RDF Schema

Now that we have got to grips with the XML syntax for conveying meta data as RDF, we can go on to look at how we might check the validity of this data. This is achieved using **RDF Schema**, which is defined by the W3C at http://www.w3.org/TR/2000/CR-rdf-schema-20000327.

> *Note that this document is a Candidate Recommendation, which means that it is to be treated as work in progress. This does not mean that we can't write programs that use this format, but we have to understand that the final version of the documentation could be completely different – the W3C puts out Candidate Recommendation documents to help the process of clarification and to spot problems.*

In that spirit, we'll have a look at RDF Schema in this chapter. We'll also look at extensions to RDF Schema, such as **DAML+OIL** (http://www.daml.org/).

Why Do We Need a Schema for RDF?

In the previous chapter we presented the idea of RDF as a *model* for meta data, alongside RDF/XML as a *syntax* which could be used to transport this model. This allowed us to represent name/value pairs of information and assign them to a resource or URI. We concluded that RDF/XML was an important standard for conveying this type of information.

For many applications RDF/XML is sufficient. The ability to transport triples is such a useful concept, who could want more? But there are many situations where we do want more. For example, there will be times when we need to know additional information about the resource being referred to, such as what it represents. For example, if we have a property that represents an author, then we may require that the value of the property is a reference to a person (and not a car or house). If we have a property that represents a birthday, then we want to check that the value is a date (and not a number or type of animal).

But we may want to check more than just the *validity* of the value – we may also want to *restrict* where certain properties can be applied. It is probably meaningless to allow a birthday property to be applied to a piece of music, for example.

The key to achieving these things is the RDF Schema specification. While the RDF Model and Syntax specification sets down how XML documents can be constructed to convey RDF, the RDF Schema document defines how we can be sure that the structure of some RDF/XML document or other conveys the correct *meaning*.

To understand what we mean by this, let's look at data integrity in a little more detail, to see why we are concerned with it.

Data Integrity

During the course of the recent British General Election the main subjects of interest were the constituencies (the locations in which the elections are taking place) and in the candidates (the people who have put themselves forward for election). Whoever wins the most votes in each constituency becomes the Member of Parliament – or MP – for that constituency. We'll use information from ePolitix.com, an apolitical provider of political news/information, to illustrate some of our points.

Remember that we are using RDF to convey name/value pairs, attached to some resource or other. Let's take a candidate in the Great Grimsby constituency – the Labour Party member Austin Mitchell, who actually won the seat. The UK's national news agency, PA News (www.pa.press.net), has a code for each constituency, and Great Grimsby's is 281. I've made up a URI to represent Grimsby (`http://www.pa.press.net/constituencies/281`). The meta data that we now have for this candidate is as follows:

```
primary key=http://www.epolitix.com/austin-mitchell
name=Austin Mitchell
candidateFor=http://www.pa.press.net/constituencies/281
memberOf=Labour Party
dob=19 September, 1934
```

As you already know from the first part of Chapter 4, RDF/XML is ideal for conveying this information across computer networks. We might use syntax like this (as in the previous chapter, assume that namespaces have all been defined correctly in the code fragments):

```
<rdf:Description rdf:about="http://www.epolitix.com/austin-mitchell">
    <epx:Name>Austin Mitchell</epx:Name>
    <epx:CandidateFor
         rdf:resource="http://www.pa.press.net/constituencies/281" />
    <epx:MemberOf>Labour Party</epx:MemberOf>
    <epx:DOB>19 September, 1934</epx:DOB>
</rdf:Description>
```

Our first data integrity problem is that we have no way of knowing whether the resource referred to in the `rdf:resource` property of the `<epx:CandidateFor>` element actually represents a constituency or not. What if, for example, we had the following:

```
<rdf:Description rdf:about="http://www.epolitix.com/austin-mitchell">
    <epx:Name>Austin Mitchell</epx:Name>
    <epx:CandidateFor rdf:resource="http://www.microsoft.com/" />
    <epx:MemberOf>Labour Party</epx:MemberOf>
    <epx:DOB>19 September, 1934</epx:DOB>
</rdf:Description>
```

As far as an RDF/XML parser is concerned this is perfectly acceptable. As long as the `rdf:resource` property is identified by a URI then everything is fine. But at the level of the meta data, the value that we have here is totally wrong, unless Austin is standing for election against Bill Gates, as the resource `http://www.microsoft.com/` is not a constituency.

Our second data integrity problem is that we might have used a property in an incorrect situation. Imagine that we had some meta data about the constituency that Austin was a candidate for:

```
<rdf:Description rdf:about="http://www.pa.press.net/constituencies/281">
    <epx:Name>Great Grimsby</epx:Name>
    <epx:MainIndustry>Fishing</epx:MainIndustry>
</rdf:Description>
```

This is fine, but what if we then saw this meta data:

```
<rdf:Description rdf:about="http://www.pa.press.net/constituencies/281">
    <epx:Name>Great Grimsby</epx:Name>
    <epx:MainIndustry>Fishing</epx:MainIndustry>
    <epx:DOB>19 September, 1934</epx:DOB>
</rdf:Description>
```

It is obvious that a parliamentary constituency doesn't have a date of birth property, but RDF/XML has no way of spotting this. As far as RDF/XML is concerned there is nothing wrong with this RDF syntax, so while RDF syntax can efficiently *convey* meta data, it is unable to *ensure its integrity.*

> **Our schema language must therefore be able to express the relationship between different items of meta data, regardless of the syntax used to express that meta data – that is, it must be able to validate *statements* and not just XML.**

Validating Statements

So what does it mean to validate statements? What sort of things should we be able to check for?

As you know from the previous chapter, statements are essentially triples, comprising a name/value pair – the predicate and the object that it refers to – and the resource that the pair refers to – the subject. To ensure the integrity of triples we only actually need to check two things:

- That the predicate is suitable for the subject, in other words that the property is acceptable for the resource.
- That the object is suitable for the predicate, in other words that the value is acceptable for the specified property.

I haven't included checking the subject since that is implied by RDF/XML anyway – the subject and predicate must both be identified by URIs, while the object can be a URI or a string literal. Here I am concentrating on checks that are beyond RDF/XML.

Checking the Predicate

Checking the predicate part of a statement requires us to check that the property being used is allowable when applied to the particular subject. In other words, is it legitimate to have a triple such as this?

```
{ epx:DOB, [http://www.pa.press.net/constituencies/281], "19 September, 1934" }
```

In this example we would like to prevent this, since we know that the subject of the statement is a constituency and that constituencies don't have birthdays.

Checking the Object

Checking the object part of a statement requires us to check that the value used refers to a resource or literal that is acceptable for the predicate. In other words, is it legitimate to have a triple such as this:

```
{
    epx:CandidateFor,
    [http://www.epolitix.com/austin-mitchell],
    [http://www.microsoft.com/]
}
```

Again, we would like to prevent this, since we know that the correct value should refer to a constituency.

Summary of Why We Need a Schema for RDF

The advantages to defining a schema that can specify how meta data should be structured, rather than how RDF/XML should be structured, mean that we have the ability to check the validity of meta data structures, even if they do not use RDF/XML to convey them. Provided that the meta data can be parsed into a set of triples – that is, into an RDF *model* – then we can validate the information against this schema.

The information we want to check against a schema is that:

- Properties applied to a resource make sense.
- Values used with properties make sense.

We can now start to look at how we might define a schema that meets these requirements.

Defining the Schema

Now that we have established why we need a way to distinctly specify RDF schemas, over and above the ability to define XML schemas, we can begin to look at how these schemas should be specified. The first question is what should we use to specify these schemas?

If you answered "RDF" you have either read the story's end, or have moved well beyond RDF guru status to a higher plain. The answer is indeed RDF. If we can specify RDF schemas in RDF itself, then we can move our schemas around and store them in a way that pays no regard to the syntax used to transmit those schemas. This means that we can use triples to define a schema, knowing that regardless of whether RDF/XML or some other syntax is used to convey those triples, the underlying meaning of the schema definition will still be understood.

Having said that, since the only widely accepted means of transporting triples is RDF/XML, we will discuss here the RDF/XML syntax for specifying an RDF schema, as defined in the W3C documentation. However, bear in mind that it is not the syntax that is important, but the underlying model.

Now we've established how we can specify the schema, let's look at how we can check properties and their values. Let's begin with the values.

Checking the Object of a Statement

The first area that we want to be able to check is that a resource being referred to by the predicate – the object, in other words – is right for the predicate. Let's return to our previous example of a candidate in the General Election:

```
<rdf:Description rdf:about="http://www.epolitix.com/austin-mitchell">
    <epx:Name>Austin Mitchell</epx:Name>
    <epx:CandidateFor
         rdf:resource="http://www.pa.press.net/constituencies/281" />
    <epx:MemberOf>Labour Party</epx:MemberOf>
    <epx:DOB>19 September, 1934</epx:DOB>
</rdf:Description>
```

Recall that we were concerned that there was no way of indicating that the value of the `epx:CandidateFor` property must be a constituency. The problem with the syntax as it stands is that we have no way of knowing what the following resource actually refers to:

```
http://www.pa.press.net/constituencies/281
```

The meta data we had for the constituency was as follows:

```
<rdf:Description rdf:about="http://www.pa.press.net/constituencies/281">
    <epx:Name>Great Grimsby</epx:Name>
    <epx:MainIndustry>Fishing</epx:MainIndustry>
</rdf:Description>
```

but there is nothing here to say that this is a constituency. The key to data integrity within RDF is to indicate to the schema processor what the **type** of the resource is.

Typing Resources

You may remember that in the last chapter we introduced the concept of a type property and typed nodes. We didn't go into much detail, but we indicated that it was possible to specify the type of a resource. The ability to specify the type of a resource is crucial to making the leap from RDF Model and Syntax to RDF Schemas.

