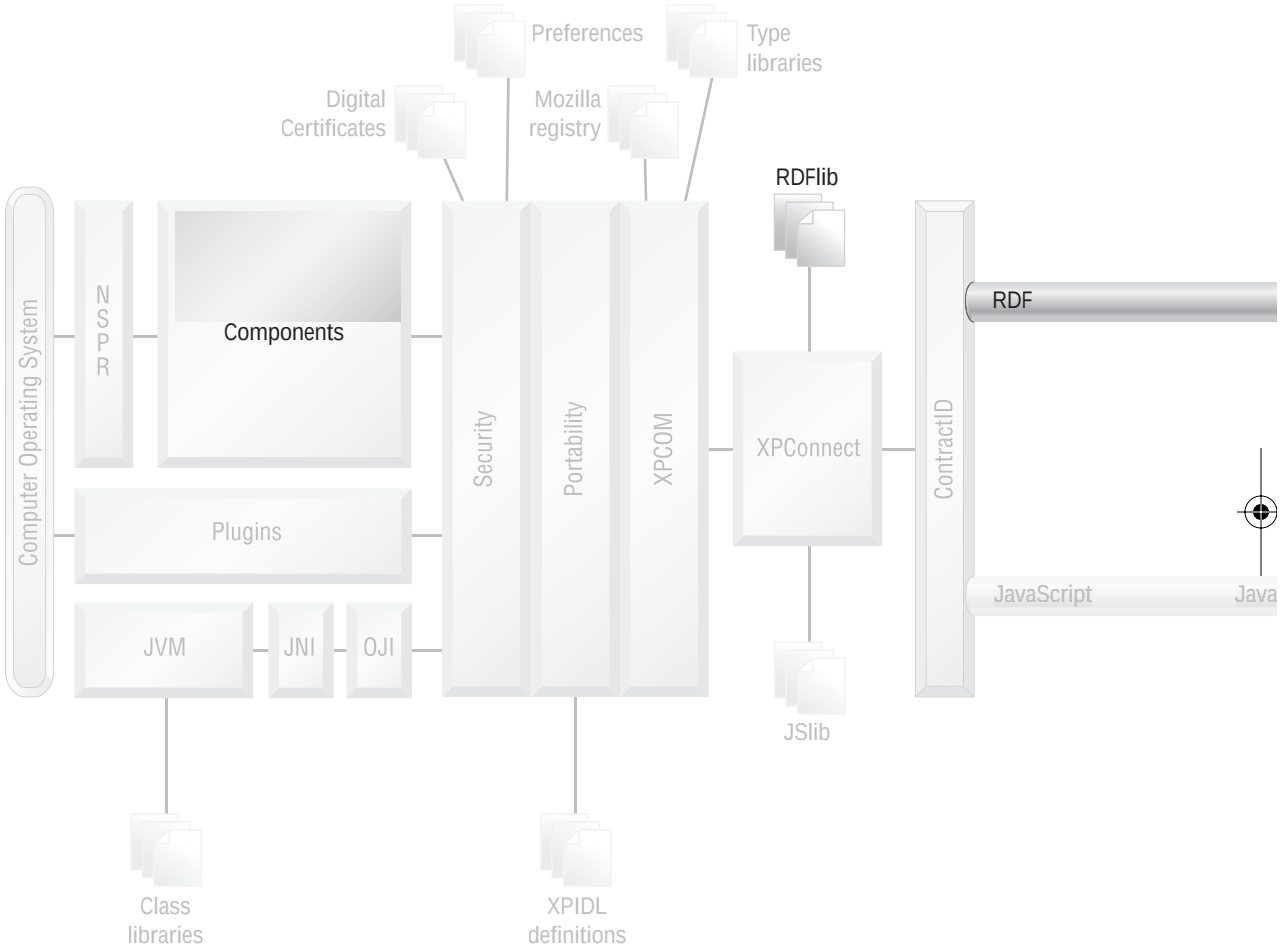
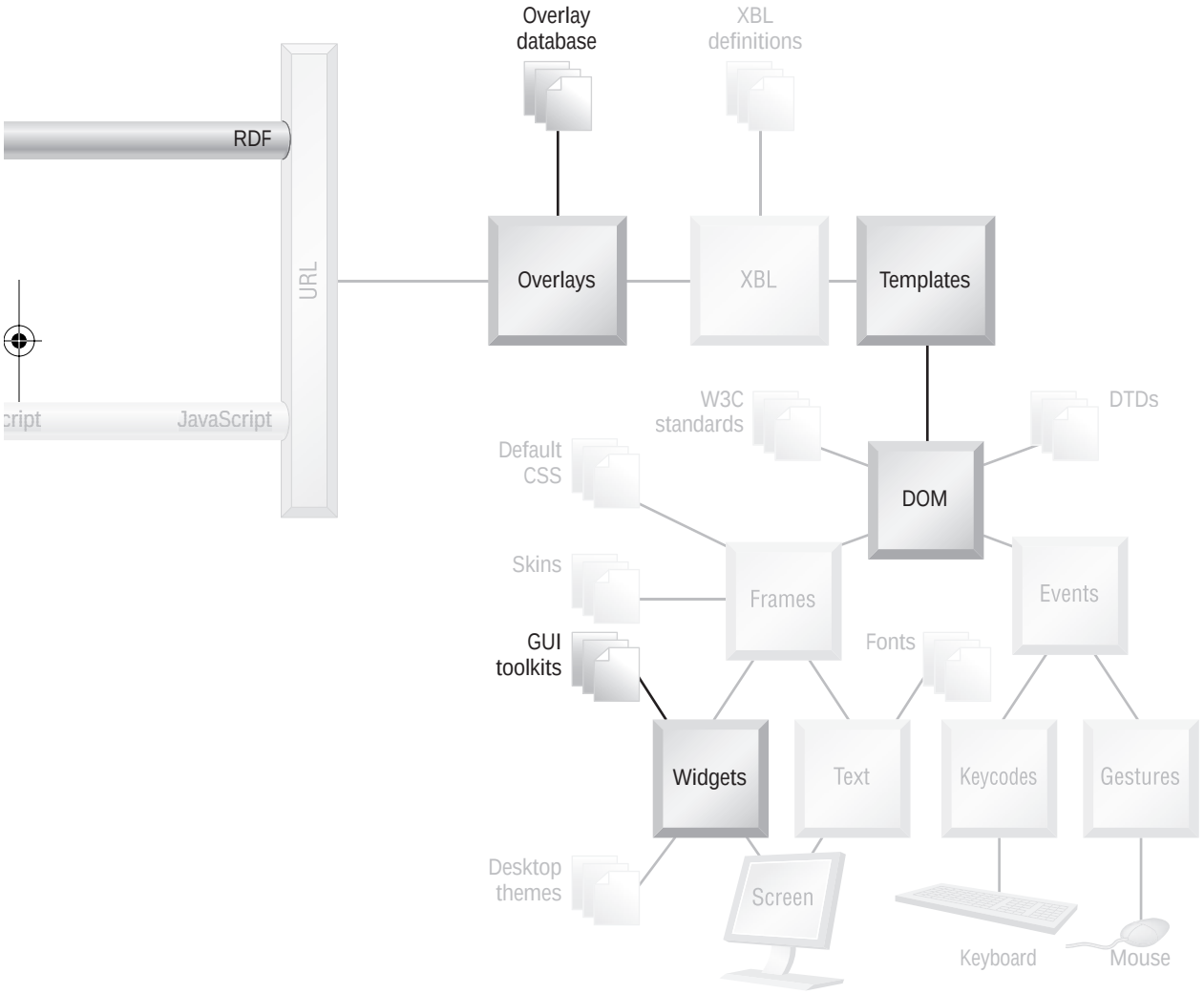

Preferences
Type libraries
Digital Certificates
Mozilla registry
RDFlib
Computer Operating System
NSPR
Components
Security
Portability
XPCOM
XPConnect
ContractID
RDF
Plugins
JavaScript
Java
JVM
JNI
OJI
JSlib
Class libraries
XPIDL definitions

CHAPTER 11

RDF



This chapter explains the basics of RDF—a significant information format used by the Mozilla platform. RDF is a W3C XML standard. It is one of the more unusual technologies in Mozilla, but correctly applied it is both powerful and convenient.

Few applications can do useful work without externally supplied information, and Mozilla applications are no different. RDF is a good way to supply small amounts of reusable information. The Mozilla platform can process that RDF information. The Mozilla platform is also partially built out of RDF information. The everyday operation of most Mozilla-based applications depends on RDF building blocks.

This chapter explains the underlying concepts, syntax, and a little about platform support. That is quite a lot by itself. This chapter is mostly RDF introduction, just as Chapter 5, Scripting, introduced JavaScript. Chapters 12, Overlays and Chrome; Chapter 14, Templates; and Chapter 16, XPCOM Objects, expand greatly on applications of RDF.

Most computer technologies can be picked up with a few glances. That strategy doesn't work if you meet something that is a bit new and unusual. In such a case, you need to slow down and absorb the material more systematically. RDF is such a technology. It is also a gateway to all the fancier features of Mozilla. Pull up a comfy chair, pour your drink of choice, and discover this technology carefully. RDF can be fascinating, expressive, and thought-provoking.

What is RDF? Well, there are many kinds of information. One arbitrary set of information categories might be *content*, *data*, and *facts*. Each is processed in a different way. Content-like information tends to be processed as a whole: Display this HTML page; play that music. Datalike information tends to be processed piecewise: Add this record to a database; sort this list of objects. Factlike information is less commonly seen. Facts are statement-like data items. Facts are used by ordinary humans in their daily lives, by specialist academics, and by technologists called knowledge engineers. Some everyday examples of facts are

I went to the shop.

The moon is made of green cheese.

Tom, Dick, and Harry are brothers.

This function is never used.

Every person must find their own path through life.

It is not important whether these facts are true when tested against the real world. It is not important where they came from or whether anyone agrees with them. The important thing is that it is possible to write them down in some useful and general way (in this case, in English). Writing facts down moves them from your head to a formal specification where they can be used. It is only *after* they are captured that you might reflect on their impor-

tance or correctness. The modeling in the "Hands On" session in this chapter shows one way that facts can be scrutinized.

The world, including computer technology, is soaked with factlike information. Nearly all of it is expressed in ways that are not specifically designed for fact processing. Generalist programmers rarely deal with specialist fact-processing systems, even though their code is full of implied facts. That implied information is merely used to get other things done. RDF, on the other hand, is an explicit fact-writing system.

A very primitive, trivial, and non-RDF example of a factlike system is the Classic Mozilla bookmarks file, stored in the user profile. Here is a snippet of that file:

```
<A HREF="http://www.mozilla.org/"
   ADD_DATE="961099870"
   LAST_VISIT="1055733093"
   ICON="http://www.mozilla.org/images/mozilla-16.png"
   LAST_CHARSET="ISO-8859-1">
  The Mozilla Organization
</A>
```

This code states information about a URL: the date added; the date last visited. These XML attributes can be seen as plain data or as facts about a URL. Although facts can be expressed in plain XML (or ancient semi-HTML, as shown), there is no hint in those syntaxes what a standard form of expression should be. RDF exists to provide that formality. The bookmarks file is not written in RDF because of backwards-compatibility issues.

In this bookmark example, many industries call the stated information *metadata*. The term metadata is supposed to help our minds separate the data a URL *represents* (content) from the data that is *about* a URL (a description). Unfortunately, if a programmer writes code to process the bookmark file, the only interesting information is the so-called metadata—the substance of that file, which is its content. To the programmer, the metadata is therefore the data to be worked on. This is a very confusing state of affairs when trying to learn RDF. One person's metadata is another's data.

In short, the term *metadata* is overused. To a programmer, the only thing in RDF that should be considered metadata is type information. Everything else is just plain data or, preferably, plain facts. No facts are special; no facts have any special "meta" status.

RDF has its own terminology. For example, here is a fact expressed in RDF:

```
<Description about="file:///local/writing/" open="true"/>
```

In simple terms, this line says that the folder `/local/writing` is open. A more precise RDF interpretation is: "There exists a subject (or resource) named `file:///local/writing`, and it has a predicate (or property) named `open`, whose object (or value) is the anonymous literal string `"true"`. That is rather awkward language, and we need to explore what it all means.

