Chapter 4

Ontologies in OWL

In this chapter, we introduce the ontology language OWL. We focus on an introduction to the syntax and on conveying an intuitive understanding of the semantics. We will also discuss the forthcoming OWL 2 standard. A formal treatment of the semantics will be done later in Chapter 5, and a discussion of OWL software tools can be found in Section 8.5.

We have seen at the end of Chapter 3 that RDF(S) is suitable for modeling simple ontologies and allows the derivation of implicit knowledge. But we have also seen that RDF(S) provides only very limited expressive means and that it is not possible to represent more complex knowledge. For example, it is not possible to model the meaning of the sentences from Fig. 4.1 in RDF(S) in a sufficiently precise way.

For modeling such complex knowledge, expressive representation languages based on formal logic are commonly used. This also allows us to do logical reasoning on the knowledge, and thereby enables the access to knowledge which is only implicitly modeled. OWL is such a language.

The acronym OWL stands for *Web Ontology Language*.[1] Since 2004 OWL is a W3C recommended standard for the modeling of ontologies, and since then has seen a steeply rising increase in popularity in many application domains. Central for the design of OWL was to find a reasonable balance between expressivity of the language on the one hand, and efficient reasoning, i.e. scalability, on the other hand. This was in order to deal with the general observation that complex language constructs for representing implicit knowledge usually yield high computational complexities or even undecidability of

[1] There exist a number of speculations about the origin of the distorted acronym. In particular, it is often said that the acronym was a reference to a character appearing in the book *Winnie the Pooh* by Alan Alexander Milne: the character is an owl which always misspells its name as wol instead of owl.
Historically correct, however, is that the acronym was originally proposed by Tim Finin in an email to www-webont-wg@w3.org, dated 27th of December 2001, which can be found under `http://lists.w3.org/Archives/Public/www-webont-wg/2001Dec/0169.html`: "I prefer the three letter WOL .... How about OWL as a variation. ... it has several advantages: (1) it has just one obvious pronunciation which is easy on the ear; (2) it opens up great opportunities for logos; (3) owls are associated with wisdom; (4) it has an interesting back story."
The mentioned background story concerns an MIT project called *One World Language* by William A. Martin from the 1970s, which was an early attempt at developing a universal language for knowledge representation.

Every project has at least one participant.
Projects are always internal or external projects.
Gisela Schillinger and Anne Eberhardt are the secretaries of Rudi Studer.
The superior of my superior is also my superior.

FIGURE 4.1: Sentences which cannot be modeled in RDF(S) in a sufficiently precise way

reasoning, and therefore unfavorable scalability properties. In order to give the user a choice between different degrees of expressivity, three sublanguages of OWL – called *species* of OWL – have been designed: OWL Full, OWL DL, and OWL Lite. OWL Full contains both OWL DL and OWL Lite, and OWL DL contains OWL Lite. The main differences between the sublanguages are summarized in Fig. 4.2. We will discuss this in more detail in Section 4.2.

We introduce OWL in this chapter by means of a syntax based on RDF. While most of the contents of this chapter should be accessible without any in-depth knowledge about RDF, the reader may occasionally want to refer to Chapter 2, and in particular to Sections 2.1 to 2.3.

4.1 OWL Syntax and Intuitive Semantics

OWL documents are used for modeling OWL ontologies. Two different syntaxes have been standardized in order to express these. One of them is based on RDF and is usually used for data exchange. It is also called *OWL RDF syntax* since OWL documents in RDF syntax are also valid RDF documents. The other syntax is called the *OWL abstract syntax* and is somewhat more readable for humans. However, it is only available for OWL DL, and it will undergo some major changes in the transition to OWL 2. In this chapter, we introduce the RDF syntax since it is more widely used. In Chapter 5 we will present yet another syntax for OWL DL which is very popular among researchers due to its conciseness and because it is stripped of some technicalities. Indeed in later chapters, we will mostly use this latter syntax. The RDF syntax which we now introduce, though, is suitable for data exchange on the Web, which is why it is so important.

An OWL ontology is basically expressed in terms of classes and properties, which we already know from RDF(S). In OWL, however, much more complex relationships between these classes and properties can be described. The sentences in Fig. 4.1 are examples of such complex relationships. We will see how they can be modeled by means of a number of constructors taken from

OWL Full

- contains OWL DL and OWL Lite,
- is the only OWL sublanguage containing all of RDFS,
- very expressive,
- semantically difficult to understand and to work with,
- undecidable,
- supported by hardly any software tools.

OWL DL

- contains OWL Lite and is contained in OWL Full,
- decidable,
- fully supported by most software tools,
- worst-case computational complexity: NExpTime.

OWL Lite

- contained in OWL DL and OWL Full,
- decidable,
- less expressive,
- worst-case computational complexity: ExpTime.

FIGURE 4.2: The three sublanguages of OWL and their most important general properties. Further details can be found in Section 4.2

formal logic. We will introduce them on an intuitive level in this chapter, and will give an in-depth formal treatment of the underlying logical aspects in Chapter 5.

4.1.1 The Header of an OWL Ontology

The header of an OWL document contains information about namespaces, versioning, and so-called annotations. This information has no direct impact on the knowledge expressed by the ontology.

Since every OWL document is an RDF document, it contains a root element. Namespaces are specified in the opening tag of the root, as in the following example.

```
<rdf:RDF
    xmlns     ="http://www.example.org/"
    xmlns:rdf ="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:xsd ="http://www.w3.org/2001/XMLSchema#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
    xmlns:owl ="http://www.w3.org/2002/07/owl#">
```

The second line in this example defines the namespace used for objects without prefix. Note the namespace which should be used for `owl`.

An OWL document may furthermore contain some general information about the ontology. This is done within an `owl:Ontology` element. We give an example.

```
<owl:Ontology rdf:about="">
  <rdfs:comment
      rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
    SWRC ontology, version of June 2007
  </rdfs:comment>
  <owl:versionInfo>v0.7.1</owl:versionInfo>
  <owl:imports rdf:resource="http://www.example.org/foo" />
  <owl:priorVersion
      rdf:resource="http://ontoware.org/projects/swrc" />
</owl:Ontology>
```

Note the first line of this example: it states that the current base URI – usually given by `xml:base` – identifies an instance of the class `owl:Ontology`.

Some header elements are inherited from RDFS, for example the following:

- `rdfs:comment`
- `rdfs:label`
- `rdfs:seeAlso`
- `rdfs:isDefinedBy`

For versioning, the following elements can be used:

```
owl:versionInfo
owl:priorVersion
owl:backwardCompatibleWith
owl:incompatibleWith
owl:DeprecatedClass
owl:DeprecatedProperty
```

`owl:versionInfo` usually has a string as object. With the statements `owl:DeprecatedClass` and `owl:DeprecatedProperty`, parts of the ontology can be described which are still supported, but should not be used any longer. The other versioning elements contain pointers to other ontologies, with the obvious meaning.

It is also possible to import other OWL ontologies using the `owl:imports` element as given in the example above. The content of the imported ontology is then understood as being part of the importing ontology.

4.1.2 Classes, Roles, and Individuals

The basic building blocks of OWL are classes and properties, which we already know from RDF(S), and individuals, which are declared as RDF instances of classes. OWL properties are also called *roles*, and we will use both notions interchangeably.

Classes are defined in OWL using `owl:Class`. The following example states the RDF triple `Professor rdf:type owl:Class`.[2]

```
<rdf:Description rdf:about="Professor">
  <rdf:type rdf:resource="&owl;Class" />
</rdf:Description>
```

Equivalently, the following short form can be used:

```
<owl:Class rdf:about="Professor" />
```

Via `rdf:about="Professor"`, the class gets assigned the name `Professor`, which can be used for references to the class. Instead of `rdf:about` it is also possible to use `rdf:ID`, if the conditions given on page 33 are observed.[3]

[2] We assume that `<!ENTITY owl 'http://www.w3.org/2002/07/owl#'>` has been declared – see Section 2.2.5.

[3] For better readability, we assume that `http://www.example.org/` is the namespace used in all our examples, as declared on page 114.

There are two predefined classes, called `owl:Thing` and `owl:Nothing`. The class `owl:Thing` is the most general class, and has every individual as an instance. The class `owl:Nothing` has no instances by definition.

`owl:Class` is a subclass of `rdfs:Class`. There are some differences, however, which we will discuss in Section 4.2 on the different sublanguages of OWL.

As in RDF, individuals can be declared to be instances of classes. This is called *class assignment*.

```
<rdf:Description rdf:about="rudiStuder">
  <rdf:type rdf:resource="Professor" />
</rdf:Description>
```

Equivalently, the following short form can be used.

```
<Professor rdf:about="rudiStuder" />
```

There are two different kinds of roles in OWL: abstract roles and concrete roles. Abstract roles connect individuals with individuals. Concrete roles connect individuals with data values, i.e. with elements of datatypes. Both kinds of roles are subproperties of `rdf:Property`. However, there are again some differences which we will discuss in Section 4.2 on the different sublanguages of OWL.