The type of a resource is specified by making a statement in which the predicate is `rdf:type`, and the object is a URI that refers to the type of the resource. Let's say that ePolitix has defined a set of resource types for this purpose, and that the URI to use with an `rdf:type` property, if you want to indicate that a resource is of type `Constituency`, is this:

```
http://www.ePolitix.com/2001/03/rdf-schema#Constituency
```

Our document could now be specified as follows:

```
<rdf:Description rdf:about="http://www.pa.press.net/constituencies/281">
    <rdf:type
         resource="http://www.ePolitix.com/2001/03/rdf-schema#Constituency" />
    <epx:Name>Great Grimsby</epx:Name>
    <epx:MainIndustry>Fishing</epx:MainIndustry>
</rdf:Description>
```

and our resource has been typed. Don't forget that, provided that the namespace `epx` has been defined as `http://www.ePolitix.com/2001/03/rdf-schema#`, we can also specify the same information using the following syntax:

```
<epx:Constituency rdf:about="http://www.pa.press.net/constituencies/281">
    <epx:Name>Great Grimsby</epx:Name>
    <epx:MainIndustry>Fishing</epx:MainIndustry>
</epx:Constituency>
```

Now that we have indicated that the resource being referred to is of type `epx:Constituency`, we need to indicate that the property `epx:CandidateFor` can only have a value of a resource that has a type of `epx:Constituency`. This is achieved by creating a statement in which the resource is the name of the property – `epx:CandidateFor` – and this resource has a predicate that indicates the type for resources assigned to that property. First we need to indicate that `epx:CandidateFor` is indeed a property.

rdf:Property

A property is specified in an RDF schema by making a statement where the URI referred to identifies the resource which represents the property, and the `rdf:type` of the resource is `rdf:Property`. Our property to indicate which constituency a candidate is standing for election in would be defined like this:

```
<rdf:Description
 rdf:about="http://www.ePolitix.com/2001/03/rdf-schema#CandidateFor"
 xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
>
    <rdf:type resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#Property" />
</rdf:Description>
```

As you are of course aware, we can also use the abbreviated syntax:

```
<rdf:Property
 rdf:about="http://www.ePolitix.com/2001/03/rdf-schema#CandidateFor"
 xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
/>
```

Using either syntax we have created a property, but we haven't yet indicated in any way that this property can only have a value that is a resource that has a type of `epx:Constituency`. To do this we add a statement to our property URI to indicate what values can be used.

It is not immediately clear why the `rdf:Property` element should be in the `rdf` namespace, instead of the `rdfs` namespace (introduced below). After all, it does seem to be part of the schema. However, since the RDF model can have properties regardless of whether there is a schema to say anything more about those properties, the URI for the property type must exist in the RDF namespace. This means that we can do this, for example:

```
<rdf:RDF>
    <rdf:Description rdf:about="http://www.pa.press.net/constituencies/281">
        <rdf:type
           resource="http://www.ePolitix.com/2001/03/rdf-schema#Constituency" />
        <epx:Name>Great Grimsby</epx:Name>
    </rdf:Description>
    <rdf:Description
         rdf:about="http://www.ePolitix.com/2001/03/rdf-schema#Name">
        <rdf:type
             resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#Property" />
    </rdf:Description>
</rdf:RDF>
```

In this document we have said that the resource `http://www.pa.press.net/constituencies/281` is of type `Constituency`, and that it has a property of `epx:Name`. We then go on to say that `epx:Name` is indeed a property. Although this is not necessary at the RDF Model and Syntax level, `Property` must be in the `rdf` namespace so as to be consistent. Otherwise we could not make the statement we have just made without introducing RDF Schema.

rdfs:range

Just as we had a specific namespace to help identify RDF/XML within an XML document, so we need an RDF Schema namespace. The `rdfs` namespace prefix can be anything we want, but the URI that it evaluates to must be:

```
http://www.w3.org/2000/01/rdf-schema#
```

The statement we need – to indicate that a property can only take on certain values – has a predicate of `rdfs:range`:

```
<rdf:Property
 rdf:about="http://www.ePolitix.com/2001/03/rdf-schema#CandidateFor"
>
    <rdfs:range
       rdf:resource="http://www.ePolitix.com/2001/03/rdf-schema#Constituency" />
</rdf:Property>
```

Now we have defined a predicate that can be used with any set of statements, but when it is used it must have as an object a resource. But that resource cannot be just any resource. That resource must somewhere appear in a triple in which the predicate is `rdf:type`, and the object is the following URI:

```
http://www.ePolitix.com/2001/03/rdf-schema#Constituency
```

As we know from the previous chapter, this long URI will often be made easier to manage through the use of XML namespaces. For example, if a school was to hold pretend – or "mock" – elections to give students experience in running campaigns, they may wish to use our `CandidateFor` property, but use their school schema for the pupil's name and date of birth:

```
<rdf:Description
 rdf:about="http://www.GrangeHill.edu/mark-birbeck"
 xmlns:e="http://www.ePolitix.com/2001/03/rdf-schema#"
 xmlns:gh="http://www.GrangeHill.edu/2001/03/rdf-schema#"
>
    <gh:Name>Mark Birbeck</gh:Name>
    <e:CandidateFor rdf:resource="http://www.pa.press.net/constituencies/281" />
    <gh:Born>28 September, 1964</gh:Born>
    <gh:Comments>A little old for this school.</gh:Comments>
</rdf:Description>
```

As the RDFS spec currently stands there can only be one `rdfs:range` predicate for a property, but there is no requirement that a property *must have* a `range` predicate. In that case the predicate could take any value.

> *There are some further restrictions on the resources that can be referred to with `rdfs:range`, but these will be discussed later in this chapter. They are important, but do not affect your understanding of the concepts so far.*

rdfs:Literal

As we know from the previous chapter, the object of a statement can either be a resource or a literal. The examples we have just given for `rdfs:range` allow us to handle resources that have a type, but we will also want to specify that a value cannot be a resource, but must be a literal. This is achieved using `rdfs:Literal`, as follows:

```
<rdf:Property rdf:about="http://www.ePolitix.com/2001/03/rdf-schema#Name">
    <rdfs:range rdf:resource="http://www.w3.org/2000/01/rdf-schema#Literal" />
</rdf:Property>
```

This indicates that the value of the `epx:Name` property can only be a string literal, and not a resource.

Summary of Object Checking

We have looked at how a predicate can be shared among different meta data models, and also how the type of the resource that the predicate refers to can be restricted. We have seen that the key to this is the ability to specify the type of a resource through the `rdf:type` predicate.

Checking the Predicate of a Statement

We can now look at how RDF schemas allow us to indicate what predicates can appear with which resources. Again the key is to indicate the *type* of the resource, and then indicate whether a property is *legitimate* for that resource.

The `CandidateFor` property that we defined in the previous section required a constituency as its value, but there was no indication of what resources it could be used with. At first sight this worked in our favor, since it allowed the property used on a commercial web site like ePolitix to be the same as the property used in a school mock election web site. Recall that ePolitix used the property like this:

```
<rdf:Description rdf:about="http://www.epolitix.com/austin-mitchell">
    <epx:Name>Austin Mitchell</epx:Name>
    <epx:CandidateFor
         rdf:resource="http://www.pa.press.net/constituencies/281" />
    <epx:MemberOf>Labour Party</epx:MemberOf>
    <epx:DOB>19 September, 1934</epx:DOB>
</rdf:Description>
```

While the school used the same property like this:

```
<rdf:Description rdf:about="http://www.GrangeHill.edu/mark-birbeck">
    <gh:Name>Mark Birbeck</gh:Name>
    <epx:CandidateFor
         rdf:resource="http://www.pa.press.net/constituencies/281" />
    <gh:Born>28 September, 1964</gh:Born>
</rdf:Description>
```

When building up common properties – such as those used in the Dublin Core (which we discussed in Chapter 4) – it will often be the case that the property definition will allow the property to appear with *any* resource. For example, the DC `Creator` property may be defined like this:

```
<rdf:Property rdf:about="http://purl.org/dc/elements/1.0/Creator" />
```

Just as with our `CandidateFor` property, the `Creator` predicate can appear in a statement with any resource. However, while this may work for very broad properties like `Creator`, it doesn't work for our political properties. It would be nonsensical, for example, if we had the following:

```
<rdf:Description  rdf:about="http://www.conservatives.org.uk/">
    <rdf:type
         resource="http://www.ePolitix.com/2001/03/rdf-schema#PoliticalParty" />
    <epx:CandidateFor
         rdf:resource="http://www.pa.press.net/constituencies/281" />
</rdf:Description>
```

What does it mean for a political party to be a candidate for a constituency? Or a house or car? We need to be able to say that only resources that have been typed with something that indicates that they are a person can have the `CandidateFor` property. This is achieved through the use of the `rdfs:domain` predicate.

rdfs:domain

The `rdfs:domain` property is used to indicate what type of resources can be associated with the subject of a particular property. For example, imagine that we have a widely accepted type value that indicates that a resource represents a human being:

```
http://www.Schemas.org/2001/01/rdf-schema#Person
```

Let's now take our statements about Austin Mitchell being a candidate for Great Grimsby and indicate that the resource that we are making statements about – Austin – represents a person:

```
<rdf:Description rdf:about="http://www.epolitix.com/austin-mitchell">
    <rdf:type resource="http://www.Schemas.org/2001/01/rdf-schema#Person" />
    <epx:Name>Austin Mitchell</epx:Name>
    <epx:CandidateFor
```

```
            rdf:resource="http://www.pa.press.net/constituencies/281" />
        <epx:MemberOf>Labour Party</epx:MemberOf>
        <epx:DOB>19 September, 1934</epx:DOB>
    </rdf:Description>
```

We also use the same URI in the `rdfs:domain` predicate of our `CandidateFor` property to indicate that this property (`CandidateFor`) is only acceptable when used with a resource that has been typed as a `Person`:

```
    <rdf:Property
         rdf:about="http://www.ePolitix.com/2001/03/rdf-schema#CandidateFor">
        <rdfs:domain
              rdf:resource="http://www.Schemas.org/2001/01/rdf-schema#Person" />
        <rdfs:range
           rdf:resource="http://www.ePolitix.com/2001/03/rdf-schema#Constituency" />
    </rdf:Property>
```

So where does this leave the school elections? If the school wants to use the `CandidateFor` property in their meta data then all they have to do is make sure that their resources are also of type `Person`:

```
    <rdf:Description rdf:about="http://www.GrangeHill.edu/mark-birbeck">
        <rdf:type resource="http://www.Schemas.org/2001/01/rdf-schema#Person" />
        <gh:Name>Mark Birbeck</gh:Name>
        <epx:CandidateFor
             rdf:resource="http://www.pa.press.net/constituencies/281" />
        <gh:Born>28 September, 1964</gh:Born>
    </rdf:Description>
```

Now the `CandidateFor` property is acceptable.