Finally, RDF is not a visual language. Mozilla cannot lay it out as for HTML or XUL. If display is required, RDF must be hooked up to XUL. RDF itself is silently processed inside the platform. Some of this internal processing happens automatically without any programmer effort. The concept of a *data source* is central to all that processing.

The NPA diagram at the start of this chapter shows that RDF support extends from the very front to the very back of Mozilla's architecture. At the back are the precious XPCOM components that the application developer can use for power manipulation of RDF content. A convenient set of scripts also exists to make this job easier. Those scripts are called RDFlib here, but they are really part of the JSLib library. RDF technology acts like a bridge between front and back ends of the platform. This is because there is direct support for RDF in both XUL and in the scriptable AOM objects. The two ends are automatically connected. The template and overlay systems, both of which manipulate XUL, also depend on RDF.

Unfortunately, Mozilla is only version 1 and RDF processing could be faster. Don't use RDF for data with millions of records; it is not a database. Performance is more than adequate for small data sets.

11.1 MOZILLA USES OF RDF

Here is a taste of what RDF can be used for.

Classic Mozilla uses RDF extensively in the construction of the Classic application suite. Some of those uses involve RDF files stored in the file system, and some do not. Uses that do create RDF files are

- User choices for window arrangement and position
- Content of the Mozilla Sidebar
- Manifest files for JAR archives, chrome packages, skins, and locales
- Overlay database for application overlays
- Search types for the Smart Browsing Navigator feature
- Searching and Viewing states in the DOM Inspector
- The Download Manager
- MIME types

In addition to these uses, Netscape 7 creates and uses many custom RDF files. Enhancements to the Classic Browser, such as those at *www.mozdev.org*, might also manipulate RDF files.

RDF is a data model as well as a file format. Mozilla's platform infrastructure uses RDF facts in a number of places, without necessarily reading any RDF documents. This infrastructure might convert a non-RDF source into RDF for internal processing. Some places where the RDF model is important are

- The XUL Overlay system described in Chapter 12, Overlays and Chrome
- The XUL Template system described in Chapter 14, Templates
- Directories and files in the local file system
- Bookmarks
- Web page navigation history
- Downloadable Character Set support
- The Mozilla registry
- The What's Related feature of the Sidebar
- Currently open windows
- The address book
- Email folders
- Email folder messages
- Email SMTP message delivery
- Email and Newsgroup accounts
- Sounds to play when email arrives

RDF is not used for any of the following tasks: permanent storage of Internet connections and subscriptions; databases of newsgroups and of newsgroup headers; databases of email folders and of email items; or the platform's Web document cache.

11.2 Learning Strategies for RDF

Learning RDF is like flying. It's tricky to get off the ground, but once you're away, it's great. Why should this be so, and what can be done to make it easier? Here are some thoughts.

XML syntax is quite verbose. By the time your eye and brain absorb all the text and punctuation in an example RDF file, it's hard to focus on the bigger picture. Even the W3C RDF standards people have acknowledged this problem. Use an informal syntax at design time, and only use the official RDF syntax during code and test. This chapter uses informal syntax everywhere except in real code.

RDF itself is frequently confused with its applications. The nature of RDF is one thing; a purpose to which RDF is put is another thing entirely. Reading about content management when trying to learn RDF is of no use. That's like trying to understand a database server by learning an accounting package. It's best to learn the fundamental technology first.

Well-known explanations of RDF are aimed at many different audiences. When reading someone else's explanation, ask yourself: Was that explanation suited to me? Don't frustrate yourself with an explanation that doesn't suit your purpose or your mindset.

RDF in its full glory is also quite big, even though it only has about ten tags. It is equal to several XML standards all at once. RDF runs all the way from simple data to schemas and meta-schemas. That is too much to absorb in one quick reading. To experiment with RDF, practice with very simple tasks to start with. Gain confidence with the basics before trying to compete with Einstein. It's just like any big technology—don't get run over.

Finally, RDF presents an unusual learning trap for those who need absolute certainty. The concepts behind RDF are very open-ended. The RDF philosophy goes on forever with many subtleties and twists. Questions about the meaning of advanced features in RDF have few real answers. Take it at face value, and just use it.

Even given all that, RDF is no big deal. There are tougher XML standards, like OWL. If you have any Prolog or AI training, then RDF will be trivial to pick up.

11.3 A TUTORIAL ON FACTS

The basic piece of syntax in XML and RDF is the element, which is frequently expressed as a tag. The basic concept unique to RDF is the fact. A fact does equal one element, but only sometimes equals one tag. What is a fact, and what can you do with one? That is first up. Experts on deductive predicate logic need only glance though this material.

A programmer can glimpse the world of facts through familiar technologies that are a little factlike. Two examples are SQL and `make`. Manipulating records (rows) in a relational database via SQL's INSERT, DELETE, and particularly SELECT is a little factlike. Retrieving rows with a query is like retrieving facts. Alternatively, stating file dependencies in a `make(1)` makefile and letting `make` deduce what is old and needs re-compilation is a little factlike. A makefile dependency rule is like a fact about files or targets. Another example of a configuration file that is factlike is the rather unreadable UNIX `sendmail.cf` configuration file.

What these systems have in common is that the preserved data items are stated independently and are multivalued—each fact has several parts (columns/targets/patterns in the respective examples). Working with a fact means working with some processing engine, like a database server, `make` program, or email routing system. That processing engine presents or crunches facts for you.

11.3.1 Facts Versus Data Structures

Facts are used to describe or model data. Programmers typically model data using data structures. Programmers who are also designers might also model data using tools like data dictionaries or UML diagrams.

Perhaps the easiest way to see how facts differ from traditional data is to write one down. Suppose a boy is at the beach with a dog and a ball. Those four real-world objects (boy, beach, dog, ball) can be stored together as a data structure, or as a fact. Consider first traditional data structures. In JavaScript, this information could be stored as an object:

```
{ boy:"Tom", dog:"Spot", ball:"tennis", beach:"Waikiki" }
```

This is also near the syntax for a suitable C/C++ `struct`. Alternately, this information could be stored in a JavaScript array:

```
[ "Tom", "Spot", "tennis", "Waikiki" ]
```

Fortunately, all the data items consist of strings, which suit the array concept. This syntax can also be used for a C/C++ array. Perl, on the other hand, has lists:

```
( "Tom", "Spot", "tennis", "Waikiki", )
```

In general, there are many ways to write the same bits of data, and each way has its benefits and restrictions. Using an *object* or *class* implies that all the data items (object properties) share the same owner and have a type each. Using an *array* implies that the data items are numbered and of the same type. Using a *list* implies that the items are ordered. Programmers choose whichever is best for a given problem.

This information can also be written as a fact, using a *tuple*. A tuple is a group of *N* items, where *N* is any whole number. The number of items is usually fixed per tuple, so a tuple can't grow or shrink. The word tuple comes from the ordered set of terms: single, double, triple, quadruple, quin*tuple*, sex*tuple*, sep*tuple*, oc*tuple*, and so on. Few computer languages support tuples directly (SQL's INSERT is one), so mathematical notation is used as syntax. There are many different mathematical notations. For example, one briefly used in the RDF standards is

```
< Tom, Spot, tennis, Waikiki >
```

That notation, and many others that are similar, can be easily confused with XML tag names. We use

```
<- Tom, Spot, tennis, Waikiki ->
```

Each of the four words in the tuple is called a *term*. Later on, these "crow's feet" brackets will remind you that certain tuples should have three terms only. No quotes are required because this is not a programming language. The terms in a tuple are ordered (like a list) but not numbered (unlike an array) and do not have fixed types (unlike a `struct` or class). The meaning of a tuple is just this: These terms are associated with each other. How they are associated is not important in the general case.

The use of angle brackets < and > hints at the big difference between tuples and other data structures. That difference is that a tuple is a declaration and a statement, like an XML tag or a class definition. The data structure

examples are merely expressions. An expression can be calculated and put into a variable. You can't put a statement into a variable. A statement just exists.

When a tuple statement is processed, it simply makes the matching fact true. If a tuple for a fact exists, the fact is said to be true. If the tuple doesn't exist, the fact is said to be false. This truth value is not stored anywhere; it is calculated as required. This arrangement lets the programmer imagine that all possible facts can exist. This is convenient for computer programming because you can process what you've got and conclude that everything else is untrue (false).

The example tuple we created makes this fact true: "Tom, Spot, tennis, and Waikiki are associated with each other." It is true because we've managed to write it down. Note that the tuple we've created contains most, but not all, of the original statement. For example, it doesn't state that Tom and Spot were at Waikiki at the same time. It is normal for any information-gathering exercise to capture the most important details first.

This example tuple has four terms in it. It could have any number of terms. To keep the example simple, we'll now reduce it to just a boy, his dog, and a ball—three terms. Where they happen to be (at a beach or otherwise) is no longer important.

Suppose that this simpler example needed to be captured more completely. A standard modeling approach is to start by identifying the nouns. From there, objects, classes, entities, tables, or types can be inferred. This can also be done for facts. A JavaScript example using objects is shown in Listing 11.1.

Listing 11.1 Objects modeling boy and dog example.

```
var boy = { Pid:1, name:"Tom", Did:null, Bid:null };
var dog = { Did:2, name:"Spot", Pid:null, Bid:null };
var ball = { Bid:5, type:"tennis", color:"green" };

boy.Did = dog;    // connect the objects up
boy.Bid = ball;
dog.Pid = boy;
dog.Bid = ball;
```

`Pid`, `Did`, and `Bid` are short for Person-id, Dog-id, and Ball-id, respectively. These ids are used to make each person unique—there might be two different dogs, or Tom might have five green tennis balls. In addition to the base objects, some effort is made to link the data. Both Tom and Spot are concerned with the same ball; Spot is Tom's dog, Tom is Spot's person.

This modeling can be repeated using tuples, as shown in Listing 11.2.

Listing 11.2 Tuples modeling boy and dog example.

```
<- 1, Tom, 2, 5 ->
<- 2, Spot, 1, 5 ->
<- 5, tennis, green ->
```

As for relational data, links (relationships) between nouns are represented with a pair of identical values. Here we use numbers as identifiers instead of the object references used in Listing 11.1. In Listing 11.2, there is a pair of 1s, a pair of 2s, and two pairs of 5s (the 5 in the third tuple is matched twice). Tuples obviously have a compact notation that is simpler to write than 3GL code. That is one of their benefits. Basic tuples have a naming problem though—there are no variable names to give hints as to what each tuple is about. So the tuple syntax is sometimes harder to read. Nevertheless, both relational databases and facts are based on the tuple concept.

These two modeling attempts, object-based and fact-based, have their uses, but overall they are not very good. The starting scenario is: "Tom and his dog Spot play with a ball." The results of the two modeling attempts are shown in Table 11.1.

Table 11.1 Example information held by object and tuple models

Object model	Tuple model
There is an object for Tom.	There is a tuple for Tom.
There is an object for Spot.	There is a tuple for Spot.
There is an object for a green tennis ball.	There is a tuple for a green tennis ball.
Tom uses-a Spot.	Tom, 1, 2, and 5 are associated.
Spot uses-a Tom.	Spot, 1, 2, and 5 are associated.
Tom uses-a green tennis ball.	Green tennis ball and 5 are associated.
Spot uses-a green tennis ball.	
(More information can be deduced.)	(More information can be deduced.)

Table 11.1 cheats a bit because the purpose of each tuple and object is assumed. The problem with both of these models is that priority is given to the *things* in the scenario (the nouns). The *relationships* between the things are a far distant second. Both models have lost a lot of relationship information. It is not captured that Tom *owns* Spot, or that Spot *plays* with the ball.

The traditional solution to this loss is to add more objects, or more tables, or more whatever. In the world of facts, the solution is to make every existing relationship a term in a tuple. Such a tuple is called a *predicate*.

11.3.2 Predicates and Triples

A special group of tuples are called predicates. Since all tuples are facts, predicates are also facts. Predicates contain terms holding relationship information as well as terms holding simple data items. The naïve way to add this relationship information is shown in Listing 11.3, which updates Listing 11.2.

