <owl:disjointWith rdf:resource="#PointAndShoot"/>
5: </owl:Class>
```

Once a given application sees this definition, it will understand that any instance of `DSLR` can never be an instance of the `PointAndShoot` camera. Also notice that `owl:disjointWith` by default is a symmetric property (more on this later): if `DSLR` is disjoint with `PointAndShoot`, then `PointAndShoot` is disjoint with `DSLR`.

5.3.4.2 Enhanced Reasoning Power 6

Similar to set operators, we will list some related reasoning powers here without using concrete examples:

- If a class C_0 is `owl:oneOf` a list of class C_1, C_2 and C_3, then each of C_1, C_2 and C_3 has `rdf:type` given by C_0.
- If a class `A` is `owl:equivalentClass` to class `B`, then a `owl:sameAs` relationship will be asserted between these two classes.
- If a class `A` is `owl:disjointWith` class `B`, then any subclass of `A` will be `owl:disjointWith` with class `B`.

Again, this is certainly not a complete list, and you will see others in your future work for sure.

5.3.5 Our Camera Ontology So Far

Let us summarize our latest camera ontology (with only the class definitions) as shown in List 5.17. Notice that in previous sections, in order to show the related language features of OWL 1, we have discussed different ways to define classes. To avoid unnecessary complexities, we have not included all of them into our current camera ontology.

List 5.17 Our current camera ontology, with class definitions only

```
1: <?xml version="1.0"?>
2: <!DOCTYPE rdf:RDF [
3:     <!ENTITY owl "http://www.w3.org/2002/07/owl#" >
4:     <!ENTITY xsd "http://www.w3.org/2001/XMLSchema#" >
5:     <!ENTITY rdfs "http://www.w3.org/2000/01/rdf-schema#" >
6:     <!ENTITY myCamera "http://www.liyangyu.com/camera#" >
7: ]>
8:
9: <rdf:RDF
9a:     xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
10:     xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
11:     xmlns:owl="http://www.w3.org/2002/07/owl#"
12:     xmlns:myCamera="http://www.liyangyu.com/camera#"
13:     xml:base="http://www.liyangyu.com/camera#">
14:
15: <owl:Class rdf:about="&myCamera;Camera">
16:   <rdfs:subClassOf>
17:     <owl:Restriction>
18:       <owl:onProperty rdf:resource="&myCamera;model"/>
19:       <owl:minCardinality
19a:           rdf:datatype="&xsd;nonNegativeInteger">
20:         1
21:       </owl:minCardinality>
22:     </owl:Restriction>
23:   </rdfs:subClassOf>
24: </owl:Class>
25:
26: <owl:Class rdf:about="&myCamera;Lens">
27: </owl:Class>
28:
29: <owl:Class rdf:about="&myCamera;Body">
30: </owl:Class>
31:
32: <owl:Class rdf:about="&myCamera;ValueRange">
33: </owl:Class>
34:
35: <owl:Class rdf:about="&myCamera;Digital">
36:   <rdfs:subClassOf rdf:resource="&myCamera;Camera"/>
37:   <rdfs:subClassOf>
38:     <owl:Restriction>
39:       <owl:onProperty
39a:           rdf:resource="&myCamera;effectivePixel"/>
40:       <owl:cardinality
40a:           rdf:datatype="&xsd;nonNegativeInteger">
41:         1
```

```
42:       </owl:cardinality>
43:     </owl:Restriction>
44:   </rdfs:subClassOf>
45: </owl:Class>
46:
47: <owl:Class rdf:about="&myCamera;Film">
48:   <rdfs:subClassOf rdf:resource="&myCamera;Camera"/>
49: </owl:Class>
50:
51: <owl:Class rdf:about="&myCamera;DSLR">
52:   <rdfs:subClassOf rdf:resource="&myCamera;Digital"/>
53: </owl:Class>
54:
55: <owl:Class rdf:about="&myCamera;PointAndShoot">
56:   <rdfs:subClassOf rdf:resource="&myCamera;Digital"/>
57: </owl:Class>
58:
59: <owl:Class rdf:about="&myCamera;Photographer">
60:   <rdfs:subClassOf
60a:        rdf:resource="http://xmlns.com/foaf/0.1/Person"/>
61: </owl:Class>
62:
63: <owl:Class rdf:about="&myCamera;Professional">
64:   <rdfs:subClassOf rdf:resource="&myCamera;Photographer"/>
65: </owl:Class>
66:
67: <owl:Class rdf:about="&myCamera;Amateur">
68:   <owl:intersectionOf rdf:parseType="Collection">
69:     <owl:Class rdf:about="http://xmlns.com/foaf/0.1/Person"/>
70:     <owl:Class>
71:       <owl:complementOf
71a:           rdf:resource="&myCamera;Professional"/>
72:     </owl:Class>
73:   </owl:intersectionOf>
74: </owl:Class>
75:
76: <owl:Class rdf:about="&myCamera;ExpensiveDSLR">
77:   <rdfs:subClassOf rdf:resource="&myCamera;DSLR"/>
78:   <rdfs:subClassOf>
79:     <owl:Restriction>
80:       <owl:onProperty rdf:resource="&myCamera;owned_by"/>
81a:           rdf:resource="&myCamera;Professional"/>
82:     </owl:Restriction>
83:   </rdfs:subClassOf>
84: </owl:Class>
85:
86: </rdf:RDF>
```

5.3.6 *Define Properties: The Basics*

Up to this point, for our project of rewriting the camera ontology using OWL 1, we have finished defining the necessary classes. It is now the time to define all the necessary properties.

Recall when creating ontologies using RDF schema, we have the following terms to use when it comes to describing a property:

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

With only three terms, a given application already shows impressive reasoning power. And more importantly, as we saw in Chap. 4, most of the reasoning power comes from the understanding of the properties by the application.

This shows an important fact: richer semantics embedded into the properties will directly result in greater reasoning capabilities. This is the reason why OWL 1, besides continuing to use these three methods, has greatly enhanced the ways to characterize a property, as we will see in this section.

The first thing to notice is the fact that defining properties using OWL 1 is quite different from defining properties using RDF schema. More specifically, when using RDFS terms, the general procedure is to define the property first and then use it to connect two things together: a given property can either connect one resource to another resource, or connect one resource to a typed or untyped value. Both connections are done by using the term `rdf:Property`.

In the world of OWL 1, two different classes are used to implement these two different connections:

- `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 datatype (typed) value.

In addition, `owl:ObjectProperty` and `owl:DatatypeProperty` are both subclasses of `rdf:Property`. For example, List 5.18 shows the definitions of `owned_by` property and `model` property (taken from List 4.13):

List 5.18 Definitions of `owned_by` and `model` property, as shown in List 4.13

```
<rdf:Property
     rdf:about="http://www.liyangyu.com/camera#owned_by">
  <rdfs:domain rdf:resource="#DSLR"/>
  <rdfs:range rdf:resource="#Photographer"/>
</rdf:Property>

<rdf:Property rdf:about="http://www.liyangyu.com/camera#model">
  <rdfs:domain rdf:resource="#Camera"/>
  <rdfs:range
        rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
</rdf:Property>
<rdfs:Datatype
      rdf:about="http://www.w3.org/2001/XMLSchema#string"/>
```

In OWL 1, these definitions will look like the ones shown in List 5.19:

List 5.19 Use OWL 1 terms to define `owned_by` and `model` property

```
<owl:ObjectProperty
     rdf:about="http://www.liyangyu.com/camera#owned_by">
  <rdfs:domain rdf:resource="#DSLR"/>
  <rdfs:range rdf:resource="#Photographer"/>
</owl:ObjectProperty>

<owl:DatatypeProperty
     rdf:about="http://www.liyangyu.com/camera#model">
  <rdfs:domain rdf:resource="#Camera"/>
  <rdfs:range
        rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
</owl:DatatypeProperty>
<rdfs:Datatype
      rdf:about="http://www.w3.org/2001/XMLSchema#string"/>
```

Notice that except using `owl:ObjectProperty` and `owl:DatatypeProperty`, the basic syntax of defining properties in both RDF schema and OWL 1 is quite similar. In fact, at this moment, we can go ahead define all the properties that appear in List 4.13. After defining these properties, we have a whole camera ontology written in OWL 1 on our hands. Our finished camera ontology is given in List 5.20.

List 5.20 Our camera ontology defined in OWL 1

```
1: <?xml version="1.0"?>
2: <!DOCTYPE rdf:RDF [
3:     <!ENTITY owl "http://www.w3.org/2002/07/owl#" >
4:     <!ENTITY xsd "http://www.w3.org/2001/XMLSchema#" >
5:     <!ENTITY rdfs "http://www.w3.org/2000/01/rdf-schema#" >
6:     <!ENTITY myCamera "http://www.liyangyu.com/camera#" >
7: ]>
8:
9: <rdf:RDF
9a:     xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
10:     xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
11:     xmlns:owl="http://www.w3.org/2002/07/owl#"
12:     xmlns:myCamera="http://www.liyangyu.com/camera#"
13:     xml:base="http://www.liyangyu.com/camera#">
14:
15:  <owl:Class rdf:about="&myCamera;Camera">
16:    <rdfs:subClassOf>
17:      <owl:Restriction>
18:        <owl:onProperty rdf:resource="&myCamera;model"/>
19:        <owl:minCardinality
19a:            rdf:datatype="&xsd;nonNegativeInteger">
20:          1
21:        </owl:minCardinality>
22:      </owl:Restriction>
23:    </rdfs:subClassOf>
24:  </owl:Class>
25:
26:  <owl:Class rdf:about="&myCamera;Lens">
27:  </owl:Class>
28:
29:  <owl:Class rdf:about="&myCamera;Body">
30:  </owl:Class>
31:
32:  <owl:Class rdf:about="&myCamera;ValueRange">
33:  </owl:Class>
34:
35:  <owl:Class rdf:about="&myCamera;Digital">
36:    <rdfs:subClassOf rdf:resource="&myCamera;Camera"/>
37:    <rdfs:subClassOf>
38:      <owl:Restriction>
```

```
39:        <owl:onProperty
39a:            rdf:resource="&myCamera;effectivePixel"/>
40:        <owl:cardinality
40a:            rdf:datatype="&xsd;nonNegativeInteger">
41:         1
42:        </owl:cardinality>
43:      </owl:Restriction>
44:    </rdfs:subClassOf>
45:  </owl:Class>
46:
47:  <owl:Class rdf:about="&myCamera;Film">
48:    <rdfs:subClassOf rdf:resource="&myCamera;Camera"/>
49:  </owl:Class>
50:
51:  <owl:Class rdf:about="&myCamera;DSLR">
52:    <rdfs:subClassOf rdf:resource="&myCamera;Digital"/>
53:  </owl:Class>
54:
55:  <owl:Class rdf:about="&myCamera;PointAndShoot">
56:    <rdfs:subClassOf rdf:resource="&myCamera;Digital"/>
57:  </owl:Class>
58:
59:  <owl:Class rdf:about="&myCamera;Photographer">
60:    <rdfs:subClassOf
60a:         rdf:resource="http://xmlns.com/foaf/0.1/Person"/>
61:  </owl:Class>
62:
63:  <owl:Class rdf:about="&myCamera;Professional">
64:    <rdfs:subClassOf rdf:resource="&myCamera;Photographer"/>
65:  </owl:Class>
66:
67:  <owl:Class rdf:about="&myCamera;Amateur">
68:    <owl:intersectionOf rdf:parseType="Collection">
69:      <owl:Class
69a:          rdf:about="http://xmlns.com/foaf/0.1/Person"/>
70:      <owl:Class>
71:        <owl:complementOf
71a:            rdf:resource="&myCamera;Professional"/>
72:      </owl:Class>
73:    </owl:intersectionOf>
74:  </owl:Class>
75:
76:  <owl:Class rdf:about="&myCamera;ExpensiveDSLR">
77:    <rdfs:subClassOf rdf:resource="&myCamera;DSLR"/>
```

```
78:    <rdfs:subClassOf>
79:      <owl:Restriction>
80:        <owl:onProperty rdf:resource="&myCamera;owned_by"/>
81:        <owl:someValuesFrom
81a:            rdf:resource="&myCamera;Professional"/>
82:      </owl:Restriction>
83:    </rdfs:subClassOf>
84:  </owl:Class>
85:
86:  <owl:ObjectProperty rdf:about="&myCamera;owned_by">
87:    <rdfs:domain rdf:resource="&myCamera;DSLR"/>
88:    <rdfs:range rdf:resource="&myCamera;Photographer"/>
89:  </owl:ObjectProperty>
90:
91:  <owl:ObjectProperty rdf:about="&myCamera;manufactured_by">
92:    <rdfs:domain rdf:resource="&myCamera;Camera"/>
93:  </owl:ObjectProperty>
94:
95:  <owl:ObjectProperty rdf:about="&myCamera;body">
96:    <rdfs:domain rdf:resource="&myCamera;Camera"/>
97:    <rdfs:range rdf:resource="&myCamera;Body"/>
98:  </owl:ObjectProperty>
99:
100: <owl:ObjectProperty rdf:about="&myCamera;lens">
101:   <rdfs:domain rdf:resource="&myCamera;Camera"/>
102:   <rdfs:range rdf:resource="&myCamera;Lens"/>
103: </owl:ObjectProperty>
104:
105: <owl:DatatypeProperty rdf:about="&myCamera;model">
106:   <rdfs:domain rdf:resource="&myCamera;Camera"/>
107:   <rdfs:range rdf:resource="&xsd;string"/>
108: </owl:DatatypeProperty>
109: <rdfs:Datatype rdf:about="&xsd;string"/>
110:
111: <owl:ObjectProperty rdf:about="&myCamera;effectivePixel">
112:   <rdfs:domain rdf:resource="&myCamera;Digital"/>
113:   <rdfs:range rdf:resource="&myCamera;MegaPixel"/>
114: </owl:ObjectProperty>
115: <rdfs:Datatype rdf:about="&myCamera;MegaPixel">
116:   <rdfs:subClassOf rdf:resource="&xsd;decimal"/>
117: </rdfs:Datatype>
118:
119: <owl:ObjectProperty rdf:about="&myCamera;shutterSpeed">
120:   <rdfs:domain rdf:resource="&myCamera;Body"/>
```

```
121:   <rdfs:range rdf:resource="&myCamera;ValueRange"/>
122: </owl:ObjectProperty>
123:
124: <owl:DatatypeProperty rdf:about="&myCamera;focalLength">
125:   <rdfs:domain rdf:resource="&myCamera;Lens"/>
126:   <rdfs:range rdf:resource="&xsd;string"/>
127: </owl:DatatypeProperty>
128: <rdfs:Datatype rdf:about="&xsd;string"/>
129:
130: <owl:ObjectProperty rdf:about="&myCamera;aperture">
131:   <rdfs:domain rdf:resource="&myCamera;Lens"/>
132:   <rdfs:range rdf:resource="&myCamera;ValueRange"/>
133: </owl:ObjectProperty>
134:
135: <owl:DatatypeProperty rdf:about="&myCamera;minValue">
136:   <rdfs:domain rdf:resource="&myCamera;ValueRange"/>
137:   <rdfs:range rdf:resource="&xsd;float"/>
138: </owl:DatatypeProperty>
139: <rdfs:Datatype rdf:about="&xsd;float"/>
140:
141: <owl:DatatypeProperty rdf:about="&myCamera;maxValue">
142:   <rdfs:domain rdf:resource="&myCamera;ValueRange"/>
143:   <rdfs:range rdf:resource="&xsd;float"/>
144: </owl:DatatypeProperty>
145: <rdfs:Datatype rdf:about="&xsd;float"/>
146:
147: </rdf:RDF>
```

At this point, we have just finished rewriting our camera ontology using OWL 1 by adding the property definitions. Compared to the ontology defined using RDFS (List 4.13), List 5.20 includes a couple of new classes. I will leave it to you to update Fig. 4.3 to show these changes.

Now, this is only part of the whole picture. OWL 1 provides much richer features when it comes to property definitions. We will discuss these features in detail in the next several sections, but here is a quick look at these features:

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

5.3.7 *Defining Properties: Property Characteristics*

5.3.7.1 Symmetric Properties

A symmetric property describes the situation where if a 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 `Photographer` class, and if photographer `A` is `friend_with` photographer `B`, photographer `B` is certainly `friend_with` photographer `A`. This is shown in List 5.21:

List 5.21 Example of symmetric property

```
1: <owl:ObjectProperty
1a:     rdf:about="http://www.liyangyu.com/camera#friend_with">
2:   <rdf:type rdf:resource=
2a:       "http://www.w3.org/2002/07/owl#SymmetricProperty"/>
3:   <rdfs:domain rdf:resource="#Photographer"/>
4:   <rdfs:range rdf:resource="#Photographer"/>
5: </owl:ObjectProperty>
```

The key to indicate this property is a symmetric property lies in line 2. The definition in List 5.21 says the follows: `friend_with` is an object property which should be used to describe instances of class `Photographer`, its values are also instances of class `Photographer`, and it is a symmetric property.

Notice List 5.21 does have a simpler form, as shown in List 5.22:

List 5.22 Example of symmetric property using a simpler form

```
1: <owl:SymmetricProperty
1a:     rdf:about="http://www.liyangyu.com/camera#friend_with">
2:   <rdfs:domain rdf:resource="#Photographer"/>
3:   <rdfs:range rdf:resource="#Photographer"/>
4: </owl:SymmetricProperty>
```

It is important to know and understand the long form shown in List 5.21. One case where this long form is useful is the case when you need to define a property to be of several types, for example, a property that is symmetric and also functional (the functional property will be explained soon). In that case, the long form is the choice, and we just have to use multiple `rdf:type` elements.

Notice owl:SymmetricProperty is a subclass of owl:ObjectProperty. Therefore, rdfs:range of a symmetric property can only be a resource, and cannot be a literal or data type.

5.3.7.2 Enhanced Reasoning Power 7

Our application sees the following instance document:

```
<myCamera:Photographer
          rdf:about="http://www.liyangyu.com/people#Liyang">
   <myCamera:friend_with
        rdf:resource="http://www.liyangyu.com/people#Connie"/>
</myCamera:Photographer>
```

The application will be able to add the following two statements:

```
<http://www.liyangyu.com/people#Connie> rdf:type
myCamera:Photographer.