As with the `rdfs:range` predicate there is no limit to the number of domains that a property can be applied to. Taking our previous example of a `Region`, we might say that a `Person` can have a property of a `Region`, and that a `Company` can also have a property of a `Region`:

```
    <rdf:Property
     rdf:about="http://www.ePolitix.com/2001/03/rdf-schema#Region"
     xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    >
        <rdfs:domain
              rdf:resource="http://www.Schemas.org/2001/01/rdf-schema#Person" />
        <rdfs:domain
              rdf:resource="http://www.Schemas.org/2001/01/rdf-schema#Company" />
    </rdf:Property>
```

But note an interesting consequence of the distributed nature of RDF – if we are running an event web site with a database of plays and exhibitions that people might want to go to, we can organize my data by region too. And to help with meta data searches we might use the ePolitix `Region` property within our meta data:

```
    <rdf:Property rdf:about="http://www.ePolitix.com/2001/03/rdf-schema#Region">
        <rdfs:domain
              rdf:resource="http://www.WhatsOnWhere.com/2001/01/rdf-schema#Event" />
    </rdf:Property>
```

Even though we didn't invent the `Region` property, and we don't own its schema definition, we have been able to use it in my schemas.

The most likely scenario though, is that organizations will define schemas for public use, and then other organizations will adopt the schemas of the most authoritative organization for a particular subject category. So the original ePolitix definition of `Region` would have been similar to the event web site definition, in that they would both have been in reference to another schema:

```
<rdf:Property rdf:about="http://www.Schemas.org/2001/01/rdf-schema#Region">
    <rdfs:domain
          rdf:resource="http://www.Schemas.org/2001/01/rdf-schema#Person" />
    <rdfs:domain
          rdf:resource="http://www.Schemas.org/2001/01/rdf-schema#Company" />
</rdf:Property>
```

(Similarly the event web site would also have taken the more authoritative schema.)

> *There are some further restrictions on the resources that can be referred with* `rdfs:domain`*, but these will be discussed later in this chapter. Again, they are important, but do not affect your understanding of the concepts so far.*

Summary of Predicate Checking

A property can be linked to one or more resource types through the `rdfs:domain` predicate. This does not prevent properties being shared amongst different schemas, but ensures that when properties are reused, they are placed into meaningful contexts.

Hierarchy of Types

So far we have seen that the typing system is key to specifying what properties are acceptable and what values a property can take. We simply make a statement about a resource that indicates what type that resource is. For example, we saw in the previous section that we could indicate that a resource was a `Person` so that we could then restrict the `CandidateFor` property to only be applicable to resources that were of type `Person`.

Let's refine this a little. It is not the case that *any* `Person` resource might be a candidate in an election. We might have a `Person` resource that represents an author of a book, or an astronaut. Ideally we would want to restrict the `CandidateFor` property to only apply to resources that are of type `Candidate`, so that if we saw this property attached to a resource of type `CircusClown` we could be sure that it was an error.

> *Of course, if the resource were of type* `CircusClown` and *of type* `Candidate`*, then we would allow the property* `CandidateFor`*.*

This doesn't also mean that we don't want to keep the type `Person`, but instead, resources of this type should perhaps have more basic information common to all people, such as a name and date of birth. The `CandidateFor` property should then only apply to resources of type `Candidate`. To create an election candidate we would then create a resource of type `Person`, and a resource of a new type, called `Candidate`. Let's see how this is done.

Resources with Multiple Types

The property definitions for predicates that can be used with resources of type `Person` would be defined by some accepted schema organization, like this:

```
<rdf:RDF>
    <rdf:Property rdf:about="http://www.Schemas.org/2001/01/rdf-schema#Name">
        <rdfs:domain
              rdf:resource="http://www.Schemas.org/2001/01/rdf-schema#Person" />
        <rdfs:range
              rdf:resource="http://www.w3.org/2000/01/rdf-schema#Literal" />
    </rdf:Property>
    <rdf:Property rdf:about="http://www.Schemas.org/2001/01/rdf-schema#DOB">
        <rdfs:domain
              rdf:resource="http://www.Schemas.org/2001/01/rdf-schema#Person" />
        <rdfs:range
              rdf:resource="http://www.w3.org/2000/01/rdf-schema#Literal" />
    </rdf:Property>
</rdf:RDF>
```

This means that we have two string literal properties – `Name` and `DOB` – which can be applied to a resource of type `Person`.

Our ePolitix `CandidateFor` property could now be limited to only be acceptable with resources of type `Candidate`, like this:

```
<rdf:Property
     rdf:about="http://www.ePolitix.com/2001/03/rdf-schema#CandidateFor">
    <rdfs:domain
          rdf:resource="http://www.ePolitix.com/2001/03/rdf-schema#Candidate" />
    <rdfs:range
       rdf:resource="http://www.ePolitix.com/2001/03/rdf-schema#Constituency" />
</rdf:Property>
```

Now all we need to do is give our election candidate two resource types, instead of one:

```
<rdf:Description
 rdf:about="http://www.epolitix.com/austin-mitchell"
 xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
 xmlns:s="http://www.Schemas.org/2001/01/rdf-schema#"
 xmlns:epx="http://www.ePolitix.com/2001/03/rdf-schema#"
>
    <rdf:type resource="http://www.Schemas.org/2001/01/rdf-schema#Person" />
    <s:Name>Austin Mitchell</s:Name>
    <s:DOB>19 September, 1934</s:DOB>
    <rdf:type resource="http://www.ePolitix.com/2001/03/rdf-schema#Candidate" />
    <epx:CandidateFor
         rdf:resource="http://www.pa.press.net/constituencies/281" />
    <epx:MemberOf>Labour Party</epx:MemberOf>
</rdf:Description>
```

A number of statements have been made here about the resource `http://www.epolitix.com/austin-mitchell` and all of them are acceptable:

- The `Name` and `DOB` statements are acceptable because the resource is of type `Person`.
- The `CandidateFor` statement is acceptable because the resource is also of type `Candidate`.
- The `MemberOf` statement is acceptable because this particular predicate currently has no restrictions on where it may appear.

However, this still does not feel complete. Just as we said earlier that we do not want the `CandidateFor` property to be applied to people who are not a `Candidate`, or to objects such as cars and houses, so too it seems that a `Candidate` will *always* be a `Person`. Ideally we'd like to say that any resource that has the type of `Candidate` automatically acquires the type of `Person`, and so any properties that are acceptable for a `Person` resource are fine for a `Candidate` resource.

In **object-oriented programming** (**OOP**) terminology this is much the same as defining a **class** that has **inherited** from another class. In fact, issues to do with OOP were taken into account when specifying the RDF Schema specification, and the documentation actually uses the term class, so let's look at that now.

Classes

If you are not familiar with OOP then let's have a quick look at some of the concepts. A class is a special kind of template from which to produce things. Remember as a child using a plastic cutter shaped like a little man to cut gingerbread men from a cake mix? Little did you know, all those years ago, that you were creating **instances** of a class. The plastic cutter defines the shape of all objects that are created with it, but it is not itself that object. The cutter is not a gingerbread man, but it is the template that you can use to create instances of gingerbread men. (Apologies for being gender-specific; I don't feel it is the place of this chapter to give voice to the hidden history of gingerbread women!)

We might also build other classes on top of our class. We might start with a template for creating faxes in Microsoft Word, and then modify it to become a template that creates faxes that demand payment on overdue accounts. The more specific fax – the one demanding money – is said to **inherit** features and properties from the more general fax. The more specific class is said to be a **subclass** of the more general one. **Inheritance** is a key concept in OOP.

Any object created using the **class definition** of a class that has inherited from another class is said to be an instance of both classes. A fax demanding money from one of your clients is said to be an object of type "fax payment demand", as well as an object of type "fax".

The template that the class definition specifies lists the properties that an instance of this class will have. Creating an object of type "gingerbread man" will create an object that has arms, legs, a body, and a head. Creating an object of type "fax" will create an object that has a fax number, a subject line, some text, and so on.

Creating an object using a class that inherits from another actually gives the object properties from both classes, or it may hide some properties from the parent class, or it may even set them to specific values. For example, an object of type "fax payment demand", might only have properties of company name and amount owed – the "text" property of "fax" might then be set automatically to contain a message that says "please pay the amount of £[amount owed] now".

Whilst an understanding of classes from the world of OOP may help you in your dealings with RDF Schema, you should be aware that there are two major differences between the notion of class within RDF and in programming.