Roles are declared similarly as classes.

```
<owl:ObjectProperty rdf:about="hasAffiliation" />
<owl:DatatypeProperty rdf:about="firstName" />
```

The first of these roles is abstract and shall express which organization(s) a given person is affiliated with. The second role is concrete and assigns first names to persons. Domain and range of roles can be declared via `rdfs:domain` and `rdfs:range` as in RDFS.[4]

[4]We assume that `<!ENTITY xsd 'http://www.w3.org/2001/XMLSchema#'>` has been declared – see Section 2.2.5.

xsd:string	xsd:boolean	xsd:decimal
xsd:float	xsd:double	xsd:dateTime
xsd:time	xsd:date	xsd:gYearMonth
xsd:gYear	xsd:gMonthDay	xsd:gDay
xsd:gMonth	xsd:hexBinary	xsd:base64Binary
xsd:anyURI	xsd:token	xsd:normalizedString
xsd:language	xsd:NMTOKEN	xsd:positiveInteger
xsd:NCName	xsd:Name	xsd:nonPositiveInteger
xsd:long	xsd:int	xsd:negativeInteger
xsd:short	xsd:byte	xsd:nonNegativeInteger
xsd:unsignedLong	xsd:unsignedInt	xsd:unsignedShort
xsd:unsignedByte	xsd:integer	

FIGURE 4.3: XML datatypes for OWL

```
<owl:ObjectProperty rdf:about="hasAffiliation">
  <rdfs:domain rdf:resource="Person" />
  <rdfs:range rdf:resource="Organization" />
</owl:ObjectProperty>
<owl:DatatypeProperty rdf:about="firstName">
  <rdfs:domain rdf:resource="Person" />
  <rdfs:range  rdf:resource="&xsd;string" />
</owl:DatatypeProperty>
```

Besides `xsd:string` it is also possible to use `xsd:integer` in OWL. Indeed all XML datatypes from Fig. 4.3 can in principle be used in OWL, but the standard does not require their support. Concrete tools typically support only a selected set of datatypes. `rdfs:Literal` can also be used as datatype.

Just as in RDF, it is also possible to explicitly declare two individuals connected by a role, as in the following example. This is called a *role assignment*. The example also shows that roles may not be functional,[5] as it is possible to give two affiliations for `rudiStuder`.

```
<Person rdf:about="rudiStuder">
  <hasAffiliation rdf:resource="aifb" />
  <hasAffiliation rdf:resource="ontoprise" />
  <firstName rdf:datatype="&xsd;string">Rudi</firstName>
</Person>
```

[5]Functionality of roles will be treated on page 135.

The class inclusions

```
<owl:Class rdf:about="Professor">
  <rdfs:subClassOf rdf:resource="FacultyMember" />
</owl:Class>
<owl:Class rdf:about="FacultyMember">
  <rdfs:subClassOf rdf:resource="Person" />
</owl:Class>
```

allow us to infer that `Professor` is a subclass of `Person`.

FIGURE 4.4: Logical inference by transitivity of `rdfs:subClassOf`

In this book, we adhere to a common notational convention that names of classes start with uppercase letters, while names for roles and individuals start with lowercase letters. This is not required by the W3C recommendation, but it is good practice and enhances readability.

4.1.3 Simple Class Relations

OWL classes can be put in relation to each other via `rdfs:subClassOf`. A simple example of this is the following.

```
<owl:Class rdf:about="Professor">
  <rdfs:subClassOf rdf:resource="FacultyMember" />
</owl:Class>
```

The construct `rdfs:subClassOf` is considered to be transitive as in RDFS. This allows us to draw simple inferences, as in Fig. 4.4. Also, every class is a subclass of `owl:Thing`, and `owl:Nothing` is a subclass of every other class.

Two classes can be declared to be disjoint using `owl:disjointWith`. This means that they do not share any individual, i.e. their intersection is empty. This allows corresponding inferences, as exemplified in Fig. 4.5.

Two classes can be declared to be equivalent using `owl:equivalentClass`. Equivalently, this can be achieved by stating that two classes are subclasses of each other. Further examples for corresponding inferences are given in Figs. 4.6 and 4.7.

4.1.4 Relations Between Individuals

We have already seen how to declare class memberships of individuals and role relationships between them. OWL also allows us to declare that two individuals are in fact the same.

The class inclusions

```
<owl:Class rdf:about="Professor">
  <rdfs:subClassOf rdf:resource="FacultyMember" />
</owl:Class>
<owl:Class rdf:about="Book">
  <rdfs:subClassOf rdf:resource="Publication" />
</owl:Class>
```

together with the statement that `FacultyMember` and `Publication` are disjoint,

```
<owl:Class rdf:about="FacultyMember">
  <owl:disjointWith rdf:resource="Publication" />
</owl:Class>
```

allow us to infer that `Professor` and `Book` are also disjoint.

FIGURE 4.5: Example of an inference with `owl:disjointWith`

The class inclusion

```
<owl:Class rdf:about="Man">
  <rdfs:subClassOf rdf:resource="Person" />
</owl:Class>
```

together with the class equivalence

```
<owl:Class rdf:about="Person">
  <owl:equivalentClass rdf:resource="Human" />
</owl:Class>
```

allows us to infer that `Man` is a subclass of `Human`.

FIGURE 4.6: Example of an inference with `owl:equivalentClass`

From

```
<Book rdf:about="http://semantic-web-book.org/uri">
  <author rdf:resource="markusKroetzsch" />
  <author rdf:resource="sebastianRudolph" />
</Book>
<owl:Class rdf:about="Book">
 <rdfs:subClassOf rdf:resource="Publication" />
</owl:Class>
```

we can infer that `http://semantic-web-book.org/uri` is a `Publication`.

FIGURE 4.7: Example of an inference with individuals

From

```
<Professor rdf:about="rudiStuder" />
<rdf:Description rdf:about="rudiStuder">
  <owl:sameAs rdf:resource="professorStuder" />
</rdf:Description>
```

we can infer that `professorStuder` is in the class `Professor`.

FIGURE 4.8: Example inference with `owl:sameAs`

```
<rdf:Description rdf:about="rudiStuder">
  <owl:sameAs rdf:resource="professorStuder" />
</rdf:Description>
```

An example of an inference with `owl:sameAs` is given in Fig. 4.8.

Let us remark that the possible identification of differently named individuals via `owl:sameAs` distinguishes OWL from many other knowledge representation languages, which usually impose the so-called *Unique Name Assumption* (UNA), i.e. in these languages it is assumed that differently named individuals are indeed different. In OWL,[6] however, differently named individuals can denote the same thing. `owl:sameAs` allows us to declare this explicitly, but it is also possible that such an identification is implicit, i.e. can be inferred from the knowledge base without being explicitly stated.

[6]RDF(S) does also *not* impose the Unique Name Assumption.

With `owl:differentFrom`, it is possible to declare that individuals are different. In order to declare that several individuals are mutually different, OWL provides a shortcut, as follows. Recall from Section 2.5.1.3 that we can use `rdf:parseType="Collection"` for representing closed lists.

```
<owl:AllDifferent>
  <owl:distinctMembers rdf:parseType="Collection">
    <Person rdf:about="rudiStuder" />
    <Person rdf:about="dennyVrandecic" />
    <Person rdf:about="peterHaase" />
  </owl:distinctMembers>
</owl:AllDifferent>
```

4.1.5 Closed Classes

A declaration like

```
<SecretariesOfStuder rdf:about="giselaSchillinger" />
<SecretariesOfStuder rdf:about="anneEberhardt" />
```

states that `giselaSchillinger` and `anneEberhardt` are secretaries of Studer. However, it does not say anything about the question whether he has more secretaries, or only those two. In order to state that a class contains only the explicitly stated individuals, OWL provides closed classes as in Fig. 4.9.

It is also possible that a closed class contains data values, i.e. elements of a datatype, which are collected into a list using `rdf:List` (cf. Section 2.5.1.3). Figure 4.10 gives an example using email addresses as strings.

The use of these constructors is restricted in OWL Lite, and we will come back to that in Section 4.2.

4.1.6 Boolean Class Constructors

The language elements described so far allow us to model simple ontologies. But their expressivity hardly surpasses that of RDFS. In order to express more complex knowledge, OWL provides logical class constructors. In particular, OWL provides language elements for logical *and*, *or*, and *not*, i.e. conjunction, disjunction, and negation. They are expressed via `owl:intersectionOf`, `owl:unionOf`, and `owl:complementOf`, respectively. These constructors allow us to combine *atomic classes* – i.e. class names – to *complex classes*. Let us remark that the use of these constructors is restricted in OWL Lite, and we will come back to that in Section 4.2.

The declaration

```
<owl:Class rdf:about="SecretariesOfStuder">
  <owl:oneOf rdf:parseType="Collection">
    <Person rdf:about="giselaSchillinger" />
    <Person rdf:about="anneEberhardt" />
  </owl:oneOf>
</owl:Class>
```

states that `giselaSchillinger` and `anneEberhardt` are the only individuals in the class `SecretariesOfStuder`. If we also add

```
<Person rdf:about="anupriyaAnkolekar" />
<owl:AllDifferent>
  <owl:distinctMembers rdf:parseType="Collection">
    <Person rdf:about="anneEberhardt" />
    <Person rdf:about="giselaSchillinger" />
    <Person rdf:about="anupriyaAnkolekar" />
  </owl:distinctMembers>
</owl:AllDifferent>
```

then it can also be inferred, e.g., that `anupriyaAnkolekar` is not in the class `SecretariesOfStuder`. Without the latter statement, such an inference is not possible, since the knowledge that the individuals are different is needed to exclude identification of `anupriyaAnkolekar` with `giselaSchillinger` or `anneEberhardt`.