Listing 11.3 Addition of relationships to boy and dog tuples.

```
<- 1, Tom, owner, 2, plays-with, 5 ->
<- 2, Spot, owned-by, 1, plays-with, 5 ->
<- 5, tennis, green ->
```

In this example the relationships have the same status as the other information. It is almost possible to read the first tuple as though it were a sentence: "Id one (Tom) is owner of id two (Spot) and plays with id five (a ball)." Clearly there is more specific and complete information here than in either of the attempts in Table 11.1. This process is similar to database entity-relationship modeling.

Here is some technical jargon: Tuples containing relationship information are called predicates because one or more terms in the tuple *predicates* a relationship between two other terms in the tuple. The relationship term by itself is also called a predicate because it is responsible for the predication effect. This is confusing unless you work out the context in which "predicate" is used. Either it refers to a whole tuple, or just to a particular term in a tuple. Here we try to use it only for the particular term. We use tuple, triple, or fact for the set of terms.

Listing 11.3 is still not ideal because some tuples have more than one relationship term. If a tuple contains more than one predicate term, then the processing of facts is not simple. Some repair of this example is therefore needed. Listing 11.4 splits the tuples up so that there is at most one predicate term per tuple.

Listing 11.4 Single predicate boy and dog tuples.

```
<- 1, Tom, owner, 2 ->
<- 1, Tom, plays-with, 5 ->
<- 2, Spot, owned-by, 1 ->
<- 2, Spot, plays-with, 5 ->
<- 5, tennis, green ->
```

At the cost of a little duplication, the predicates are now separated, and the tuples are perhaps even simpler to read. In database design, an equivalent process is called normalization. In software design, it is called factoring. In all cases, it is a divide-and-conquer information-reduction strategy. This refinement process is not complete, however. More can be done. These tuples can be cleaned up so that every tuple has exactly three items ($N = 3$). Listing 11.5 does this cleanup.

Listing 11.5 Predicate triples for boy and dog example.

```
<- 1, is-named, Tom ->
<- 1, owner, 2 ->
<- 1, plays-with, 5 ->
<- 2, is-named, Spot ->
<- 2, owned-by, 1 ->
```

```
<- 2, plays-with, 5 ->
<- 5, type-of, tennis ->
<- 5, color-of, green ->
```

All these tuples are now triples. Triples with a predicate item are a well-understood starting point for all factlike systems. Using predicate triples is like using first normal form for a database schema ("every table has a unique key …") or identifying nonvalue classes in an OO design ("All classes with identity…"). When thinking about facts, start with predicate triples ("A fact has three terms …"), not with general tuples.

Note that it is easy to get carried away with predicates and relationships. The last two triples in Listing 11.4 have very weak relationships. *Tennis* and *green* are more like simple descriptive properties than first-class data items like persons and dogs. Just as you can have too much normalization or too many trivial objects, you can have too many trivial facts. If trivial facts are the interesting part of your problem, however, then use them freely.

Because triples are so widely used, their three data items have formal names. The relationship-like item is called the predicate, as before. The first data item is called the *subject*, and the third data item is called the *object*. This use of the term "object" derives from the grammar of spoken language not from technology engineering. The answer to the question "Which of these three is a triple about?" is a matter of personal philosophy. The most common answer is that it is about the subject. In Listing 11.5, the predicate data items have been chosen so that the subject always comes first. That is a convention you should always follow.

Finally, facts that are predicates can be written in a number different ways. For example, in the Prolog computer language, they can be written:

```
predicate(subject, object)
plays-with(1,5)
```

Alternately, in Lisp or Scheme, they can be written:

```
(predicate subject object)
(plays-with 1 5)
```

Predicates can also be written in English:

```
subject predicate object
1 plays with 5
```

And, of course, using RDF, predicates can be written in XML. One of several options is to use a single tag:

```
<Description about="subject" predicate="object"/>
<Description about="1" plays-with="5"/>
```

To write down a predicate in convenient shorthand, the informal tuple notation of this chapter can also be used, or the punctuation can be stripped away, leaving the simple *N*-Triple syntax used by some RDF experts:

```
   tuple: <- subject, predicate, object ->
N-Triple: subject predicate object
```

If you prefer to stick to real-language syntax, then the simplest and clearest notations are those of Prolog and Lisp, which have been used for decades for fact-based processing. The alternative is RDF itself.

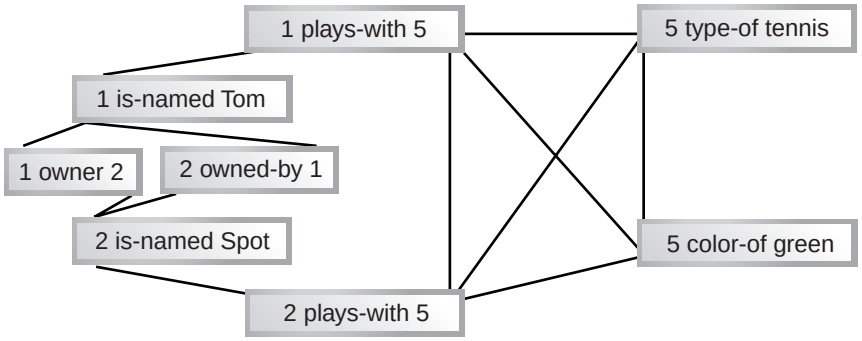


Fig. 11.1 Graph of boy-dog tuple links.

11.3.3 Three Ways to Organize Facts

So far, all that has been achieved in this tutorial is to identify what a good format for a fact is. How are you to store such facts in a computer? There are several ways to do so.

The first way to store facts is as a set of independent items. In relational technology that means as separate rows in a three-column table; in object technology, as a collection of items, say in a Bag or Set; and in plain data structure technology, as a simple list. Listing 11.5 is a written version of such a simple set.

Such a simple approach is very flexible. More facts can be added at any time. Facts can be deleted. There is no internal structure to maintain. Such a solution is like an ordinary bucket (a pail). You can pour facts into and out of the bucket as you please.

One of the chief benefits of using a bucket is that fact sets can be trivially merged. You can pour facts into the bucket from several sources. The result is just one big collection of facts. When facts are poured out of the bucket, all facts appear together, regardless of origin. This is just a simple union of two sets.

The second way to store facts is to recognize that there are links between them that create an overall structure. This structure can be stored as a traditional data structure, with pointers or references between the tuples. Because the links are quite general, the structure is a graph, rather than a simple tree or list. Recall that a graph is the most general way to represent data. Graphs have edges (lines) and vertices (junction points or nodes), and that's all. Both edges and vertices might be labeled. Such a graph can also be displayed visually. Figure 11.1 shows the links between facts from Listing 11.5.

This is a rather complex diagram for a system consisting of only a boy, a dog, and a ball. In fact, this diagram is six lines short; there should be three more "1" lines and three more "2" lines, for a total of 18 lines. A simplification strategy is to add intersection points by breaking out the ids from the tuples. Figure 11.2 shows this improved diagram, which reduces the total line count from 18 to 12.

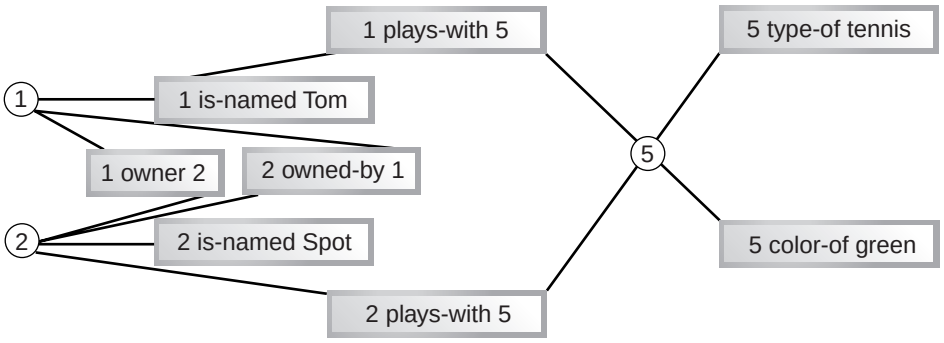


Fig. 11.2 Reduced graph of boy-dog tuple links.

The strategy of breaking out items from the tuples seems to have worked a little, so let us continue it. All of the items can be broken out, not just those with identifiers. Furthermore, it can be seen that some tuples are opposites (`owner` and `owned-by`, in this example). Such duplication can be removed by giving a line a direction. Follow an arrow one way for one predicate; follow it the reverse way for the opposite predicate. Figure 11.3 shows these improvements.

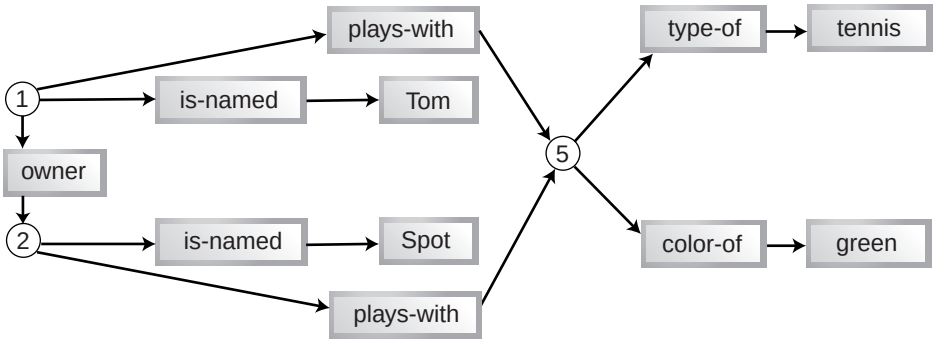


Fig. 11.3 Much reduced graph of boy-dog tuple links.

As a final improvement, note that every predicate has exactly one arrow in and one arrow out. The predicate might as well be used as a label for the arrow that passes through it, and save some boxes. That last step is done in Figure 11.4, which also slightly reorganizes the left side of the diagram.

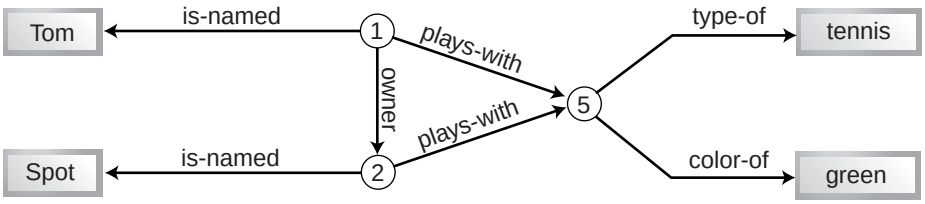


Fig. 11.4 RDF graph of boy-dog tuple links.

The predicate relationships in Figure 11.4 are very clear. There are peripheral predicates that merely add information to a given identifier (`is-named`, `type-of`, `colour-of`), and more critical predicates (`plays-with`, `owner`) that say something about the relationships between the identifiers under scrutiny. The decision to use an identifier for each real-world thing being modeled (decided in Listings 11.1 and 11.2), and to focus on the identifier in the diagram (done in Figure 11.2) have both paid off. Those identifiers have turned out to be vital for diagramming purposes.

The graph in Figure 11.4 follows the official RDF graph notation. Circles or ellipses are used for identifiers, and squares are used for literal values. A better system for identifier and predicate names is needed, though. We can use URLs as identifiers. That will come shortly.

The benefit of a graph representation of RDF data is that scripts can navigate intelligently through the set of facts, using or ignoring each connection, as the need determines.

There is a third way to organize facts, which is often used in Mozilla RDF documents. Drop all the facts onto a kitchen table randomly. Now take a needle and thread and run the thread through any terms that are relevant to a given topic. If you need to know about that topic, just pick the thread up and those terms and their related facts are lifted out from the full set. Continuing the boy and dog example, Figure 11.5 shows an imaginary line that connects all the number identifiers.

There is no syntax in RDF for implementing such lines. Instead, this effect is achieved with special RDF tags called *containers*. Since RDF can only represent facts, containers are made out of facts. Any term of a fact can appear in one or more containers, although the subject term is the common choice. The other fact terms are stored as usual. Figure 11.6 is the same as Figure 11.5, except that it shows the container expressed in the same way as the other facts.

The term holding the word `Container` is the start point for the container. The only thing that distinguishes a container from other facts is the way predicates are named. Each fact that represents one contained item is automatically given a number as a name. Unlike the numbers the example has used as placeholders for subject and object identifiers, the numbered predicates used by a container really are expressed using numbers in the RDF syntax.