<http://www.liyangyu.com/people#Connie> myCamera:friend_with
<http://www.liyangyu.com/people#Liyang>.
```

5.3.7.3 Transitive Properties

A transitive property describes the situation where, 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 example, photography is a fairly expensive hobby for most of us, and which camera to buy depends on which one offers a better ratio of quality to price. Therefore, even though a given camera is very expensive, if it provides excellent quality and performance, the ratio could be high. On the other hand, a PointAndShoot camera has a very appealing price but it may not offer you that much room to discover your creative side; therefore may not have a high ratio at all.

Let us define a new property called betterQPRatio to capture this part of the knowledge in our camera ontology. Obviously, this property should be able to provide us a way to compare two different cameras. Furthermore, we will also declare it to be a transitive property; therefore if camera A is betterQPRatio than camera B, and camera B is betterQPRatio than camera C, it should be true that Camera A is betterQPRatio than camera C.

List 5.23 shows the syntax we use in OWL 1 to define such a property:

List 5.23 Example of transitive property

```
1: <owl:ObjectProperty
1a:     rdf:about="http://www.liyangyu.com/camera#betterQPRatio">
2:   <rdf:type rdf:resource=
2a:      "http://www.w3.org/2002/07/owl#TransitiveProperty"/>
3:   <rdfs:domain rdf:resource="#Camera"/>
4:   <rdfs:range rdf:resource="#Camera"/>
5: </owl:ObjectProperty>
```

Not much explanation is needed. Again, `owl:TransitiveProperty` is a sub-class of `owl:ObjectProperty`. Therefore, `rdfs:range` of a transitive property can only be a resource, and cannot be a literal or a data type.

5.3.7.4 Enhanced Reasoning Power 8

Our application collects the following statement from one instance document:

```
<myCamera:DSLR
          rdf:about="http://www.liyangyu.com/camera#Nikon_D300">
  <myCamera:betterQPRatio
        rdf:resource="http://www.liyangyu.com/camera#Nikon_D70"/>
</myCamera:DSLR>
```

and in another RDF document, our application finds this statement:

```
<DSLR rdf:about="http://www.liyangyu.com/camera#Nikon_D70"
      xmlns="http://www.liyangyu.com/camera#">
  <betterQPRatio>
     <DSLR rdf:about=
           "http://www.liyangyu.com/camera#Nikon_D40"/>
  </betterQPRatio>
</DSLR>
```

our application will add the following statement:

```
<http://www.liyangyu.com/camera#Nikon_D300>
myCamera:betterQPRatio
<http://www.liyangyu.com/camera#Nikon_D40>.
```

Notice the usage of the namespace in the second instance file. Since the namespace attribute (`xmlns`) is added, there is no need to use QNames, which is used in the first instance file.

Furthermore, although these two statements are from two different instance files, our application is still able to draw the conclusion based on our camera ontology. Clearly, distributed information over the Web is integrated and processed by the machine because of two facts: first, we have expressed the related knowledge in our ontology, and second, even though the information is distributed all over the Web, the idea of using URIs to identify resources is the clue that connects them all.

5.3.7.5 Functional Properties

A functional property describes the situation where, for any given instance, there is at most one value for a property. In other words, it defines a many-to-one relation: there is at most one unique `rdfs:range` value for each `rdfs:domain` instance.

A good example would be our `manufactured_by` property. A camera includes a lens and a camera body, both of which include a number of different parts, and these parts can indeed be made in different countries around the world. If we ignore this complexity at this point, we can simply say that one given camera can only have one manufacturer, such as Nikon D300 is manufactured by Nikon Corporation. Clearly, different cameras can be manufactured by the same manufacturer.

List 5.24 shows a revised definition of `manufactured_by` property:

List 5.24 Example of functional property

```
1: <owl:ObjectProperty
1a:   rdf:about="http://www.liyangyu.com/camera#manufactured_by">
2:   <rdf:type rdf:resource=
2a:      "http://www.w3.org/2002/07/owl#FunctionalProperty"/>
3:   <rdfs:domain rdf:resource="#Camera"/>
4: </owl:ObjectProperty>
```

To see another example of a functional property, let us revisit the definition of property `effectivePixel`. Clearly, for a given digital camera, it has only one `effectivePixel` value, and we have indicated this fact by defining `Digital` and `effectivePixel` as shown in List 5.25:

List 5.25 Definitions of `Digital` class and `effectivePixel` property

```
<owl:Class rdf:about="http://www.liyangyu.com/camera#Digital">
  <rdfs:subClassOf rdf:resource="#Camera"/>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#effectivePixel"/>
      <owl:cardinality rdf:datatype=
          "http://www.w3.org/2001/XMLSchema#nonNegativeInteger">
         1
      </owl:cardinality>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>

<owl:ObjectProperty
     rdf:about="http://www.liyangyu.com/camera#effectivePixel">
  <rdfs:domain rdf:resource="#Digital"/>
  <rdfs:range
        rdf:resource="http://www.liyangyu.com/camera#MegaPixel"/>
</owl:ObjectProperty>
<rdfs:Datatype
      rdf:about="http://www.liyangyu.com/camera#MegaPixel">
  <rdfs:subClassOf
        rdf:resource="http://www.w3.org/2001/XMLSchema#decimal"/>
</rdfs:Datatype>
```

As you can see, `owl:cardinality` is used to accomplish the goal. In fact, this is equivalent to the following definitions shown in List 5.26:

List 5.26 Definition of `Digital` class and `effectivePixel` property (equivalent to List 5.25)

```
<owl:Class rdf:about="http://www.liyangyu.com/camera#Digital">
   <rdfs:subClassOf rdf:resource="#Camera"/>
</owl:Class>

<owl:FunctionalProperty
     rdf:about="http://www.liyangyu.com/camera#effectivePixel">
  <rdfs:domain rdf:resource="#Digital"/>
  <rdfs:range
        rdf:resource="http://www.liyangyu.com/camera#MegaPixel"/>
</owl:FunctionalProperty>
<rdfs:Datatype
      rdf:about="http://www.liyangyu.com/camera#MegaPixel">
  <rdfs:subClassOf
        rdf:resource="http://www.w3.org/2001/XMLSchema#decimal"/>
</rdfs:Datatype>
```

Therefore, a class with a property that has an `owl:cardinality` equal to 1 is the same as the class which has the same property defined as a functional property. OWL not only provides us with a much richer vocabulary to express more complex knowledge, but also gives us different routes to accomplish the same goal.

Notice `owl:FunctionalProperty` is a subclass of `rdf:Property`. Therefore, `rdfs:range` of a functional property can be a resource, a literal or a data type.

5.3.7.6 Enhanced Reasoning Power 9

Our application collects the following statement from one instance document:

```
<myCamera:DSLR
          rdf:about="http://www.liyangyu.com/camera#Nikon_D300">
  <myCamera:manufactured_by
           rdf:resource="http://dbpedia.org/resource/Nikon"/>
</myCamera:DSLR>
```

and in another RDF document, our application finds this statement:

```
<DSLR rdf:about="http://www.liyangyu.com/camera#Nikon_D300"
      xmlns="http://www.liyangyu.com/camera#">
<manufactured_by rdf:resource=
                 "http://www.freebase.com/view/en/nikon"/>
</DSLR>
```

our application will add the following statement:

```
<http://dbpedia.org/resource/Nikon> owl:sameAs
<http://www.freebase.com/view/en/nikon>.
```

Since property `manufactured_by` is defined as a functional property, and since the two instance files are both describing the same resource identified by `myCamera:Nikon_D300`, it is therefore straightforward to see the reason why the above statement can be inferred.

Notice that the URI `http://dbpedia.org/resource/Nikon`, coined by DBpedia, represents Nikon Corporation. Similarly, the following URI was created by freebase[8] to represent the same company,

```
http://www.freebase.com/view/en/nikon
```

[8] http://www.freebase.com/

and freebase is another experimental Web site in the area of the Semantic Web. Understanding the fact these two URIs represent the same company in real life is not a big deal for human minds and eyes. However, for machines to understand this fact is great progress, and we can easily imagine how much this will help us in a variety of applications.

5.3.7.7 Inverse Property

An inverse property describes the situation where 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`, than we can define an inverse property of `owned_by`, say, `own`, to indicate that the `Photographer own` the camera. This example is given in List 5.27.

List 5.27 Example of inverse property

```
1: <owl:ObjectProperty
1a:     rdf:about="http://www.liyangyu.com/camera#owned_by">
2:   <rdfs:domain rdf:resource="#DSLR"/>
3:   <rdfs:range rdf:resource="#Photographer"/>
4: </owl:ObjectProperty>
5: <owl:ObjectProperty
5a:     rdf:about="http://www.liyangyu.com/camera#own">
6:   <owl:inverseOf rdf:resource="#owned_by"/>
7:   <rdfs:domain rdf:resource="#Photographer"/>
8:   <rdfs:range rdf:resource="#DSLR"/>
9: </owl:ObjectProperty>
```

Notice that compared to the definition of property `owned_by`, property `own`'s values for `rdfs:domain` and `rdfs:range` are flipped from that in `owned_by`. Also note that `owl:inverseOf` is a property, not a class (recall that `owl:FunctionalProperty` is a subclass of `rdf:Property`). Therefore, it cannot be used to connect any `rdfs:domain` to any `rdfs:range`; it is only used as a constraint when some other property is being defined, as shown in List 5.27.

5.3.7.8 Enhanced Reasoning Power 10

Our application collects the following statement from a given instance document:

```
<myCamera:Photographer
          rdf:about="http://www.liyangyu.com/people#Liyang">
  <myCamera:own
     rdf:resource="http://www.liyangyu.com/camera#Nikon_D300"/>
</myCamera:Photographer>
```

and once it realizes the fact that `own` is an inverse property of `owned_by`, it will add the following statement, without us doing anything:

```
<http://www.liyangyu.com/camera#Nikon_D300> myCamera:owned_by
<http://www.liyangyu.com/people#Liyang>.
```

5.3.7.9 Inverse Functional Property

Recall the functional property discussed earlier: for a given `rdfs:domain` value, there is a unique `rdfs:range` value. An inverse functional property, as its name suggests, is just the opposite of functional property: for a given `rdfs:range` value, the value of the `rdfs:domain` property must be unique.

Let us go back to the camera review example, and also assume the reviewers themselves are often photographers. We would like to assign a unique reviewer ID to each photographer, so when they submit a review for a given camera, they can add their reviewer IDs into the submitted RDF documents.

The `reviewerID` property, in this case, should be modeled as an inverse functional property. Therefore, if two photographers have the same `reviewerID`, these two photographers should be the same person. List 5.28 shows the definition of `reviewerID` property.

List 5.28 Example of inverse functional property

```
1: <owl:DatatypeProperty
1a:     rdf:about="http://www.liyangyu.com/camera#reviewerID">
2:   <rdf:type rdf:resource=
2a:    http://www.w3.org/2002/07/owl#InverseFunctionalProperty"/>
3:   <rdfs:domain rdf:resource="#Photographer"/>
4:   <rdfs:range
4a:      rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
5: </owl:DatatypeProperty>
6: <rdfs:Datatype
6a:      rdf:about="http://www.w3.org/2001/XMLSchema#string"/>
```

In fact, we can make an even stronger statement about photographers and their reviewer IDs: not only is one reviewer ID is used to identify just one photographer, each photographer has only one reviewer ID. Therefore, we can define the `reviewerID` property as both a functional and an inverse functional property as shown in List 5.29.

List 5.29 A property can be both a functional and an inverse functional property

```
1: <owl:DatatypeProperty
1a:     rdf:about="http://www.liyangyu.com/camera#reviewerID">
2:   <rdf:type rdf:resource=
2a:       "http://www.w3.org/2002/07/owl#FunctionalProperty"/>
3:   <rdf:type rdf:resource=
3a:    "http://www.w3.org/2002/07/owl#InverseFunctionalProperty"/>
4:   <rdfs:domain rdf:resource="#Photographer"/>
5:   <rdfs:range
5a:      rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
6: </owl:DatatypeProperty>
7: <rdfs:Datatype
7a:      rdf:about="http://www.w3.org/2001/XMLSchema#string"/>
```

It is important to understand the difference between functional and inverse functional properties. A good example you can use to make the difference clear is the birthdate property: any given person can have only one birthdate; therefore, birthdate is a functional property. Is the birthdate property also an inverse functional property? Certainly not, because many people can have the same birthdate; if it were indeed an inverse functional property, then for a given date, only one person could be born on that date.

Similarly, e-mail as a property, should be an inverse functional property, because an e-mail address belongs to only one person. However, e-mail cannot be a functional property since one person can have several e-mail accounts, like most of us do.

Most ID-like properties are functional properties and inverse functional properties at the same time. For example, think of social security number, student ID, driver's license and passport number, just to name a few.

Finally, notice that `owl:InverseFunctionalProperty` is a subclass of `rdf:Property`. Therefore, `rdfs:range` of an inverse functional property can be a resource, a literal or a data type.

5.3.7.10 Enhanced Reasoning Power 11

Our application collects the following statement from one instance document:

```
<myCamera:Photographer
          rdf:about="http://www.liyangyu.com/camera#Liyang">
  <myCamera:reviewerID>reviewer-0910</myCamera:reviewerID>
</myCamera:Photographer>
```

and in another RDF document, our application finds this statement:

```
<myCamera:Photographer
          rdf:about="http://liyangyu.com/foaf.rdf#Liyang">
  <myCamera:reviewerID>reviewer-0910</myCamera:reviewerID>
</myCamera:Photographer>
```

our application will add the following statement:

```
<http://www.liyangyu.com/camera#Liyang> owl:sameAs
<http://liyangyu.com/foaf.rdf#Liyang>.
```

Since property `reviewerID` is defined as an inverse functional property, the reason behind the above statement is obvious.

5.3.8 Camera Ontology Written Using OWL 1

At this point, we have covered most of the OWL 1 language features, and our current version of the camera ontology is given in List 5.30. Notice that the class definitions are not changed (compared to List 5.17), but we have added some new properties and also modified some existing properties.

List 5.30 Camera ontology written in OWL 1