FIGURE 4.9: Example inference with closed classes

```
<owl:Class rdf:about="emailsAuthor">
  <owl:DataRange>
    <owl:oneOf>
      <rdf:List>
        <rdf:first rdf:datatype="&xsd;string"
          >pascal@pascal-hitzler.de</rdf:first>
        <rdf:rest>
          <rdf:List>
            <rdf:first rdf:datatype="&xsd;string"
              >markus@korrekt.org</rdf:first>
            <rdf:rest>
              <rdf:List>
                <rdf:first rdf:datatype="&xsd;string"
                  >mail@sebastian-rudolph.de</rdf:first>
                <rdf:rest rdf:resource="&rdf;nil" />
              </rdf:List>
            </rdf:rest>
          </rdf:List>
        </rdf:rest>
      </rdf:List>
    </owl:oneOf>
  </owl:DataRange>
</owl:Class>
```

FIGURE 4.10: Classes via `oneOf` and datatypes

The conjunction `owl:intersectionOf` of two classes consists of exactly those objects which belong to both classes. The following example states that `SecretariesOfStuder` consists of exactly those objects which are both `Secretaries` and `MembersOfStudersGroup`.

```
<owl:Class rdf:about="SecretariesOfStuder">
  <owl:intersectionOf rdf:parseType="Collection">
    <owl:Class rdf:about="Secretaries" />
    <owl:Class rdf:about="MembersOfStudersGroup" />
  </owl:intersectionOf>
</owl:Class>
```

An example of an inference which can be drawn from this is that all instances of the class `SecretariesOfStuder` are also in the class `Secretaries`. The example just given is a short form of the following statement.

```
<owl:Class rdf:about="SecretariesOfStuder">
  <owl:intersectionOf rdf:parseType="Collection">
    <owl:Class rdf:about="Secretaries" />
    <owl:Class rdf:about="MembersOfStudersGroup" />
  </owl:intersectionOf>
</owl:Class>
```

Certainly, it is also possible to use Boolean class constructors together with `rdfs:subClassOf`. The following example with `owl:unionOf` describes that professors are actively teaching or retired. Note that it also allows the possibility that a retired professor is still actively teaching. Also, it allows for the possibility that there are teachers who are not professors.

```
<owl:Class rdf:about="Professor">
  <rdfs:subClassOf>
    <owl:Class>
      <owl:unionOf rdf:parseType="Collection">
        <owl:Class rdf:about="ActivelyTeaching" />
        <owl:Class rdf:about="Retired" />
      </owl:unionOf>
    </owl:Class>
  </rdfs:subClassOf>
</owl:Class>
```

The use of `owl:unionOf` together with `rdfs:subClassOf` in the example just given thus states only that every `Professor` is in at least one of the classes `ActivelyTeaching` and `Retired`.

The complement of a class can be declared via `owl:complementOf`, which corresponds to logical negation: The complement of a class consists of exactly those objects which are *not* members of the class itself. The following example states that no faculty member can be a publication. It is thus equivalent to the statement made using `owl:disjointWith` in Fig. 4.5, that the classes `FacultyMember` and `Publication` are disjoint.

```
<owl:Class rdf:about="FacultyMember">
  <rdfs:subClassOf>
    <owl:Class>
      <owl:complementOf rdf:resource="Publication" />
    </owl:Class>
  </rdfs:subClassOf>
</owl:Class>
```

Correct use of `owl:complementOf` can be tricky. Consider, for instance, the following example.

```
<owl:Class rdf:about="Male">
  <owl:complementOf rdf:resource="Female" />
</owl:Class>
<Penguin rdf:about="tweety" />
```

From these statements it cannot be concluded that `tweety` is an instance of `Female`. However, it can also not be concluded that `tweety` is *not* `Female`, and hence it cannot be concluded that `tweety` is `Male`.

Now add the following statements to the ones just given, which state the obvious facts that `Furniture` is not `Female`, and that `myDesk` is a `Furniture`.

```
<owl:Class rdf:about="Furniture">
  <rdfs:subClassOf>
    <owl:Class>
      <owl:complementOf rdf:resource="Female" />
    </owl:Class>
  </rdfs:subClassOf>
</owl:Class>
<Furniture rdf:about="myDesk" />
```

From the combined statements, however, we can now conclude that `myDesk` is `Male` – because it is known *not* to be `Female`. If you contemplate this, then you will come to the conclusion that it is usually incorrect to model `Male` as the complement of `Female`, simply because there are things which are neither

From the declarations

```
<owl:Class rdf:about="Professor">
  <rdfs:subClassOf>
    <owl:Class>
      <owl:unionOf rdf:parseType="Collection">
        <owl:intersectionOf rdf:parseType="Collection">
          <owl:Class rdf:about="Person" />
          <owl:Class rdf:about="FacultyMember" />
        </owl:intersectionOf>
        <owl:intersectionOf rdf:parseType="Collection">
          <owl:Class rdf:about="Person" />
          <owl:complementOf rdf:resource="PhDStudent">
        </owl:intersectionOf>
      </owl:Class>
    </owl:intersectionOf>
  </rdfs:subClassOf>
</owl:Class>
```

we can infer that every `Professor` is a `Person`.

FIGURE 4.11: Example inference using nested Boolean class constructors

`Male` nor `Female` – such as `myDesk`. It would be more appropriate to simply declare `Male` and `Female` to be disjoint, or alternatively, to declare `Male` to be equivalent to the intersection of `Human` and the complement of `Female`.

Boolean class constructors can be nested arbitrarily deeply; see Fig. 4.11 for an example.

4.1.7 Role Restrictions

By role restrictions we understand another type of logic-based constructors for complex classes. As the name suggests, role restrictions are constructors involving roles.

The first role restriction is derived from the universal quantifier in predicate logic and defines a class as the set of all objects for which the given role only attains values from the given class. This is best explained by an example, like the following which states that examiners must always be professors. More precisely, it states that *all* examiners of an exam must be professors.[7]

[7]This actually includes those exams which have *no* examiner.

```
<owl:Class rdf:about="Exam">
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="hasExaminer" />
      <owl:allValuesFrom rdf:resource="Professor" />
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>
```

In order to declare that any exam must have at least one examiner, OWL provides role restrictions via `owl:someValuesFrom`, which is closely related to the existential quantifier in predicate logic.

```
<owl:Class rdf:about="Exam">
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="hasExaminer" />
      <owl:someValuesFrom rdf:resource="Person" />
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>
```

Using `owl:allValuesFrom`, we can say something about *all* of the examiners. Using `owl:someValuesFrom`, we can say something about *at least one* of the examiners. In a similar way we can also make statements about the number of examiners. The following example declares an upper bound on the number; more precisely it states that an exam must have a maximum of two examiners.

```
<owl:Class rdf:about="Exam">
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="hasExaminer" />
      <owl:maxCardinality rdf:datatype="&xsd;nonNegativeInteger">
        2
      </owl:maxCardinality>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>
```

It is also possible to declare a lower bound, e.g., that an exam must cover at least three subject areas.

```
<owl:Class rdf:about="Exam">
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="hasTopic" />
      <owl:minCardinality rdf:datatype="&xsd;nonNegativeInteger">
        3
      </owl:minCardinality>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>
```

Some combinations of restrictions are needed frequently, so that OWL provides shortcuts. If we want to declare that an exam covers *exactly* three subject areas, then this can be done via `owl:cardinality`.

```
<owl:Class rdf:about="Exam">
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="hasTopic" />
      <owl:cardinality rdf:datatype="&xsd;nonNegativeInteger">
      3
      </owl:cardinality>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>
```

Obviously, this can also be expressed by combining `owl:minCardinality` with `owl:maxCardinality` as in Fig. 4.12.

The restriction `owl:hasValue` is a special case of `owl:someValuesFrom`, for which a particular individual can be given as value for the role. The following example declares that `ExamStuder` consists of those things which have `rudiStuder` as examiner.

```
<owl:Class rdf:about="ExamStuder">
  <owl:equivalentClass>
    <owl:Restriction>
      <owl:onProperty rdf:resource="hasExaminer" />
      <owl:hasValue rdf:resource="rudiStuder" />
    </owl:Restriction>
  </owl:equivalentClass>
</owl:Class>
```

```
<owl:Class rdf:about="Exam">
  <rdfs:subClassOf>
    <owl:intersectionOf rdf:parseType="Collection">
      <owl:Restriction>
        <owl:onProperty rdf:resource="hasTopic" />
        <owl:minCardinality rdf:datatype="&xsd;nonNegativeInteger">
          3
        </owl:minCardinality>
      </owl:Restriction>
      <owl:Restriction>
        <owl:onProperty rdf:resource="hasTopic" />
        <owl:maxCardinality rdf:datatype="&xsd;nonNegativeInteger">
          3
        </owl:maxCardinality>
      </owl:Restriction>
    </owl:intersectionOf>
  </rdfs:subClassOf>
</owl:Class>
```

FIGURE 4.12: `owl:cardinality` expressed using `owl:minCardinality` and `owl:maxCardinality`

In this case an exam belongs to the class `ExamStuder` even if it has another examiner besides `rudiStuder`.

The example just given can also be expressed using `owl:someValuesFrom` and `owl:oneOf`.

```
<owl:Class rdf:about="ExamStuder">
  <owl:equivalentClass>
    <owl:Restriction>
      <owl:onProperty rdf:resource="hasExaminer" />
      <owl:someValuesFrom>
        <owl:oneOf rdf:parseType="Collection">
          <owl:Thing rdf:about="rudiStuder" />
        </owl:oneOf>
      </owl:someValuesFrom>
    </owl:Restriction>
  </owl:equivalentClass>
</owl:Class>
```

We give an extended example with role restriction in order to display the expressivity of these language constructs. We consider three colleagues and the role `likesToWorkWith`. Figure 4.13 shows the example ontology. If we now additionally define

```
<Person rdf:about="anton">
  <likesToWorkWith rdf:resource="doris" />
  <likesToWorkWith rdf:resource="dagmar" />
</Person>
<Person rdf:about="doris">
  <likesToWorkWith rdf:resource="dagmar" />
  <likesToWorkWith rdf:resource="bernd" />
</Person>
<Person rdf:about="gustav">
  <likesToWorkWith rdf:resource="bernd" />
  <likesToWorkWith rdf:resource="doris" />
  <likesToWorkWith rdf:resource="desiree" />
</Person>
<Person rdf:about="charles" />
<owl:Class rdf:about="FemaleColleagues">
  <owl:oneOf rdf:parseType="Collection">
    <Person rdf:about="dagmar" />
    <Person rdf:about="doris" />
    <Person rdf:about="desiree" />
  </owl:oneOf>
</owl:Class>
<owl:AllDifferent>
  <owl:distinctMembers rdf:parseType="Collection">
    <Person rdf:about="anton" />
    <Person rdf:about="bernd" />
    <Person rdf:about="charles" />
    <Person rdf:about="dagmar" />
    <Person rdf:about="desiree" />
    <Person rdf:about="doris" />
  </owl:distinctMembers>
</owl:AllDifferent>
```

FIGURE 4.13: Example ontology for role restrictions

```
<owl:Class rdf:about="Class1">
  <owl:equivalentClass>
    <owl:Restriction>
      <owl:onProperty rdf:resource="likesToWorkWith" />
      <owl:someValuesFrom rdf:resource="FemaleColleagues" />
    </owl:Restriction>
  </owl:equivalentClass>
</owl:Class>
```

then we can infer that `anton`, `doris` and `gustav` are in `Class1`.