The first and most obvious is that whilst a class in an object-oriented language would have functions – or methods – associated with it, RDF classes have no such thing. Classes written in Java or C++ might have a method such as `postToAccounts()` on an `Invoice` class, but with RDF classes there are only properties.

The second difference is that whilst a class is defined as a set of properties (and methods) in OOP languages, with RDF a property is defined as being applicable to a class. This makes sense if you think back to what we want to check for – whether the predicate part of a triple is valid for the subject. But it also makes sense in the context of the distributed nature of the Web, in that I can make *my* properties valid for *your* classes.

We'll now go through how each of these concepts is represented in RDF Schema. We'll now look at:

- Defining a class.
- Creating subclasses from a class.
- Creating instances of a class, or instantiating a class.

Defining a Class

Let's return to our General Election candidate example:

```
<rdf:Description
 rdf:about="http://www.epolitix.com/austin-mitchell"
 xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
 xmlns:s="http://www.Schemas.org/2001/01/rdf-schema#"
 xmlns:epx="http://www.ePolitix.com/2001/03/rdf-schema#"
>
    <rdf:type resource="http://www.Schemas.org/2001/01/rdf-schema#Person" />
    <s:Name>Austin Mitchell</s:Name>
    <s:DOB>19 September, 1934</s:DOB>
    <rdf:type resource="http://www.ePolitix.com/2001/03/rdf-schema#Candidate" />
    <epx:CandidateFor rdf:resource="http://www.pa.press.net/constituencies/281" />
    <epx:MemberOf>Labour Party</epx:MemberOf>
</rdf:Description>
```

Our candidate resource has been defined as being of type `Person` and `Candidate`. We would now like to say that this resource is only of type `Candidate`, and that in turn any resource of type `Candidate` is a `Person`. This is achieved through the OOP technique of inheritance, or subclassing.

In the world of OOP we can take a class and create a more specific class from it. In our example we need a class to represent resources of type `Person`, which is then used as the basis for a class that represents resources of type `Candidate`. Let's define the class `Person`.

rdfs:Class

RDF Schema allows us to define a class using a resource that has a type of `rdfs:Class`. This definition probably sounds like it won't stand up in court – unless the court is in Alice's Wonderland – since all we have said is that a class is defined as a class of type class! However, we have to assume here that an RDF parser that understands schemas will understand the key concept of the `rdf:type` property. In that case our schema processor will understand that any resource with a type of `rdfs:Class` is a class.

Using the URI for the RDF Schema namespace, the fictitious schema organization we introduced earlier could use the `rdf:type` property to specify a `Person` class:

```
<rdf:Description
 rdf:about="http://www.Schemas.org/2001/01/rdf-schema#Person"
 xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
>
    <rdf:type resource="http://www.w3.org/2000/01/rdf-schema#Class" />
</rdf:Description>
```

Alternatively, we can use a typed node to define the class, like this:

```
<rdfs:Class
 rdf:about="http://www.Schemas.org/2001/01/rdf-schema#Person"
 xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
/>
```

We have now "created" a class. In actual fact, all that we have done is created a triple that says that the resource `s:Person` has an `rdf:type` predicate with a value of `rdfs:Class` – but that's good enough! We now have a class from which we can derive others.

Creating Subclasses

We discussed earlier that one of the most powerful features of OOP was the ability to define a class that inherits properties from some other class. If we take our `Candidate` class, we can see that properties like date of birth and name are not features of the candidate *as a candidate*. A person could still have a birthday and a name even if they didn't stand in an election.

Equally, the fact that the person is a member of a political party is not a feature of being a candidate. Most people who are members of political parties never stand for office.

Inheritance allows us to manage information much more neatly, by designing a class structure that uses inheritance to logically separate properties out. In our example we began with a class of `Person`, but now we might inherit from that to create a class called `Politician`, before finally ending with a class called `Candidate` – based on the `Politician` class. We could then attach the properties "name" and "date of birth" to the `Person` class, and then have the "member of" property assigned only to the `Politician` class.

rdfs:subClassOf

We can use RDF/XML to define these levels of inheritance using the `rdfs:subClassOf` predicate, as follows:

```
<rdf:RDF
 xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
 xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
>
    <rdfs:Class rdf:about="http://www.Schemas.org/2001/01/rdf-schema#Person" />

    <rdfs:Class
          rdf:about="http://www.ePolitix.com/2001/03/rdf-schema#Politician">
        <rdfs:subClassOf
         rdf:resource="http://www.Schemas.org/2001/01/rdf-schema#Person"
        />
    </rdfs:Class>

    <rdfs:Class
```

```
            rdf:about="http://www.ePolitix.com/2001/03/rdf-schema#Candidate">
          <rdfs:subClassOf
           rdf:resource="http://www.ePolitix.com/2001/03/rdf-schema#Politician"
          />
      </rdfs:Class>
  </rdf:RDF>
```

Using this schema we are able to say that the ePolitix `Candidate` class is a subclass of the ePolitix `Politician` class, which in turn is a subclass of the Schemas.org `Person` class. This means that classes of the more specialized type (`Candidate`) are automatically classes of the more general type (`Person`). And the consequence of *that* is that any property that can take a `Politician` class as a value, can also take a `Candidate` class as a value, and any property that can take a `Person` class as its value will accept `Candidates` ***and*** `Politicians`.

Before we go on, I'm going to make an assumption for my examples that will allow me to abbreviate the text a little. As things stand the name of the file in which all of the above statements have been made is irrelevant, since the `about` attribute is used to identify resources. However, if I was to place all of this RDF/XML into a file called `http://www.ePolitix.com/2001/03/rdf-schema` then we could shorten all of the above URLs, as follows:

```
<rdf:RDF
 xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
 xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
>
    <rdfs:Class rdf:about="http://www.Schemas.org/2001/01/rdf-schema#Person" />

    <rdfs:Class rdf:ID="Politician">
        <rdfs:subClassOf
           rdf:resource="http://www.Schemas.org/2001/01/rdf-schema#Person" />
    </rdfs:Class>

    <rdfs:Class rdf:ID="Candidate">
        <rdfs:subClassOf rdf:resource="#Politician" />
    </rdfs:Class>
</rdf:RDF>
```

As you can see, by containing everything within the same file we can use a **fragment identifier** rather than a long URL. It's important to note, however, that `rdf:about` attributes must then become `rdf:ID` attributes.

Getting back to our example, for completeness let's show how the properties are defined. First the properties for a `Person`, which we have already seen:

```
<rdf:RDF>
    <rdf:Property rdf:about="http://www.Schemas.org/2001/01/rdf-schema#Name">
        <rdfs:domain
             rdf:resource="http://www.Schemas.org/2001/01/rdf-schema#Person" />
        <rdfs:range
              rdf:resource="http://www.w3.org/2000/01/rdf-schema#Literal" />
    </rdf:Property>
    <rdf:Property rdf:about="http://www.Schemas.org/2001/01/rdf-schema#DOB">
        <rdfs:domain
              rdf:resource="http://www.Schemas.org/2001/01/rdf-schema#Person" />
        <rdfs:range
              rdf:resource="http://www.w3.org/2000/01/rdf-schema#Literal" />
    </rdf:Property>
</rdf:RDF>
```

Next, resources with a type of `Politician` can have a `MemberOf` predicate:

```
<rdf:Property rdf:ID="MemberOf">
    <rdfs:domain rdf:resource="#Politician" />
    <rdfs:range rdf:resource="#PoliticalParty" />
</rdf:Property>
```

Note that I have made it so that the range of values that are acceptable for the `MemberOf` property are resources that have an `rdf:type` of `epx:PoliticalParty`. I haven't bothered to show that class, but add it here to indicate how the system keeps extending.

Finally, as we saw before, our ePolitix `CandidateFor` property is limited to resources of type `Candidate`, like this:

```
<rdf:Property rdf:ID="CandidateFor">
    <rdfs:domain rdf:resource="#Candidate" />
    <rdfs:range rdf:resource="#Constituency" />
</rdf:Property>
```

Now that we have defined our classes we need only use the `Candidate` class, like this:

```
<rdf:Description
 rdf:about="http://www.epolitix.com/austin-mitchell"
 xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
 xmlns:s="http://www.Schemas.org/2001/01/rdf-schema#"
 xmlns:epx="http://www.ePolitix.com/2001/03/rdf-schema#"
>
    <rdf:type resource="http://www.ePolitix.com/2001/03/rdf-schema#Candidate" />
    <s:Name>Austin Mitchell</s:Name>
    <s:DOB>19 September, 1934</s:DOB>
    <epx:CandidateFor
         rdf:resource="http://www.pa.press.net/constituencies/281" />
    <epx:MemberOf rdf:resource="http://www.ePolitix.com/parties#labour" />
</rdf:Description>
```

Although we are using predicates that are only acceptable with different classes, we no longer need to specify all of those classes, since the resource `http://www.epolitix.com/austin-mitchell` is now of type `Candidate`, `Politician`, and `Person`.

Remote Inheritance

The beauty of using `rdf:resource` to specify the class that we are subclassing from is that it could be a class not under our control. As we just saw, we were able to subclass from the `Person` class, even though it was not within our schema.

And we can take this further. You might want to create and maintain meta data about cabinet ministers – that is, those politicians who are members of the British government, and you might decide to create a class called `Minister` that is a subclass of the ePolitix `Politician` class:

```
<rdf:RDF
 xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
 xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
>
```

```
    <rdfs:Class rdf:about="http://mydata.com/2001/01/rdf-schema#Minister">
        <rdfs:subClassOf
         rdf:resource="http://www.ePolitix.com/2001/03/rdf-schema#Politician"
        />
    </rdfs:Class>
</rdf:RDF>
```

As you can see, you have created a class from one of my classes, which was in turn created from someone else's class. This works in much the same way as you might inherit from someone else's classes if you use a Java or C++ library. But rather than you having to have the library of classes on your computers for the subclassing to work, RDF Schema is using URIs to create a distributed network of classes. This is an incredibly powerful feature.