```
1: <?xml version="1.0"?>
2: <!DOCTYPE rdf:RDF [
3:     <!ENTITY owl "http://www.w3.org/2002/07/owl#" >
4:     <!ENTITY xsd "http://www.w3.org/2001/XMLSchema#" >
5:     <!ENTITY rdfs "http://www.w3.org/2000/01/rdf-schema#" >
6:     <!ENTITY myCamera "http://www.liyangyu.com/camera#" >
7: ]>
8:
9: <rdf:RDF
9a:     xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
10:     xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
11:     xmlns:owl="http://www.w3.org/2002/07/owl#"
12:     xmlns:myCamera="http://www.liyangyu.com/camera#"
13:     xml:base="http://www.liyangyu.com/camera#">
14:
15:  <owl:Class rdf:about="&myCamera;Camera">
16:    <rdfs:subClassOf>
17:      <owl:Restriction>
18:        <owl:onProperty rdf:resource="&myCamera;model"/>
19:        <owl:minCardinality
20:             rdf:datatype="&xsd;nonNegativeInteger">
21:          1
22:        </owl:minCardinality>
23:      </owl:Restriction>
24:    </rdfs:subClassOf>
25:  </owl:Class>
26:
27:  <owl:Class rdf:about="&myCamera;Lens">
28:  </owl:Class>
29:
30:  <owl:Class rdf:about="&myCamera;Body">
31:  </owl:Class>
32:
33:  <owl:Class rdf:about="&myCamera;ValueRange">
34:  </owl:Class>
35:
36:  <owl:Class rdf:about="&myCamera;Digital">
37:    <rdfs:subClassOf rdf:resource="&myCamera;Camera"/>
38:    <rdfs:subClassOf>
39:      <owl:Restriction>
40:        <owl:onProperty
41:        <owl:cardinality
42:             rdf:datatype="&xsd;nonNegativeInteger">
43:          1
44:        </owl:cardinality>
```

```
45:      </owl:Restriction>
46:    </rdfs:subClassOf>
47:  </owl:Class>
48:
49:  <owl:Class rdf:about="&myCamera;Film">
50:    <rdfs:subClassOf rdf:resource="&myCamera;Camera"/>
51:  </owl:Class>
52:
53:  <owl:Class rdf:about="&myCamera;DSLR">
54:    <rdfs:subClassOf rdf:resource="&myCamera;Digital"/>
55:  </owl:Class>
56:
57:  <owl:Class rdf:about="&myCamera;PointAndShoot">
58:    <rdfs:subClassOf rdf:resource="&myCamera;Digital"/>
59:  </owl:Class>
60:
61:  <owl:Class rdf:about="&myCamera;Photographer">
62:    <rdfs:subClassOf
62a:         rdf:resource="http://xmlns.com/foaf/0.1/Person"/>
63:  </owl:Class>
64:
65:  <owl:Class rdf:about="&myCamera;Professional">
66:    <rdfs:subClassOf rdf:resource="&myCamera;Photographer"/>
67:  </owl:Class>
68:
69:  <owl:Class rdf:about="&myCamera;Amateur">
70:    <owl:intersectionOf rdf:parseType="Collection">
71:      <owl:Class
71a:          rdf:about="http://xmlns.com/foaf/0.1/Person"/>
72:      <owl:Class>
73:        <owl:complementOf
73a:            rdf:resource="&myCamera;Professional"/>
74:      </owl:Class>
75:    </owl:intersectionOf>
76:  </owl:Class>
77:
78:  <owl:Class rdf:about="&myCamera;ExpensiveDSLR">
79:    <rdfs:subClassOf rdf:resource="&myCamera;DSLR"/>
80:    <rdfs:subClassOf>
81:      <owl:Restriction>
82:        <owl:onProperty rdf:resource="&myCamera;owned_by"/>
83:        <owl:someValuesFrom
83a:            rdf:resource="&myCamera;Professional"/>
84:      </owl:Restriction>
```

```
85:     </rdfs:subClassOf>
86:   </owl:Class>
87:
88:   <owl:ObjectProperty rdf:about="&myCamera;owned_by">
89:     <rdfs:domain rdf:resource="&myCamera;DSLR"/>
90:     <rdfs:range rdf:resource="&myCamera;Photographer"/>
91:   </owl:ObjectProperty>
92:
93:   <owl:ObjectProperty rdf:about="&myCamera;manufactured_by">
94:     <rdf:type rdf:resource="&owl;FunctionalProperty"/>
95:     <rdfs:domain rdf:resource="&myCamera;Camera"/>
96:   </owl:ObjectProperty>
97:
98:   <owl:ObjectProperty rdf:about="&myCamera;body">
99:     <rdfs:domain rdf:resource="&myCamera;Camera"/>
100:    <rdfs:range rdf:resource="&myCamera;Body"/>
101:  </owl:ObjectProperty>
102:
103:  <owl:ObjectProperty rdf:about="&myCamera;lens">
104:    <rdfs:domain rdf:resource="&myCamera;Camera"/>
105:    <rdfs:range rdf:resource="&myCamera;Lens"/>
106:  </owl:ObjectProperty>
107:
108:  <owl:DatatypeProperty rdf:about="&myCamera;model">
109:    <rdfs:domain rdf:resource="&myCamera;Camera"/>
110:    <rdfs:range rdf:resource="&xsd;string"/>
111:  </owl:DatatypeProperty>
112:  <rdfs:Datatype rdf:about="&xsd;string"/>
113:
114:  <owl:ObjectProperty rdf:about="&myCamera;effectivePixel">
115:    <rdfs:domain rdf:resource="&myCamera;Digital"/>
116:    <rdfs:range rdf:resource="&myCamera;MegaPixel"/>
117:  </owl:ObjectProperty>
118:  <rdfs:Datatype rdf:about="&myCamera;MegaPixel">
119:    <rdfs:subClassOf rdf:resource="&xsd;decimal"/>
120:  </rdfs:Datatype>
121:
122:  <owl:ObjectProperty rdf:about="&myCamera;shutterSpeed">
123:    <rdfs:domain rdf:resource="&myCamera;Body"/>
124:    <rdfs:range rdf:resource="&myCamera;ValueRange"/>
125:  </owl:ObjectProperty>
126:
127:  <owl:DatatypeProperty rdf:about="&myCamera;focalLength">
128:    <rdfs:domain rdf:resource="&myCamera;Lens"/>
```

```
129:   <rdfs:range rdf:resource="&xsd;string"/>
130: </owl:DatatypeProperty>
131: <rdfs:Datatype rdf:about="&xsd;string"/>
132:
133: <owl:ObjectProperty rdf:about="&myCamera;aperture">
134:   <rdfs:domain rdf:resource="&myCamera;Lens"/>
135:   <rdfs:range rdf:resource="&myCamera;ValueRange"/>
136: </owl:ObjectProperty>
137:
138: <owl:DatatypeProperty rdf:about="&myCamera;minValue">
139:   <rdfs:domain rdf:resource="&myCamera;ValueRange"/>
140:   <rdfs:range rdf:resource="&xsd;float"/>
141: </owl:DatatypeProperty>
142: <rdfs:Datatype rdf:about="&xsd;float"/>
143:
144: <owl:DatatypeProperty rdf:about="&myCamera;maxValue">
145:   <rdfs:domain rdf:resource="&myCamera;ValueRange"/>
146:   <rdfs:range rdf:resource="&xsd;float"/>
147: </owl:DatatypeProperty>
148: <rdfs:Datatype rdf:about="&xsd;float"/>
149:
150: <owl:ObjectProperty rdf:about="&myCamera;own">
151:   <owl:inverseOf rdf:resource="&myCamera;owned_by"/>
152:   <rdfs:domain rdf:resource="&myCamera;Photographer"/>
153:   <rdfs:range rdf:resource="&myCamera;DSLR"/>
154: </owl:ObjectProperty>
155:
156: <owl:DatatypeProperty rdf:about="&myCamera;reviewerID">
157:   <rdf:type rdf:resource="&owl;FunctionalProperty"/>
158:   <rdf:type rdf:resource="&owl;InverseFunctionalProperty"/>
159:   <rdfs:domain rdf:resource="&myCamera;Photographer"/>
160:   <rdfs:range rdf:resource="&xsd;string"/>
161: </owl:DatatypeProperty>
162: <rdfs:Datatype rdf:about="&xsd;string"/>
163:
164: </rdf:RDF>
```

5.4 OWL 2 Web Ontology Language

In this section, we discuss the latest W3C standard, the OWL 2 language. We first discuss the new features in general, and then move on to each individual language feature. Similarly, we use examples to illustrate the usage of these new features.

5.4.1 What Is New in OWL 2

OWL 2 offers quite an impressive list of new features, which can be roughly categorized into the following five major categories:

1. Syntactic sugar to make some common statements easier to construct.
   These features are called syntactic sugar because these new constructs do not alter the reasoning process built upon the ontology that uses these constructs; they are there to make the language easier to use. You will see more details in the next few sections.

2. New constructs that improve expressiveness.
   These features indeed increase the expressiveness. Examples include a collection of new properties, such as reflexive property, irreflexive property and asymmetric property, just to name a few. Also, the new qualified cardinality constraints greatly enhance the expressiveness of the language, as do the new features such as property chains and keys.

3. Extended support for datatypes.
   This includes more built-in datatypes being offered by OWL 2. In addition, OWL 2 allows users to define their own datatypes when creating ontologies. As you will see in later sections, these features can be very powerful to use.

4. Simple metamodeling capabilities and extended annotation capabilities.
   The metamodeling capability includes a new feature called punning. Annotation is also quite powerful in OWL 2. More specifically, you can add annotation to axioms, add domain and range information to annotation properties, and you can also add annotation information to annotations themselves.

5. New sublanguages: the profiles.
   Another feature offered by OWL 2 is its sublanguages, namely, OWL 2 EL, OWL 2 QL and OWL 2 PL. These language profiles offer different levels of tradeoff between expressiveness and efficiency, and therefore offer more choices to the users.

5.4.2 New Constructs for Common Patterns

This part of the new features is commonly referred to as the *syntactic sugar*, meaning that these features are simply shorthand; they do not change the expressiveness or the semantics. You can accomplish the same goals using OWL 1, but these constructs can make your ontology document more concise, as you will see in this section.

5.4.2.1 Common Pattern: Disjointness

In OWL 1, we can use `owl:disjointWith` to specify the fact that two classes are disjoint (see List 5.16). However, this can be used only on two classes. Therefore, to specify that several classes are mutually disjoint, `owl:disjointWith` has to be used on all the possible class pairs. For instance, if we have four classes, we need to use `owl:disjointWith` altogether six times.

OWL 2 provides a new construct called `owl:AllDisjointClasses` so we can do this with more ease. For example, List 5.16 specifies the fact that `DSLR` and `PointAndShoot` are disjoint to each other. For illustration purpose, let us say now we want to specify the fact that `Film` camera, `DSLR` camera and `PointAndShoot` camera are all disjoint (here we make the things a lot simpler by ignoring the fact that a `DSLR` camera can be used as a `PointAndShoot` camera, and also that a `Film` camera can be a `PointAndShoot` camera). List 5.31 shows how this has to be done using OWL 1's `owl:disjointWith` construct:

List 5.31 Using OWL 1's `owl:disjontWith` to specify three classes are pairwise disjoint

```
<owl:Class rdf:about="&myCamera;DSLR">
  <owl:disjointWith rdf:resource="&myCamera;PointAndShoot"/>
</owl:Class>

<owl:Class rdf:about="&myCamera;DSLR">
  <owl:disjointWith rdf:resource="&myCamera;Film"/>
</owl:Class>

<owl:Class rdf:about="&myCamera;PointAndShoot">
  <owl:disjointWith rdf:resource="&myCamera;Film"/>
</owl:Class>
```

Using OWL 2's construct, this can be as simple as shown in List 5.32.

List 5.32 Example of using `owl:AllDisjointClasses`

```
<owl:AllDisjointClasses>
  <owl:members rdf:parseType="Collection">
    <owl:Class rdf:about="&myCamera;DSLR"/>
    <owl:Class rdf:about="&myCamera;PointAndShoot"/>
    <owl:Class rdf:about="&myCamera;Film"/>
  </owl:members>
</owl:AllDisjointClasses>
```

If we have four classes which are pairwise disjoint, instead of following the pattern shown in List 5.31 and using six separate statements, we can simply add one more line in List 5.32.

Another similar feature is OWL 2's `owl:disjointUnionOf` construct. Recall List 5.10, where we defined one class called `CameraCollection`. This class obviously includes both the extension of `Digital` and the extension of `Film`. However, List 5.10 does not specify the fact that any given camera cannot be a digital camera and at the same time a film camera as well.

Now, again for simplicity, let us assume that `Film` camera, `DSLR` camera and `PointAndShoot` camera are all disjoint; what should we do if we want to define our `CameraCollection` class as a union of class `Film`, class `DSLR` and class `PointAndShoot`, and also indicate the fact that all these classes are pairwise disjoint?

If we do this by using only OWL 1 terms, we can first use `owl:unionOf` to include all the three classes, and then we can use three pairwise disjoint statements to make the distinction clear. However, this solution is as concise as the one shown in List 5.33, which uses OWL 2's `owl:disjointUnionOf` operator.

List 5.33 Example of using `owl:disjointUnionOf` operator

```
<owl:Class rdf:about="&myCamera;CameraCollection">
  <owl:disjointUnionOf>
    <owl:members rdf:parseType="Collection">
      <owl:Class rdf:about="&myCamera;DSLR"/>
      <owl:Class rdf:about="&myCamera;PointAndShoot"/>
      <owl:Class rdf:about="&myCamera;Film"/>
    </owl:members>
  </owl:disjointUnionOf>
  </owl:Class>
```

As we discussed at the beginning of this section, these constructs are simply shortcuts which do not change semantics or expressiveness. Therefore, there is no change in the reasoning power. Any reasoning capability we mentioned when discussing OWL 1 is still applicable here.

5.4.2.2 Common Pattern: Negative Assertions

Another important syntax enhancement from OWL 2 is the so-called *negative fact assertions*. To understand this, recall that OWL 1 provides the means to specify the value of a given property for a given individual; however, it does not offer a construct to *directly* state that an individual does not hold certain values for certain properties. It is true that you can still use only OWL 1's constructs to do this; however, that is normally not the most convenient and straightforward way.

To appreciate the importance of negative fact assertions, consider this statement: Liyang as a photographer does *not* own a Canon EOS-7D camera. This kind of native property assertions are very useful since they can explicitly claim that some fact is not true. In a world with open-end assumptions where anything is possible, this is certainly important.

Since `owl:ObjectProperty` and `owl:DatatypeProperty` are the two possible types of a given property, OWL 2 therefore provides two constructs as follows:

```
owl:NegativeObjectPropertyAssertion
owl:NegativeDataPropertyAssertion
```

List 5.34 shows how to use `owl:NegativePropertyAssertion` to specify the fact that Liyang as a photographer does *not* own a Canon EOS-7D camera (notice that namespace definitions are omitted, which can be found in List 5.30).

List 5.34 Example of using `owl:NegativePropertyAssertion`

```
1: <myCamera:Photographer rdf:about="http://liyangyu.com#liyang">
2: </myCamera:Photographer>
3:
4: <myCamera:DSLR
4a:         rdf:about="http://dbpedia.org/resource/Canon_EOS_7D">
5: </myCamera:DSLR>
6:
7:  <owl:NegativeObjectPropertyAssertion>
8:    <owl:sourceIndividual rdf:resource=
8a:                          "http://liyangyu.com#liyang"/>
9:    <owl:assertionProperty rdf:resource="&myCamera;own"/>
10:   <owl:targetIndividual rdf:resource=
10a:                  "http://dbpedia.org/resource/Canon_EOS_7D"/>
11: </owl:NegativeObjectPropertyAssertion>
```

Notice that lines 1–2 define the `Photographer` resource, and lines 4–5 define the `DSLR` resource. Again, `http://dbpedia.org/resource/Canon_EOS_7D` was coined by DBpedia project, and we are reusing it here to represent the Canon EOS-7D camera. Lines 7–11 state that the `Photographer` resource does not own the `DSLR` resource.

Obviously, the following OWL 2 constructs have to be used together to specify that two individuals are not connected by a property:

```
owl:NegativeObjectPropertyAssertion
owl:sourceIndividual
owl:assertionProperty
owl:targetIndividual
```

Similarly, `owl:NegativeDataPropertyAssertion` is used to say one resource does not have a specific value for a given property. For example, we can say Nikon D300 does not have an `effectivePixel` of 10, as shown in List 5.35.

List 5.35 Example of using owl:NegativeDataPropertyAssertion

```
<owl:NegativeDataPropertyAssertion>
  <owl:sourceIndividual
      rdf:resource="http://dbpedia.org/resource/Nikon_D300"/>
  <owl:assertionProperty rdf:resource="effectivePixel"/>
  <owl:targetValue
     rdf:datatype="http://www.liyangyu.com/camera#MegaPixel">
     10
  </owl:targetValue>
</owl:NegativeDataPropertyAssertion>
```

Again, as a summary, the following OWL 2 constructs have to be used together to finish the task:

```
owl:NegativeDataPropertyAssertion
owl:sourceIndividual
owl:assertionProperty
owl:targetValue
```

5.4.3 Improved Expressiveness for Properties

Recall that compared to RDF Schema, one of the main features offered by OWL 1 is the enhanced expressiveness around property definition and restrictions. These features have greatly improved the reasoning power as well.

Similarly, OWL 2 offers even more constructs for expressing additional restrictions on properties and new characteristics of properties. These features have become the centerpiece of the OWL 2 language, and we cover them in great detail in this section.

5.4.3.1 Property Self Restriction

OWL 1 does not allow the fact that a class is related to itself by some property. However, this feature can be useful in many applications. A new property called `owl:hasSelf` is offered by OWL 2 for this reason.

More specifically, `owl:hasSelf` has the type of `rdf:Property`, and its `rdfs:range` can be any resource. Furthermore, a class expression defined by using `owl:hasSelf` restriction specifies the class of all objects that are related to themselves via the given property.

Our camera ontology does not have the need to use `owl:hasSelf` property, but let us take a look at one example where this property can be useful.

For instance, in computer science, a thread is defined as a running task within a given program. Since a thread can create another thread, multiple tasks can be running at the same time. If we were to define an ontology for computer programming, we could use List 5.36 to define a class called `Thread`:

List 5.36 Example of `owl:hasSelf`

```
<owl:Class rdf:about="&myExample;Thread">
  <owl:equivalentClass>
    <owl:Restriction>
      <owl:onProperty rdf:resource="&myExample;create"/>
      <owl:hasSelf rdf:datatype="&xsd;boolean">true</owl:hasSelf>
    </owl:Restriction>
  </owl:equivalentClass>
</owl:Class>
```

which expresses the idea that all threads can create threads.

5.4.3.2 Property Self Restriction: Enhanced Reasoning Power 12

Our application sees the following statement from an instance document (notice the definition of `Thread` is given in List 5.36):

```
<myExample:Thread
   rdf:about="http://www.liyangyu.com/myExample#webCrawler">
</myExample:Thread>
```

The application will be able to add the following fact automatically:

```
<http://www.liyangyu.com/myExample#webCrawler>
myExample:create <http://www.liyangyu.com/myExample#webCrawler>.
```

5.4.3.3 Property Cardinality Restrictions

Let us go back to List 5.30 and take a look at the definition of `Professional` class. Now, instead of simply saying it is a sub-class of `Photographer`, we would like to

say that any object of `Professional` photographer should own at least one `DSLR` camera. This can be done by using terms from the OWL 1 vocabulary, as shown in List 5.37.

List 5.37 A new definition of `Professional` class in our camera ontology

```
<owl:Class rdf:about="&myCamera;Professional">
  <rdfs:subClassOf rdf:resource="&myCamera;Photographer"/>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="&myCamera;own"/>
      <owl:minCardinality rdf:datatype="&xsd;nonNegativeInteger">
        1
      </owl:minCardinality>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>
```

This is obviously a more expressive definition. Also, given the `rdfs:range` value of property `own` is specified as `DSLR` (see List 5.30), we know that any camera owned by a `Professional` photographer has to be a `DSLR` camera.

Now, what if we want to express the idea that a `Professional` photographer is someone who owns at least one `ExpensiveDSLR` camera? It turns out this is not doable by solely using the terms from the OWL 1 vocabulary, since it does not provide a way to further specify the class type of the instances to be counted, which is required for this case.

Similar requirements are quite common for other applications. For example, we may have the need to specify the fact that a marriage has exactly two persons, one is a female and one is a male. These category of cardinality restrictions are called *qualified cardinality restrictions*, where not only the count of some property is specified, but also the class type (or data range) of the instances to be counted has to be restrained.

OWL 2 provides the following constructs to implement qualified cardinality restrictions:

```
owl:minQualifiedCardinality
owl:maxQualifiedCardinality
owl:qualifiedCardinality
```

List 5.38 shows how `owl:minQualifiedCardinality` is used to define the class `Professional` photographer.

List 5.38 Example of using `owl:minQualifiedCardinality` constraint

```
<owl:Class rdf:about="&myCamera;Professional">
  <rdfs:subClassOf rdf:resource="&myCamera;Photographer"/>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:minQualifiedCardinality
           rdf:datatype="&xsd;nonNegativeInteger">
        1
      </owl:minQualifiedCardinality>
      <owl:onProperty rdf:resource="&myCamera;own"/>
      <owl:onClass rdf:resource="&myCamera;ExpensiveDSLR"/>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>
```

Comparing List 5.38 with List 5.37, you can see that `owl:onClass` is the key construct, which is used to specify the type of the instance to be counted.

Notice that `owl:minQualifiedCardinality` is also called the *at-least* restriction, for obvious reasons. Similarly, `owl:maxQualifiedCardinality` is the *at-most* restriction and `owl:qualifiedCardinality` is the *exact* cardinality restriction. We can replace the at-least restriction in List 5.38 with the other two restrictions respectively to create different definitions of `Professional` class. For example, when `owl:maxQualifiedCardinality` is used, a `Professional` can own at most one `ExpensiveDSLR`, and when `owl:qualifiedCardinality` is used, exactly one `ExpensiveDSLR` should be owned.

5.4.3.4 Property Cardinality Restrictions: Enhanced Reasoning Power 13

Qualified cardinality restrictions can greatly improve the expressiveness of a given ontology, and therefore, the reasoning power based on the ontology is also enhanced.

In the medical field, for example, using qualified cardinality restrictions, we can specify in an ontology the fact that a human being has precisely two limbs which are of type `Leg` and two limbs which are of type `Arm`. It is not hard to imagine this kind of precise information can be very useful for any application that is built to understand this ontology.

Let us understand more about this reasoning power by again using our camera ontology example. Our application sees the following statement from an instance document (notice the definition of `Professional` is given in List 5.38):

```
<myCamera:Professional
          rdf:about="http://www.liyangyu.com/people#Liyang">
</myCamera:Professional>
```

The application will be able to add the following two statements:

```
<http://www.liyangyu.com/people#Liyang> myCamera:own _x.
_x rdf:type myCamera:ExpensiveDSLR.
```

If we submit a query to ask who owns an expensive DSLR camera, this resource, http://www.liyangyu.com/people#Liyang, will be returned as one solution, even though the instance document does not explicitly claim this fact.

5.4.3.5 More About Property Characteristics: Reflexive, Irreflexive and Asymmetric Properties

Being reflexive in real life is quite common. For example, any set is a subset of its self. Also, in a given ontology, every class is its own subclass. If you use a reasoner on our camera ontology given in List 5.30 (we will see how to do this in later chapters), you can see statement like the following,

```
<http://www.liyangyu.com/camera#Camera> rdfs:subClassOf
<http://www.liyangyu.com/camera#Camera> .
```

These are all example of reflexive relations. Furthermore, properties can be reflexive, which means a reflexive property relates everything to itself. For this purpose, OWL 2 provides the `owl:ReflexiveProperty` construct so that we can use it to define reflexive properties.

For the purpose of our camera ontology, none of the properties we have so far is a reflexive property. Nevertheless, the following shows one example of how to define http://example.org/example1#hasRelative as a reflexive property:

```
<owl:ReflexiveProperty
     rdf:about="http://example.org/example1#hasRelative"/>
```

Clearly, every person has himself or herself as a relative, including any individual from the animal world. Also, understand that `owl:ReflexiveProperty` is a subclass of `owl:ObjectProperty`. Therefore, `rdfs:range` of a reflexive property can only be a resource, and cannot be a literal or a data type.

With the understanding or reflexive property, it is easier to understand an irreflexive property. More precisely, no resource can be related to itself by an irreflexive property. For example, nobody can be his own parent:

```
<owl:IrreflexiveProperty
     rdf:about="http://example.org/example1#hasParent"/>
```

Again, owl:IrreflexiveProperty is a subclass of owl:ObjectProperty. Therefore, rdfs:range of an irreflexive property can only be a resource, and cannot be a literal or a data type.

Finally, let us discuss asymmetric properties. Recall by using OWL 1 terms, we can define symmetric properties. For example, if resource A is related to resource B by this property, resource B will be related to A by the same property as well.

Besides symmetric relationships, there are asymmetric ones in the real world. A property is an asymmetric property if it connects A with B, but never connects B with A.

A good example is the owned_by property. Based on the definition in List 5.30, a DSLR camera is owned by a Photographer. Furthermore, we understand that the owned_by relationship should not go the other way around, i.e., a Photographer instance is owned_by a DSLR camera. To ensure this, owned_by can also be defined as a asymmetric property, as shown in List 5.39.

List 5.39 Example of using owl:AsymmetricProperty

```
<owl:AsymmetricProperty rdf:about="&myCamera;owned_by">
  <rdfs:domain rdf:resource="&myCamera;DSLR"/>
  <rdfs:range rdf:resource="&myCamera;Photographer"/>
</owl:AsymmetricProperty>
```

Again, owl:AsymmetricProperty is a subclass of owl:ObjectProperty. Therefore, rdfs:range of a asymmetric property can only be a resource, rather than a literal or a data type.

5.4.3.6 More About Property Characteristics: Enhanced Reasoning Power 14

The benefits of these property characteristics are quite obvious. For example, a major benefit is related to the open-world assumption that OWL makes. In essence, the open-world assumption means that from the absence of a statement alone, a deductive reasoner cannot infer that the statement is false.

This assumption implies a significant amount of computing work for any given reasoner, simply because there are so many unknowns. Therefore, if we can eliminate some unknowns, we will achieve better computing efficiency.

owl:IrreflexiveProperty and owl:AsymmetricProperty can help us in this regard. There will be more statements tagged with a clear true/false flag, and more queries can be answered with certainty. Notice, for example, asymmetric is stronger than simply not symmetric.

owl:ReflexiveProperty can also help us when it comes to reasoning. First notice that it is not necessarily true that every two individuals who are related by a reflexive property are identical. For example, the following statement

```
<http://www.liyangyu.com/people#Liyang> example1:hasRelative
<http://www.liyangyu.com/people#Connie>.
```

is perfectly fine, and the subject and object of this statement each represent a different resource in the real world.

Furthermore, at least the following statements can be added by a given application that understands a reflexive property:

```
<http://www.liyangyu.com/people#Liyang> example1:hasRelative
<http://www.liyangyu.com/people#Liyang>.
<http://www.liyangyu.com/people#Connie> example1:hasRelative
<http://www.liyangyu.com/people#Connie>.
```

Therefore, for a given query about `example1:hasRelative`, you will see more statements (facts) returned as solutions.

5.4.3.7 Disjoint Properties

In Sect. 5.4.2.1, we presented some OWL 2 language constructs that one can use to specify that a set of classes are mutually disjoint. Experiences from real applications suggest that it is also quite useful to have the ability to express the same disjointness of properties. For example, two properties are disjoint if there are no two individual resources that can be connected by both properties.

More specifically, OWL 2 offers the following constructs for this purpose:

```
owl:propertyDisjointWith
owl:AllDisjointProperties
```

`owl:propertyDisjointWith` is used to specify that two properties are mutually disjoint, and it is defined as a property itself. Also, `rdf:Property` is specified as the type for both its `rdfs:domain` and `rdfs:range` value. Given the fact that both `owl:ObjectProperty` and `owl:DatatypeProperty` are subclasses of `rdf:Property`, `owl:propertyDisjointWith` can therefore be used to specify the disjointness of both data type properties and object properties.

A good example from our camera ontology is the `owned_by` and `own` property. Given this statement,

```
<http://dbpedia.org/resource/Nikon_D300> myCamera:owned_by
<http://www.liyangyu.com/people#Liyang>.
```

we know the following statement should not exist:

```
<http://dbpedia.org/resource/Nikon_D300> myCamera:own
<http://www.liyangyu.com/people#Liyang>.
```

Therefore, property owned_by and property own should be defined as disjoint properties. List 5.40 shows the improved definition of property owned_by.

List 5.40 Example of using owl:propertyDisjointWith

```
<owl:AsymmetricProperty rdf:about="&myCamera;owned_by">
  <owl:propertyDisjointWith rdf:resource="&myCamera;own"/>
  <rdfs:domain rdf:resource="&myCamera;DSLR"/>
  <rdfs:range rdf:resource="&myCamera;Photographer"/>
</owl:AsymmetricProperty>
```

The syntax of using owl:propertyDisjointWith on data type properties is quite similar to the one shown in List 5.40, and we will not present any example here.

owl:AllDisjointProperties has a similar syntax as its counterpart, i.e., owl:AllDisjointClasses. For example, the following shows how to specify that a given group of object properties are pairwise disjoint:

```
<owl:AllDisjointProperties>
  <owl:members rdf:parseType="Collection">
    <owl:ObjectProperty rdf:about="&example;property1"/>
    <owl:ObjectProperty rdf:about="&example;property2"/>
    <owl:ObjectProperty rdf:about="&example;property3"/>
  </owl:members>
</owl:AllDisjointProperties>
```

Finally, notice that owl:AllDisjointProperties can be used on data type properties with the same syntax as shown above.

5.4.3.8 Disjoint Properties: Enhanced Reasoning Power 15

The benefits of disjoint properties are again related to the open-world assumption, and these properties can help us to eliminate unknowns. For example, given the definition of owned_by property as shown in List 5.40 and the following statement,

```
<http://dbpedia.org/resource/Nikon_D300> myCamera:owned_by
<http://www.liyangyu.com/people#Liyang>.
```

an application will be able to flag the following statement to be false:

```
<http://dbpedia.org/resource/Nikon_D300> myCamera:own
<http://www.liyangyu.com/people#Liyang>.
```

5.4.3.9 Property Chains

Property chain is a very useful feature introduced by OWL 2. It provides a way for us to define a property in terms of a chain of object properties that connect resources.

One common example used to show the power of property chain is the `hasUncle` relationship. More specifically, assume we have defined the following object properties:

```
example:hasParent rdf:type owl:ObjectProperty.
example:hasBrother rdf:type owl:ObjectProperty.
```

where `example` is a namespace prefix. Now, given the following statements,

```
example:Joe rdf:type foaf:Person;
                    example:hasParent example:John.
example:John rdf:type foaf:Person;
                    example:hasBrother example:Tim.
```

and as human readers, we should be able to understand the fact that Joe has an uncle named Tim.

How can we make our application understand this fact? We could have defined a new property, `example:hasUncle`, as follows:

```
example:hasUncle rdf:type owl:ObjectProperty.
```

And then we could have added one statement to explicitly specify the following fact:

```
example:Joe example:hasUncle example:Tim.
```

This is, however, not the preferred solution. First, an application should be smart enough to infer this fact, and manually adding it seems to be redundant. Second, we can add the facts that are obvious to us, but what about the facts that are not quite obvious? One of the main benefits of having ontology is to help us to find all the implicit facts, especially those that are not too apparent to us.

OWL 2 offers us the property chain feature for this kind of situation. Instead of defining `example:hasUncle` property as above, we can define it by using a property chain as shown in List 5.41 (notice that List 5.41 also includes the definitions of `example:hasParent` and `example:hasBrother` properties).

List 5.41 Example of using `owl:propertyChainAxiom`

```
<owl:ObjectProperty rdf:about="&example;hasParent">
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="&example;hasBrother">
</owl:ObjectProperty>

<rdf:Description rdf:about="&example;hasUncle">
  <owl:propertyChainAxiom rdf:parseType="Collection">
    <owl:ObjectProperty rdf:about="&example;hasParent"/>
    <owl:ObjectProperty rdf:about="&example;hasBrother"/>
  </owl:propertyChainAxiom>
</rdf:Description>
```

List 5.41 defines `example:hasUncle` as a property chain consisting of `example:hasParent` and `example:hasBrother`; any time `example:hasParent` and `example:hasBrother` exist, `example:hasUncle` exists. Therefore, if resource `A` `example:hasParent` resource `B` and resource `B` `example:hasBrother` resource `C`, then `A` `example:hasUncle` resource `C`, a fact that we no longer need to add manually.

With this basic understanding about property chain, let us explore how we can use it in our camera ontology.

If you are into the art of photography, you probably use SLR cameras. An SLR camera, as we know, has a camera body and a removable lens. Therefore, a photographer normally owns a couple of camera bodies, and a collection of camera lenses. One of these lenses will be mounted to one particular camera body to make up a complete camera.

Using our current camera ontology, we can have the following statements:

```
<http://www.liyangyu.com/people#Liyang> myCamera:own
<http://www.liyangyu.com/camera#Nikon_D300>.