Note that we cannot infer that `charles` is in `Class1`. At the same time, we also cannot infer that he is *not* in `Class1`. In fact we cannot infer any statement about `charles` belonging to `Class1` or not. The reason for this lies in the so-called *Open World Assumption* (OWA): It is implicitly assumed that a knowledge base may always be incomplete. In our example this means that `charles` *could* be in the relation `likesToWorkWith` to an instance of the class `FemaleColleagues`, but this relation is simply not (or not yet) known.

Let us dwell for a moment on this observation, because the OWA can easily lead to mistakes in the modeling of knowledge. In other paradigms, like databases, usually the *Closed World Assumption* (CWA) is assumed, which means that the knowledge base is considered to be complete concerning all relevant knowledge. Using the CWA, one could infer that `charles` is indeed *not* in `Class1`, because there is no known female colleague `charles likesToWorkWith`. The choice of OWA for OWL, however, is reasonable since the World Wide Web is always expanding rapidly, i.e. new knowledge is added all the time.

The OWA obviously also impacts on other situations. If, for example, an ontology contains the statements

```
<Professor rdf:about="rudiStuder" />
<Philosopher rdf:about="mikeStange" />
```

then we cannot infer anything about the membership (or non-membership) of `mikeStange` in the class `Professor`, because further knowledge, which may not yet be known to us, could state or allow us to infer such membership (or non-membership).

Now consider the following `Class2`.

```
<owl:Class rdf:about="Class2">
  <owl:equivalentClass>
    <owl:Restriction>
      <owl:onProperty rdf:resource="likesToWorkWith" />
      <owl:allValuesFrom rdf:resource="FemaleColleagues" />
    </owl:Restriction>
  </owl:equivalentClass>
</owl:Class>
```

We can infer that `doris` and `gustav` do *not* belong to `Class2`. Because of the OWA we cannot say anything about the membership of `anton` or `charles` in `Class2`.

If we define

```
<owl:Class rdf:about="Class3">
  <owl:equivalentClass>
    <owl:Restriction>
      <owl:onProperty rdf:resource="likesToWorkWith" />
      <owl:hasValue rdf:resource="doris" />
    </owl:Restriction>
  </owl:equivalentClass>
</owl:Class>
```

then `anton` and `gustav` belong to `Class3` because both like to work with `doris` (among others).

If we define

```
<owl:Class rdf:about="Class4">
  <owl:equivalentClass>
    <owl:Restriction>
      <owl:onProperty rdf:resource="likesToWorkWith" />
      <owl:minCardinality rdf:datatype="&xsd;nonNegativeInteger">
        3
      </owl:minCardinality>
    </owl:Restriction>
  </owl:equivalentClass>
</owl:Class>
```

then `gustav` belongs to `Class4` because he is the only one we know has at least three colleagues he `likesToWorkWith`.

If we define

```
<owl:Class rdf:about="Class5">
  <owl:equivalentClass>
    <owl:Restriction>
      <owl:onProperty rdf:resource="likesToWorkWith" />
      <owl:maxCardinality rdf:datatype="&xsd;nonNegativeInteger">
        0
      </owl:maxCardinality>
    </owl:Restriction>
  </owl:equivalentClass>
</owl:Class>
```

then due to the OWA we cannot infer that `charles` is in `Class5`.

`Class5` could equivalently be defined as follows – note that there are different ways to say the same thing.

```
<owl:Class rdf:about="Class5">
  <owl:equivalentClass>
    <owl:Restriction>
      <owl:onProperty rdf:resource="likesToWorkWith" />
      <owl:allValuesFrom rdf:resource="&owl;Nothing" />
    </owl:Restriction>
  </owl:equivalentClass>
</owl:Class>
```

The use of the constructs `owl:minCardinality`, `owl:maxCardinality` and `owl:cardinality` is restricted in OWL Lite; see Section 4.2 for details.

4.1.8 Role Relationships

Roles can be related in various ways. In particular, `rdfs:subPropertyOf` can also be used in OWL. The following example states: examiners of an event are also present at the event.

```
<owl:ObjectProperty rdf:about="hasExaminer">
  <rdfs:subPropertyOf rdf:resource="hasParticipant" />
</owl:ObjectProperty>
```

Similarly it is possible to state that two roles are in fact identical. This is done by using `owl:equivalentProperty` instead of `rdfs:subPropertyOf`.

Two roles can also be inverse to each other, i.e. can state the same relationship but with arguments exchanged. This is declared using `owl:inverseOf`.

```
<Exam rdf:about="semanticWebExam">
  <hasExaminer rdf:resource="rudiStuder" />
</Exam>
<owl:ObjectProperty rdf:about="hasExaminer">
  <rdfs:subPropertyOf rdf:resource="hasParticipant" />
</owl:ObjectProperty>
<owl:ObjectProperty rdf:about="hasParticipant">
  <owl:equivalentProperty rdf:resource="hasAttendee" />
</owl:ObjectProperty>
<owl:ObjectProperty rdf:about="hasAttendee">
  <owl:inverseOf rdf:resource="participatesIn" />
</owl:ObjectProperty>
```

FIGURE 4.14: Example using role relationships

```
<owl:ObjectProperty rdf:about="hasExaminer">
  <owl:inverseOf rdf:resource="examinerOf" />
</owl:ObjectProperty>
```

Figure 4.14 shows another example of the use of role relations. In this case `semanticWebExam` and `rudiStuder` are in the relation `hasParticipant` and also in the equivalent relation `hasAttendee`. Consequently, `rudiStuder` and `semanticWebExam` can be inferred to be in the relation `participatesIn`.

The use of role relationships is restricted in OWL DL and OWL Lite; see Section 4.2 for details.

4.1.9 Role Characteristics

OWL allows us to declare that roles have certain characteristics. This includes the specification of domain and range as well as characteristics like transitivity and symmetry.

We have already talked briefly about using `rdfs:range` and `rdfs:domain`. Let us now have a closer look at their semantics. Consider the statement

```
<owl:ObjectProperty rdf:about="isMemberOf">
  <rdfs:range rdf:resource="Organization" />
</owl:ObjectProperty>
```

which is equivalent to the following.

```
<owl:Class rdf:about="&owl;Thing">
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="isMemberOf" />
      <owl:allValuesFrom rdf:resource="Organization" />
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>
```

Now what happens if we also declare that `five isMemberOf PrimeNumbers`?

```
<number rdf:about="five">
  <isMemberOf rdf:resource="PrimeNumbers" />
</number>
```

From this, OWL allows us to infer that `PrimeNumbers` is an `Organization`! This is obviously an undesired result, which comes from the use of `isMemberOf` within two very different contexts: the first statement declares `isMemberOf` as a role which is used for making statements about memberships in organizations, while the second statement talks about a different domain, namely, numbers. The example is comparable to the one given at the end of Section 2.4.5.

Please note that the example just given does not yield a formal contradiction. In order to arrive at an inconsistency, one could additionally declare that `PrimeNumbers` are not in the class `Organization`.

Similar considerations also hold for `rdfs:domain`.

Let us now return to the characteristics which roles in OWL can be declared to have. They are transitivity, symmetry, functionality, and inverse functionality. We explain their meaning using the examples in Fig. 4.15.

Symmetry states: if A and B are in a symmetric role relationship, then B and A (in reverse order) are also in the same role relationship. In the example `peterHaase` is in a `hasColleague` relationship with `steffenLamparter`, i.e. `peterHaase` has `steffenLamparter` as colleague, and by symmetry we obtain that `steffenLamparter` has `peterHaase` as colleague.

Transitivity means: if A and B are in some transitive role relationship, and B and C are in the same role relationship, then A and C are also related via the same role. In the example, since we know that `steffenLamparter hasColleague peterHaase` and `peterHaase hasColleague philippCimiano`, we obtain by transitivity of the role `hasColleague` that `steffenLamparter` also `hasColleague philippCimiano`.

Functionality of a role means: if A and B are related via a functional role, and A and C are related by the same role, then B and C are identical in the

```
<owl:ObjectProperty rdf:about="hasColleague">
  <rdf:type rdf:resource="&owl;TransitiveProperty" />
  <rdf:type rdf:resource="&owl;SymmetricProperty" />
</owl:ObjectProperty>
<owl:ObjectProperty rdf:about="hasProjectLeader">
  <rdf:type rdf:resource="&owl;FunctionalProperty" />
</owl:ObjectProperty>
<owl:ObjectProperty rdf:about="isProjectLeaderFor">
  <rdf:type rdf:resource="&owl;InverseFunctionalProperty" />
</owl:ObjectProperty>
<Person rdf:about="peterHaase">
  <hasColleague rdf:resource="philippCimiano" />
  <hasColleague rdf:resource="steffenLamparter" />
  <isProjectLeaderFor rdf:resource="neOn" />
</Person>
<Project rdf:about="x-Media">
  <hasProjectLeader rdf:resource="philippCimiano" />
  <hasProjectLeader rdf:resource="cimianoPhilipp" />
</Project>
```

FIGURE 4.15: Role characteristics

sense of `owl:sameAs`. In the example we can conclude that `philippCimiano` and `cimianoPhilipp` are identical since `hasProjectLeader` is functional.

Inverse functionality of a role R is equivalent to the inverse of R being functional. In the example, we could have omitted the declaration of inverse functionality of `isProjectLeaderFor` and instead state the following.

```
<owl:ObjectProperty rdf:about="isProjectLeaderFor">
  <owl:inverseOf rdf:resource="hasProjectLeader" />
</owl:ObjectProperty>
```

Since it is declared that the role `hasProjectLeader` is functional, the inverse role `isProjectLeaderFor` would automatically be inverse functional.

Note that it does usually not make sense to declare transitive roles to be functional. It is, however, not explicitly forbidden in OWL.