Multiple Inheritance

If you are familiar with OOP concepts then you will know that a class can be defined as being a subclass of more than one other class – known as **multiple inheritance**. This is not the same as the example we have been using, where `Candidate` is a subclass of `Politician`, which is in turn a subclass of `Person`. Instead, multiple inheritance involves creating a class from two or more other classes at the same time.

An example might be a defending candidate, who is both a candidate in the election but also the previous winner for that constituency. RDF Schema allows a class to be a subclass of as many classes as you like:

```
<rdf:RDF
 xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
 xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
>
    <rdfs:Class
      rdf:about="http://www.ePolitix.com/2001/03/rdf-schema#DefendingCandidate">
        <rdfs:subClassOf
         rdf:resource="http://www.ePolitix.com/2001/03/rdf-schema#Candidate"
        />
        <rdfs:subClassOf
         rdf:resource="http://www.ePolitix.com/2001/03/
                                                rdf-schema#ElectedRepresentative"
        />
    </rdfs:Class>
</rdf:RDF>
```

Versioning Schemas

Note the (invented) URLs we have been using for these illustrations, such as `http://mydata.com/2001/01/rdf-schema#Minister` and `http://www.ePolitix.com/2001/03/rdf-schema#Politician`.

They contain a year and month, in much the same way the URIs for the RDF namespaces do. This is not a requirement of your schema URIs, but it certainly helps if you change your schemas over time. You can release a new schema at a new URL safe in the knowledge that anyone who has derived classes from your previous schema will be unaffected. The RDF Schema specification takes this idea further, and suggests that if everyone adopts the approach of freezing the schema at a particular URL, then RDF software could cache the schema model without having to retrieve the RDF/XML every time it needs to use it.

Using techniques that we have seen so far – inheriting classes – we can link a newer schema to an older one. This would mean that rather than creating a brand new class with no relationship to a previous class, you could build a new class with new properties, but derive it from an old class with its old properties. This would mean that the base type of both classes would be the same.

For example, imagine that we want to create a new version of the ePolitix `Candidate` class. We could derive this class from our older class, like this:

```
<rdf:RDF
 xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
 xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
>
    <rdfs:Class
          rdf:about="http://www.ePolitix.com/2002/07/rdf-schema#Candidate">
        <rdfs:subClassOf
         rdf:resource="http://www.ePolitix.com/2001/03/rdf-schema#Candidate"
        />
    </rdfs:Class>
</rdf:RDF>
```

Then all ePolitix `Candidate` resources will have a common base class, regardless of whether they use the old or the new schema.

RDF Schema also provides a technique to do the same thing with properties – we can say that one property is based on another property. This allows us to either create a new property based on the existing one in a new schema, or it allows us to create a more specific property from a general one. An example might be a property that holds a value that indicates an organization that a person is a member of:

```
<rdf:Property rdf:ID="MemberOfOrganisation">
    <rdfs:domain rdf:resource="#Person" />
    <rdfs:range rdf:resource="#Organisation" />
</rdf:Property>
```

which is then used to create another property to show the more specific *party* membership of politicians:

```
<rdf:Property rdf:ID="MemberOfParty">
    <rdfs:subPropertyOf rdf:resource="#MemberOfOrganisation" />
</rdf:Property>
```

The `MemberOfParty` property has all of the attributes of the `MemberOfOrganisation` property, but of course it could also have its own features.

The `rdfs:subPropertyOf` property looks like this in the RDF schema:

```
<rdf:Property rdf:ID="subPropertyOf">
  <rdfs:range
        rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#Property"/>
  <rdfs:domain
        rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#Property"/>
</rdf:Property>
```

Instantiating a Class

We've already seen how to create instances of a class when we discussed the `rdf:type` property. For completeness let's look at it again, since I want to add a few other comments.

As you've seen, we can say that a resource is an instance of a class by giving it an `rdf:type` property:

```
<rdf:Description rdf:about="http://www.epolitix.com/austin-mitchell">
    <rdf:type resource="http://www.ePolitix.com/2001/03/rdf-schema#Candidate" />
    <s:Name>Austin Mitchell</s:Name>
    <epx:MemberOf>Labour Party</epx:MemberOf>
    <s:DOB>19 September, 1934</s:DOB>
</rdf:Description>
```

However, unlike most OOP languages, we can say that an object is an instance of two classes. This is not the same as creating a class with multiple inheritance – as you can do with some OOP languages – since you would still then create an instance of the new class. With RDF Schema we only have to add more type properties to increase the classes that the object is an instance of:

```
<rdf:Description rdf:about="http://www.epolitix.com/austin-mitchell">
    <rdf:type resource="http://www.ePolitix.com/2001/03/rdf-schema#Candidate" />
    <s:Name>Austin Mitchell</s:Name>
    <epx:MemberOf>Labour Party</epx:MemberOf>
    <s:DOB>19 September, 1934</s:DOB>
    <rdf:type resource="http://www.Schemas.org/2001/01/rdf-schema#Husband" />
    <s:Wife rdf:resource=" http://www.epolitix.com/linda-mcdougall" />
</rdf:Description>
```

Rather than creating a new class that is of type `CandidateWhoIsAlsoAHusband`, we have instead created an object that is an instance of both the `Candidate` and `Husband` classes. When processing software encounters this RDF/XML it will know that the class instance must match the requirements of both classes.

Summary of Classes

We have looked at how the schema definitions used in RDF Schema specify classes. We have seen how we can say that a resource is an instance of one or more classes, and we've begun to look at how we might specify a class. So far we have seen how we can create a hierarchy of classes, but we haven't seen how we indicate the properties of a class. We'll look at that now.

Using the Schema on the Schema

When we were discussing `rdfs:domain` and `rdfs:range` I made a note that there were further restrictions on using these properties, and that I would explain what they were later. These restrictions are that the values referred to by the resource attributes must be resources of type `rdfs:Class` – in other words the values used in defining a schema are themselves determined by the schema. Let's look at this more closely.

Recall that we indicated the type of a `Constituency` as follows:

```
<rdf:Description rdf:about="http://www.pa.press.net/constituencies/281">
    <rdf:type
         resource="http://www.ePolitix.com/2001/03/rdf-schema#Constituency" />
    <epx:Name>Great Grimsby</epx:Name>
    <epx:MainIndustry>Fishing</epx:MainIndustry>
</rdf:Description>
```

We then used the same URI to indicate that values used with the `epx:CandidateFor` property had to refer to resources that had an `rdf:type` property set in the same way as the Great Grimsby constituency:

```
<rdf:Property
     rdf:about="http://www.ePolitix.com/2001/03/rdf-schema#CandidateFor">
    <rdfs:domain
          rdf:resource="http://www.Schemas.org/2001/01/rdf-schema#Person" />
    <rdfs:range
       rdf:resource="http://www.ePolitix.com/2001/03/rdf-schema#Constituency" />
</rdf:Property>
```

We can put these two pieces together and use the URI for Great Grimsby in some meta data, like this:

```
<rdf:Description rdf:about="http://www.epolitix.com/austin-mitchell">
    <epx:Name>Austin Mitchell</epx:Name>
    <epx:CandidateFor
         rdf:resource="http://www.pa.press.net/constituencies/281" />
    <epx:MemberOf>Labour Party</epx:MemberOf>
    <epx:DOB>19 September, 1934</epx:DOB>
</rdf:Description>
```

Looking at both fragments of code, we are able to establish:

- That the resource referenced in the `epx:CandidateFor` property must have a type of `epx:Constituency`.
- That `http://www.pa.press.net/constituencies/281` does indeed represent a constituency, and not a boat or a book.

However, if we are going to check the type of the resource referred to by the URI of this statement:

```
    <epx:CandidateFor
         rdf:resource="http://www.pa.press.net/constituencies/281" />
```

why shouldn't we check the type of the resource referenced by this statement:

```
    <rdfs:range
       rdf:resource="http://www.ePolitix.com/2001/03/rdf-schema#Constituency" />
```

Just as we want to differentiate a constituency from boats and books, surely it is even more important to ensure that the resources referred to in `rdfs:range` – in this case `epx:Constituency` – do actually exist as types? The same applies to `rdfs:domain` – it is important that we determine that the resource referred to here is really a class, otherwise it makes no sense to allow it to have properties:

```
    <rdfs:domain
          rdf:resource="http://www.Schemas.org/2001/01/rdf-schema#Person" />
```

All of this is achieved by specifying RDF schema restrictions on the components of the RDF schema. We know that there are properties called `rdfs:domain` and `rdfs:range`, and that they can only be used inside an `rdf:Property` definition, so let's specify that:

```
<rdf:RDF
 xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
 xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
>
    <rdf:Property rdf:about="http://www.w3.org/2000/01/rdf-schema#domain">
        <rdfs:domain
           rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#Property" />
    </rdf:Property>
    <rdf:Property rdf:about="http://www.w3.org/2000/01/rdf-schema#range">
        <rdfs:domain
           rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#Property" />
    </rdf:Property>
</rdf:RDF>
```

We also know that both properties have a value that must be an `rdfs:Class`:

```
<rdf:RDF
 xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
 xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
>
    <rdf:Property rdf:about="http://www.w3.org/2000/01/rdf-schema#domain">
        <rdfs:domain
              resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#Property" />
        <rdfs:range resource="http://www.w3.org/2000/01/rdf-schema#Class" />
    </rdf:Property>
    <rdf:Property rdf:about="http://www.w3.org/2000/01/rdf-schema#range">
        <rdfs:domain
              resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#Property" />
        <rdfs:range resource="http://www.w3.org/2000/01/rdf-schema#Class" />
    </rdf:Property>
</rdf:RDF>
```

You'll find in the RDF Schema specification more information on the domain and range values for the elements of the schema. If you will be using RDF schema a lot then it is worth studying the RDF/XML Recommendation at http://www.w3.org/2000/01/rdf-schema to get a handle on the syntax. We'll look at them briefly here.