<http://www.liyangyu.com/camera#Nikon_D300> myCamera:lens
<http://www.liyangyu.com/camera#Nikon_Lens_10-24mm>.
```

which specify the fact that `http://www.liyangyu.com/people#Liyang` owns a Nikon D300 camera, which uses a Nikon 10–24 mm zoom lens.

As human readers, by reading these two statements, we also understand that `http://www.liyangyu.com/people#Liyang` not only owns the Nikon D300 camera, but also owns the Nikon 10–24 mm zoom lens.

To let our application understand this fact, instead of adding a simple `myCamera:hasLens` property and then manually adding a statement to explicitly specify the lens and photographer ownership, the best solution is to use the property chain to define `myCamera:hasLens` property as shown in List 5.42.

List 5.42 Use property chain to define `myCamera:hasLens` property

```
<rdf:Description rdf:about="&myCamera;hasLens">
  <owl:propertyChainAxiom rdf:parseType="Collection">
    <owl:ObjectProperty rdf:about="&myCamera;own"/>
    <owl:ObjectProperty rdf:about="&myCamera;lens"/>
  </owl:propertyChainAxiom>
</rdf:Description>
```

With this definition in place, machine can reach the same understanding as we have, without the need to manually add the statement.

With the knowledge about property chains, we need to think carefully when defining properties. It is always good to consider the choices between whether to define it as a simple property, or to use property chain for the property. Using property chain will make our ontology more expressive and powerful when inferences are made. Finally, notice that property chain is only used on object properties, not on data type properties.

5.4.3.10 Property Chains: Enhanced Reasoning Power 16

The reasoning power provided by property chain is quite obvious. Given the definition of `myCamera:hasLens` (see List 5.42), if our application sees the following statements:

```
<http://www.liyangyu.com/people#Liyang> myCamera:own
<http://www.liyangyu.com/camera#Nikon_D300>.

<http://www.liyangyu.com/camera#Nikon_D300> myCamera:lens
<http://www.liyangyu.com/camera#Nikon_Lens_10-24mm>.
```

it will add the following statement automatically:

```
<http://www.liyangyu.com/people#Liyang> myCamera:hasLens
<http://www.liyangyu.com/camera#Nikon_Lens_10-24mm>.
```

5.4.3.11 Keys

OWL 2 allows keysSee owl003AhasKey to be defined for a given class. The `owl:hasKey` construct, more specifically, can be used to state that each named instance of a given class is uniquely identified by a property or a set of properties, which can be both data properties or object properties, depending on the specific application.

For example, in our camera ontology, we can use the `myCamera:reviewerID` property as the key for `Photographer` class, as shown in List 5.43.

List 5.43 Example of using `owl:hasKey`

```
<owl:Class rdf:about="&myCamera;Photographer">
  <owl:intersectionOf rdf:parseType="Collection">
    <owl:Class rdf:about="http://xmlns.com/foaf/0.1/Person"/>
    <owl:Class>
      <owl:hasKey rdf:parseType="Collection">
        <owl:DatatypeProperty rdf:about="&myCamera;reviewerID"/>
      </owl:hasKey>
    </owl:Class>
  </owl:intersectionOf>
</owl:Class>
```

With this definition in place, we can use `myCamera:reviewerID` property to uniquely identify any named `myCamera:Photographer` instance. Notice that in this example, we don't have the need to use multiple properties as a key, but you can if your application requires it (notice the `rdf:parseType` attribute for `owl:hasKey` construct).

It is also important to understand the difference between `owl:hasKey` and the `owl:InverseFunctionalProperty` axiom. The main difference is that the property or properties used as the key can only be used with those named individuals of the class on which `owl:hasKey` is defined. On the other hand, it is often true that an `owl:InverseFunctionalProperty` is used on a blank node, as we will see in Chap. 7.

5.4.3.12 Keys: Enhanced Reasoning Power 17

The benefit of having `owl:hasKey` is quite obvious: if two named instances of the class have the same values for each key property (or a single key property), these two individuals are the same. This can be easily understood without any example.

Notice that this is quite useful if two instances of the class are actually harvested from different instance documents over the Web. The identical key values of these two individuals tell us that these two instances, although each has a different URI, actually represent the same resource in the real world. This is one of the reasons why a Linked Data Web is possible, as we will see in later chapters.

5.4.4 Extended Support for Datatypes

As we know, OWL 1 depends on XML Schema (represented by `xsd:` prefix) for its built-in datatypes, and it has generally been working well. However, with more experience gained from ontology development in practice, some further

Table 5.1 Datatypes supported by OWL 2

Category	Supported datatypes
Decimal numbers and integers	`xsd:decimal`, `xsd:integer`, `xsd:nonNegativeInteger`, `xsd:nonPositiveInteger`, `xsd:positiveInteger`, `xsd:negativeInteger`, `xsd:long`, `xsd:int`, `xsd:short`, `xsd:byte`, `xsd:unsignedLong`, `xsd:unsignedInt`, `xsd:unsignedShort`, `xsd:unsignedByte`
Float-point numbers	`xsd:double`, `xsd:float`
Strings	`xsd:string`, `xsd:normalizedString`, `xsd:token`, `xsd:language`, `xsd:Name`, `xsd:NCName`, `xsd:NMTOKEN`
Boolean values	`xsd:boolean`
Binary data	`xsd:hexBinary`, `xsd:base64Binary`
IRIs	`xsd:anyURI`
Time instants	`xsd:dateTime`, `xsd:dateTimeStamp`
XML literals	`rdf:XMLLiteral`

requirements about datatypes have been identified. These new requirements can be summarized as follows:

- A wider range of supported datatypes is needed.
- The capability of adding constraints on datatypes should be supported.
- The capability of creating new user-defined datatypes is also required.

OWL 2 has provided answers to these requirements. The related new features are covered in this section in detail.

5.4.4.1 Wider Range of Supported Datatypes and Extra Built-in Datatypes

OWL 2 provides a wider range of supported datatypes, which are again borrowed from XML Schema Datatypes. Table 5.1 summarizes all the datatypes currently supported by OWL 2, and for details, you can refer to the related OWL 2 specification.[9]

Two new built-in types, namely, `owl:real` and `owl:rational`, are added by OWL 2. The definitions of these two types are shown in Table 5.2.

5.4.4.2 Restrictions on Datatypes and User-Defined Datatypes

OWL 2 allows users to define new datatypes by adding constraints on existing ones. The constraints are added via the so-called *facets*, another concept borrowed from XML schema.

[9] http://www.w3.org/TR/2009/REC-owl2-syntax-20091027/

Table 5.2 Two new built-in datatypes of OWL 2

Datatype	Definition
`owl:real`	The set of all real numbers
`owl:rational`	The set of all rational numbers, it is a subset of `owl:real`, and it contains the value of `xsd:decimal`

Restrictions on XML elements are called facets. The four bounds facets, for example, restrict a value to a specified range:

```
xsd:minInclusive, xsd:minExclusive
xsd:maxInclusive, xsd:maxExclusive
```

Notice that `xsd:minInclusive` and `xsd:maxInclusive` specify boundary values that are included in the valid range, and values that are outside the valid range are specified by `xsd:minExclusive` and `xsd:maxExclusive` facets.

Obviously, the above four bounds facets can be applied only to numeric types; other datatypes may have their own specific facets. For instance, `xsd:length`, `xsd:minLength` and `xsd:maxLength` are the three length facets that can be applied to any of the string-based types.

We will not get into much more detail about facets, but you can always learn more about them from the related XML Schema specifications. For our purpose, we are more interested in the fact that OWL 2 allows us to specify restrictions on datatypes by means of constraining facets.

More specifically, `owl:onDatatype` and `owl:withRestrictions` are the two main OWL 2 language constructs for this purpose. By using these constructs, we can, in fact, define new datatypes.

Let us take a look at one such example. We will define a new datatype called `AdultAge`, where the age has a lower bound of 18 years old. List 5.44 shows how this is done.

List 5.44 Example of using `owl:onDatatype` and `owl:withRestrictions` to define new datatype

```
<rdfs:Datatype rdf:about="&example;AdultAge">
  <owl:onDatatype rdf:resource="&xsd;integer"/>
  <owl:withRestrictions rdf:parseType="Collection">
    <rdf:Description>
      <xsd:minInclusive
           rdf:datatype="&xsd;integer">18</xsd:minInclusive>
    </rdf:Description>
  </owl:withRestrictions>
</rdfs:Datatype>
```

With this definition, `AdultAge`, as a user-defined datatype, can be used as the `rdfs:range` value for some property, like any other built-in datatype. To make this

more interesting, we can use another facet to add an upper bound, thereby creating another new datatype called `PersonAge`, as shown in List 5.45.

List 5.45 Another example of using `owl:onDatatype` and `owl:withRestrictions` to define new datatype

```
<rdfs:Datatype rdf:about="&example;PersonAge">
  <owl:onDatatype rdf:resource="&xsd;integer"/>
  <owl:withRestrictions rdf:parseType="Collection">
    <rdf:Description>
      <xsd:minInclusive
           rdf:datatype="&xsd;integer">0</xsd:minInclusive>
    </rdf:Description>
    <rdf:Description>
      <xsd:maxInclusive
           rdf:datatype="&xsd;integer">150</xsd:maxInclusive>
    </rdf:Description>
  </owl:withRestrictions>
 </rdfs:Datatype>
```

In our camera ontology, we can change the definition of `myCamera:MegaPixel` datatype to make it much more expressive. For example, we can say that any digital camera's effective pixel value should be somewhere between 1.0 megapixel and 24.0 megapixel, as shown in List 5.46:

List 5.46 Define `myCamera:MegaPixel` as a new datatype

```
<rdfs:Datatype rdf:about="&myCamera;MegaPixel">
  <owl:onDatatype rdf:resource="&xsd;integer"/>
  <owl:withRestrictions rdf:parseType="Collection">
    <rdf:Description>
      <xsd:minInclusive
           rdf:datatype="&xsd;decimal">1.0</xsd:minInclusive>
    </rdf:Description>
    <rdf:Description>
      <xsd:maxInclusive
           rdf:datatype="&xsd;decimal">24.0</xsd:maxInclusive>
    </rdf:Description>
  </owl:withRestrictions>
</rdfs:Datatype>
```

Similarly, we can define another new datatype called `myCamera:CameraModel`, which, for example, should be an `xsd:string` with `xsd:maxLength` no longer than 32 characters. We will leave this as an exercise for you — at this point, it should be not difficult at all.

5.4.4.3 Data Range Combinations

Just as new classes can be constructed by combining existing ones, new datatypes can be created by combining existing datatypes. OWL 2 provides the following constructs for this purpose:

`owl:datatypeComplementOf`
`owl:intersectionOf`
`owl:unionOf`

These are quite straightforward to understand. `owl:unionOf`, for example, creates a new datatype by using a union on existing data ranges.

List 5.47 shows the definition of a new datatype called `MinorAge`, which is created by combining two existing datatypes:

List 5.47 Example of using `owl:intersectionOf` and `owl:datatypeComplementOf` to define new datatype

```
<rdfs:Datatype rdf:about="&example;MinorAge">
  <owl:equivalentClass>
    <owl:intersectionOf rdf:parseType="Collection">
      <rdfs:Datatype rdf:about="&example;PersonAge"/>
      <rdfs:Datatype>
        <owl:datatypeComplementOf
             rdf:resource="&example;AdultAge"/>
      </rdfs:Datatype>
    </owl:intersectionOf>
  </owl:equivalentClass>
</rdfs:Datatype>
```

Therefore, a person who has a `MinorAge` is younger than 18 years old.

5.4.5 *Punning and Annotations*

5.4.5.1 Understanding Punning

OWL 1 (more specifically, OWL 1 DL) has strict rules about separation of namespaces. For example, a URI cannot be typed as both a class and an individual in the same ontology.

OWL 2 relaxes this requirement: you can the same IRI for entities of different kinds, thus treating, for example, a resource as both a class and an individual of a class. This feature is referred to as *punning*.

Let us take a look at one example. Recall we have borrowed this URI from DBpedia project, `http://dbpedia.org/resource/Nikon_D300`, to represent a Nikon D300 camera. And obviously, it is an instance of class `myCamera:DSLR`:

```
<myCamera:DSLR
          rdf:about="http://dbpedia.org/resource/Nikon_D300"/>
```

However, there is not just one Nikon D300 camera in the world, Nikon must have produced thousands of them. For example, I have one Nikon D300 myself. I can use the following URI to represent this particular Nikon D300:

```
http://liyangyu.com/resource/Nikon_D300
```

Therefore, it is natural for me to make the following statement:

```
<rdf:Description
     rdf:about="http://www.liyangyu.com/camera#Nikon_D300">
  <rdf:type
     rdf:resource="http://dbpedia.org/resource/Nikon_D300"/>
</rdf:Description>
```

Clearly, `http://dbpedia.org/resource/Nikon_D300` represents a class in this statement. Therefore, this same URI can represent both a class and an individual resource.

Notice, however, the reasoning engine will interpret them as two different and independent entities, that is, entities that are not logically connected but just happen to have the same-looking name. Statements we make about the individual nature of Nikon D300 do not affect the class nature of Nikon D300, and vice versa.

There are also some restrictions in OWL 2 about punning:

- One IRI cannot denote both a datatype property and an object project.
- One IRI cannot be used for both a class and a datatype.

So why is punning useful to us? To put it simply, punning can be used for stating facts about classes and properties themselves.

For example, when we treat http://dbpedia.org/resource/Nikon_D300 as an instance of class myCamera:DSLR, we are using myCamera:DSLR as a metaclass. In fact, punning is also referred to as *metamodelingSee* punning.

Metamodeling is related to annotations (more about annotation in the next section). They both provide ways to associate additional information with classes and properties, and the following rules-of-the-thumb are often applied to determine when to use which construct:

- Metamodeling should be used when the information attached to entities should be considered as part of the domain.
- Annotations should be used when the information attached to entities should not be considered as part of the domain and should not contribute to the logical consequences of the underlying ontology.

As a quick example, the facts that my Nikon D300 is a specific instance of the class of Nikon D300 and Nikon D300 in general is a digital SLR camera are statements about the domain. These facts are therefore better represented in our camera ontology by using metamodeling. In contrast, a statement about who created the IRI that represents a Nikon D300 camera does not describe the actual domain itself, and it should be represented via annotation.

5.4.5.2 OWL Annotations, Axioms About Annotation Properties

Annotation is not something new, but it is enhanced by OWL 2. In this section, we first discuss annotations by OWL 1 (you can still find them in ontologies created by using OWL 1), and we then cover annotation constructs provided by OWL 2.

OWL 1 allows classes, properties, individuals and ontology headers to be annotated with useful information such as labels, comments, authors, creation date, etc. This information could be important if the ontology is to be reused by someone else.

Notice that OWL 1 annotation simply associates property–value pairs to ontology entities, or to the entire ontology itself. This information is merely for human eyes and is not part of the semantics of the ontology, and will therefore be ignored by most reasoning engines. The commonly used annotation constructs offered by OWL 1 are summarized in Table 5.3.

These constructs are quite straightforward and easy to use. For example, the following shows annotation property rdfs:comment is used to add information to myCamera:Lens class, providing a natural language description of its meaning:

```
<owl:Class rdf:about="&myCamera;Lens">
    <rdfs:comment>represents the set of all camera lenses.
    </rdfs:comment>
</owl:Class>
```

Table 5.3 OWL 1's annotation properties

Annotation property	Usage
`owl:versionInfo`	Provides basic information for version control purpose
`rdfs:label`	Supports a natural language label for the resource/property
`rdfs:comment`	Supports a natural language comment about a resource/property
`rdfs:seeAlso`	Provides a way to identify more information about the resource
`rdfs:isDefinedBy`	Provides a link pointing to the source of information about the resource

You will see more examples of using these properties in later chapters.

OWL 1 also offers the ability to create user-defined annotation properties. More specifically, a user-defined annotation property should be defined by using the `owl:AnnotationProperty` construct, before we can use this property. List 5.48 shows an example of using a user-defined annotation property:

List 5.48 Example of using `owl:AnnotationProperty` to declare a user-defined annotation property