The use of role characteristics is restricted in OWL DL and OWL Lite; see Section 4.2 for details.

4.1.10 Types of Inferences

To date, there is no standardized query language for OWL. While we discuss proposals for expressive query languages for OWL in Chapter 7, we briefly

discuss here what types of simple queries are commonly considered to be important when working with OWL. These are also supported by the usual software tools, as listed in Section 8.5. It is customary to distinguish between two types of simple queries, those involving individuals, and those using only schema knowledge.

Queries not involving individuals are concerned with classes and their relationships. We can distinguish between querying about the *equivalence* of two classes in the sense of `owl:equivalentClass` and querying about a subclass relationship in the sense of `rdfs:subClassOf`. We have given examples for this in Figs. 4.4, 4.6 and 4.11. Computing all subclass relationships between named classes is called *classifying* the ontology. Figure 4.5 gives an example asking about the *disjointness* of classes in the sense of `owl:disjointWith`.

We will see in Chapter 5 that querying for *global consistency*, i.e. for *satisfiability* of a knowledge base, is of central importance. Global consistency means the absence of contradictions.

Checking for *class consistency* is usually done in order to debug an ontology. A class is called *inconsistent*[8] if it is equivalent to `owl:Nothing`, which usually happens due to a modeling error. The following is a simple example of a class inconsistency caused by erroneous modeling.

```
<owl:Class rdf:about="Book">
  <rdfs:subClassOf rdf:resource="Publication" />
  <owl:disjointWith rdf:resource="Publication" />
</owl:Class>
```

Note that the knowledge base is not inconsistent: if there are no books, (and only in this case), the knowledge is consistent. That is because if we had a book, then it would also be a `Publication` by the `rdfs:subclassOf` statement. But this is impossible because `Publication` and `Book` are disjoint by the other statement. So, since there can be no book, `Book` is equivalent to `owl:Nothing`.

Queries involving individuals are of particular importance for practical applications. Such queries usually ask for all *known instances of a given class*, also known as *instance retrieval*. We have given examples for this in Figs. 4.7 and 4.8, and also in Section 4.1.7. Instance retrieval is closely related to *instance checking* which, given a class and an individual, decides whether the individual belongs to the class.

[8] In this case it is sometimes said that the class or ontology is *incoherent*.

4.2 OWL Species

We have remarked that there are three official sublanguages of OWL, and we have mentioned some of their conceptual differences in Fig. 4.2. We will now discuss the syntactic differences and their significance.

4.2.1 OWL Full

In OWL Full, all OWL language constructs can be used and mixed freely with all RDF(S) language constructs, as long as the resulting document is valid RDF(S).

Due to the unrestricted use of OWL and RDF(S) constructs in OWL Full, several problems arise, which suggest the design of more restrictive sublanguages. These problems are caused by the fact that OWL has been designed for the modeling of expressive knowledge and the resulting ability to access implicit knowledge by logical inference. For OWL Full, however, drawing inferences is in general undecidable, and indeed there is no software tool which supports the entire unwieldy semantics of OWL Full.

One of the reasons for the undecidability of OWL Full is that type separation is not enforced, i.e. in OWL Full individuals, classes, and roles can be mixed freely, and it is, e.g., possible to use an identifier as an individual in one statement, and as a role in the next statement. Consequently, the classes `owl:Thing` and `rdfs:Resource` are equivalent in OWL Full, as are `owl:Class` and `rdfs:Class`. Further, `owl:DatatypeProperty` is a subclass of `owl:ObjectProperty`, which in turn is equivalent to `rdf:Property`.

Complete type separation, however, is not desired in all cases, and OWL Full accommodates this. To give an example, it is sometimes necessary to make statements about a class, in which this class appears syntactically as an individual. In the following, we use roles to assign linguistic information to the class `Book`, thereby providing the basis for a multilingual system.

```
<owl:Class rdf:about="Book">
  <germanName rdf:datatype="&xsd;string">Buch</germanName>
  <frenchName rdf:datatype="&xsd;string">livre</frenchName>
</owl:Class>
```

The use of classes (or roles) as individuals is called *metamodeling*, and it allows us to talk about classes of classes.

Hardly any of the current inference engines support OWL Full, and in particular these do not provide its complete semantics. It would be unreasonable to expect this to change any time soon. This has only little impact on practical applications since for metamodeling and other constructs not supported

in OWL DL it is easy to provide workarounds outside the knowledge base: In the example just given, metamodeling is only used for providing background information, e.g., for multi-language user dialogues. In this case, the multilingual data could still be stored in and extracted from the OWL knowledge base, while inferencing over this knowledge is not needed.

Despite the lack of automated reasoning support, OWL Full is used for conceptual modeling in cases where automated reasoning support is not required.

4.2.2 OWL DL

The use of some language elements from OWL Full is restricted in OWL DL. This concerns mainly type separation and the use of RDF(S) language constructs, but also other restrictions. OWL DL was designed to be decidable, i.e. for any inference problem from Section 4.1.10 there exists an always terminating algorithm for deciding it. We discuss this in more detail in Chapter 5.

The following conditions must be satisfied such that an OWL Full document is a valid OWL DL document.

- Restricted use of RDF(S) language constructs: Only those RDF(S) language constructs may be used which are specifically allowed for OWL DL. These are essentially all those RDF(S) language constructs which we have used in our examples throughout this chapter. The use of `rdfs:Class` and `rdf:Property` is forbidden.

- Type separation and declaration: Type separation must be respected in OWL DL, i.e. it must be clearly distinguished between individuals, classes, abstract roles, concrete roles, datatypes, and the ontology characteristics specified in the header. In addition, classes and roles must be declared explicitly.

- Restricted use of concrete roles: The role characteristics `owl:inverseOf`, `owl:TransitiveProperty`, `owl:InverseFunctionalProperty`, as well as `owl:SymmetricProperty` must not be used for concrete roles.

- Restricted use of abstract roles: Cardinality restrictions expressed via `owl:cardinality`, `owl:minCardinality`, or via `owl:maxCardinality` must not be used with transitive roles, inverses of transitive roles, or superroles of transitive roles.[9]

[9] This somewhat strange restriction – which in exact form is a bit more complicated than this – is necessary to ensure decidability of OWL DL; see Section 5.1.4.3.

4.2.3 OWL Lite

OWL Lite was intended to be an easy to implement sublanguage containing the most important language constructs. However, it turned out that OWL Lite is essentially as difficult to deal with as OWL DL, and so it does not come as a surprise that it plays only a minor role in practice.

The following conditions must be satisfied such that an OWL Full document is a valid OWL Lite document.

- All restrictions imposed for OWL DL must be respected.
- Restricted use of class constructors: `owl:unionOf`, `owl:complementOf`, `owl:hasValue`, `owl:oneOf`, `owl:disjointWith`, `owl:DataRange` must not be used.
- Restricted use of cardinality restrictions: They can only be used with the numbers 0 and 1.
- Mandatory use of class names: In some situations, class names must be used:
  in the subject of `owl:equivalentClass` and `rdfs:subClassOf`,
  in the object of `rdfs:domain`.
- Mandatory class names or role restrictions: In some situations, class names or role restrictions must be used:
  in the object of `owl:equivalentClass`, `rdfs:subClassOf`, `rdf:type`, `owl:allValuesFrom`, `owl:someValuesFrom`, `rdfs:range`.
  Additionally, `owl:intersectionOf` must be used only for class names and role restrictions.

4.3 The Forthcoming OWL 2 Standard

The Web Ontology Language is currently undergoing a revision by means of a working group of the World Wide Web Consortium.[10] The forthcoming revision, originally called OWL 1.1 and now christened OWL 2, is essentially a small extension of the original version, which we will call OWL 1 in the following. At the time of this writing (June 2009), the working group has produced so-called Candidate Recommendations of the standard, which are very likely to be close to the final outcome. However, some things may still change, and so the following introduction to OWL 2 can only reflect the current state of the standardization process.

[10] `http://www.w3.org/2007/OWL/`

Let us first note that OWL 2 introduces a new syntax, called the *functional style syntax*, which will replace the OWL 1 abstract syntax. However, OWL 2 also comes with an RDF syntax, and we use this for the introduction. There will probably also be an XML syntax for OWL 2, which we do not discuss here.

4.3.1 OWL 2 DL

We introduce OWL 2 DL, which is backward compatible with OWL 1 DL, but extends it with some additional features. We thus describe only the new language features.

4.3.1.1 Type Separation, Punning and Declarations

OWL 1 DL imposes type separation, as discussed in Section 4.2.2, i.e. class names, role names, and individual names must be distinct. OWL 2 relaxes this requirement such that a class name, for example, may also occur as a role name. However, they are treated as distinct.[11] This is called *punning*.

Consider the following example.

```
<owl:Class rdf:about="Professor" />
<Professor rdf:about="rudiStuder" />
<owl:Class rdf:about="Institute" />
<owl:ObjectProperty rdf:about="Professor" />
<Institute rdf:about="aifb">
  <Professor rdf:resource="rudiStuder" />
</Institute>
```

In this example, `Professor` occurs both as class name and as abstract role name. This is not allowed in OWL 1 DL, but possible in OWL 1 Full. Intuitively, however, it seems reasonable to treat the role `Professor`[12] as distinct from the class `Professor`.

In OWL 2 DL, it is possible to use `Professor` both as role and as class name, and these are assumed to be distinct. Thus, the example code above is valid OWL 2 DL. However, it is not allowed that a name stands for both an abstract and a concrete role. Likewise, it is not allowed that a name stands for both a class and a datatype.

In OWL 2, classes, datatypes and roles must be declared as such. Individuals can also be declared, as follows, though this is optional.

[11] Note that if a class name is also used as a role name, they are identified by the same URI, i.e. they are *the same resource* in the sense of RDF. Nevertheless, in OWL 2 DL we consider them *semantically distinct*, i.e. we have two different views on the same resource.