Other Schema Elements

The most important parts of RDF Schema have now been introduced. For the sake of completeness, in this section we will run quickly through the remaining elements. The following schema excerpts can be found in http://www.w3.org/2000/01/rdf-schema which is why they use the `rdf:ID` attribute, rather than full URIs in `rdf:about` attributes.

Resources

This first group of schema items relates to the definition of resources.

rdfs:Resource

Although we have acted as if any URI that appears in an `rdf:about` or `rdf:ID` attribute identifies a resource, it is possible to more explicitly state that a URI refers to a resource. Although a bit self-referential, `rdfs:Resource` is defined as a class:

```
<rdfs:Class rdf:ID="Resource" />
```

This therefore also gives us the `rdf:type`:

http://www.w3.org/2000/01/rdf-schema#Resource

rdfs:label

In some situations a human-readable version of a URI would be required, perhaps to display on a data entry form or in a report. To achieve this, the schema allows for a label to be attached to a resource. For example:

```
<rdf:RDF
 xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
 xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
>
    <rdfs:Class rdf:about="http://mydata.com/2001/01/rdf-schema#Minister">
        <rdfs:label>Government Minister</rdfs:label>
        <rdfs:subClassOf
         rdf:resource="http://www.ePolitix.com/2001/03/rdf-schema#Politician"
        />
    </rdfs:Class>
</rdf:RDF>
```

Note how the RDF schema limits the range of the `rdfs:label` property to a string literal:

```
<rdf:Property rdf:ID="label">
    <rdfs:domain rdf:resource="#Resource"/>
    <rdfs:range rdf:resource="#Literal"/>
</rdf:Property>
```

rdfs:comment

RDF Schema also provides a facility for comments to be added to resources. Again the property can only be used on a resource of type `rdf:Resource` and its value must be a string literal:

```
<rdf:Property rdf:ID="comment">
  <rdfs:domain rdf:resource="#Resource"/>
  <rdfs:range rdf:resource="#Literal"/>
</rdf:Property>
```

Schema Control

This next group of schema items is intended to relate to the use of schemas.

rdfs:seeAlso

The `rdfs:seeAlso` property indicates a resource that may contain more information. Exactly what that information is remains undefined – it will depend on the application. The domain and range of this property are as follows:

```
<rdf:Property rdf:ID="seeAlso">
    <rdfs:range rdf:resource="#Resource"/>
    <rdfs:domain rdf:resource="#Resource"/>
</rdf:Property>
```

rdfs:isDefinedBy

Although the exact workings of `rdfs:seeAlso` are undefined, a sub-property of that property does deal directly with schema. As with `rdf:seeAlso`, this property can be applied to any instance of `rdfs:Resource` and may have as its value any `rdfs:Resource`. The most common anticipated usage is to identify an RDF schema:

```
<rdf:Property rdf:ID="isDefinedBy">
    <rdfs:subPropertyOf rdf:resource="#seeAlso"/>
    <rdfs:range rdf:resource="#Resource"/>
    <rdfs:domain rdf:resource="#Resource"/>
</rdf:Property>
```

The main use of this property will be when it is not obvious what schema should be used for a property or class. We can express our earlier candidate example using a typed node, as follows:

```
<epx:Candidate
 rdf:about="http://www.epolitix.com/austin-mitchell"
 xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
 xmlns:s="http://www.Schemas.org/2001/01/rdf-schema#"
 xmlns:epx="http://www.ePolitix.com/2001/03/rdf-schema#"
>
    <s:Name>Austin Mitchell</s:Name>
    <s:DOB>19 September, 1934</s:DOB>
    <epx:CandidateFor
         rdf:resource="http://www.pa.press.net/constituencies/281" />
    <epx:MemberOf rdf:resource="http://www.ePolitix.com/parties#labour" />
</epx:Candidate>
```

Then it is pretty straightforward to see where the schemas are located since they are identified by the XML namespace declarations. However, if we had represented the same information like this, it is not so clear where the schema that defines `epx:Candidate` is located:

```
<rdf:Description
 rdf:about="http://www.epolitix.com/austin-mitchell"
 xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
 xmlns:s="http://www.Schemas.org/2001/01/rdf-schema#"
 xmlns:epx="http://www.ePolitix.com/2001/03/rdf-schema#"
>
    <rdf:type resource="http://www.ePolitix.com/2001/03/rdf-schema#Candidate" />
    <s:Name>Austin Mitchell</s:Name>
    <s:DOB>19 September, 1934</s:DOB>
```

```
        <epx:CandidateFor
             rdf:resource="http://www.pa.press.net/constituencies/281" />
        <epx:MemberOf rdf:resource="http://www.ePolitix.com/parties#labour" />
    </rdf:Description>
```

In this case it is useful to indicate where the schema for the type can be located, like this:

```
    <rdf:Description
     rdf:about="http://www.ePolitix.com/2001/03/rdf-schema#Candidate"
     xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    >
        <rdfs:isDefinedBy
              rdf:resource="http://www.ePolitix.com/2001/03/rdf-schema" />
    </rdf:Description>
```

Constraints

We have seen two types of limitation on the relationships that can be described with RDF Schema, and they are `rdfs:domain` and `rdfs:range`. However, RDF Schema has a facility to allow other constraints to be created, although it does not define how those constraints should be handled by a particular RDF processing application.

rdfs:ConstraintResource

The first element in this group is a base class from which other constraint classes can be derived.

```
    <rdfs:Class rdf:ID="ConstraintResource">
        <rdfs:subClassOf rdf:resource="#Resource"/>
    </rdfs:Class>
```

The purpose is simply that a parser will know that it has a constraint if it encounters a class that is based on this class. As RDF Schema stands at the moment though, there is no way for the parser to then "discover" how it should process that constraint. RDF Schema in its basic form does derive some classes – the `rdfs:domain` and `rdfs:range` constraints – from this class though.

rdfs:ConstraintProperty

The `ConstraintProperty` class is also used as a basis from which to derive other classes – in this case classes that are properties that in some way constrain values in RDF schema. Note that this class is both a constraint resource and a property:

```
    <rdfs:Class rdf:ID="ConstraintProperty">
        <rdfs:subClassOf
              rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#Property"/>
        <rdfs:subClassOf rdf:resource="#ConstraintResource"/>
    </rdfs:Class>
```

This class forms the basis for the two constraints that you have already seen – `rdfs:domain` and `rdfs:range`. I'll show their schema definitions here for completeness. First `rdfs:domain`:

```
    <rdfs:ConstraintProperty rdf:ID="domain">
        <rdfs:range rdf:resource="#Class" />
        <rdfs:domain
              rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#Property" />
    </rdfs:ConstraintProperty>
```

and then `rdfs:range`:

```
<rdfs:ConstraintProperty rdf:ID="range">
    <rdfs:range rdf:resource="#Class" />
    <rdfs:domain
          rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#Property" />
</rdfs:ConstraintProperty>
```

RDF Elements

RDF Schema also defines the basic RDF types.

rdf:Property

RDF Schema defines a class for `rdf:Property` as follows:

```
<rdfs:Class rdf:about="http://www.w3.org/1999/02/22-rdf-syntax-ns#Property">
    <rdfs:subClassOf rdf:resource="#Resource"/>
</rdfs:Class>
```

rdf:value

RDF Schema defines the `rdf:value` property as follows:

```
<rdf:Property rdf:about="http://www.w3.org/1999/02/22-rdf-syntax-ns#value" />
```

rdf:Statement, rdf:subject, rdf:predicate, and rdf:object

The elements required to create reified statements, in other words to create representations of the statements, are defined in RDF Schema like this:

```
<rdfs:Class rdf:about="http://www.w3.org/1999/02/22-rdf-syntax-ns#Statement">
    <rdfs:subClassOf rdf:resource="#Resource"/>
</rdfs:Class>

<rdf:Property rdf:about="http://www.w3.org/1999/02/22-rdf-syntax-ns#subject">
    <rdfs:domain
          rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#Statement"/>
    <rdfs:range rdf:resource="#Resource"/>
</rdf:Property>

<rdf:Property rdf:about="http://www.w3.org/1999/02/22-rdf-syntax-ns#predicate">
    <rdfs:domain
          rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#Statement"/>
    <rdfs:range
          rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#Property"/>
</rdf:Property>

<rdf:Property rdf:about="http://www.w3.org/1999/02/22-rdf-syntax-ns#object">
    <rdfs:domain
          rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#Statement"/>
</rdf:Property>
```

Container Elements

All of the RDF container elements – `rdf:Bag`, `rdf:Seq`, and `rdf:Alt` – are based on a class called `rdfs:Container`. This makes it easier for the parser to spot resources that are of type container, regardless of which type of container is used. The schema therefore defines the base class first:

```
<rdfs:Class rdf:ID="Container">
  <rdfs:subClassOf rdf:resource="#Resource"/>
</rdfs:Class>
```

It then defines the `rdf:Bag`, `rdf:Seq`, and `rdf:Alt` classes:

```
<rdfs:Class rdf:about="http://www.w3.org/1999/02/22-rdf-syntax-ns#Bag">
    <rdfs:subClassOf rdf:resource="#Container"/>
</rdfs:Class>

<rdfs:Class rdf:about="http://www.w3.org/1999/02/22-rdf-syntax-ns#Seq">
    <rdfs:subClassOf rdf:resource="#Container"/>
</rdfs:Class>

<rdfs:Class rdf:about="http://www.w3.org/1999/02/22-rdf-syntax-ns#Alt">
    <rdfs:subClassOf rdf:resource="#Container"/>
</rdfs:Class>
```