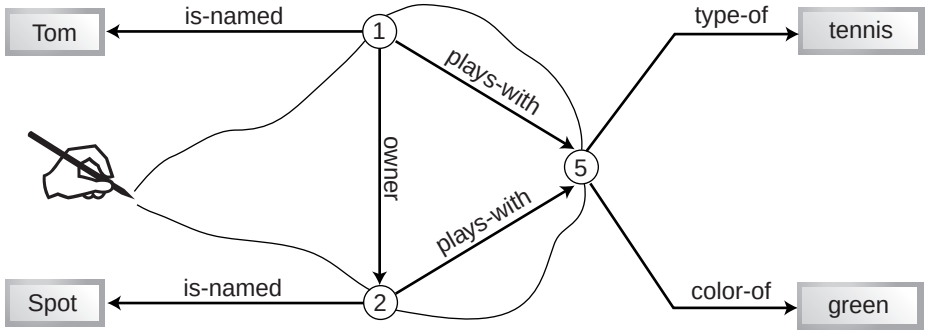


Fig. 11.5 RDF graph showing like terms connected by a line.

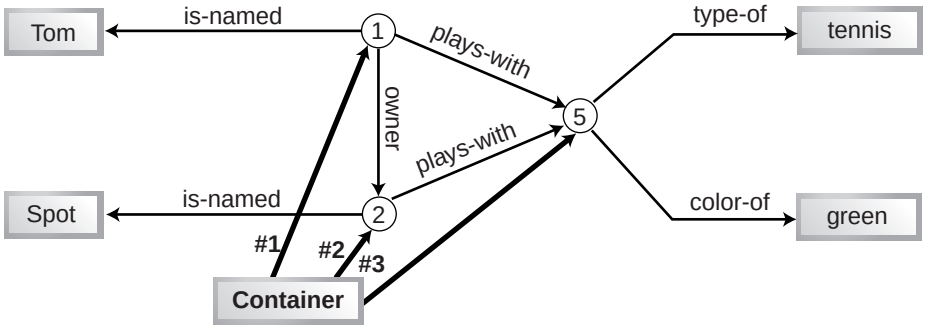


Fig. 11.6 RDF graph showing like terms connected by a container.

Containers are simple structural mechanisms. They also support querying a fact set. An application programmer can use a container as a partial index, or as an iterator, or as a relational view. A given RDF document can have unlimited containers.

To summarize, facts can be stored as a simple set of triples or as a complex graph that revolves around identifiers. In between is the partial structure of a container, which is like drawing a travel route on a map. A set of facts is called a fact store. Complex fact stores are called knowledge bases, just as a data collection is a database. Simple RDF documents are fact stores; RDF documents that include RDF schema tags are knowledge bases.

11.3.4 Facts About Facts

Facts can describe other facts. Knowing this is sometimes useful, but only for special applications. Most of the time it's better to ignore it. Some of the more standard effects are described here. If your brain is full by now, skip this section, take a break, and resume with the next section. Otherwise, onward.

In the boy and dog example, many extra facts that could be stated explicitly are implied by the stated facts. Some of these extra facts are the result of design observations, and some are almost automatically true. The following examples are based on this single fact:

```
<- 1, is-named, Tom ->
```

One set of design observations that yields extra facts is type information. A programmer developing a fact set can choose to add types. For example, these facts might follow from the preceding single fact:

```
<- 1, is-type, integer ->
<- Tom, is-type, string ->
```

These facts give a type to the subject and object data items in the earlier fact. These facts provide extended information about another fact. They are equivalent to a data dictionary or a schema. Unlike database and OO design, these facts are not separate in any sense to other "plain" facts. They can be stored with any other facts. Such facts might be used by a Mozilla/RDF application that is a data modeling tool, which needs to manipulate schemas.

To repeat the remarks at the start of this chapter, many industries call this stated information *metadata*. The term *metadata* is supposed to help our minds separate the data the URL *represents* from the data that is *about* the URL, as in the example. Unfortunately, if a programmer writes code to process a file of so-called metadata, the only interesting information is the metadata itself—the contents of that file. To the programmer, the metadata is the data. This is a very confusing state of affairs, not only because one person's metadata is another person's data but also because RDF is about facts, and not really about plain data at all.

In short, the term *metadata* is overused. To a programmer, the only thing in RDF that should be considered as metadata is type information. Everything else is just plain data or, preferably, plain facts. No facts are special; no facts have any special "meta" status.

A second set of facts are these three, which are automatically true:

```
<- example-fact, subject, 1 ->
<- example-fact, predicate, is-named ->
<- example-fact, object, Tom ->
```

Here, `example-fact` means the preceding example fact. The `subject` predicate says that 1 is the subject for that fact. The `predicate` predicate says that `is-named` is the predicate for that example fact. In other words, these facts are facts about the example fact. This process is called *reification* (from *reify*), which loosely means "exposing the facts about this fact so that we can work on it."

Reification is similar to extracting metadata, but far messier. To see why, consider this problem. Earlier it was said that all stated facts are true, and all unstated facts are false. Does an example fact without the first reification fact above have a subject or not? If the reification fact doesn't exist, then the

answer should be false. But the example fact exists, so assumptions about its composition are surely true. There is an apparent contradiction (but the logic is flawed). Such thinking gets a practical person nowhere. Neither does asking whether the reification facts should themselves be reified. These kinds of facts might be used by a Mozilla/RDF application interested in textual analysis or natural language processing, but that's all.

A further set of design observations that yields extra facts is the matter of names. A programmer developing a fact set might choose to name features of the fact set. Using the same single example fact, one might name the subject and object data items, perhaps if data are to be presented in columns with column headings, or if some King is handing out titles to his nobles.

```
<- 1, is-named, person-id ->
<- Tom, is-named, person-name ->
<- example-fact, is-named, person-definition ->
```

This is also very messy. The original fact states that there is a person named Tom. But according to the first fact here, the string "Tom" is named (has the type) `person-name`, and the name of the string "1" used to recognize Tom is `person-id`. Also, the whole initial fact that states "a person exists named Tom with identity 1," is itself named `person-definition`. To summarize, Tom's name is `"Tom"`, `"Tom"`'s name is `person-name`, and "Tom is named `"Tom"`" is named `person-definition`. This subtle thinking is not of practical use to programmers.

Next, note that most aspects of most data modeling systems can be expressed as predicate facts. For example, you can use the following predicates to state facts describing an OO model:

```
is-a has-a uses-a instance-of
```

These other predicates might instead assist a relational model:

```
has-key has-foreign-key one-to-many one-to-one has-optional
```

Using such predicates, it is possible to add complex layers of meaning (like object-orientedness) over the top of the basic fact system. This is very messy and not for beginners. RDF includes a few features like this itself, but for ordinary use they should be avoided. Use of these predicates might appear to be metadata, but metadata should use "is-a" (and other relationships) as an object term, not as a predicate term. A correct metadata example fact is "*UML arrow 5* has-feature *is-a*." A messier, layered solution is just to state directly "*UML entity 3* is-a *UML entity 2*."

Finally, new facts can be derived from existing facts. In the boy and dog example, this fact might be implied because of the other facts stated:

```
<- Tom, plays-with, Spot ->
```

Whether this fact is implied depends on what assumptions are made about the safety of leaping to such conclusions. After all, Tom and Spot are playing with

the same ball, so they're probably playing with each other. Such derived facts are the responsibility of deductive systems, and neither RDF nor Mozilla does any real deduction.

When facts are collected together for processing, they are held in a *fact store*. A fact store is the fact equivalent of a database, and is usually memory-based rather than disk-based.

To summarize this tutorial on facts, predicate triples are a useful subset of simple tuples. Facts are expressed in such triples. RDF documents contain facts. Facts are statements, not data, and when stated are true. Complex aspects of facts are easily revealed, but they are not that useful. Using facts as information about other, named, nonfact information (like URLs) is normal. Using facts as information about other facts should wait until simpler uses are mastered.

11.3.5 Queries, Filters, and Ground Facts

Storing facts in a fact store is useless if you can't get them out. Programmers need a way of extracting important facts and ignoring the rest.

RDF documents are XML content and therefore can be navigated by hand, or queried. The simplest way to navigate XML by hand is to use the DOM standards. The simplest way to query an XML document is to use a search method like `getElementById()`, or something fancier like XML Query or XPath. None of these approaches is used for RDF.

Instead, RDF documents are read or produced as a stream of facts. A programmer would rather receive only the content that is needed, not everything under the sun. For RDF, that means only the facts that are needed. To restrict the supplied stream of facts, a matching process is required, one that throws away irrelevant information. Such a process is a kind of query or filtering system, like SQL or `grep(1)`.

Query systems aren't explored in this chapter, but they rely on the concept of a *ground fact*. A ground fact (sometimes called a concrete fact) is a fact that is fully known. All the facts discussed so far in this chapter are ground facts.

As an example, consider this statement: "Tom is the owner of Spot." It is easy to identify a subject (Tom), object (Spot), and predicate (owner). Everything is known about the statement, and so the statement is said to be *ground*. This means it has a solid basis. An equivalent ground fact can be written down right away:

```
<- Tom, owner, Spot ->
```

By comparison, a trickier statement might be: "Tom owns a dog." The subject, object, and predicate can still be identified so the statement is ground. An equivalent fact is

```
<- Tom, owner, dog ->
```

If, in this case, you happen to know that there are many dogs, then the question "Which dog does Tom own?" is left unanswered. In that case, "Tom

owns a dog" is not ground because no object (a particular dog) can be identified. A lawyer would say, "That statement is groundless because you can't produce a specific dog that Tom owns." That is his way of saying that you are too vague to be trusted. In such a case, the best fact you can write down is

```
<- Tom, owner, ??? ->
```

The question marks just indicate that something is missing; they are not special syntax. There is very little you can do if handed such an incomplete fact. The opposite of ground is *not ground*, so this fact is not ground.

This partial-fact problem can be turned on its head. If the computer knows which dog Tom owns, but you don't, then the incomplete fact can be given to the computer to fix. The computer runs some code that compares the incomplete fact (called a *goal*) against all the available facts and returns all the facts that would ground it. This is called *unification*; in Mozilla's simple system, it's a matching process. You would find out all the dogs that Tom owns, or even all the pets that Tom owns, or possibly everything that Tom owns. How much you get back just depends on what facts are in the fact store. Mozilla's query and filtering systems do this for you.

RDF ground facts are queried or filtered by programmers using a fact or facts that are not ground and that are called a goal. RDF itself can state both ground and not ground facts, but facts that are not ground are rare and generally bad design. Mozilla-specific features are required to support goals.

To summarize, facts can be stored like data and examined with a matching system that unifies a goal against ground facts. That is all the ground facts about facts.

11.4 RDF SYNTAX

RDF's syntax is based on several standards and nonstandards.

The core standard at work is, of course, RDF. This W3C standard consists of two main parts and has evolved in two stages.

The first stage of standardization consisted of draft standards developed in 1999 and 2000. There are two main documents:

- *http://www.w3.org/TR/1999/REC-rdf-syntax-19990222.* This document is the "RDF 1.0 Model and Syntax Final Recommendation." Model just means underlying conceptual design.
- *http://www.w3.org/TR/2000/CR-rdf-schema-20000327.* This document was not finalized for several years. It provides complex schema operations on RDF tags and is different from XML Schema.

The second stage of RDF standardization consisted of expanding and completing the existing documents. These newer documents were finalized in 2003:

- *http://www.w3.org/TR/rdf-syntax-grammar/*, "RDF/XML Syntax Specification (Revised)," is an update to the preceding 1999 document.
- `http://www.w3.org/TR/rdf-schema/`, "RDF Vocabulary Description Language 1.0: RDF Schema," is a completion of the earlier RDF schema document.
- There are also several explanatory documents at `http://www.w3.org/RDF/` that analyze RDF from different points of view.

Of these five items, Mozilla implements nearly all the first item (the 1999 recommendation) and a little of the new features of the third item (the Revised recommendation).

Other standards that work closely with RDF are XML Namespaces and XML Schemas. Mozilla implements XML Namespaces and XML Schema, but the XML Schema support is not used in any way for RDF processing. Mozilla's XUL also has some syntax support for RDF.