[12] In this case, we cannot maintain our notational convention from page 118 to write roles lowercase and classes uppercase.

```
<rdf:Description rdf:about="rudiStuder">
  <rdf:type rdf:resource="&owl;NamedIndividual" />
</rdf:Description>
```

Alternatively, declaration of an individual can be done via the following shortcut.

```
<owl:NamedIndividual rdf:about="rudiStuder" />
```

4.3.1.2 Disjoint Classes

In OWL 1, `owl:disjointWith` can be used to declare two classes to be disjoint. If several classes should be declared to be mutually disjoint, however, a lot of `owl:disjointWith` statements are needed. Hence OWL 2 introduces `owl:AllDisjointClasses` as a shortcut which allows us to declare several classes to be mutually disjoint. Say, for example, that the classes `UndergraduateStudent`, `GraduateStudent` and `OtherStudent` should be declared as mutually disjoint. Then this can be done as follows.

```
<owl:AllDisjointClasses>
  <owl:members rdf:parseType="Collection">
    <owl:Class rdf:about="UndergraduateStudent" />
    <owl:Class rdf:about="GraduateStudent" />
    <owl:Class rdf:about="OtherStudent" />
  </owl:members>
</owl:AllDisjointClasses>
```

In OWL 1 the union of classes can be described using `owl:unionOf`. OWL 2 allows us to use `owl:disjointUnionOf` to declare a class the disjoint union of some other classes. Consider, for example, the situation that each `Student` is exactly in one of `UndergraduateStudent`, `GraduateStudent`, `OtherStudent`. Then this can be written in OWL 2 DL as follows.

```
<owl:Class rdf:about="Student">
  <owl:disjointUnionOf rdf:parseType="Collection">
    <owl:Class rdf:about="UndergraduateStudent" />
    <owl:Class rdf:about="GraduateStudent" />
    <owl:Class rdf:about="OtherStudent" />
  </owl:disjointUnionOf>
</owl:Class>
```

Obviously, the same effect can be obtained using `owl:unionOf` together with an `owl:AllDisjointClasses` statement.

4.3.1.3 Role Characteristics and Relationships

In OWL 1, it is possible to declare roles to be transitive, symmetric, functional or inverse functional. OWL 2 furthermore allows declarations of roles to be

- asymmetric, via `owl:AsymmetricProperty`, meaning that if A is related to B via such a role, then B is never related to A via this a role,
- reflexive, via `owl:ReflexiveProperty`, meaning that every individual A is related to itself via such a role, and
- irreflexive, via `owl:IrreflexiveProperty`, meaning that no individual is related to itself via such a role.

Recall that in OWL 1, as discussed in Section 4.2.2, transitivity, symmetry and inverse functionality must not be used for concrete roles. Likewise, asymmetry, reflexivity and irreflexivity must not be used for concrete roles. This leaves only functionality for both concrete and abstract roles.

It should be noted that reflexivity is a very strong statement since it refers to *every* possible individual, not just to individuals of a particular class. In many applications, it is more appropriate to use a more "local" notion of reflexivity, as provided by the Self construct that is introduced in Section 4.3.1.7 below.

Related to inverse functionality is `owl:hasKey`, which allows us to say that certain roles are keys for named instances of classes. More precisely, given a class `AClass`, a set of roles `r1`,...,`rn` is said to be a *key* for `AClass`, if no two named instances of `AClass` coincide on all values of all the (concrete or abstract) roles `r1`,...,`rn`. The syntax is the following.

```
<owl:Class rdf:about="AClass">
  <owl:hasKey rdf:parseType="Collection">
    <owl:ObjectProperty rdf:about="key1" />
    <owl:ObjectProperty rdf:about="key2" />
    <owl:DatatypeProperty rdf:about="key3" />
  </owl:hasKey>
</owl:Class>
```

Note the differences between using a key and using inverse functionality: Keys apply only to explicitly named instances of a class, while inverse functionality is also applicable to instances whose existence may only be implied, e.g., by means of `owl:someValuesFrom`. Another difference is that keys can involve several roles. Also note that concrete roles can be used for keys, while inverse functionality is forbidden for them.

Roles can also be declared to be disjoint, which means that two individuals A and B cannot be in relationship with respect to both roles. The following states, for example, that it is impossible that somebody teaches and attends a course at the same time.

```
<owl:ObjectProperty rdf:about="attendsCourse">
  <owl:propertyDisjointWith rdf:resource="teachesCourse" />
</owl:ObjectProperty>
```

As for classes, there is a shortcut notation to declare a number of roles to be mutually disjoint. The syntax is as follows.

```
<owl:AllDisjointProperties>
  <owl:members rdf:parseType="Collection">
    <owl:ObjectProperty rdf:about="attendsCourse" />
    <owl:ObjectProperty rdf:about="teachesCourse" />
    <owl:ObjectProperty rdf:about="skipsCourse" />
  </owl:members>
</owl:AllDisjointProperties>
```

Both `owl:propertyDisjointWith` and `owl:AllDisjointProperties` can also be used with concrete roles. Stating disjointness of an abstract and a concrete role is not useful (and not allowed) since abstract and concrete roles are always disjoint.[13]

OWL 2 furthermore sports four predefined roles:

- `owl:topObjectProperty`, called the top abstract role. It connects all possible pairs of individuals. Every abstract role is related to this role via `rdfs:subPropertyOf`.
- `owl:topDataProperty`, called the top concrete role. It connects all possible individuals with all datatype literals. Every concrete role is related to this role via `rdfs:subPropertyOf`.
- `owl:bottomObjectProperty`, called the bottom abstract role. It does not connect any pair of individuals. This role is related to any other abstract role via `rdfs:subPropertyOf`.
- `owl:bottomDataProperty`, called the bottom concrete role. It does not connect any individual with a literal. This role is related to any other concrete role via `rdfs:subPropertyOf`.

[13] Some further restrictions apply, which we discuss in Section 5.1.4.

4.3.1.4 Inverse Roles

OWL 1 allows the declaration of a role as the inverse of another role. In OWL 2, we can also refer to the inverse of a role without naming it. The following example states that if the event A has the person B as examiner (i.e. A is an exam), then B participates in A. Note that not every B participating in some event A implies that B is an examiner in A.

```
<owl:ObjectProperty rdf:about="hasExaminer">
  <rdfs:subPropertyOf>
    <owl:ObjectProperty>
      <owl:inverseOf rdf:resource="participatesIn" />
    </owl:ObjectProperty>
  </rdfs:subPropertyOf>
</owl:ObjectProperty>
```

This construction is not allowed for concrete roles.

4.3.1.5 Role Chains

OWL 2 allows us to express role chains, in the sense of concatenation of roles.[14] The classic example would be to express that whenever a person's parent has a brother, then that brother is the person's uncle. The syntax for this is as follows.

```
<owl:ObjectProperty rdf:about="hasUncle">
  <owl:propertyChainAxiom rdf:parseType="Collection">
    <owl:ObjectProperty rdf:resource="hasParent" />
    <owl:ObjectProperty rdf:about="hasBrother" />
  </owl:propertyChainAxiom>
</owl:ObjectProperty>
```

It is certainly possible to include more roles in the chain. However, concrete roles must not be used.

From a logical perspective, role chains are the most substantial improvement of OWL 2 compared to OWL 1. They can be understood as a broad generalization of transitivity, since, e.g.,

[14] Some restrictions apply, which we discuss in Section 5.1.4.

```
<owl:ObjectProperty rdf:about="hasAncestor">
  <owl:propertyChainAxiom rdf:parseType="Collection">
    <owl:ObjectProperty rdf:resource="hasAncestor" />
    <owl:ObjectProperty rdf:about="hasAncestor" />
  </owl:propertyChainAxiom>
</owl:ObjectProperty>
```

is equivalent to stating that the role `hasAncestor` is transitive.

4.3.1.6 Qualified Cardinality Restrictions

OWL 1 allows cardinality restrictions which are called *unqualified*, since they do not allow us to declare the target class of the role onto which the cardinality restriction is imposed. Have a look at the example on page 127, which states that `Exam` is a subclass of those things which have at most two objects attached via the `hasExaminer` role. With qualified cardinality restrictions we can say also something about the class these objects belong to. The following example states that each `Exam` has at most two elements from the class `Professor` related to it via the `hasExaminer` role. Note that this would allow further things to be related to an `Exam` via the `hasExaminer` role – as long as they are not in the class `Professor`.

```
<owl:Class rdf:about="Exam">
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="hasExaminer" />
      <owl:maxQualifiedCardinality
          rdf:datatype="&xsd;nonNegativeInteger">
        2
      </owl:maxQualifiedCardinality>
      <owl:onClass rdf:resource="Professor" />
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>
```

Similar constructions can be made with `owl:minQualifiedCardinality` and `owl:qualifiedCardinality`. They can also be used for concrete roles, using `owl:onDataRange` instead of `owl:onClass`.

4.3.1.7 The Self Construct

With the Self construct it can be stated that some individuals are related to themselves under a given role.[15] The typical example is that of persons committing suicide: they can be characterized by stating that these are all those people who killed themselves, as in the following example.

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

4.3.1.8 Negated Role Assignments

OWL 1 allows us to express that two individuals are related by some role. OWL 2 furthermore allows us to express that two individuals are *not* related by some role. The following example states that `anupriyaAnkolekar` and `sebastianRudolph` are *not* colleagues.

```
<owl:NegativePropertyAssertion>
  <owl:sourceIndividual rdf:about="anupriyaAnkolekar" />
  <owl:assertionProperty rdf:about="hasColleague" />
  <owl:targetIndividual rdf:about="sebastianRudolph" />
</owl:NegativePropertyAssertion>
```

The same is possible with concrete roles, where `owl:targetIndividual` is replaced by `owl:targetValue`.

4.3.1.9 Datatypes

Most XML Schema datatypes from Fig. 4.3 are supported in OWL 2 DL. Exceptions are some types relating to date and time: not supported are `xsd:time`, `xsd:date`, `xsd:gYear`, `xsd:gMonth`, `xsd:gDay`, `xsd:gMonthDay`, and `xsd:gYearMonth`. Furthermore, OWL 2 introduces the following new datatypes.[16]

[15] Some restrictions apply, which we discuss in Section 5.1.4.