```
1: <?xml version="1.0"?>
2: <rdf:RDF
2a:     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:      xmlns:dc="http://www.purl.org/metadata/dublin-core#"
6:      xmlns:myCamera="http://www.liyangyu.com/camera#"
7:      xml:base="http://www.liyangyu.com/camera#">
8:
9:   <owl:AnnotationProperty rdf:about=
9a:            "http://www.purl.org/metadata/dublin-core#date">
10:  </owl:AnnotationProperty>
11:
12:  <owl:Class rdf:about="http://www.liyangyu.com/camera#Lens">
13:    <dc:date rdf:datatype=
13a:               "http://www.w3.org/2001/XMLSchema#date">
13b:     2009-09-10</dc:date>
14:  </owl:Class>
```

The goal is to annotate the date on which class `Lens` was created. To do so, we reuse the terms in Dublin Core and explicitly declare `dc:date` as a user-defined annotation property (lines 9–10) and then use it on class `Lens` (line 13) to signal the date when this class is defined.

With the understanding about annotations in OWL 1, let us take a look at what is offered by OWL 2. An obvious improvement is that OWL 1 did not allow annotations of axioms, but OWL 2 does. As a summary, OWL 2 allows for

annotations on ontologies, entities (classes, properties and individuals), anonymous individuals, axioms and also on annotations themselves.

Without covering all these new features in detail, we will concentrate on how to add annotation information on axioms and how to make statements about annotations themselves.

To add annotation information about a given axiom, we need the following OWL 2 language constructs:

```
owl:Axiom
owl:annotatedSource
owl:annotatedProperty
owl:annotatedTarget
```

List 5.49 shows some possible annotation on the following axiom:

```
<owl:Class rdf:about="&myCamera;DSLR">
  <rdfs:subClassOf rdf:resource="&myCamera;Digital"/>
</owl:Class>
```

List 5.49 Annotations on a given axiom

```
<owl:Axiom>
   <owl:annotatedSource rdf:resource="&myCamera;DSLR"/>
   <owl:annotatedProperty rdf:resource="&rdfs;subClassOf"/>
   <owl:annotatedTarget rdf:resource="&myCamera;Digital"/>
   <rdfs:comment>
     States that every DSLR is a Digital camera.
   </rdfs:comment>
 </owl:Axiom>
```

This probably reminds you of the RDF reification vocabulary that we discussed in Chap. 2. They indeed share some similarities. For example, such annotations are often used in tools to provide natural language text to be displayed in `help` windows.

OWL 2 also allows us to add axioms about annotation properties. For example, we can specify the domain and range of a given annotation property. In addition, annotation properties can participate in an annotation property hierarchy. All these can all be accomplished by using the constructs that you are already familiar with: `rdfs:subPropertyOf`, `rdfs:domain` and `rdfs:range`.

If an annotation property's `rdfs:domain` value has been specified, that annotation property can only be used to add annotations to the entities whose type is the specified type. Similarly, if the `rdfs:range` property of an annotation property has been specified, the added annotation information can only assume values that have the type specified by its `rdfs:range` property. Since these are all quite straightforward, we will skip the examples.

Finally, understand that the annotations we have discussed here carry no semantics in the OWL 2 Direct Semantics (more on this in later sections), with the exception of axioms about annotation properties. These special axioms have no semantic meanings in the OWL 2 Direct Semantics, but they do have the standard RDF semantics in the RDF-based Semantics, via the mapping RDF vocabulary.

5.4.6 Other OWL 2 Features

5.4.6.1 Entity Declarations

As developers, we know most programming languages require us to declare a variable first before we can actually use it. However, this is not the case when developing ontologies using OWL 1: we can use any entity, such as a class, an object property or an individual anywhere in the ontology without any prior announcement.

This can be understood as a convenience for the developers. However, the lack of error checking could also be a problem. In practice, for example, if any entity were mistyped in a statement, there would be no way of catching that error at all.

For this reason, OWL 2 has introduced the notion of *entity declaration*. The idea is that every entity contained in an ontology should be declared first before it can be used in that ontology. Also, a specific type (class, datatype property, object property, datatype, annotation property or individual) should be associated with the declared entity. With this information, OWL 2 supporting tools can check for errors and consistency before the ontology is actually being used.

For example, in our camera ontology, the class `myCamera:Camera` should be declared as follows before its complete class definition:

```
<owl:Class rdf:about="&myCamera;Camera"/>
```

Similarly, the following statements declare a new datatype, an object property and a user-defined annotation property:

```
<rdfs:Datatype rdf:about="&myCamera;MegaPixel"/>
<owl:ObjectProperty rdf:about="&myCamera;own"/>
<owl:AnnotationProperty
     rdf:about="http://www.purl.org/metadata/dublin-core#date"/>
```

To declare an individual, a new OWL construct, `owl:NamedIndividual` can be used. The following statement declares a Nikon D300 camera will be specified in the ontology as an individual:

```
<owl:NamedIndividual
     rdf:about="http://dbpedia.org/resource/Nikon_D300"/>
```

Table 5.4 Top and bottom object/data properties

Type	Usage
`owl:topObjectProperty` (universal object property)	All pairs of individuals are connected by `owl:topObjectProperty`
`owl:bottomObjectProperty` (empty object property)	No individuals are connected by `owl:bottomObjectProperty`
`owl:topDataProperty` (universal data property)	All individuals are connected with all literals by `owl:topDataProperty`
`owl:bottomDataProperty` (empty data property)	No individual is connected with a literal by `owl:bottomDataProperty`

Finally, understand that these declarations are optional, and they do not affect the meanings of OWL 2 ontologies and therefore have no effect on reasoning either. However, using declaration is always recommended to ensure the quality of the ontology.

5.4.6.2 Top and Bottom Properties

OWL 1 has built-in top and bottom entities for classes, namely, `owl:Thing` and `owl:Nothing`. `owl:Thing` represents an universal class, and `owl:Nothing` represents an empty class.

In addition to the above class entities, OWL 2 provides top and bottom object and data properties. These constructs and their usage are summarized in Table 5.4.

5.4.6.3 Imports and Versioning

Imports and versioning are important aspects of the ontology management task. In this section, we first discuss how imports and versioning are handled in OWL 1, since quite a lot of ontologies are created when only OWL 1 is available. We then examine the new features of imports and versioning provided by OWL 2.

To understand handling imports and versioning in OWL 1, we first must understand several related concepts. One of these concepts is the ontology name, which is typically contained in a section called *ontology header*.

The ontology header of a given ontology is part of the ontology document, and it describes the ontology itself. For example, List 5.50 can be the ontology header of our camera ontology:

List 5.50 Ontology header of our camera ontology

```
1: <owl:Ontology rdf:about="">
2:   <owl:versionInfo>v.10</owl:versionInfo>
3:   <rdfs:comment>our camera ontology</rdfs:comment>
4: </owl:Ontology>
```

Line 1 of List 5.50 declares an RDF resource of type `owl:Ontology`, and the name of this resource is given by its `rdf:about` attribute. Indeed, as with anything else in the world, an ontology can be simply treated as a resource. Therefore, we can assign a URI to it and describe it by using the terms from the OWL vocabulary.

Notice in List 5.50, the URI specified by `rdf:about` attribute points to an empty string. In this case, the base URI specified by `xml:base` attribute (line 13, List 5.30) will be taken as the name of this ontology. This is also part of the reason why we have line 13 in List 5.30.

With this said, the following statement will be created by any parser that understands OWL ontology:

```
<http://www.liyangyu.com/camera> rdf:type owl:Ontology.
```

and a given class in this ontology, such as `Camera` class, will have the following URI:

http://www.liyangyu.com/camera#Camera

Similarly, a given property in this ontology, such as `model` property, will have the following URI:

http://www.liyangyu.com/camera#model

and this is exactly what we want to achieve.

Now, with this understanding about the name of a given ontology, let us study how `owl:imports` works in OWL 1. List 5.51 shows a new ontology header which uses `owl:imports` construct:

List 5.51 Our camera ontology header which uses `owl:imports`

```
1: <owl:Ontology rdf:about="">
2:   <owl:versionInfo>v.10</owl:versionInfo>
3:   <rdfs:comment>our camera ontology</rdfs:comment>
4:   <owl:imports
4a:      rdf:resource="http://www.example.org/exampleOntology"/>
5: </owl:Ontology>
```

Clearly, line 4 of List 5.51 tries to import another ontology into our camera ontology. Notice this is used only as an example to show the usage of `owl:imports` construct; there is currently no real need for our camera ontology to import another ontology yet.

First off, understand `owl:imports` is a property with class `owl:Ontology` as both its `rdfs:domain` and `rdfs:range` value. It is used to make reference to another

OWL ontology that contains definitions, and those definitions are considered part of the definitions of the importing ontology. The `rdf:resource` attribute of `owl:imports` specifies the URI (name) of the ontology being imported.

In OWL 1, importing another ontology is done by "name and location". In other words, the importing ontology is required to contain a URI that points to the location of the imported ontology, and this location should match with the name of the imported ontology as well.

One way to understand this "name and location" rule is to think about the possible cases where our camera ontology (List 5.30) is imported by other ontologies. For example, every such importing ontology has to have a statement like this:

```
<owl:imports rdf:resource="http://www.liyangyu.com/camera"/>
```

The importing ontology then expects our camera ontology to have a name specified by `http://www.liyangyu.com/camera`, and our camera ontology has to be located at `http://www.liyangyu.com/camera` as well. This is why we use an empty `rdf:about` attribute (see List 5.50) and at the same time, we specify the `xml:base` attribute on line 13 of List 5.30. By doing so, we can guarantee the location and the name of our camera ontology match each other (since the value of `xml:base` is taken as the name of the ontology), and every importing ontology can find our camera ontology successfully.

This coupling of names and locations in OWL 1 works well when ontologies are published at a fixed location on the Web. However, applications quite often use ontologies off-line (the ontology has been downloaded to some local ontology repositories before hand). Also, ontologies can be moved to other locations. Therefore, in real application world, ontology names and their locations may not match at all.

This has forced the users to manually adjust the names of ontologies and the `owl:imports` statements in the importing ontologies. This is obviously a cumbersome solution to the situation. In addition, these problems get more acute when considering the multiple versions of a given ontology. The specification of OWL 1 provides no guidelines on how to handle such cases at all.

OWL 2's solution is quite simple: it specifies that importing another ontology should be implemented by the location, rather than the name, of the imported ontology.

To understand this, we need to start from ontology version management in OWL. More specifically, in OWL 1, a simple construct called `owl:versionInfo` is used for version management, as shown on line 2 of List 5.51. In OWL 2, on the other hand, a new language construct, `owl:versionIRI`, is introduced to replace `owl:versionInfo`. For example, our camera ontology can have a ontology header as shown in List 5.52.

List 5.52 Ontology header of our camera ontology using

```
<owl:Ontology rdf:about="">
  <owl:versionIRI>
    http://www.liyangyu.com/camera/v1
  </owl:versionIRI>
  <rdfs:comment>our camera ontology</rdfs:comment>
</owl:Ontology>
```

With the usage of `owl:versionIRI`, each OWL 2 ontology has two identifiers: the usual ontology IRI that identifies the name of the ontology, and the value of `owl:versionIRI`, which identifies a particular version of the ontology.

In our case (List 5.52), since the `rdf:about` attribute points to an empty string, the IRI specified by `xml:base` attribute (line 13, List 5.30) is taken as the name of this ontology, which will remain stable. The second identifier for this ontology, i.e., `http://www.liyangyu.com/camera/v1`, is used to represent the current version.

OWL 2 has specified the following rules when publishing an ontology:

- An ontology should be stored at the location specified by its `owl:versionIRI` value.
- The latest version of the ontology should be located at the location specified by the ontology IRI.
- If no version IRI is ever used, the ontology should be located at the location specified by the ontology IRI.

With this said, the importing schema is quite simple:

- If it does not matter which version is desired, the ontology IRI should be the used as the `owl:imports` value.
- If a particular version is needed, the particular version IRI should be used as the `owl:imports` value.

In OWL 2, this is called the "importing by location" rule. With this schema, publishing a new current version of an ontology involves placing the new ontology at the appropriate location as identified by the version IRI, and replacing the ontology located at the ontology URI with this new ontology.

Before we move on to the next topic, there are two more things we need to know about `owl:imports`. First off, notice `owl:imports` property is transitive, that is, if ontology A imports ontology B, and B imports C, then ontology A imports both B and C.

Second, it is true that `owl:imports` property includes other ontologies whose content is assumed to be part of the current ontology, and the imported ontologies provide definitions that can be used directly. However, `owl:imports` does not provide any shorthand notation when it comes to actually using the terms from the imported ontology. Therefore, it is common to have a corresponding namespace declaration for any ontology that is imported.

5.4.7 OWL Constructs in Instance Documents

There are several terms from the OWL vocabulary that can be used in instance documents. These terms can be quite useful, and we cover them here in this section.

The first term is `owl:sameAs`, which is often used to link one individual to another, indicating the two URI references actually refer to the same resource in the world. Obviously, it is unrealistic to assume everyone will use the same URI to represent the same resource; thus URI aliases cannot be avoided in practice, and `owl:sameAs` is a good way to connect these aliases together.

List 5.53 describes Nikon D300 camera as a resource, and it also indicates that the URI coined by DBpedia in fact represents exactly the same resource:

List 5.53 Example of using `owl:sameAs`

```
1: <?xml version="1.0"?>
2: <rdf:RDF
2a:     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:      xmlns:myCamera="http://www.liyangyu.com/camera#">
6:
7:  <rdf:Description
7a:      rdf:about="http://www.liyangyu.com/camera#Nikon_D300">
8:     <rdf:type
8a:         rdf:resource="http://www.liyangyu.com/camera#DSLR"/>
9:     <owl:sameAs
9a:        rdf:resource="http://dbpedia.org/resource/Nikon_D300"/>
10: </rdf:Description>
11:
12: </rdf:RDF>
```

Based on List 5.53, the following two URIs actually represent the same camera, namely, the Nikon D300:

`http://www.liyangyu.com/camera#Nikon_D300`
`http://dbpedia.org/resource/Nikon_D300`

If you happen to have read some earlier documents about OWL, you might have come across another OWL term called `owl:sameIndividualAs`. In fact, `owl:sameAs` and `owl:sameIndividualAs` have the same semantics, and in the published W3C standards, `owl:sameAs` has replaced `owl:sameIndividualAs`. Therefore, avoid using `owl:sameIndividualAs`, and use `owl:sameAs` instead.

`owl:sameAs` can also be used to indicate that two classes denote the same concept in the real world. For example, List 5.54 defines a new class called `DigitalSingleLensReflex`, and it has the same intentional meaning as the class `DSLR`:

List 5.54 Use `owl:sameAs` to define a new class `DigitalSingleLensReflex`

```
1: <?xml version="1.0"?>
2: <rdf:RDF
2a:     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:      xmlns:myCamera="http://www.liyangyu.com/camera#">
6:
7:  <owl:Class rdf:about=
7a:     "http://www.liyangyu.com/camera#DigitalSingleLensReflex">
8:    <owl:sameAs
8a:         rdf:resource="http://www.liyangyu.com/camera#DSLR"/>
9:  </owl:Class>
10:
11: </rdf:RDF>
```

Notice the definition in List 5.55 is quite different from the one in List 5.54:

List 5.55 Use `owl:equivalentClass` to define class `DigitalSingleLensReflex`

```
1: <?xml version="1.0"?>
2: <rdf:RDF
2a:     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:      xmlns:myCamera="http://www.liyangyu.com/camera#">
6:
7: <owl:Class rdf:about=
7a:     "http://www.liyangyu.com/camera#DigitalSingleLensReflex">
8:   <owl:equivalentClass
8a:       rdf:resource="http://www.liyangyu.com/camera#DSLR"/>
9: </owl:Class>
10:
11: </rdf:RDF>
```

List 5.55 defines a new class by using `owl:equivalentClass`. With the term `owl:equivalentClass`, the two classes, namely, `DigitalSingleLensReflex` and `DSLR`, will now have the same class extension (the set of all the instances of a given class is called its extension); however, they do not necessarily denote the same concept at all.

Also notice that `owl:sameAs` is not only used in instance documents, it can be used in ontology documents as well, as shown in List 5.54.

`owl:differentFrom` property is another OWL term that is often used in instance documents. It is the opposite of `owl:sameAs`, and it is used to indicate that two URIs refer to different individuals. List 5.56 shows one example:

List 5.56 Example of using `owl:differentFrom`