These standards provide a set of XML tags from which facts can be constructed. Fact subjects and fact objects can be expressed as attribute values, or (for objects) as text nodes enclosed in start and end tags. The standards, however, provide only a few special-purpose predicates. All the rest of the predicates must be supplied by the application programmer. This means that names for extra XML tags and/or extra XML attributes must be defined.

These extra names form a vocabulary. Such a set of names can be specified in an XML Schema document or in an RDF document. Some existing vocabularies have well-known URLs and specifications to encourage their reuse. The most famous example is the Dublin Core, which is a set of keywords. It is used primarily by librarians and archivists for catalogues, and consists of predicates like "Title" and "Author."

Mozilla does not use the Dublin Core for its vocabulary. It does not allow XML Schema to be used to specify a vocabulary. Instead, a few vocabularies are built into the platform directly. The RDF processing that Mozilla does also allows the application programmer to make up names on the spot without any formal definition existing. So predicate names can be made up in the same way that JavaScript variable names can be made up.

11.4.1 Syntax Concepts

An RDF document is an XML document, and RDF is an application of XML. The XML Namespace identifier used for RDF in Mozilla is

```
http://www.w3.org/1999/02/22-rdf-syntax-ns#
```

RDF documents should have an `.rdf` suffix. The MIME types Mozilla recognizes for RDF are

```
text/rdf
text/xml
```

The official MIME type is not yet supported as of 1.4. That type is

```
application/rdf+xml
```

11.4.1.1 Tags The primary goal of RDF is to provide a syntax that allows facts to be specified. XML facts could be represented a number of ways. Listing 11.6 illustrates some imaginary options:

Listing 11.6 Potential XML syntax for facts.

```
<fact subject="…" predicate="…" object="…"/>

<fact>
  <subject …/>
  <predicate …/>
  <object …/>
</fact>

<subject … predicate="…" object="…"/>
```

None of the forms in Listing 11.6 is used for RDF. RDF syntax uses this form instead:

```
<fact subject="…">
  <predicate>object</predicate>
</fact>
```

This is conceptual syntax only, not actual RDF. A syntactically correct RDF fact matching the conceptual syntax is

```
<Description about="http://www.mozilla.org/">
  <NC:LastVisited>10 January 2004</NC:LastVisited>
</Description>
```

This syntax choice provides options for nesting facts inside other facts. It makes a number of syntax shortcuts possible. It mimics some technical aspects of the Web environment. Unfortunately, terminology is a real challenge for beginners. The words in this conceptual syntax are not used in RDF. Worse, the different bits of this syntax are described with RDF-specific ideas, not with the fact concepts that appear earlier in this chapter. That is a very confusing state of affairs.

RDF uses different ideas because it attempts to reuse thinking from Web technologies; RDF was originally created to address issues in the Web environment. On the one hand, this reuse does create a somewhat familiar environment for developers. On the other hand, RDF is still about facts, and no amount of clever naming can hide that. Table 11.2 compares these RDF names with fact concepts and Web terminology.

The recommended way to handle this is to wear two hats. When thinking very generally, RDF syntax is best seen as a set of facts, so use the fact con-

Table 11.2 RDF terminology

Fact concept	RDF term	Web terms borrowed from	RDF syntax
Fact	Description	Description of a document or record	<Description>, <Seq>, <Alt>, <Bag>
Subject	Resource	URL	about=, id=
Predicate	Property (and resource)	Object property, CSS property, XML attribute	user-defined
Object	Value (or resource)	Property value, attribute value	resource=, plain text

cepts. In specific examples where the set of facts forms a simple tree, it is best to see it as a hierarchy of resources and properties, so use RDF terms. Because most RDF documents are small or highly structured or both, this simpler case is easy to get away with.

We can practice this thinking on the simple `<Description>` tag stated earlier. In fact terminology, it is interpreted as follows. The `<Description>` tag defines the subject term of a fact. If that tag has any contents, that content represents the remaining terms of one or more facts. The `<NC:LastVisited>` tag is a predicate term, and the plain string "`10 January 2004`" is an object term. By comparison, in RDF terminology, the `<Description>` tag is interpreted this way. The `<Description>` tag identifies a resource. That resource may have properties, given by other tags. The `<NC:LastVisited>` tag is one such property, and it has the value "`10 January 2004`".

An advantage of the RDF terminology is that most predicate names in Mozilla's RDF are property-like. Although `color` (or any word) can be used as a predicate, `color` sounds more like a DOM or CSS property name than a predicate relationship. Even so, in RDF, the terms *predicate* and *property* are harmlessly interchangeable.

The complete list of basic RDF tags is as follows:

```
<RDF> <Description> <Seq> <Bag> <Alt> <li>
```

The last four tags are redundant and can be expressed using `<Description>`, so RDF has a very small number of tags. Although predicate tags are application-specific, RDF does predefine some predicates. These predicate tags are named

```
<Statement> <subject> <predicate> <object>
```

These tags are used for reification of facts. `<Statement>` reifies a fact. The other three reify one term each of a triple. Mozilla does not support any of these four tags.

11.4.1.2 Containers RDF supports containers. Containers are a lazy person's way of writing and collecting repetitious facts. A container consists of a `<Bag>`, `<Seq>`, or `<Alt>` tag. The container and its content together make a collection. Such a collection can be put where the object appears in a fact. Such a container looks like this:

```
<Description>
  <Bag>
    <li>object 1</li>
    <li>object 2</li>
    <li>object 3</li>
  </Bag>
</Description>
```

In a normal fact, there is a one-to-one mapping between the fact's subject and object (between resource and property value). That means one object per subject. Containers change this so that there can be a one-to-many mapping. That means at least zero objects per subject. Containers are the RDF equivalent of a list or an array. If a container is used, it is up to the application to know about it or to detect it and react appropriately.

An obvious use of a container is to track the empty seats in a theatre or plane booking system. Each seat is a resource; such a system needs to manage all the seats, whether full or empty. A list of yet-to-be allocated seats can be maintained in a container separate from the facts about the seats themselves. An example RDF fragment is shown in Listing 11.7.

Listing 11.7 Two facts specified with a single RDF `<description>` tag.

```
<Description id="seat:A1">
  <aisle>true</aisle>
</Description>
<Description id="seat:A2">
  <booked>Tim</booked>
  <aisle>false</aisle>
</Description>
<Description id="seat:A3">
  <aisle>false</aisle>
</Description>

<Description id="seat:vacancies">
  <Bag>
    <li resource="seat:A1"/>
    <li resource="seat:A3"/>
  </Bag>
</Description>
```

This example states whether each seat is an aisle seat, and if the seat is booked, it adds the name of the person booked. The `seat:` syntax is an imaginary application-specific URL scheme. Only seat A2 is booked. The `<Bag>` container holds references to the two unbooked seats. The syntax used for the `<li>` tag is one of the several shorthands that RDF supports.

So containers are also used to give programmers access to subsets of facts. A container's held items are the subjects of another set of other facts, and those other facts can be accessed by grabbing the container and looking through it. Containers can be viewed as a simple data structure for facts and as a simple navigation mechanism. In the booking system, the programmer can book a seat by first checking it is in the vacancies container, then adding a `booked` predicate to the seat's `<Description>`, and then removing it from that container.

In object-oriented terms, the container has a uses-a relationship with each collected item. The collected items are not in any way hidden inside the container.

Container tags and their collection of items can always be replaced with an equivalent set of plain facts. See the specific tags.

11.4.1.3 Identifiers The boy and dog example discussed earlier in this chapter made some effort to provided identifiers for each significant item modeled by the set of facts. This emphasis on identifiers is much stronger in RDF than it is in most XML applications. Identifiers are critical to RDF and are used two ways.

The first use of identifiers identifies whole facts. This is achieved by adding an id attribute to the tag holding a particular fact. The RDF document's URL, plus #, plus the id's value uniquely identifies that fact worldwide. According to IETF RFC 2396, such a URL represents a document fragment, not a complete resource. RDF breaks that rule—such a URL is considered a whole resource even though it is part of a larger document. RDF documents therefore can be viewed as a bundle of resources—a resource collection. Each resource is a single fact. This is very different from HTML. In HTML, an id is just a single tag, and `<A>` without HREF just marks progress points inside a single document. An example of an RDF identifier is

```
<Description ID="printEnabled" … />
```

This RDF file might store information about the printing subsystem, and the `ID` makes this specific fact locatable via a meaningful name.

The second use of RDF identifiers is to replace local literals in a given RDF file. The example facts so far noted generally work with information stored right in that fact. For example, "tennis" (for tennis ball) is a piece of information stored directly in one fact in Listing 11.5.

Such immediacy does not need to be the case. For example, several facts have been considered about Tom. Tom's physical body does not appear in the fact—just a number (1) used to identify him. It's understood that this number stands in for him. RDF provides a better identifier for Tom than a simple number; a URL can be used instead. This URL represents Tom just as a simple number can. Perhaps it's a `mailto:` address or a URL that retrieves a personnel record. Any facts whose subject is a string matching that URL are about Tom. In Web terms, Tom is a resource, and a URL locates him.

RDF goes further, though. The fact's object can be a URL, just as the subject can. For subject and object, this is straightforward. Tom's dog has a URL and so does Tom's ball.

More subtly, RDF allows the fact predicate/property to have a URL. The URL of a predicate is just a pointer to a location where that predicate is fully described. The URL is the predicate's identifier and stands in for the actual predicate, which might be held by some regulatory body, a standards organization, or even an MIS server. Actually, RDF insists that the predicate part of a fact be represented with an id, and that id must be a URL. For readability, the URL of a predicate usually contains a word that expresses its general meaning, like *www.example.com/#Owner*.

Facts can therefore be expressed entirely in URL identifiers if necessary. This is really no more than using pointers to the terms the facts are composed of. This kind of RDF document is one source of confusion for beginners because such a file seems to have some direct interaction with the Web. The URLs in such a document are no more significant than strings that hold a street address. There is no automatic Web navigation or other magic, unless the software processing the RDF file adds it. Mozilla does not add any such processing. RDF documents do not need access to the Web. URLs are just data items in RDF.

There is one complexity, however. Any URL in an RDF document might be a fact in another RDF file. Facts in documents can therefore refer to each other. In ordinary applications, this is a bad idea because it creates a spaghetti mess of fact dependencies. In specialized cases, like authentication protocols, it might be more necessary. It makes sense for facts in one document to contain URLs to metadata facts or schema items in some other, authoritative resource, but that's about as far as it should go.

These stand-in URLs can have the more general syntax of a URI (Universal Resource Identifier). A URI is a URL or URN (Universal Resource Name). The benefits of this broader possibility are explored shortly. The example of a URL has only been used because it is familiar.

It is possible for a fact to have a missing term in an RDF document. This happens when a container or `<Description>` tag with no identifier is used. The container tag represents an extra subject in the graph of facts and without some identification; it is anonymous. An RDF document with no anonymous facts is ground. For query purposes, it is highly recommended that all RDF documents be ground. Therefore, container tags should not be left without identifiers.

11.4.2 `<RDF>`

The `<RDF>` tag is the container tag for a whole RDF document and is required. Because RDF documents contain several namespaces, it is common to always use a namespace prefix for RDF. Thus,

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

This tag has no special attributes of its own. The only things that appear in it are XML namespace declarations, which add a vocabulary (additional tags) to the RDF document. These namespace declarations assist RDF in the same way the DTDs assist HTML. Table 11.3 shows all the namespaces used in the Mozilla Platform.

Table 11.3 XML Namespaces Used for RDF Vocabulary

URL of Namespace	xmlns Prefix	Defined Where?	Use
http://www.w3.org/1999/02/22-rdf-syntax-ns#	RDF	www.w3.org	Core RDF support
http://home.netscape.com/WEB-rdf#	Web	Hard-coded	Bookmarks and timestamps
http://www.mozilla.org/rdf/chrome#	Chrome	Hard-coded	Managing chrome packages and overlays
http://home.netscape.com/NC-rdf#	nc	Hard-coded	General purpose
http://www.mozilla.org/LDAPATTR-rdf#	ldapattr	JavaScript, based on LDAP properties	email LDAP support in Mail &News client
http://www.mozilla.org/inspector#	ins	JavaScript	DOM Inspector

Except for the first entry, none of these URLs exists as documents. The URLs containing "netscape" are a legacy of Netscape Communicator 4.x. Prefixes are suggestions based on existing conventions. Of these prefixes, `web`, `chrome`, and `nc` are used the most in Mozilla. To use a namespace, you must know the keywords it supplies. Those keywords are remarked on under "Predicate Tags." An application programmer is free to add namespaces as needed because namespaces are just arbitrary `xmlns` strings.