[16] The datatypes `owl:rational` and `rdf:XMLLiteral` are currently *at risk*, i.e. they may be dropped in the final version.

- `owl:real`: the set of all real numbers
- `owl:rational`: the set of all rational numbers
- `rdf:PlainLiteral`: a string enriched with a language tag following BCP 47.[17]
- `rdf:XMLLiteral`: borrowed from RDF
- `xsd:dateTimeStamp`: based on `xsd:dateTime` but requires the specification of time zones.

It is not possible to explicitly write down literals for the datatype `owl:real`. However, such values can come about implicitly when dealing with integers and rationals, which is why they were included.

In addition to these basic datatypes, OWL 2 supports the use of *constraining facets*, which are actually borrowed from XML Schema, to further restrict datatype values. The example in Fig. 4.16 describes the class `Teenager` as the intersection of the class `Person` with all things which have an age between 12 and 19, where 19 is included in the range, but 12 is not. Other constraining facets for numeric datatypes are `xsd:maxInclusive` and `xsd:minExclusive`. Constraining facets for string datatypes are `xsd:minLength`, `xsd:maxLength`, `xsd:length`, and `xsd:pattern`. The latter refers to a selection based on matching a regular expression. Further information on constraining facets can be found in literature on XML Schema.

OWL 2 furthermore allows us to refer to the complement of a datatype, using `owl:datatypeComplementOf`. The following example specifies that the deficit on a bank account cannot be a positive integer.

```
<owl:DatatypeProperty rdf:about="deficit">
  <rdfs:domain rdf:resource="BankAccount" />
  <rdfs:range>
    <rdfs:Datatype>
      <owl:datatypeComplementOf rdf:resource="&xsd;positiveInteger" />
    </rdfs:Datatype>
  </rdfs:range>
</owl:DatatypeProperty>
```

The range then includes all negative integers and zero, but it also contains all strings and other values from other datatypes which are not positive integers.

Similarly, datatypes can be intersected using `owl:intersectionOf` with a list of datatypes in the object place. Likewise, `owl:unionOf` can be used for datatypes. These expressions can be nested.

[17] http://www.rfc-editor.org/rfc/bcp/bcp47.txt

```
<owl:Class rdf:about="Teenager">
  <owl:equivalentClass>
    <owl:Class>
      <owl:intersectionOf rdf:parseType="Collection">
        <rdf:Description rdf:about="Person" />
        <owl:Restriction>
          <owl:onProperty rdf:resource="hasAge" />
          <owl:someValuesFrom>
            <rdfs:Datatype>
              <owl:onDataType rdf:resource="&xsd;integer" />
              <owl:withRestrictions rdf:parseType="Collection">
                <xsd:minExclusive rdf:datatype="&xsd;integer">
                  12
                </xsd:minExclusive>
                <xsd:maxInclusive rdf:datatype="&xsd;integer">
                  19
                </xsd:maxInclusive>
              </owl:withRestrictions>
            </rdfs:Datatype>
          </owl:someValuesFrom>
        </owl:Restriction>
      </owl:intersectionOf>
    </owl:Class>
  </owl:equivalentClass>
</owl:Class>
```

FIGURE 4.16: Datatype constraining facets: restricting the allowed range of an integer value

4.3.2 OWL 2 Profiles

OWL 2 profiles are sublanguages of OWL 2. In this sense, OWL 2 DL, OWL 1 DL and OWL 1 Lite could be understood as profiles of OWL 2. The forthcoming standard will furthermore comprise three designated profiles which have been chosen for their favorable computational properties. We briefly present them in the following, but it should be noted that we omit details of the definitions, since our goal is to provide a rough intuition about them, rather than a comprehensive treatment.

4.3.2.1 OWL 2 EL

OWL 2 EL allows polynomial time algorithms for all standard inference types, such as satisfiability checking, classification, and instance checking. It was designed as a language that is particularly suitable for defining ontologies that include very large class and role hierarchies while using only a limited

amount of OWL features. A typical example application is the large medical ontology SNOMED CT (see Section 9.5) that defines more than one hundred thousand classes and roles.

The following language elements can be used in OWL 2 EL.

- `owl:Class`, `owl:Thing`, and `owl:Nothing`
- `rdfs:subClassOf`, `owl:equivalentClass`, `owl:disjointWith`, `owl:AllDisjointClasses`, and `owl:intersectionOf`
- `owl:someValuesFrom`, `owl:hasValue`, `owl:hasSelf`, and `owl:oneOf` (with exactly one individual or value)
- `owl:ObjectProperty`, `owl:DatatypeProperty`, `rdfs:domain`, `rdfs:range` (subject to some restrictions when using role chains), `owl:topObjectProperty`, `owl:topDataProperty`, `owl:bottomObjectProperty`, and `owl:bottomDataProperty`
- `rdfs:subPropertyOf`, `owl:equivalentProperty`, `owl:propertyChainAxiom`, `owl:TransitiveProperty`, `owl:ReflexiveProperty`, `owl:hasKey`, and for concrete roles also `owl:FunctionalProperty`
- all language elements needed for stating class and role relationship assignments for individuals using `owl:sameAs`, `owl:differentFrom` and `owl:AllDifferent`, `owl:NegativePropertyAssertion`, and for stating the basic assignment of an individual to a class, of two individuals to be related by an abstract role, and an individual and a datatype literal to be related by a concrete role
- class, role, and individual declarations
- many of the predefined OWL 2 datatypes

Note that in particular the use of `owl:allValuesFrom`, `owl:unionOf`, and `owl:complementOf` is disallowed. Cardinality restrictions also must not be used. Most role characteristics, including disjointness and inverses of roles, are also not allowed. For preserving its good computational properties, the datatypes supported by OWL 2 EL have been chosen to ensure that their intersection is either empty or infinite. This specifically excludes a number of numerical datatypes such as `xsd:int`, `xsd:byte`, and `xsd:double`. The usage of constraining facets is disallowed for the same reason.

4.3.2.2 OWL 2 QL

OWL 2 QL allows conjunctive query answering (see Section 7.2) to be implemented using conventional relational database systems. It also features polynomial time algorithms for all standard inference types. OWL 2 QL has

been designed for data-driven applications, and offers a convenient option for vendors of RDF stores to include some amount of OWL support without sacrificing the advantages of a database-like implementation.

- All predefined classes and roles can be used.
- OWL 2 QL imposes different restrictions on the subject and the object part of `rdfs:subClassOf` statements.
  - On the subject side, it is only allowed to use class names and `owl:someValuesFrom`, though in the latter case the target class must be `owl:Thing`.
  - On the object side, it is allowed to use class names. It is also allowed to use `owl:someValuesFrom` with target class as on the object side, `owl:complementOf` with target class as on the subject side, and `owl:intersectionOf` with the intersecting classes as on the object side.
- `owl:equivalentClass`, `owl:disjointWith`, `owl:AllDisjointClasses` can only be used with class expressions as allowed on the subject side of `rdfs:subClassOf`.
- `rdfs:subPropertyOf` and `owl:equivalentProperty` are allowed, as well as `rdfs:domain` (restricted to object side class expressions for concrete roles), `rdfs:range`, `owl:inverseOf`, and expressions involving `owl:propertyDisjointWith` and `owl:AllDisjointProperties`. Abstract roles can be declared to be symmetric, asymmetric, and reflexive.
- Assignments of individuals to be members of a class, and of individuals to be related to individuals or datatype literals via roles are allowed.
- `owl:differentFrom` and `owl:AllDifferent` can be used.
- Many of the predefined OWL 2 datatypes can be used.

Note that in addition to the subject and object side restrictions on the use of `rdfs:subClassOf`, it is not allowed to use `owl:allValuesFrom`, `owl:oneOf`, `owl:hasValue`, `owl:unionOf`, `owl:hasSelf`, `owl:hasKey`, and cardinality restrictions including functional and inverse functional roles. Transitivity and `owl:propertyChainAxiom`, `owl:sameAs`, and negative property assignments must not be used. The available datatypes and the use of facets are restricted in a similar way as for OWL 2 EL.

4.3.2.3 OWL 2 RL

OWL 2 RL allows standard inference types to be implemented with polynomial time algorithms using rule-based reasoning engines in a relatively straightforward way. It has been designed to allow the easy adoption of OWL

by vendors of rule-based inference tools, and it provides some amount of interoperability with knowledge representation languages based on rules (see Chapter 6).

OWL 2 RL is defined as a restriction of OWL 2 DL which mainly impacts on the use of `rdfs:subClassOf`. We present the main restrictions in the following.

- `owl:Thing` and `owl:Nothing` can be used. Top and bottom roles are disallowed.
- As for OWL 2 QL, the subject and the object sides of `rdfs:subClassOf` bear different restrictions.
  - On the subject side, we can use class names, `owl:Nothing` (but not `owl:Thing`), `owl:oneOf` and `owl:hasValue`. It is allowed to use `owl:intersectionOf`, `owl:unionOf`, `owl:someValuesFrom`; however, the involved class expressions must again be subject side class expressions; if `owl:someValuesFrom` is used with a concrete role, only datatype literals can be used.
  - On the object side, we can use class names and `owl:Nothing` (but not `owl:Thing`). It is allowed to use `owl:hasValue` and also `owl:allValuesFrom` for concrete roles, but it is restricted to object side class expressions for abstract roles. The only cardinality restriction allowed is `owl:maxCardinality`, and it is furthermore restricted to the cardinalities 0 and 1. For abstract roles, qualified cardinality restrictions can only be used with subject side class expressions as the target class.
- `owl:equivalentClass` can only be used with class expressions which are both subject and object side class expressions. `owl:disjointWith` and `owl:AllDisjointClasses` are restricted to subject side class expressions. `owl:disjointUnionOf` is disallowed.
- `owl:hasKey` can only be used with subject side class expressions.
- `rdfs:domain` and `rdfs:range` can only be used with object side class expressions. There are almost no further restrictions on using language elements for roles. `rdfs:subPropertyOf` and `owl:equivalentProperty`, inverse roles, and `owl:propertyChainAxiom` are supported. Roles can be declared transitive, symmetric, asymmetric, irreflexive, functional, and inverse functional.
- Many of the predefined OWL 2 datatypes can be used.
- Assignments of class membership of individuals can only be used with object side class expressions. `owl:NegativePropertyAssertion` is disallowed. There are no further restrictions on assignments and on the use of `owl:sameAs`, `owl:differentFrom` and `owl:AllDifferent`.