```
1: <?xml version="1.0"?>
2: <rdf:RDF
2a:     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:      xmlns:myCamera="http://www.liyangyu.com/camera#">
6:
7:  <rdf:Description
7a:      rdf:about="http://www.liyangyu.com/camera#Nikon_D3X">
8:    <rdf:type
8a:        rdf:resource="http://www.liyangyu.com/camera#DSLR"/>
9:    <owl:differentFrom
9a:     rdf:resource="http://www.liyangyu.com/camera#Nikon_D3S"/>
10: </rdf:Description>
11:
12: </rdf:RDF>
```

The code snippet in List 5.56 clearly states the following two URIs represent different resources in the real world, so there is no confusion even though these two URIs do look like each other a lot (notice that D3X and D3S are both real DSLRs by Nikon):

```
http://www.liyangyu.com/camera#Nikon_D3X
http://www.liyangyu.com/camera#Nikon_D3S
```

The last OWL term to discuss here is `owl:AllDifferent`, a special built-in OWL class. To understand it, we need to again mention the so-called *Unique-Names* assumption, which typically holds in the world of database applications, for example. More specifically, this assumption says that individuals with different names are indeed different individuals.

However, this is not the assumption made by OWL, which actually follows the non-unique-names assumption: even if two individuals (or classes or properties) have different names, they can still be the same individual. This can be derived by

inference, or explicitly asserted by using `owl:sameAs`, as shown in List 5.53. The reason why we adopt the non-unique-names assumption in the world of the Semantic Web is simply because it is the most plausible one to make in the given environment.

However, there are some cases where the unique-names assumption does hold. To model this situation, one solution is to repeatedly use `owl:differentFrom` on all the individuals. However, this solution will likely create a large number of statements, since all individuals have to be declared pairwise disjoint.

A special class called `owl:AllDifferent` is provided for this kind of situation. This class has one built-in property called `owl:distinctMembers`, and an instance of `owl:AllDifferent` is linked to a list of individuals by property `owl:distinctMembers`. The intended meaning of such a statement is that all individuals included in the list are all different from each other. An example is given in List 5.57.

List 5.57 Example of using `owl:AllDifferent` and `owl:distinctMembers`

```
1: <?xml version="1.0"?>
2: <rdf:RDF
2a:     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:      xmlns:myCamera="http://www.liyangyu.com/camera#">
6:
7:  <owl:AllDifferent>
8:    <owl:distinctMembers rdf:parseType="Collection">
9:      <myCamera:DSLR
9a:       rdf:about="http://www.liyangyu.com/camera#Nikon_D3"/>
10:     <myCamera:DSLR
10a:      rdf:about="http://www.liyangyu.com/camera#Nikon_D3X"/>
11:     <myCamera:DSLR
11a:      rdf:about="http://www.liyangyu.com/camera#Nikon_D3S"/>
12:     <myCamera:DSLR
12a:     rdf:about="http://www.liyangyu.com/camera#Nikon_D300S"/>
13:     <myCamera:DSLR
13a:      rdf:about="http://www.liyangyu.com/camera#Nikon_D300"/>
14:     <myCamera:DSLR
14a:      rdf:about="http://www.liyangyu.com/camera#Nikon_D700"/>
15:   </owl:distinctMembers>
16: </owl:AllDifferent>
```

Clearly, List 5.57 has accomplished the goal with much more ease. Remember, `owl:distinctMembers` is a special syntactical construct added for convenience and should always be used together with an `owl:AllDifferent` instance as its subject.

5.4.8 OWL 2 Profiles

5.4.8.1 Why Do We Need All These?

An important issue when designing an ontology language is the tradeoff between its expressiveness and the efficiency of the reasoning process. It is generally true that the richer the language is, the more complex and time-consuming the reasoning becomes. Sometimes, the reasoning can become complex enough that it is computationally impossible to finish the reasoning process. The goal therefore is to design a language that has sufficient expressiveness and that is also simple enough to be supported by reasonably efficient reasoning engines.

This is also inevitably the case with OWL: some of its constructs are very expressive; however, they can lead to uncontrollable computational complexities. The tradeoff between the reasoning efficiency and the expressiveness has led to the definitions of different subsets of OWL, and each one of these subsets is aimed at the different levels of this tradeoff.

This has also been the case for OWL 1. Again, since quite a lot ontologies were created when only OWL 1 was available, we first briefly discuss the OWL 1 language subsets, and we then move on to the profiles provided by OWL 2.

5.4.8.2 Assigning Semantics to OWL Ontology: Description Logic vs. RDF-Based Semantics

Once we have an ontology written in OWL (be it OWL 1 or OWL 2), we have two alternative ways to assign meanings to this ontology. The first one is called the *Direct Model–Theoretic Semantics*; the other one is called *RDF-based semantics*.

The reason behind this dual-assignment was largely due to the fact that OWL was originally designed to use a notational variant of Description Logic*See* DL (DL), which has been extensively investigated in the literature, and its expressiveness and computational properties are well-understood. Meanwhile, it was also very important for OWL to be semantically compatible with existing Semantic Web languages such as RDF and RDFS. The semantic differences between DL and RDF made it difficult to satisfy both requirements, and the solution chosen by W3C was to provide two coexisting semantics for OWL, therefore two ways of assigning semantics to a given OWL ontology.

For OWL 1, this dual-assignment directly results in two different dialects of OWL 1, namely *OWL 1 DL* and *OWL 1 Full*. More specifically, OWL 1 DL refers to the OWL 1 ontologies interpreted by using Direct Semantics, and OWL 1 Full refers to those interpreted by using RDF-based semantics.

This dual-assignment continues to be true in OWL 2. Similarly, *OWL 2 DL* refers to the OWL 2 ontologies that have their semantics assigned by using Direct Semantics, and *OWL 2 Full* refers to those ontologies that have their semantics assigned by using RDF-based semantics.

5.4.8.3 Three Faces of OWL 1

At this point, we understand OWL 1 has two different language variants: OWL 1 DL and OWL 1 Full.

OWL 1 DL is designed for users who need maximum expressiveness together with guaranteed computational completeness and *decidability*, meaning that all conclusions are guaranteed to be computable and all computations will be finished in finite time. To make sure this happens, OWL 1 DL supports all OWL 1 language constructs, but they can be used under certain constraints. For example, one such constraint specifies that a class may be a subclass of many classes, but it cannot be an instance of any class (more details coming up).

On the other hand, OWL 1 Full provides maximum expressiveness; there are no syntactic restrictions on the usage of the built-in OWL 1 vocabulary and the vocabulary elements defined in the ontology. However, since it uses the RDF-compatible semantics, its reasoning can be undecidable. In addition, it does not have the constraints that OWL 1 DL has, thereby adding an extra source of undecidability. At the time of this writing, no complete implementation of OWL 1 Full exists, and it is not clear whether OWL 1 Full can be implemented at all in practice.

With the above being said, OWL 1 DL seems to be a good choice if a decidable reasoning process is desired. However, reasoning in OWL 1 DL has a high worst-case computational complexity. To ease this concern, a fragment of OWL 1 DL was proposed by the OWL Working Group, and this subset of OWL 1 DL is called *OWL 1 Lite*.

Therefore, the *three faces of OWL 1* are given by OWL 1 Lite, OWL 1 DL and OWL 1 Full. The following summarizes the relations between these three faces:

- Every legal OWL 1 Lite feature is a legal OWL 1 DL feature; therefore, every legal OWL 1 Lite ontology is a legal OWL 1 DL ontology.
- OWL 1 DL and OWL 1 Full have the same language features; therefore every legal OWL 1 DL ontology is a legal OWL 1 Full ontology.
- Every valid OWL 1 Lite conclusion is a valid OWL 1 DL conclusion.
- Every valid OWL 1 DL conclusion is a valid OWL 1 Full conclusion.

Let us now discuss OWL 1's three faces in more detail. This will not only help you to understand ontologies that are developed by using only OWL 1 constructs, it will also help you to better understand OWL 2 language profiles as well.

- OWL 1 Full

The entire OWL 1 language we have discussed in this chapter is called OWL 1 Full, with every construct we have covered in this chapter being available to the ontology developer. It also allows us to combine these constructs in arbitrary ways with RDF and RDF schema, including mixing the RDF schema definitions with OWL definitions. Any legal RDF document is therefore a legal OWL 1 Full document.

- OWL 1 DL

As we mentioned, OWL 1 DL has the same language feature as OWL 1 Full does, but it has restrictions about the ways in which the constructs from OWL 1 and

RDF can be used. More specifically, the following rules must be observed when building ontologies:

– No arbitrary combination is allowed: a resource can be only a class, a datatype, a datatype 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 (no punning is allowed).
– Restrictions on functional property and inverse functional property: recall these two properties are subclasses of `rdf:Property`, therefore they can connect resource to resource or resource to value. However, in OWL 1 DL, they can only be used with object properties, not datatype properties.
– Restriction on transitive property: `owl:cardinality` cannot be used with transitive properties or their subproperties because these subproperties are transitive properties by implication.
– Restriction on `owl:imports`: if `owl:imports` is used by an OWL 1 DL ontology to import an OWL 1 Full ontology, the importing ontology will not be qualified as an OWL 1 DL.

In addition, OWL 1 Full does not put any constraints on annotation properties; however, OWL 1 DL does:

– Object properties, datatype properties, annotation properties and ontology properties must be mutually disjoint. For example, a property cannot be at the same time a datatype property and an annotation property.
– Annotation properties must not be used in property axioms. In other words, no subproperties or domain/range constraints for annotation properties can be defined.
– Annotation properties must be explicitly declared, as shown in List 5.48.
– The object of an annotation property must be either a data literal, a URI reference or an individual; nothing else is permitted.

• OWL 1 Full

OWL 1 Lite is a further restricted subset of OWL 1 DL, and the following are some of the main restrictions:

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

Remember, you can always find a full list of the features supported by the three faces of OWL 1 from OWL 1's official specifications. It is up to you to understand each version in detail and thus make the right decision in your design and development work.

Recall that List 5.30 is an ontology written only by using OWL 1 features. Let us decide what face this ontology has. Clearly, it is not OWL 1 Lite since we used

`owl:hasValue`, and it is also not OWL 1 DL since we also used functional property on `owl:DatatypeProperty`. Therefore, our camera ontology shown in List 5.30 is an OWL 1 Full version ontology.

There are indeed tools that can help you to decide the particular species of a given OWL 1 ontology. For example, you can find one such tool as this location:

`http://www.mygrid.org.uk/OWL/Validator`

You can try to validate our camera ontology and see its species decided by this tool. This will certainly enhance your understanding about OWL 1 species.

5.4.8.4 Understanding OWL 2 Profiles

Recall we mentioned that for any ontology created by using OWL 1, we had two alternative ways to assign semantics to this ontology. The same situation still holds for OWL 2. In addition, the first method is still called the Direct Model-Theoretic Semantics, and it is specified by the W3C recommendation OWL 2 Web Ontology Language Direct Semantics.[10] The second one is again called RDF-based Semantics, and it is specified by the W3C recommendation OWL 2 Web Ontology Language RDF-Based Semantics.[11] Also, *OWL 2 DL* refers to those OWL 2 ontologies interpreted by using Direct Semantics, and *OWL 2 Full* refers to those ontologies interpreted by using RDF-based semantics. Another way to understand this is to consider the fact that the direct model-theoretic semantics assigns meaning to OWL 2 ontologies by using Description Logic, therefore the name OWL 2 DL.

The differences between these two semantics are generally quite slight. For example, given an OWL 2 DL ontology, inferences drawn using Direct Semantics remain valid inferences under RDF-based semantics. As developers, we need to understand the following about OWL 2 DL vs. OWL 2 Full:

- OWL 2 DL can be viewed as a syntactically restricted version of OWL 2 Full. The restrictions are added and designed to make the implementation of OWL 2 reasoners easier.

More specifically, the reasoners built upon OWL 2 DL can return all "yes or no" answers to any inference request, while OWL 2 Full can be undecidable. At the time of this writing, there are production quality reasoners that cover the entire OWL 2 DL language, but there are no such reasoners for OWL 2 Full yet.

- Under OWL 2 DL, annotations have no formal meaning. However, under OWL Full, there are some extra inferences that can be drawn.

In addition to OWL 2 DL, OWL 2 further specifies language profiles. An OWL 2 *profile* is a trimmed down version of the OWL 2 language that trades some

[10] http://www.w3.org/TR/2009/REC-owl2-direct-semantics-20091027/

[11] http://www.w3.org/TR/2009/REC-owl2-rdf-based-semantics-20091027/

expressive power for efficiency of reasoning. In computational logic, profiles are usually called *fragment*s or *sublanguage*s.

The OWL 2 specification offers three different profiles, and they are called *OWL 2 EL*, *OWL 2 QL* and *OWL 2 RL*. To guarantee a scalable reasoning capability, each one of these profiles has its own limitations regarding its expressiveness. In the next section, we take a closer look at all these three profiles, and we also briefly summarize the best scenarios for using each specific profile. For the details of each profile, you can always consult OWL 2's official specification, i.e., OWL 2 Web Ontology Language Profiles.[12]

5.4.8.5 OWL 2 EL, QL and RL

- OWL 2 EL

OWL 2 EL is designed with very large ontologies in mind. For example, life sciences commonly require applications that depend on large ontologies. These ontologies normally have a huge number of classes and complex structural descriptions. Classification is the main goal of the related applications. With the restrictions added by OWL 2 EL, the complexity of reasoning algorithms (including query answering algorithms) is known to be worst-case polynomial; therefore these algorithms are often called *PTime-complete* algorithms.

More specifically, the following key points summarize the main features of OWL 2 EL:

– allow `owl:someValuesFrom` to be used with class expression or data range;
– allow `owl:hasValue` to be used with individual or literal;
– allow the use of self-restriction `owl:hasSelf`;
– property domains, class/property hierarchies, class intersections, disjoint classes, property chains and keys are fully supported.

And these features are not supported by OWL 2 EL:

– `owl:allValuesFrom` is not supported on both class expression and data range;
– none of the cardinality restrictions is supported;
– `owl:unionOf` and `owl:complementOf` are not supported;
– disjoint properties are not supported;
– irreflexive object properties, inverse object properties, functional and inverse functional object properties, symmetric object properties and asymmetric object properties are not supported;
– the following datatypes are not supported: `xsd:double`, `xsd:float`, `xsd:nonPositiveInteger`, `xsd:positiveInteger`, `xsd:short`, `xsd:long`, `xsd:int`, `xsd:byte`, `xsd:unsignedLong`, `xsd:boolean`, `xsd:unsignedInt`, `xsd:`

[12] http://www.w3.org/TR/2009/REC-owl2-profiles-20091027/

negativeInteger, xsd:unsignedShort, xsd:unsignedByte and xsd:language.

- OWL 2 QL

OWL 2 QL is designed for those applications that involve classical databases and that also need to work with OWL ontologies. For these applications, the interoperability of OWL with database technologies becomes their main concern because the ontologies used in these applications are often used to query large sets of individuals. Therefore, querying answering against large volumes of instance data is the most important reasoning task for these applications.

OWL 2 QL can guarantee polynomial-time performance as well, and this performance is again based on the limited expressive power. Nevertheless, the language constructs supported by OWL 2 QL can represent key features of Entity-relationship and UML diagrams; therefore, it can be used directly as a high level database schema language as well.

More specifically, the following key points summarize the main features of OWL 2 QL:

– allow owl:someValuesFrom to be used, but with restrictions (see below);
– property domains and ranges, property hierarchies, disjoint classes or equivalence of classes (only for subclass-type expressions), symmetric properties, reflexive properties, irreflexive properties, asymmetric properties and inverse properties are supported.

And these features are not supported by OWL 2 QL:

– owl:someValuesFrom is not supported when used on a class expression or a data range in the subclass position;
– owl:allValuesFrom is not supported on both class expression and data range;
– owl:hasValue is not supported when used on an individual or a literal;
– owl:hasKey is not supported;
– owl:hasSelf is not supported;
– owl:unionOf and owl:oneOf are not supported;
– none of the cardinality restrictions is supported;
– property inclusions involving property chains;
– transitive property, functional and inverse functional properties are not supported;
– the following datatypes are not supported: xsd:double, xsd:float, xsd:nonPositiveInteger, xsd:positiveInteger, xsd:short, xsd:long, xsd:int, xsd:byte, xsd:unsignedLong, xsd:boolean, xsd:unsignedInt, xsd:negativeInteger, xsd:unsignedShort, xsd:unsignedByte and xsd:language.

- OWL 2 RL

OWL 2 RL is designed for those applications that require scalable reasoning without sacrificing too much expressive power. Therefore, OWL 2 applications that

Table 5.5 Syntactic restrictions on class expressions in OWL 2 RL

Subclass expressions	Super-class expressions
A class other than `owl:Thing` An enumeration of individuals (`owl:oneOf`) Intersection of class expressions (`owl:intersectionOf`) Union of class expressions (`owl:unionOf`) Existential quantification to a class expression (`owl:someValuesFrom`) Existential quantification to a data range (`owl:someValuesFrom`) Existential quantification to an individual (`owl:hasValue`) Existential quantification to a literal (`owl:hasValue`)	A class other than `owl:Thing` Intersection of classes (`owl:intersectionOf`) Negation (`owl:complementOf`) Universal quantification to a class expression (`owl:allValuesFrom`) Existential quantification to an individual (`owl:hasValue`) At-most 0/1 cardinality restriction to a class expression Universal quantification to a data range (`owl:allValuesFrom`) Existential quantification to a literal (`owl:hasValue`) At-most 0/1 cardinality restriction to a data range

are willing to trade the full expressiveness of the language for efficiency, and RDF (S) applications that need some added expressiveness from OWL 2 are all good candidates for this profile. OWL 2 RL can also guarantee polynomial-time performance.

The design goal of OWL 2 RL is achieved by restricting the use of constructs to certain syntactic positions. Table 5.5, taken directly from OWL 2's official profile specification document, uses `owl:subClassOf` as an example to show the usage patterns that must be followed by the subclass and super-class expressions used with `owl:subClassOf` axiom.

All axioms in OWL 2 RL are constrained in the similar pattern. And furthermore:

- property domains and ranges only for subclass-type expressions; property hierarchies, disjointness, inverse properties, symmetry and asymmetric properties, transitivity properties, property chains, functional and inverse functional properties, irreflexive properties fully supported;
- disjoint unions of classes and reflexive object properties are not supported;
- finally, `owl:real` and `owl:rational` as datatypes are not supported.

5.4.9 Our Camera Ontology in OWL 2

As this point, we have covered the major features offered by OWL 2. Our camera ontology (List 5.30) has also been rewritten using OWL 2 features, as shown in List 5.58. At this point, there are not many tools that support OWL 2 ontologies yet, so we simply list the new camera ontology here and leave it to you to validate it and decide its species, once the related tools are available.

List 5.58 Our camera ontology written using OWL 2 features