There is one more property that relates to the members of a container, which we saw in the previous chapter. As you remember, to specify the following property we can use syntax like this:

```
<rdf:Description rdf:about="http://www.ePolitix.com/Articles/0000001AEA32.htm">
  <dc:Subject>
    <rdf:Bag>
      <rdf:li>foot-and-mouth</rdf:li>
      <rdf:li>agriculture</rdf:li>
    </rdf:Bag>
  </dc:Subject>
</rdf:Description>
```

or this:

```
<rdf:Description rdf:about="http://www.ePolitix.com/Articles/0000001AEA32.htm">
  <dc:Subject>
    <rdf:Bag rdf:_1="foot-and-mouth" rdf:_2="agriculture" />
  </dc:Subject>
</rdf:Description>
```

Either of these two ways of specifying the property amounts to the same set of triples when parsed. RDF Schema calls this property a **container membership property**, and defines it as follows:

```
<rdfs:Class rdf:ID="ContainerMembershipProperty">
    <rdfs:subClassOf
          rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#Property"/>
</rdfs:Class>
```

Note that the definition does not limit the domain of this property to a class of type `rdfs:Container`, but it should actually be like this:

```
<rdfs:Class rdf:ID="ContainerMembershipProperty">
    <rdfs:subClassOf rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-
ns#Property" />
    <rdfs:domain rdf:resource="#Container" />
</rdfs:Class>
```

Of course, as RDF Schema stands, there is no easy way to automatically generate all the triples for the properties that are derived from this class:

```
<rdf:Description rdf:about=" http://www.w3.org/1999/02/22-rdf-syntax-ns#_1">
    <rdf:type resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#Property"/>
    <rdfs:subPropertyOf rdf:resource="#ContainerMembershipProperty"/>
</rdf:Description>
<rdf:Description rdf:about=" http://www.w3.org/1999/02/22-rdf-syntax-ns#_2">
    <rdf:type resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#Property"/>
    <rdfs:subPropertyOf rdf:resource="#ContainerMembershipProperty"/>
</rdf:Description>
<rdf:Description rdf:about=" http://www.w3.org/1999/02/22-rdf-syntax-ns#_3">
    <rdf:type resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#Property"/>
    <rdfs:subPropertyOf rdf:resource="#ContainerMembershipProperty"/>
</rdf:Description>
…
```

The closest we can get would be to use the `aboutEachPrefix`:

```
<rdf:Description
     rdf:aboutEachPrefix="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
    <rdf:type resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#Property" />
    <rdfs:subPropertyOf rdf:resource="#ContainerMembershipProperty" />
</rdf:Description>
```

but this would allow non-integers as the final part of the fragment identifier:

```
<rdf:Description rdf:about="http://www.ePolitix.com/Articles/0000001AEA32.htm">
  <dc:Subject>
    <rdf:Bag rdf:_yt7ER4="foot-and-mouth" rdf:_ab113zy="agriculture" />
  </dc:Subject>
</rdf:Description>
```

It is highly likely that this type of constraint will be possible in a future version of RDF Schema.

Typing in RDF and RDF Schema

The only remaining item to discuss is `rdf:type`. RDF Schema defines this as follows:

```
<rdf:Property rdf:about="http://www.w3.org/1999/02/22-rdf-syntax-ns#type">
    <rdfs:range rdf:resource="#Class" />
</rdf:Property>
```

which implies that the only resources that can be used as values for the `rdf:type` predicate are those that are of type `rdfs:Class`.

I've purposefully left this until the end because it points to a glaring gap between RDF syntax and RDF Schema. From this schema constraint it would appear that an RDF/XML document can only use the `rdf:type` property if it is using an RDF schema. This may well be the intention, but it is not referred to at all in the RDF Model and Syntax document. In that document it seems that a resource can be typed, without the type URI itself being checked – in other words you can produce RDF/XML documents that have typed resources but on which you don't want to impose a schema.

At first sight this does not seem to be a problem – give schema to those who want it, and omit if for those who don't. The problem is, what should a schema processor do with a document that uses `rdf:type`, but was intended for a non-schema aware processor? How is it to know that the resource referred to in the type attribute is *not* a schema, but just some made up URI? And what if there *is* a schema at the address referred to by the URI? It may well be the case that some RDF model makes use of types that originate in some schema, but it does not want to be validated against that schema – how is our parser to know?

Inferencing

Anther point worth mentioning about RDF schemas – and currently quite contentious – is that although the most obvious way to use schemas is to validate an RDF model, they could also be used to draw conclusions about an RDF model – or inference.

Take the schema statements that we gave earlier for a party member:

```
<rdf:Property rdf:ID="MemberOf">
    <rdfs:domain rdf:resource="#Politician" />
    <rdfs:range rdf:resource="#PoliticalParty" />
</rdf:Property>
```

Now imagine that our inferencing software was given the following model:

```
<rdf:Description
 rdf:about="http://www.epolitix.com/austin-mitchell"
 xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
 xmlns:s="http://www.Schemas.org/2001/01/rdf-schema#"
 xmlns:epx="http://www.ePolitix.com/2001/03/rdf-schema#"
>
    <s:Name>Austin Mitchell</s:Name>
    <s:DOB>19 September, 1934</s:DOB>
    <epx:MemberOf rdf:resource="http://www.ePolitix.com/parties#labour" />
</rdf:Description>
```

Note that there is no type information, but given the presence of the `epx:MemberOf` predicate, we could use the schema statement to infer that:

- The resource identified by `http://www.epolitix.com/austin-mitchell` is of type `epx:Politician`.
- The resource identified by `http://www.ePolitix.com/parties#labour` is of type `epx:PoliticalParty`.

Although this usage of RDF Schema has obvious uses, there are problems with merging validation and inferencing into one set of definitions. The main problem as far as the current state of RDF Schema is concerned relates to the `rdfs:domain` property. Recall that this property can appear more than once in an `rdf:Property` definition, reflecting the fact that a property can be valid on more than one class:

```
<rdf:Property
 rdf:about="http://www.ePolitix.com/2001/03/rdf-schema#Region"
 xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
>
    <rdfs:domain
          rdf:resource="http://www.Schemas.org/2001/01/rdf-schema#Person" />
    <rdfs:domain
          rdf:resource="http://www.Schemas.org/2001/01/rdf-schema#Company" />
</rdf:Property>
```

As far as validation is concerned, this simply means that the `epx:Region` property may appear on a `Person` class or a `Company` class. But what would the same schema mean when performing inferencing? The occurrence of the `epx:Region` property would mean that the resource with this property is either a `Person` or a `Company`, but we cannot conclude which.

There is a proposal with the W3C's working group on RDF to resolve this by saying that if a resource has the `epx:Region` property it is ***both*** a `Person` and a `Company`. The consequence of this when using the schema for validation would be that the `epx:Region` property would only be applicable to classes which are of type `Person` ***and*** of type `Company`. This seems to me to actually weaken RDF Schema as a whole, and instead the situation is best resolved by adding a layer of classes to RDF Schema specifically for inferencing. We will look at how further languages can be layered on top of RDF Schema in the next section.

New Layers on RDF Schema

RDF Schema provides a number of useful features that can be used to restrict meta data. We have seen that a limit can be placed on the possible values that can be used with a predicate, and we have also seen that it is possible to specify which predicates can apply to which resource types.

But whilst this is an important layer to build on top of RDF, for many applications it will not be enough. For example, we may want to say that if a person has credit card information in their meta data, then they must also have an address. Restrictions such as this cannot be specified with the current set of constraints used in RDF Schema. RDF Schema was deliberately scaled back to provide a minimum set of features. The idea is to enable the development of a number of ontology languages, but that each such language would have the same basic notions of Class, Property, domain and range.

We may also want to use the schema information for different applications. As we saw in the previous section, we may want to use the schema to say that any resource that has a constituency property is a Member of Parliament. We noted in the previous section that simple inferences such as this could be carried out using RDF Schema as it stands, but more complex derivations could not. What if we want to say that any MP, who won their seat by only 5% more votes than whoever came second, is in a precarious situation come the next election?

That may seem a little long-winded, but there are many situations where the ability to either define a new class from another class with additional restrictions – a `child` is a `person` whose `age` is less than 18 – or to infer something from a combination of values – if you are `male` and your `father` is the same `person` as my `father` then you are my `brother` – will be extremely useful, if not essential.

RDF Schema was designed with extension in mind, and you'll find some extensions at the SemanticWeb.org web site (http://www.semanticweb.org). We will finish this chapter with a brief look at two of these extensions – OIL and DAML.

An Example RDF Schema Extension: OIL

The OIL initiative is funded by the European Union IST program for Information Society Technologies under the On-To-Knowledge project and IBROW. More information can be found at http://www.ontoknowledge.org/oil.

It's not completely clear what OIL stands for – it could be Ontology Inference Layer, or Ontology Interchange Language. Regardless of the exact definition, the concept of **ontologies** is key.

Ontologies

An ontology is a set of agreed terms. For example, we might create an ontology for the animal kingdom that gives us terms such as animal, mammal, fish and carnivore. Similarly, the concepts we have seen for politics – politician, candidate, minister, and so on – could also comprise an ontology, because it is an agreed set of terms. Arranging the terms hierarchically allows us to determine that, for example, mammals are animals, or candidates are politicians.