The contents of the `<RDF>` tag is a set of child tags. Each of those child tags must be either `<Description>` or one of the collection tags `<Seq>`, `<Bag>`, or `<Alt>`.

11.4.3 <Description>

The `<Description>` tag is the heart of RDF. A `<Description>` tag represents one or more facts and can contain zero or more child tags. Each child tag is a predicate (RDF property). Each child tag implies one complete fact, with the `<Description>` tag as subject in each case. An example of two facts is layed out in Listing 11.8.

Listing 11.8 Two facts specified with a single RDF <description> tag.

```
<fact subject="…">
  <property1 …>object1</property1>
  <property2 …>object2</property2>
</fact>
```

This is pseudo-code, not plain RDF. In this example, *property* has been written instead of predicate, since they are interchangeable. The example makes it clear why RDF uses the term property: the fact appears to hold two properties for its subject. In reality this syntax just states two different facts with the same subject. It just saves some typing.

Because the role of the `<fact>` pseudo-code is taken by the RDF `<Description>` tag, that tag is therefore like a container. It is not like other RDF containers because it has no special semantics at all and contains predicates/properties, not subjects or objects.

`<Description>` has the following special attributes.

```
ID about type
```

Every `<Description>` tag should have an `ID` or an `about` attribute. If both are missing, the subject of the stated fact will be anonymous. It will not be visible to the rest of the world, and inside the document it represents a term that is not ground.

The `ID` attribute has a name as its value; this name is appended to the RDF document's URL to create a unique URL for the whole fact. That constructed URL is also the URL of the stated fact's subject. Applying an `ID` to `<Description>` only makes sense when the `<Description>` tag has exactly one property. A fact subject stated with `ID` is visible to the whole Web.

The `about` attribute specifies the subject of the stated fact. It takes a full URI as its value. If the `about` attribute is used instead of the `ID` attribute, the whole fact has no URL of its own. A fact subject stated with `about` is not visible to the Web.

The `type` attribute records the type of the fact's object (its value). The value/object normally appears as XML content between the property start and end tags. The `type` attribute states what kind of thing the value/object is. If it is present, its value should be a URI. Mozilla does nothing with the `type` except store and retrieve it. It is not integrated with any schema definition. The `type` attribute has one additional role. It is also a property or predicate. Its use as shown here is really a shorthand notation, as described below.

Mozilla does not support these attributes:

```
aboutEach aboutEachItem bagID
```

All but `bagID` are deprecated in the most recent RDF standards. `bagID` is used for reification of RDF facts. Mozilla does not do this.

11.4.3.1 Shorthand Notation The <Description> tag and its contents can be reduced to a single <Description/> tag with no contents. This can be done by treating the property/predicate and value/object as an XML attribute.

This RDF fragment is a single fact stating that Tom owns Spot:

```
<Description about="www.test.com/#Tom">
  <ns:Owns>Spot</ns:Owner>
</Description>
```

The subject and predicate are defined by URLs; the object is defined by a literal. The predicate belongs to a declared namespace with ns as prefix, so the full URL of the predicate is not obvious. This example can be shortened to a single tag:

```
<Description about="www.test.com/#Tom" ns:Owns="Spot"/>
```

Note the namespace prefix before the Owns attribute name. That is standard XML but is not often seen in practical examples. This shorthand can be used only if the value/object is a literal. It does not work if the value/object is a URI.

11.4.4 Predicate/Property Tags

Predicate or property tags must be supplied by the application developer. The easiest way to do this is to find an existing set of tags useful for your purpose. For informal use, it is possible to make up tags as you go, but properly defining the namespace that you use is a sign of good design and ensures that the purpose of your RDF content is clear.

RDF supplies XML attributes that can be added to property tags. Just as the observes attribute turns any XUL tag into an observer, so too do RDF custom properties affect application-defined property tags. The list of such special attributes is

```
ID parseType
```

The ID attribute has the same purpose for the predicate tag that it has for the <Description> tag. It is used when the parent <Description> tag has more than one predicate tag, and therefore represents more than one fact. The ID on a specific predicate tag identifies the matching fact globally.

The parseType attribute provides a hint to the RDF parser. It is different from the type predicate, discussed in the next topic. It states how the XML text string holding the value/object should be interpreted. It can be set to one of these four values:

```
Literal Resource Integer Date
```

Literal means the value is an arbitrary string; it is the default. Resource means that the value is a URI. Those two options are standard. Integer and Date are Mozilla enhancements. Integer reads the string as a 32-bit signed

integer. `Date` reads the string to be a date. Such a date can be in any of several formats, but Mozilla's support is incomplete. The safest format is to use the UTC output of the UNIX `date(1)` command, and change "UTC" in the resultant string to "UT" or "GMT." The `Date` option does not accommodate Unicode characters, only ASCII ones.

11.4.4.1 Existing Predicates RDF itself provides the `type` predicate. This predicate matches the `type` attribute of the `<Description>` tag. Its use in that tag is actually shorthand for

```
<rdf:type>value</rdf:type>
```

where `rdf` is the prefix used for the RDF namespace. Because `type` is a predicate, it can be applied to all fact subjects. Such a use can extend the basic type system of RDF, either with application-specified facts, or with RDF Schema, if that were implemented. It is not much used in Mozilla applications.

The namespaces noted in Table 11.3 also provide sets of predicates. Because predicates are ultimately just data items, these names do not "do" anything by themselves.

In Chapter 9, Commands, we noted that many command names exist in the Mozilla Platform, but those commands are all tied to specific application code, such as the Navigator, Messenger, and Composer clients. The same is true of predicates. Many exist, but their uses are all tied to specific application code.

These predicates are not centrally listed anywhere, not even in the source code. Since it is easy to make new predicates up (just add them to the `.rdf` and to the code), it is a matter of curiosity to track down the ones that Mozilla uses. Some examples of their use appear in Chapter 12, Overlays and Chrome.

A new Mozilla application that uses RDF should have a formal data model, and/or a data dictionary that states the predicates used in the application. The application programmer might look to the existing Mozilla predicates as a source of inspiration or to ensure the new application is similar, but there is no need to use the same names.

The only time predicate names must be followed exactly is when RDF files in existing formats are to be created, like `mimeTypes.rdf`. In such cases, it is enough to create a sample file using the Classic Browser, and then to examine the predicates generated. A more thorough alternative is to look at the source code.

11.4.4.2 Shorthand Notation The RDF `resource` attribute can be added to predicate tags. It works like the `about` attribute on the `<Description>` tag, except that it specifies the value/object of the fact. When the `resource` attribute is added, no XML content is required for the object, and the object must be a URI, not a literal. An unshortened example is

```
<ns:Owns parseType="Resource">www.test.com/#Spot</ns:Owns>
```

and the equivalent shortened form is

```
<ns:Owns rdf:resource="www.test.com/#Spot"/>
```

This use of the `resource` attribute is straightforward. The `resource` attribute has a further use which is more complicated.

In the simple case, the object/value specified with the resource attribute participates only in one fact—the current one. The URI of that object might, however, be specified as a subject in another fact. In that case, the URI's resource has two roles: It is both subject and object. A very confusing shorthand syntax rule says this: If XML attribute-value pairs appear in the predicate tag, and the resource attribute is also specified, then those attribute-value pairs act as though they appeared in the `<Description>` tag where the URI is a subject.

This means that the final shorthand technique noted for the `<Description>` tag can also be used inside predicate/property tags. However, another `<Description>` tag, elsewhere in the document, is affected as a result.

This is messy and complex, and not worth exploring unless your application is ambitious. It is designed to reduce the total number of tags required in a set of nested tags. In theory, this makes the RDF document more human-readable, but that is debateable. For more on this subject, read section 2.2.2 of the original RDF standard.

11.4.5 `<Seq>`, `<Bag>`, `<Alt>`, and `<li>`

`<Seq>`, `<Bag>`, and `<Alt>` are RDF's three container tags.

- `<Seq>` is a sequence or ordered list. The contained items are ordered. One possible use is history information, like a list of commands recently executed.
- `<Bag>` is a simple collection. There are no restrictions on contained items.
- `<Alt>` stands for alternative. It is a simple collection with no restrictions except that the contained items are all considered equivalent to each other in some application-specific way. One possible use of `<Alt>` is support for a message stated in multiple languages.

These container tags provide a way to organize objects and subjects into groups and to write many similar facts in a compact notation. All three containers can contain duplicate terms.

Containers contain items. Each item in a container is enclosed in an RDF `<li>` tag, just as each item in an HTML `<UL>` or `<OL>` list is enclosed in an HTML `<LI>` tag. Each item in a container is an object. Therefore, `<li>` is a delimiter for an object. Because `<Description>` tags or container tags can substitute for objects, containers can be nested. Listing 11.9 shows a single container.

Listing 11.9 Example of an RDF container.

```
<Description about="www.example.com/#Tom">
  <ns:Owns>
    <Bag ID="Dogs">
      <li>Spot</li>
      <li>Fido</li>
      <li>Cerberus</li>
    </Bag>
  </ns:Owns>
</Description>
```

Tom owns Spot; Tom owns Fido; Tom owns Cerebus. That should be three facts. Containers are easy to write down. Unfortunately, containers are slightly ugly to implement. The equivalent facts are shown in Listing 11.10.

Listing 11.10 Equivalent facts for ownership of three dogs.

```
<- "www.example.com/#Tom", ns:Owns, "Dogs" ->
<- "Dogs", rdf:_1, "Spot" ->
<- "Dogs", rdf:_2, "Fido" ->
<- "Dogs", rdf:_3, "Cerberus" ->
```

RDF can only state facts, so containers must be implemented with facts. This is accomplished by manufacturing an extra term to stand for the container itself. The `<Description>`'s fact has this subject as object. Whole facts are also manufactured. In turn, these facts tie the manufactured subject term to each of the container items. These new facts are therefore one-to-one, as all facts really are. These manufactured facts and terms are automatically added to the fact store created when the RDF document is read.

Two things are missing from this manufacturing strategy: The manufactured subject needs an ID, or at least a literal value, and the new facts need predicate or property names.

The first gap is filled easily: The container tag must be supplied with an ID or an about attribute by the document creator. If it isn't, it remains anonymous, and the RDF document is not ground. Anonymity should be avoided because it prevents Mozilla templates from working.

The second gap is filled when the RDF document is interpreted by software. If that software meets the RDF standard, then it will generate predicates for these new facts as it goes. These predicates will be named _1, _2, _3, and so forth. They exist in the RDF namespace, so they will be called `rdf:_1`, `rdf:_2`, and so forth, assuming `rdf` is the prefix selected for that namespace. That is the origin of those names in Listing 11.10.

Listing 11.11 shows facts equivalent to Listing 11.9 after that RDF fragment has been read. The RDF standard has some diagrams showing graphs for container-style facts and is worth glancing at.

Listing 11.11 Equivalent RDF facts to a container of three items.

```
<Description about="www.test.com/#Tom">
  <ns:Owns resource="Dogs"/>
</Description>
<Description about="Dogs" rdf:_1="Spot"/>
<Description about="Dogs" rdf:_2="Fido"/>
<Description about="Dogs" rdf:_3="Cerberus/>
```

Shorthand can be used for the `<rdf:_1>` predicate tag because its object value is a literal. Full shorthand cannot be used for the id of `Dogs` in the first fact because it is a URL fragment, not a literal. Whether you do or don't use containers is a design choice, but they are somewhat neater than the equivalent `<Description>` tags.

Because the container items and the `<Description>` subject are sorted into separate facts, they are not directly connected. An application cannot find a single fact in Listing 11.11 that states that Tom owns Spot. This means that an application looking for such a fact must either know the structure of the fact graph and navigate through it, including knowing where the containers are, or make an extensive analysis of the content. The former strategy is obviously far more efficient.

The following attribute applies to container tags:

```
ID type
```

The `ID` attribute has the same purpose as for `<Description>`.