```
1: <?xml version="1.0"?>
2: <!DOCTYPE rdf:RDF [
3:     <!ENTITY owl "http://www.w3.org/2002/07/owl#">
4:     <!ENTITY xsd "http://www.w3.org/2001/XMLSchema#">
5:     <!ENTITY rdfs "http://www.w3.org/2000/01/rdf-schema#">
6:     <!ENTITY myCamera "http://www.liyangyu.com/camera#">
7: ]>
8:
9: <rdf:RDF
9a:     xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
10:     xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
11:     xmlns:owl="http://www.w3.org/2002/07/owl#"
12:     xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
13:     xmlns:myCamera="http://www.liyangyu.com/camera#"
14:     xml:base="http://www.liyangyu.com/camera#">
15:
16:   <owl:Ontology rdf:about="">
17:     <owl:versionIRI>
18:       http://www.liyangyu.com/camera/v1
19:     </owl:versionIRI>
20:     <rdfs:comment>our camera ontology</rdfs:comment>
21:   </owl:Ontology>
22:
23:   <owl:Class rdf:about="&myCamera;Camera"/>
24:   <owl:Class rdf:about="&myCamera;Lens"/>
25:   <owl:Class rdf:about="&myCamera;Body"/>
26:   <owl:Class rdf:about="&myCamera;ValueRange"/>
27:   <owl:Class rdf:about="&myCamera;Digital"/>
28:   <owl:Class rdf:about="&myCamera;Film"/>
29:   <owl:Class rdf:about="&myCamera;DSLR"/>
30:   <owl:Class rdf:about="&myCamera;PointAndShoot"/>
31:   <owl:Class rdf:about="&myCamera;Photographer"/>
32:   <owl:Class rdf:about="&myCamera;Professional"/>
33:   <owl:Class rdf:about="&myCamera;Amateur"/>
34:   <owl:Class rdf:about="&myCamera;ExpensiveDSLR"/>
35:
36:   <owl:AsymmetricProperty rdf:about="&myCamera;owned_by"/>
37:   <owl:ObjectProperty rdf:about="&myCamera;manufactured_by"/>
38:   <owl:ObjectProperty rdf:about="&myCamera;body"/>
39:   <owl:ObjectProperty rdf:about="&myCamera;lens"/>
40:   <owl:DatatypeProperty rdf:about="&myCamera;model"/>
41:   <owl:ObjectProperty rdf:about="&myCamera;effectivePixel"/>
42:   <owl:ObjectProperty rdf:about="&myCamera;shutterSpeed"/>
43:   <owl:DatatypeProperty rdf:about="&myCamera;focalLength"/>
```

```
44:   <owl:ObjectProperty rdf:about="&myCamera;aperture"/>
45:   <owl:DatatypeProperty rdf:about="&myCamera;minValue"/>
46:   <owl:DatatypeProperty rdf:about="&myCamera;maxValue"/>
47:   <owl:ObjectProperty rdf:about="&myCamera;own"/>
48:   <owl:DatatypeProperty rdf:about="&myCamera;reviewerID"/>
49:
50:   <rdfs:Datatype rdf:about="&xsd;string"/>
51:   <rdfs:Datatype rdf:about="&myCamera;MegaPixel"/>
52:   <rdfs:Datatype rdf:about="&xsd;float"/>
53:
54:   <owl:Class rdf:about="&myCamera;Camera">
55:     <rdfs:subClassOf>
56:       <owl:Restriction>
57:         <owl:onProperty rdf:resource="&myCamera;model"/>
58:         <owl:minCardinality
59:                   rdf:datatype="&xsd;nonNegativeInteger">
60:          1
61:         </owl:minCardinality>
62:       </owl:Restriction>
63:     </rdfs:subClassOf>
64:   </owl:Class>
65:
66:   <owl:Class rdf:about="&myCamera;Lens">
67:   </owl:Class>
68:
69:   <owl:Class rdf:about="&myCamera;Body">
70:   </owl:Class>
71:
72:   <owl:Class rdf:about="&myCamera;ValueRange">
73:   </owl:Class>
74:
75:   <owl:Class rdf:about="&myCamera;Digital">
76:     <rdfs:subClassOf rdf:resource="&myCamera;Camera"/>
77:     <rdfs:subClassOf>
78:       <owl:Restriction>
79:         <owl:onProperty
79a:             rdf:resource="&myCamera;effectivePixel"/>
80:         <owl:cardinality
81:              rdf:datatype="&xsd;nonNegativeInteger">
82:           1
83:         </owl:cardinality>
84:       </owl:Restriction>
85:     </rdfs:subClassOf>
```

```
86:   </owl:Class>
87:
88:   <owl:Class rdf:about="&myCamera;Film">
89:     <rdfs:subClassOf rdf:resource="&myCamera;Camera"/>
90:   </owl:Class>
91:
92:   <owl:Class rdf:about="&myCamera;DSLR">
93:     <rdfs:subClassOf rdf:resource="&myCamera;Digital"/>
94:   </owl:Class>
95:
96:   <owl:Class rdf:about="&myCamera;PointAndShoot">
97:     <rdfs:subClassOf rdf:resource="&myCamera;Digital"/>
98:   </owl:Class>
99:
100:  <owl:Class rdf:about="&myCamera;Photographer">
101:    <owl:intersectionOf rdf:parseType="Collection">
102:      <owl:Class
102a:          rdf:about="http://xmlns.com/foaf/0.1/Person"/>
103:      <owl:Class>
104:        <owl:hasKey rdf:parseType="Collection">
105:          <owl:DatatypeProperty
105a:             rdf:about="&myCamera;reviewerID"/>
106:        </owl:hasKey>
107:      </owl:Class>
108:    </owl:intersectionOf>
109:  </owl:Class>
110:
111:
112:  <owl:Class rdf:about="&myCamera;Professional">
113:    <rdfs:subClassOf rdf:resource="&myCamera;Photographer"/>
114:    <rdfs:subClassOf>
115:      <owl:Restriction>
116:        <owl:minQualifiedCardinality
116a:           rdf:datatype="&xsd;nonNegativeInteger">
117:          1
118:        </owl:minQualifiedCardinality>
119:        <owl:onProperty rdf:resource="&myCamera;own"/>
120:        <owl:onClass rdf:resource="&myCamera;ExpensiveDSLR"/>
121:      </owl:Restriction>
122:    </rdfs:subClassOf>
123:  </owl:Class>
124:
125:
126:  <owl:Class rdf:about="&myCamera;Amateur">
```

```
127:    <owl:intersectionOf rdf:parseType="Collection">
128:      <owl:Class
128a:          rdf:about="http://xmlns.com/foaf/0.1/Person"/>
129:      <owl:Class>
130:        <owl:complementOf
130a:            rdf:resource="&myCamera;Professional"/>
131:      </owl:Class>
132:    </owl:intersectionOf>
133:  </owl:Class>
134:
135:  <owl:Class rdf:about="&myCamera;ExpensiveDSLR">
136:    <rdfs:subClassOf rdf:resource="&myCamera;DSLR"/>
137:    <rdfs:subClassOf>
138:      <owl:Restriction>
139:        <owl:onProperty rdf:resource="&myCamera;owned_by"/>
140:        <owl:someValuesFrom
140a:            rdf:resource="&myCamera;Professional"/>
141:      </owl:Restriction>
142:    </rdfs:subClassOf>
143:  </owl:Class>
144:
145:  <owl:AsymmetricProperty rdf:about="&myCamera;owned_by">
146:    <owl:propertyDisjointWith rdf:resource="&myCamera;own"/>
147:      <rdfs:domain rdf:resource="&myCamera;DSLR"/>
148:      <rdfs:range rdf:resource="&myCamera;Photographer"/>
149:  </owl:AsymmetricProperty>
150:
151:  <owl:ObjectProperty rdf:about="&myCamera;manufactured_by">
152:    <rdf:type rdf:resource="&owl;FunctionalProperty"/>
153:    <rdfs:domain rdf:resource="&myCamera;Camera"/>
154:  </owl:ObjectProperty>
155:
156:  <owl:ObjectProperty rdf:about="&myCamera;body">
157:    <rdfs:domain rdf:resource="&myCamera;Camera"/>
158:    <rdfs:range rdf:resource="&myCamera;Body"/>
159:  </owl:ObjectProperty>
160:
161:  <owl:ObjectProperty rdf:about="&myCamera;lens">
162:    <rdfs:domain rdf:resource="&myCamera;Camera"/>
163:    <rdfs:range rdf:resource="&myCamera;Lens"/>
164:  </owl:ObjectProperty>
165:
166:  <owl:DatatypeProperty rdf:about="&myCamera;model">
167:    <rdfs:domain rdf:resource="&myCamera;Camera"/>
```

```
168:    <rdfs:range rdf:resource="&xsd;string"/>
169:  </owl:DatatypeProperty>
170:  <rdfs:Datatype rdf:about="&xsd;string"/>
171:
172:  <owl:ObjectProperty rdf:about="&myCamera;effectivePixel">
173:    <rdfs:domain rdf:resource="&myCamera;Digital"/>
174:    <rdfs:range rdf:resource="&myCamera;MegaPixel"/>
175:  </owl:ObjectProperty>
176:
177:  <rdfs:Datatype rdf:about="&myCamera;MegaPixel">
178:    <owl:onDatatype rdf:resource="&xsd;integer"/>
179:    <owl:withRestrictions rdf:parseType="Collection">
180:      <rdf:Description>
181:        <xsd:minInclusive rdf:datatype="&xsd;decimal">
182:          1.0
183:        </xsd:minInclusive>
184:      </rdf:Description>
185:      <rdf:Description>
186:        <xsd:maxInclusive rdf:datatype="&xsd;decimal">
187:          24.0
188:        </xsd:maxInclusive>
189:      </rdf:Description>
190:    </owl:withRestrictions>
191:  </rdfs:Datatype>
192:
193:  <owl:ObjectProperty rdf:about="&myCamera;shutterSpeed">
194:    <rdfs:domain rdf:resource="&myCamera;Body"/>
195:    <rdfs:range rdf:resource="&myCamera;ValueRange"/>
196:  </owl:ObjectProperty>
197:
198:  <owl:DatatypeProperty rdf:about="&myCamera;focalLength">
199:    <rdfs:domain rdf:resource="&myCamera;Lens"/>
200:    <rdfs:range rdf:resource="&xsd;string"/>
201:  </owl:DatatypeProperty>
202:  <rdfs:Datatype rdf:about="&xsd;string"/>
203:
204:  <owl:ObjectProperty rdf:about="&myCamera;aperture">
205:    <rdfs:domain rdf:resource="&myCamera;Lens"/>
206:    <rdfs:range rdf:resource="&myCamera;ValueRange"/>
207:  </owl:ObjectProperty>
208:
209:  <owl:DatatypeProperty rdf:about="&myCamera;minValue">
210:    <rdfs:domain rdf:resource="&myCamera;ValueRange"/>
211:    <rdfs:range rdf:resource="&xsd;float"/>
```

```
212:  </owl:DatatypeProperty>
213:  <rdfs:Datatype rdf:about="&xsd;float"/>
214:
215:  <owl:DatatypeProperty rdf:about="&myCamera;maxValue">
216:    <rdfs:domain rdf:resource="&myCamera;ValueRange"/>
217:    <rdfs:range rdf:resource="&xsd;float"/>
218:  </owl:DatatypeProperty>
219:  <rdfs:Datatype rdf:about="&xsd;float"/>
220:
221:  <owl:ObjectProperty rdf:about="&myCamera;own">
222:    <owl:inverseOf rdf:resource="&myCamera;owned_by"/>
223:    <rdfs:domain rdf:resource="&myCamera;Photographer"/>
224:    <rdfs:range rdf:resource="&myCamera;DSLR"/>
225:  </owl:ObjectProperty>
226:
227:  <owl:DatatypeProperty rdf:about="&myCamera;reviewerID">
228:    <rdf:type rdf:resource="&owl;FunctionalProperty"/>
229:    <rdf:type rdf:resource="&owl;InverseFunctionalProperty"/>
230:    <rdfs:domain rdf:resource="&myCamera;Photographer"/>
231:    <rdfs:range rdf:resource="&xsd;string"/>
232:  </owl:DatatypeProperty>
233:  <rdfs:Datatype rdf:about="&xsd;string"/>
234:
235:  <rdf:Description rdf:about="&myCamera;hasLens">
236:    <owl:propertyChainAxiom rdf:parseType="Collection">
237:      <owl:ObjectProperty rdf:about="&myCamera;own"/>
238:      <owl:ObjectProperty rdf:about="&myCamera;lens"/>
239:    </owl:propertyChainAxiom>
240:  </rdf:Description>
241:
242: </rdf:RDF>
```

Compare this ontology with the one shown in List 5.30; the differences you see are part of the new features offered by OWL 2.

5.5 Summary

We have covered OWL in this chapter, including both OWL 1 and OWL 2. As an ontology development language, OWL fits into the world of the Semantic Web just as RDFS does. However, compared to RDFS, OWL provides a much greater expressiveness, together with much more powerful reasoning capabilities.

The first part of this chapter presents OWL 1, since most of the available ontologies are written by using OWL 1, and quite a few development tools at this point still only support OWL 1. More specifically, you should understand the following main points about OWL 1:

- understand the key terms and related language constructs provided by OWL 1, and understand how to use these terms and language constructs to define classes and properties;
- understand the enhanced expressiveness and reasoning power offered by OWL 1 ontologies, compared to the ontologies defined by using RDFS.

The second part of this chapter focuses on OWL 2. Make sure you understand the following about OWL 2:

- new features provided by OWL 2, such as a collection of new properties, extended support for datatypes and simple metamodeling capabilities, etc.;
- understand how to use the added new features to define ontologies with more expressiveness and enhanced reasoning power.

This chapter also discusses the topic of OWL language profiles. Make sure you understand the following main points about OWL profiles:

- the concept of OWL language profiles, and why these profiles are needed;
- language features and limitations of each profile, and which specific profile should be selected for a given task.

With the material presented in this chapter and Chap. 4, you should be technically sound when it comes to ontology development. In Chap. 14, we present a methodology that will help you further with ontology design and development. Together, this will prepare you well for your work on the Semantic Web.