Note that top roles and reflexive roles are specifically excluded. This restriction is not motivated by computational properties (inferencing would still be polynomial if they were included), but by implementation-specific considerations: various rule-based systems are based on pre-computing and storing all logical consequences that can be expressed as assertional facts – a method known as *materialization* – and this approach works less well if some language constructs entail a very large number of such consequences. The available datatypes and the use of facets are restricted in a similar way as for OWL 2 EL.

4.3.3 OWL 2 Full

OWL 2 Full, syntactically, is the union of OWL 2 DL and RDFS. Semantically, i.e. in terms of inferences derivable from such ontologies, OWL 2 Full is compatible with OWL 2 DL in the sense that the OWL 2 Full semantics allows us to draw all inferences which can be drawn using the OWL 2 DL semantics (which is presented in the next chapter).

It can be expected that OWL 2 Full will play a similar role as OWL 1 Full for applications, i.e. it will probably be mainly used for conceptual modeling in cases where automated reasoning is not required.

4.4 Summary

In this chapter we have introduced the Web Ontology Language OWL using the normative RDFS syntax following the W3C recommendation from 2004. We put the focus on modeling with OWL DL, since this sublanguage is currently the most important one. We have also presented the other sublanguages OWL Full and OWL Lite and discussed their differences.

Besides introducing the syntax of the language constructs, we have also exemplified in all cases how logical inferences can be drawn from OWL ontologies. We will give this a thorough and formal treatment in Chapter 5.

We briefly mentioned important types of queries for OWL, and we will follow up on this in Chapter 7. We also presented the forthcoming revision of OWL, called OWL 2.

An introductory text like this cannot and may not present all the details of a rich language such as OWL, and so we have omitted several aspects the understanding of which is not central for an introduction to the language. The most important of our omissions are the following.

- Besides `owl:ObjectProperty` and `owl:DatatypeProperty`, there also exists `owl:AnnotationProperty`, instances of which can be used to annotate the whole ontology, single statements, or single entities. They do

not affect the logical meaning of OWL ontologies, i.e. they do not infringe on the semantics which is presented in the next chapter. A typical example would be `rdfs:comment`, and also the other header elements which can be used. In particular OWL 2 provides rich support for annotation properties, which can be freely defined, for example to give additional human-readable information, such as comments or provenance information, to statements and entities.

- There are some global syntactic constraints concerning the use of transitive roles and subrole relationships. We actually explain them in detail in the next chapter, when we discuss the semantics of OWL. In a nutshell, cyclic dependencies of subrole relationships are problematic if they involve OWL 2 role chains. Transitive roles must not occur in cardinality restrictions or the OWL 2 Self construct.

Advice on engineering ontologies can be found in Chapter 8, which also contains a discussion of available OWL Tools.

4.4.1 Overview of OWL 1 Language Constructs

Language constructs with restricted use in OWL Lite are marked by a ⋆. Note that this does not cover all the restrictions listed in Section 4.2.3.

4.4.1.1 Header

`rdfs:comment`	`rdfs:label`
`rdfs:seeAlso`	`rdfs:isDefinedBy`
`owl:versionInfo`	`owl:priorVersion`
`owl:backwardCompatibleWith`	`owl:incompatibleWith`
`owl:DeprecatedClass`	`owl:DeprecatedProperty`
`owl:imports`	

4.4.1.2 Relations Between Individuals

`owl:sameAs` `owl:differentFrom`
`owl:AllDifferent` together with `owl:distinctMembers`

4.4.1.3 Class Constructors and Relationships

`owl:Class`	`owl:Thing`	`owl:Nothing`
`rdfs:subClassOf`	`owl:disjointWith`⋆	`owl:equivalentClass`
`owl:intersectionOf`	`owl:unionOf`⋆	`owl:complementOf`⋆

Role restrictions using `owl:Restriction` and `owl:onProperty`:

`owl:allValuesFrom`	`owl:someValuesFrom`	`owl:hasValue`
`owl:cardinality`⋆	`owl:minCardinality`⋆	`owl:maxCardinality`⋆

`owl:oneOf`⋆, for datatypes together with `owl:DataRange`⋆

4.4.1.4 Role Constructors, Relationships and Characteristics

`owl:ObjectProperty`	`owl:DatatypeProperty`
`rdfs:subPropertyOf`	`owl:equivalentProperty`
`rdfs:domain`	`rdfs:range`
`owl:TransitiveProperty`	`owl:SymmetricProperty`
`owl:FunctionalProperty`	`owl:InverseFunctionalProperty`
`owl:inverseOf`	

4.4.1.5 Allowed Datatypes

The standard only requires the support of `xsd:string` and `xsd:integer`.

`xsd:string`	`xsd:boolean`	`xsd:decimal`
`xsd:float`	`xsd:double`	`xsd:dateTime`
`xsd:time`	`xsd:date`	`xsd:gYearMonth`
`xsd:gYear`	`xsd:gMonthDay`	`xsd:gDay`
`xsd:gMonth`	`xsd:hexBinary`	`xsd:base64Binary`
`xsd:anyURI`	`xsd:token`	`xsd:normalizedString`
`xsd:language`	`xsd:NMTOKEN`	`xsd:positiveInteger`
`xsd:NCName`	`xsd:Name`	`xsd:nonPositiveInteger`
`xsd:long`	`xsd:int`	`xsd:negativeInteger`
`xsd:short`	`xsd:byte`	`xsd:nonNegativeInteger`
`xsd:unsignedLong`	`xsd:unsignedInt`	`xsd:unsignedShort`
`xsd:unsignedByte`	`xsd:integer`	

4.4.2 Overview of Additional OWL 2 Language Constructs

4.4.2.1 Declaring Individuals

`owl:NamedIndividual`

4.4.2.2 Class Relationships

`owl:disjointUnionOf`	`owl:AllDisjointClasses`	`owl:members`

4.4.2.3 Role Characteristics and Relationships

`owl:AsymmetricProperty`	`owl:ReflexiveProperty`	
`owl:IrreflexiveProperty`		
`owl:topObjectProperty`	`owl:topDataProperty`	
`owl:bottomObjectProperty`	`owl:bottomDataProperty`	
`owl:propertyDisjointWith`	`owl:AllDisjointProperties`	
`owl:propertyChainAxiom`	`owl:hasKey`	`owl:inverseOf`

4.4.2.4 Role Restrictions

owl:maxQualifiedCardinality	owl:minQualifiedCardinality
owl:qualifiedCardinality	owl:onClass
owl:onDataRange	owl:hasSelf

4.4.2.5 Role Assignments

owl:NegativePropertyAssertion	owl:sourceIndividual
owl:assertionProperty	owl:targetIndividual
owl:targetValue	

4.4.2.6 Datatype Restrictions

owl:onDataType	owl:withRestrictions
owl:datatypeComplementOf	

4.4.2.7 Additional Datatypes

owl:real	owl:rational	rdf:PlainLiteral
rdf:XMLLiteral	xsd:dateTimeStamp	

4.5 Exercises

Exercise 4.1 Use OWL DL to model the following sentences:

- The class `Vegetable` is a subclass of `PizzaTopping`.
- The class `PizzaTopping` does not share any elements with the class `Pizza`.
- The individual `aubergine` is an element of the class `Vegetable`.
- The abstract role `hasTopping` is only used for relationships between elements of the classes `Pizza` and `PizzaTopping`.
- The class `VegPizza` consists of those elements which are in the class `NoMeatPizza` and in the class `NoFishPizza`.
- The role `hasTopping` is a subrole of `hasIngredient`.

Exercise 4.2 Decide which of the following statements would be reasonable in the context of the ontology from Exercise 4.1.

- The role `hasIngredient` is transitive.
- The role `hasTopping` is functional.
- The role `hasTopping` is inverse functional.

Exercise 4.3 Use OWL DL to model the following sentences.

- Every pizza has at least two toppings.
- Every pizza has `tomato` as topping.
- Every pizza in the class `PizzaMargarita` has exactly `tomato` and `cheese` as toppings.

Exercise 4.4 Consider the example in Fig. 4.7. Show that the given inference can be drawn using the formal semantics of RDFS.

Exercise 4.5 Install Protégé and KAON2 on your computer. Use Protégé to input the example from Fig. 4.11. Then use KAON2 to show that the given inference is correct.

4.6 Further Reading

We will give a thorough treatment of OWL semantics in Chapter 5, and also further literature references on semantics. So for the time being we will simply give pointers to the original documents with the W3C specification.

- [OWL] is the central website for OWL.
- [MvH04] gives an overview of OWL.
- [SD04] contains a complete description of all OWL language constructs.
- [SMW04] shows how to use OWL for knowledge representation.
- [HHPS04] describes the semantics of OWL, which we will cover in Chapter 5. It also presents the abstract syntax for OWL, which we do not treat in this book.

The current state of discussion on the forthcoming OWL 2 standard can be found on the Web pages of the W3C OWL working group.[18] The current key documents are the following.

[18] `http://www.w3.org/2007/OWL`

- [MPSP09] is the central document which introduces OWL 2 in functional style syntax.
- [PSM09] describes how the functional style syntax translates from and to the RDF syntax.
- [MCGH+09] specifies the different profiles of OWL 2.
- [SHK09] describes conformance conditions for OWL 2 and introduces the format of OWL 2 test cases which are provided along with the OWL 2 documents.
- [HKP+09] is a general introduction to OWL 2.

Exercises 4.1 to 4.3 were inspired by [RDH+04].