The `type` attribute is not set by the document creator. It is set automatically to the name of the container tag (e.g., `rdf:Bag`), which is the type of that tag. This attribute is a predicate and value pair for the `<Description>` subject and is, thus, a fact in its own right. That extra fact is stated not with type as predicate, but with the special value `instanceOf`. For the example of three dogs, that fact is

```
<- "Dogs",
   http://www.w3.org/1999/02/22-rdf-syntax-ns#instanceOf,
   http://www.w3.org/1999/02/22-rdf-syntax-ns#Bag
->
```

That fact can be used by the application programmer to detect the existence and kind of the container tag. In Mozilla, such direct detection is not usually necessary because the platform supplies utility objects that do the job for you.

Container tags can also use the general property = value shorthand that applies to `<Description>` tags. The following attributes apply to `<li>` tags:

```
parseType resource
```

These attributes act the same as they do for predicate/property tags, and support the same shorthand forms.

That concludes the discussion on RDF syntax.

11.5 RDF EXAMPLES

Several examples are presented to show how the syntax and concepts work together.

11.5.1 A URL Example: The Download Manager

Classic Mozilla's Download Manager presents a clean example of an RDF document that manages Web-based resources. The Download Manager is available only in version 1.2.1 and greater. It tracks progress and completion of downloaded files. It is turned on with a preference under Edit | Preferences | Navigator | Downloads.

The Download Manager consists of a single XUL window and an RDF file. The window appears when saving a URL to the local file system, perhaps with File | Save Page As .. from the Navigator menu system. It can also be opened directly with Tools | Download Manager. The RDF file is called `downloads.rdf` and is stored in the user's profile directory. The code for the Download Manager is in the chrome file `comm.jar`. The GUI is implemented with a XUL `<tree>` tag.

To see this at work, open the Download Manager and remove all listed items by selecting them and clicking the "Remove From List" button. Using a text editor, open the `downloads.rdf` file. It contains nothing but namespace declarations and an empty collection which is a `<Seq>` tag. Listing 11.12 shows this file.

Listing 11.12 "Empty" `downloads.rdf` file.

```
<?xml version="1.0"?>
<RDF:RDF
  xmlns:NC="http://home.netscape.com/NC-rdf#"
  xmlns:RDF="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
>
  <RDF:Seq about="NC:DownloadsRoot">
  </RDF:Seq>
</RDF:RDF>
```

Next, view any remote Web page such as *www.mozilla.org*. Save the page to a local file. Using a text editor, reopen the `downloads.rdf` file when the save operation is complete. A `<Description>` tag and contents, and an `<li>` collection item have been added. The `<Description>` tag states eight facts (spot them) about the locally downloaded file. The `<li>` item states a further fact: The fact subject that is the downloaded file is also an object in the sequence collection. This sequence is used to find all the files recorded in the document.

After a single file has been downloaded, the RDF appears as in Listing 11.13.

Listing 11.13 `downloads.rdf` file after a single complete download.

```
<?xml version="1.0"?>
<RDF:RDF
  xmlns:NC="http://home.netscape.com/NC-rdf#"
  xmlns:RDF="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
>
  <RDF:Seq about="NC:DownloadsRoot">
    <RDF:li resource="C:\tmp\test_save.html"/>
  </RDF:Seq>
  <RDF:Description about="C:\tmp\test_save.html"
                   NC:Name="test_save.html"
                   NC:ProgressMode="none"
                   NC:StatusText="Finished"
                   NC:Transferred="1KB of  1KB">
    <NC:URL resource="http://www.mozilla.org/"/>
    <NC:File resource="C:\tmp\test_save.html"/>
    <NC:DownloadState NC:parseType="Integer">1</NC:DownloadState>
    <NC:ProgressPercent NC:parseType="Integer">100</NC:ProgressPercent>
  </RDF:Description>
</RDF:RDF>
```

If you are using Microsoft Windows, don't be confused by path names prefixed with `C:` (or any drive letter). This is just a variation on URL syntax invented by Microsoft that Mozilla supports. Such things are equivalent to a `file:///C|/` prefix and are still URLs.

Try viewing and saving any Web page, and then deleting entries with the Download Manager. It is easy to see how the RDF data file and XUL window are coordinated. Shut down the Download Manager and carefully hand-edit the `downloads.rdf` file so that one sequence item and the matching `<Description>` is removed. Restart the Download Manager to see the effect.

When downloading Internet files to local disk, there is sometimes a long pause before the FilePicker dialog box appears. This is a bug and occurs when the Download Manager's RDF file has grown large. Delete or empty the file to improve response times.

The Download Manager could be implemented without using RDF. RDF is used because it means that only a small amount of code is needed. The extensive RDF infrastructure inside the Mozilla Platform makes retrieving and storing information in RDF format easy.

11.5.2 Using URNs for Plain Data

The whole point of using RDF is to get some perhaps fact-driven processing done in an application. Although URLs have their uses, most applications work on traditional data. Traditional data are also expressible in RDF.

Why use RDF for traditional data? The RDF infrastructure in Mozilla allows data stored in RDF to be pooled and dynamically updated. This pool of data can be used from several points in an application at once, or even from

several different applications. The RDF infrastructure also provides extensive automatic parsing and management of RDF content. The application programmer's script does not have to perform any low-level operations. Most importantly, RDF is the basis for Mozilla's XUL Template system, which provides automated display of RDF content.

11.5.2.1 Summary of Useable Types in Mozilla RDF To complete previous remarks on RDF types, the available types include the following:

- **Literal types**. `Literal`, `Resource`, `Integer`, `Date`, `Blob`. `XMLLiteral` is not supported.
- **Fact component types**. `Property`, `Bag`, `Seq`, `Alt`. List is not supported.
- **Fact types**. `Statement` is not supported.
- **Reification types**. Subject, predicate, and object are not supported.

The `Blob` type holds an array of binary data. It cannot be specified in an RDF document or from JavaScript. It is a Mozilla extension that can only be used from C/C++ code. One use of the `Blob` type occurs in the Classic Mail & News client. There, attachments are treated as fact objects of type `Blob`.

Types that are supported in RDF are specified with the `<Description>` tag's type attribute, which is the same as the built-in `rdf:type` predicate.

None of the type features of RDF Schema are supported. In general, an application programmer should store content in RDF as a plain XML string and do any necessary conversion in application code.

11.5.2.2 URNs When identifiers were discussed earlier, all the examples used URLs. URLs are useful if the stored facts are about Web or Internet resources. For more ordinary applications that only work with plain data, URNs are recommended instead of URLs.

Identifiers in RDF are actually URIs, not URLs. Recall that a URI (a Uniform Resource Identifier) is either a URL (a Uniform Resource Locator) or a URN (a Uniform Resource Name).

A URL ties a resource to a particular access point and an access method such as HTTP. The resource at that point might change over time, like a Web page. A URN, on the other hand, is just a name for an unchangeable thing that exists as a concept. If that concept has a real-world equivalent, that is at most convenient.

To a programmer, a URN is a variable name for a constant piece of data. Although URNs are supposed to be globally unique, you can make up your own provided your application is isolated from the rest of the world. This is like making up your own domain name or your own IP address. See IETF RFC 2611 for the URN registration body. In Mozilla, URNs are short, which suits programmers. Their syntax is

```
urn:{namespace}:{name}
```

where {namespace} is a string, and {name} is a string. {namespace} cannot be "urn" and cannot contain colons (:). {name} can be anything, including colons, which can be initially confusing. RFC 2141 describes the exact syntax, which allows for an ASCII, alphanumeric, case-insensitive namespace and an ASCII-punctuated case-insensitive name. {name} is optional in Mozilla. There are no "relative" URNs. Some punctuation marks are banned. See also RFC 2379. Two examples of URNs are

```
urn:myapp:runstate
urn:myapp:perfdialog:response
```

In the second URN, the {name} part is constructed to look like a second namespace plus a final name. In standards terms, this is just an illusion, but in programming terms it is a useful way to divide up a large number of URNs into subcategories. This kind of subdivision can have as many levels as necessary.

URNs are useful in RDF. They turn a verbose, Web-specific document with a worldwide focus into a general-purpose document about local information. An example of a URN fact is

```
<Description about="urn:mozilla:skin:modern/1.0"
     chrome:author="mozilla.org"/>
```

Here, chrome: is an XML namespace, "mozilla.org" is a literal, and "skin:modern/1.0" is the name of the thing the URN describes. Although this is a fact, not data, such a simple arrangement can be compared to a line of JavaScript code:

```
mozilla["skin:modern/1.0"].author = "mozilla.org";
```

It can also be compared to a more verbose line like this:

```
urn.mozilla.skin["modern/1.0"].chrome.author = "mozilla.org";
```

Of course, this simple code does not match all the behavior that facts have; it just reflects one possible use of a particular fact.

There is a URL scheme called data URLs. It is documented in IETF RFC 2397. This scheme offers a tempting way to pretend that raw data are also a URL. It might seem that this is a way to name a resource after its own contents. Data URLs are not useful in RDF, except possibly as objects, because they are not unique identifiers. Avoid them.

11.5.3 A URN Example: MIME Types

MIME types (and file suffixes) are used in the Classic Browser to handle foreign documents to the correct application for viewing. An example is a Microsoft Word .DOC document, which on Microsoft Windows is handed to winword.exe for display. MIME types are configured under Edit | Preferences | Navigator | Helper Applications.

As for the Download Manager, the MIME types system has a GUI and a file component. The GUI component is a panel in the Preferences dialog box; the file component is the file `mimeTypes.rdf` stored in the Classic Browser's profile directory. Code for this system is in `comm.jar` in the chrome, in files prefixed `pref-application`.

This subsystem of Mozilla can be exercised in a similar way to the Download Manager. The significant difference is that the RDF data model uses URNs instead of URLs. That data model is a set of facts, whose subjects are a hierarchy of URNs. An example of that hierarchy, for a configuration of two types only, is shown in Listing 11.14.

Listing 11.14 Hierarchy of URNs for a two-type MIME configuration.

```
urn:mimetypes
urn:mimetypes:root
urn:mimetypes:text/plain
urn:mimetypes:application/octet-stream
urn:mimetypes:handler:text/plain
urn:mimetypes:handler:application/octet-stream
urn:mimetypes:externalApplication:text/plain
urn:mimetypes:externalApplication:application/octet-stream
```

Because these URNs are just names, their apparent hierarchy has no technical meaning. It just serves to tell the reader that something is going on. There is also real hierarchy at work, one that is made of facts. If you draw the RDF graph for a `mimeTypes.rdf` file (try that), you will see that the facts make up a simple hierarchy, with a little bit of cross-referencing. This hierarchy is very much like the `window.navigator.mimeTypes` DOM extension used by Web developers. In fact, that array is populated from the `mimeTypes.rdf` data.

As for the Download Manager, a `<Seq>` container captures the full set of MIME types in an easy-to-retrieve structure. If multiple Classic Browser windows are open, and content of several special types is downloaded, then all those windows will use a single pool of RDF MIME type data to determine how to display that content.

11.5.4 RDF Application Areas

Mozilla uses RDF as an implementation technology for a number of features, but those features could have been created without it. Why does RDF exist, really? The answer is that there are some application-level problems for which RDF is a supposed solution.

According to the RDF specification, the primary application use of RDF is library catalogs. Such catalogs need a formal and flexible data model that everyone agrees on because such catalogs are increasingly Web-enabled and interconnected.

The most visible result of this application area is the Dublin Core. This is a data model broadly agreed to by the library industry and suitable for storing information about any kind of publication, including Web pages. In RDF terms, it is a list of predicate names that should be used for this type of application. See *www.purl.org* (that's P-URL not Perl).

A second use of RDF, according to the RDF specification, is Content Management. This is the process of adding review-generated information to content so that management decisions can be made about it. Simple examples of review information are secrecy ratings and maturity ratings. RDF is an obvious way to attach such information to content URLs via facts. A software system can then choose to supply or not supply the content, depending on the clearance or age of the reader.

In practice, there are many mechanisms for managing Web content, and RDF is not yet a clear winner, or even a major player.

A more successful use of RDF is to maintain information hierarchies. The volunteer-run Web catalog *www.dmoz.org* stores its entire subject hierarchy, including individual URL entries, as a single large RDF file.

