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 SecretariesOfStudor consists of exactly those objects which are both Secretaries and MembersOfStudorsGroup.

```
<owl:Class rdf:about="SecretariesOfStudor">
 <owl:intersectionOf rdf:parseType="Collection">
  <owl:Class rdf:about="Secretaries" />
  <owl:Class rdf:about="MembersOfStudorsGroup" />
 </owl:intersectionOf>
</owl:Class>
```

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

```
<owl:Class rdf:about="SecretariesOfStudor">
 <owl:intersectionOf rdf:parseType="Collection">
  <owl:Class rdf:about="Secretaries" />
  <owl:Class rdf:about="MembersOfStudorsGroup" />
 </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="FemaleColleaguen" />
    </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*.