Within an ontology we can go further than just defining a set of terms – we can also specify how they relate to each other. So we might say, for example, that an animal that is a carnivore cannot also be a herbivore. We might also specify conclusions that can be drawn, for example, if the food source of an animal is only other animals then we can infer that the animal is a carnivore.

Whilst RDF Schema has certain features to allow the building of simple ontologies, OIL has been created with ontologies in mind, and adds more functionality. Let's have a look at some of the features of OIL.

Defining an Ontology

We won't go into OIL in great detail, since the purpose of this chapter is RDF Schema. Instead we'll look at a few features of OIL and then see how they can be implemented by extending RDF Schema.

OIL ontologies do not actually *require* RDF. The language provides a simple English-type syntax to express rules in. However, these sentences are very easily mapped to RDF and the OIL documentation discusses how the English-type syntaxes can be represented with RDF/XML. We'll explore both means of representation as we look at some of the features of OIL.

The first feature to look at is the class definition. At its simplest a class can be defined as a specialisation of another class (words that are part of OIL's semantics get shown in bold in the OIL language):

```
class-def lion
     subclass-of animal
```

You probably didn't have to spend long thinking about that one to see how we might model this using RDF Schema:

```
<rdf:Class rdf:ID="lion">
    <rdf:subClassOf rdf:resource="#animal">
</rdf:Class>
```

Let's look at some more of OIL's features. We said that OIL could define a class as a set of objects that meet certain criteria. Let's say that the class of `herbivores` has as its members all objects that are *not* a member of the class `carnivores`:

```
class-def herbivore
     subclass-of animal
     subclass-of NOT carnivore
```

We could specify this using the OIL RDF Schema, as follows:

```
<rdf:RDF
 xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
 xmlns:rdfs="http://www.w3.org/TR/1999/PR-rdf-schema-19990303#"
 xmlns:oil="http://www.ontoknowledge.org/oil/rdf-schema"
>
    <rdfs:Class rdf:ID="herbivore">
        <rdf:type rdf:resource="http://www.ontoknowledge.org/oil/rdf-
schema/#DefinedClass"/>
        <rdfs:subClassOf rdf:resource="#animal"/>
        <rdfs:subClassOf>
            <oil:NOT>
                <oil:hasOperand rdf:resource="#carnivore"/>
            </oil:NOT>
        </rdfs:subClassOf>
    </rdfs:Class>
</rdf:RDF>
```

First we define the class. This class is then made a subclass of two other classes. The first is pretty easy to see, and is the `animal` class. However, note the second – the class is effectively an anonymous resource which represents all objects that are *not* members of the `carnivore` class. The ability to subclass from an anonymous resource is perfectly valid in ordinary RDF Schema – what is new here is the ability to define one set of objects as the negation of some other set of objects. The class `herbivore` is defined as all `animals` that are *not* `carnivores`.

We'll now look at how OIL handles what we would call properties in RDF Schema. The definition of a `carvivore` is any `animal` that eats others:

```
class-def defined carnivore
    subclass-of animal
    slot-constraint eats
        value-type animal
```

The OIL RDF Schema version of this is:

```
<rdfs:Class rdf:ID="carnivore">
    <rdfs:subClassOf rdf:resource="#animal"/>
    <oil:hasSlotConstraint>
        <oil:ValueType>
            <oil:hasProperty rdf:resource="#eats"/>
            <oil:hasClass rdf:resource="#animal"/>
        </oil:ValueType>
    </oil:hasSlotConstraint>
</rdfs:Class>
```

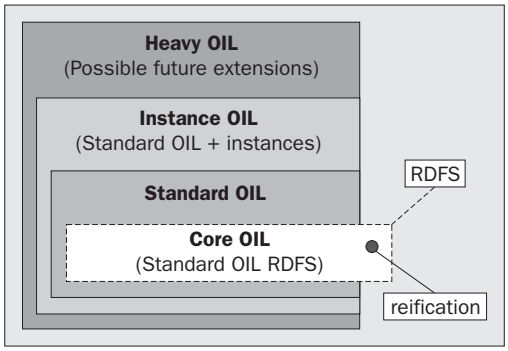
In OIL terminology a slot is basically a property. **Slot constraints** allow the value of a particular property to limited – or in this case inferred. So any class that is derived from the class `animal`, which has a property of `eats` and a value of a class that is also derived from the class `animal` can be concluded to be a `carnivore`. We merely need to extend our definition of `lion` as follows, to allow us to infer that `lion`s are `carnivore`s:

```
<rdfs:Class rdf:ID="lion">
    <rdfs:subClassOf rdf:resource="#animal"/>
    <oil:hasSlotConstraint>
        <oil:ValueType>
            <oil:hasProperty rdf:resource="#eats"/>
            <oil:hasClass rdf:resource="#herbivore"/>
        </oil:ValueType>
    </oil:hasSlotConstraint>
</rdfs:Class>
```

Just to reinforce the fact that the OIL elements in this schema are extensions of RDF Schema, let's look at the schema definition for one of the elements in an earlier example. The `hasOperand` property was used within `<oil:NOT>` to link the Boolean expression – in this case *not* – with another expression – in this case the class `carnivore`. The `hasOperand` property is defined as being allowable on classes of type `BooleanExpression`, and the values that can be used with `hasOperand` must be `Expressions`:

```
<rdf:Property rdf:ID="hasOperand">
    <rdfs:domain rdf:resource="#BooleanExpression" />
    <rdfs:range rdf:resource="#Expression" />
</rdf:Property>
```

The range of features that OIL defines is quite wide, although the plan is that further levels build on top of OIL, as the following diagram from the OIL web site shows:



I'll leave you to explore the web site if you want to find out more, but before I do, let's finish on one more example from OIL, which is the very useful ability to define inverse relations. At their simplest these can be used to say things like, if class `A` eats class `B`, then class `B` can be said to be eaten by class `A`. This is extremely useful in many areas – not just ontologies – and can be expressed in OIL the following way:

```
slot-def eats
    inverse is-eaten-by
```

and using RDFS, like this:

```
<rdf:Property rdf:ID="eats">
    <oil:inverseRelationOf rdf:resource="#is-eaten-by"/>
</rdf:Property>
<rdf:Property rdf:ID="is-eaten-by"/>
```

As you can see we have two properties `eats` and `is-eaten-by`. However, one is defined to be the inverse of the other, so that from our previous example:

```
<rdfs:Class rdf:ID="lion">
    <rdfs:subClassOf rdf:resource="#animal"/>
    <oil:hasSlotConstraint>
        <oil:ValueType>
            <oil:hasProperty rdf:resource="#eats"/>
            <oil:hasClass rdf:resource="#herbivore"/>
        </oil:ValueType>
    </oil:hasSlotConstraint>
</rdfs:Class>
```

we can conclude that `herbivores` are eaten by `lions`.

OIL and RDF Schema

We have said that OIL extends RDF Schema, but be aware that this would require an RDF processor that knew how to handle the OIL extensions. Although an RDF Schema processor would have no problems processing an ontology defined using OIL, there is much that it would miss out.

For example, in our illustrations above, we showed that the `hasOperand` property could be restricted to classes of a certain type. However, that doesn't help the processor when trying to work out what to *do* with such a property. All a simple RDF Schema processor could do would be to say that an OIL model was consistent from an RDF point of view (no properties are assigned to classes that shouldn't be, for example) but it couldn't draw anything more from the document. It certainly couldn't, for example, conclude that some object is an `herbivore,` because it is not a `carnivore`, or that some object is an `animal` because it is eaten by `lions`. A further layer of processing software would be needed.

An Example RDF Schema Extension: DAML

The US Department of Defence has a research department called DARPA – the Defence Advanced Research Projects Agency. In August 2000 they kicked off a program to develop a language that would better describe the relationships between objects. The language was the DARPA Agent Markup Language – or DAML.

Many regard agents as the key software that will make the Semantic Web work. The idea is that one day you should be able to say to your agent software that you need to organize a business meeting with me in New York, and your software would then check availability dates with my diary agent, check flight information with a travel agent, reserve hotel rooms with a hotel agent, and then confirm all of this information with you and me.

As you can imagine, with the web as it stands the biggest barrier to this type of technology is not so much with the reasoning software, but the fact that most information sources are not talking the same language – the very problem that we encountered at the beginning of our discussion on RDF. The DAML project aims to provide a language that can be used to define further languages that enable the communication of agents.

We won't go into the details on DAML here, but if you are interested you can get more information at http://www.daml.org. For the purposes of our discussion on RDF Schema it is worth just pointing out that DAML is simply a thin layer on top of RDF Schema, providing only a small number of features. These features include simple things such as providing the ability to restrict the number of occurrences there are of a particular property, and the addition of XML Schema data types.

DAML is significant not so much for these additions to RDF Schema, but more because it is itself the basis for further extensions. The DAML project provides a number of further languages for different areas that will hopefully all come together to enable agent technology. One example is DAML+S – or DAML Services – which allows a data service to indicate to other services what features it is capable of supporting. Another example is DAML+OIL, which incorporates all of the elements of OIL that we saw in the previous section, but builds them onto DAML, rather than straight onto RDF Schema. These extensions – and many others – are outlined on the DAML web site.

Summary

We have looked in this chapter at how RDF Schema uses RDF to encode schemas that can be used to check the intention of some RDF-encoded meta data, rather than simply its format, as XML Schema might do. We also saw that RDF Schema is intended to be extendable, and that a great deal of work is going on to extend RDF Schema, particularly in the worlds of knowledge representation and logic. We looked at OIL, which is used to define ontologies, and at DAML, which adds a small number of features to RDF Schema to make it easier to define further languages to facilitate agent technologies to communicate.

In the next chapter we will look at some of the issues that arise when processing the RDF/XML documents that carry our meta data. In particular, we will look at parsing.