RDF has not yet broken out as a vital piece of Internet infrastructure; at the moment it is just a useful tool. When developing Mozilla applications, consider RDF in that light.

That concludes the discussion of RDF examples.

11.6 HANDS ON: NOTETAKER: DATA MODELS

This "Hands On" session gives you an example of the modeling process required to create a set of RDF facts.

We've experimented extensively with GUI elements of the NoteTaker tool, and with the scripting environment, but now its time to go back to the design phase. We have no clear statement yet what data NoteTaker manipulates. We choose RDF as the final storage format for that data. We need a data model that tells us what RDF facts are needed. Facts are the modeling language we'll use.

In Mozilla, RDF is strongest when it is used inside the platform's own install area, although it can be passed across the Internet as well. It is also strong when the amount of data is small. Those strengths suit us because the NoteTaker tool will store the few notes that the user creates in their Mozilla user profile, on the local disk. No Internet access will be required.

The choice of facts as a modeling language is not a casual choice. We could as easily have chosen an object-oriented approach, or a relational approach like SQL. But facts have one clear advantage over those other systems: They map directly to RDF syntax. UML is good for objects; Entity Relationship Modeling is good for relational databases; facts are good for RDF.

The process we use to find, build, and implement the model follows.

1. Write down everything about the data.
2. Pick out the significant words (terms).
3. Construct some useful facts.
4. Ensure that those facts are about resources.
5. Consider how those facts need to be accessed.
6. Arrange the facts into structures that support that access.
7. Translate the results into pure RDF.

Data modeling usually requires several drafts, but here we'll get it mostly right to start with, and highlight a few common pitfalls and blind alleys as we go.

The NoteTaker data model is very simple for us to state in words. Each note is associated with one URL. It has a short and a long description (summary and details). It has a position on the screen (top, left, width, height). It also has keywords associated with it.

Keywords are just descriptive words added by the user that classify the URL to which the note belongs. If two keywords appear on the same note, those two keywords are said to be related. This "relatedness" goes beyond a single note. After two keywords appear together somewhere, that is evidence that they are related in a general sense. Related keywords can give the user guidance. When the user picks a keyword for a note, any related keywords are displayed at the same time. The user can then check if any of those other keywords should be added to the note at the same time.

This keyword system is used only a little by the NoteTaker tool. It is merely a reminder system for the user and an exploration of XUL and RDF technology. A version of NoteTaker created after the example in this book will support a Note Manager window. Such a window is used to list all existing notes, order and manage those notes, search for notes by keyword, and retrieve and display URLs for specific notes.

In the NoteTaker GUI, two other pieces of data exist. They appear as checkable boxes in the Edit dialog box under "Chop Query" and "Home Page." These items are not properties of the note, but rather control how the note is created.

If "Chop Query" is ticked, then a new note's URL will have any HTTP GET parameters removed. For example,

```
http://www.test.com/circuits.cgi?voltage=240V;amps=50mA
```

would be reduced to

```
http://www.test.com/circuits.cgi
```

The constructed note would appear on any page whose URL is prefixed with this shorter URL. If "Home Page" is ticked, then the URL will be reduced even more, to

```
http://www.test.com/
```

In this case, the note will only appear on the home page. If the URL contains an individual user's directory like this example:

```
http://www.test.com/~fred/mytests/test2.htm
```

then ticking "Home Page" will reduce this URL to

```
http://www.test.com/~fred/
```

For both of these options, only the most specific note for a given URL will be shown when that URL is displayed. In summary, after these choices are made in the GUI, the results are implicit in the URL for the note. Therefore, they add no further data to the model.

That concludes our descriptive overview of the data.

This overview of the data needs to be reduced to the three components of facts. Step 2 involves picking out the significant terms. Because facts include relationship information, we need to spot that as well, not just the nouns (entities, objects, or resources). Finding the nouns is a useful beginning point, though. Our first guesses in this case are as follows:

```
note keyword URL summary details top left width height
```

To these noun-like items, we add some relationship guesses:

```
related-keyword note-data note-for-a-url
```

Now to assemble these terms into facts—step 3. Facts can be viewed as subject-predicate-object triples (the general terminology) or as resource-property-value triples (the RDF terminology). Here we will see how both of these views are useful. First, we test each of the found terms for useful meaning by asking three questions:

1. Could this term be a thing of its own (a subject or resource)?
2. Could this term be a relationship (a predicate)?
3. Could this term be a descriptive feature of something else (a property)?

Question 3 will help us separate out the "weaker" terms so that we set a reasonable limit on what we model. Table 11.4 shows the early results of these questions.

Table 11.4 XML namespaces used for RDF vocabulary

Term	Thing?	Relationship?	Feature?
note	✓		✓ 3.
keyword	✓		1.
URL	✓		✓ 3.
summary	2.		✓

Table 11.4 XML namespaces used for RDF vocabulary (Continued)

Term	Thing?	Relationship?	Feature?
details	2.		✓
top	2.		✓
left	2.		✓
width	2.		✓
height	2.		✓
related-keyword		✓	✓
note-data		✓ 4.	
note-for-a-url		✓	✓

After this initial guesswork, we have some results and some outstanding issues. First the results: keyword is a thing; note-data is a relationship; and summary, details, top, left, width, and height are all descriptive features. We're sure of these results because each relevant row in the table has one tick only.

Next, the issues:

1. We suspect that a keyword can't be a feature of another term because that would make it a property in RDF (a fact predicate). Although there's nothing legally wrong with that, properties are generally expected to have well-known names, like "color." We know that there can be zero or more keywords per note, each with a different user-entered name. That means there is no single well-known name. Keyword is therefore not a good example of a feature. We would have this problem with any data item that has a many-to-one relationship with another data item.
2. These terms have no features of their own, so we can't see any reason to elevate them to the status of a thing. It seems pretty obvious that they're properties of something else, like a URL or a note.
3. We're still a bit confused about note and URL. Which belongs to which, or are they separate? We'd better assume they're both things for the minute.
4. Note-data sounds a bit uncertain. It doesn't identify the data or its relationship. In fact, it's not a concrete piece of information at all; it's vague. We've accidentally introduced a metadata concept: "notes have data." Summary, by comparison, is a concrete piece of data for a note. We'll throw note-data away. Metadata is never required for simple applications.

Out of this analysis, the facts we've identified are shown in Listing 11.15.

Listing 11.15 Starting facts for NoteTaker data model.

```
<- note, ?, URL ->
<- URL, ?, note ->
<- note, URL, ? ->
<- URL, note, ? ->

<- keyword, ?, ? ->
<- note, summary, ? ->
<- note, details, ? ->
<- note, top, ? ->
<- note, left, ? ->
<- note, width, ? ->
<- note, height, ? ->
<- ?, related-keyword, ? ->
<- ?, note-for-a-url, ? ->
```

The first four facts are possibilities that reflect our uncertainty about how notes and URLs are related. We need to fill in the unknowns, using our knowledge of the application's needs. We'll come back to these, after we've done the easy bits.

The six facts from `summary` to `height` are easy. They will hold a simple value each, so their object component is not a URI. It must be one of the types that Mozilla supports (`Literal`, `Integer`, `Date`, `Blob`). We'll just choose `Literal` (which is a string). So an example is

```
<- note, top, Literal ->
```

The related-keyword fact relates two keywords together. It seems obvious that the subject and object of this fact should be a keyword. We'll reduce the spelling `related-keyword` to `related`, for brevity:

```
<- keyword, related, keyword ->
```

The keyword fact presents us with a naming problem. We know that a keyword must be named by a URI because it's a thing (a resource). Unless some server specifies all the keywords in the world (not our case, and not even practical), it must be a URN rather than a URL. What is that URN? We haven't got one. We'll have to construct it out of the data the user supplies. We'll make it by prefixing the keyword string with `urn:notetaker:keyword:`. For example, the keyword `foo` will have URI:

```
urn:notetaker:keyword:foo
```

We also need access to the keyword string itself. It's not obvious now, but in later chapters we'll see that it's hard to extract a substring out of an RDF URI. So we'll have a property named label, and we'll hold the keyword as a separate string in it:

```
<- urn:notetaker:keyword:{keystring}, label, "{keystring}" ->
```

Finally, there is the complexity of a note and a URL. Each note has one URL, and each URL has one note. We need to know if they're separate. If they are separate, we'll need some kind of cross-table relationship between them. If they're not separate, then one will probably be a property of the other.

If note and URL are separate, then both will have a URI, so let's test that possibility. The URI for a URL is obvious—it's just the URL itself. What is the URI for a note? It will have to be a URN (it could be a file: URL, but that would be unusual). What will that URN be? Ultimately, we don't have a name for the note, we'd have to manufacture an arbitrary one (how?), or make the note anonymous (but then the final RDF document wouldn't be ground). Both of those options are ugly.

The truth is that the note lacks identity of its own and must therefore be dependent on the URL. Because it has no identity, it can't appear as a subject or object in any fact. Since a note doesn't have an "own value," it can't be a literal either. The whole concept of a note as a thing collapses to nothing. We throw it out of the model, leaving only the URL that the note is for. That resolves most of our issues with the first four facts in Listing 11.13.

This last point may seem very surprising, especially if you've done any object-oriented or relational modeling before. Aren't notes central to the whole NoteTaker application? Aren't we at liberty to create whatever objects/entities we like? The answers are Yes to the former and No to the latter. Yes, notes are a concept of the application, but as it turns out, not of the data model. No, we're not at liberty to create whatever entities we like because we're not working with a pure fact system like Prolog or Lisp, we're working with RDF. In RDF, one concept exists already before modeling starts: the concept of a URI-based resource. We're not free to create first-class objects; we're only free to create things with URIs, or else plain literals. Although we have a concept of a note in the NoteTaker application, in the RDF data model everything significant must be a URI. The only reason we managed to keep "keyword" as a data model concept is because we found a URI for each one.

The moral of this modeling process is simple: If you are not modeling a URL, you must make a URN for it, treat it as a literal, or forget it.

Listing 11.16 shows what's left when all of these changes are made:

Listing 11.16 Completed data facts for the NoteTaker data model.

```
<- URL, summary, Literal ->
<- URL, details, Literal ->
<- URL, top, Literal ->
<- URL, left, Literal ->
<- URL, width, Literal ->
<- URL, height, Literal ->
<- keyword, label, Literal ->
<- keyword, related, keyword ->
```

The obvious missing fact is some link between keyword and the note-laden URL. In Table 11.4 we put off thinking about this link because it wasn't clear how a keyword could be related to another fact when it's a many-to-one relationship. We can't use a keyword as a property/predicate because it's URN changes for each keyword. We could try either of these:

```
<- URL, keyword, keyword-urn ->
<- keyword-urn, note-url, URL->
```

Each URL (and note) would have zero or more instances of the first fact; alternately (or also), each keyword would have zero or more instances of the second fact.

In this proposed solution, there are multiple "keyword" properties per note (per URL). Could we store those keywords in an RDF container, like `<Seq>`? The answer is: not easily. If we did, there would be one such container for each URL with a note defined. What would those containers have as names? Such names would need to be URNs, and so we would need to construct them. We would need to do something like concatenating the URL for the note to the end of some prefix. That is possible, but messy, and we would run into string-processing problems in later chapters. We are better off sticking to a simple solution. We avoid `<Seq>` because such sequences would be repeated. We stick with the proposed solution, but we'll be lazy and only use the first fact:

```
<- URL, keyword, keyword-urn ->
```

We have now finished step 4 of our seven-step modeling process—we have captured all the necessary information for the data model. An example of the facts we'll store for one note with two keywords "test" and "cool" is shown in Listing 11.17.

Listing 11.17 Example facts for one NoteTaker note.

```
<- http://saturn/test.html, summary, "My Summary" ->
<- http://saturn/test.html, details, "My Details" ->
<- http://saturn/test.html, top, "100" ->
<- http://saturn/test.html, left, "90" ->
<- http://saturn/test.html, width, "80" ->
<- http://saturn/test.html, height, "70" ->
<- http://saturn/test.html, keyword, urn:notetaker:keyword:test ->
<- http://saturn/test.html, keyword, urn:notetaker:keyword:cool ->

<- urn:notetaker:keyword:test, label, "test" ->
<- urn:notetaker:keyword:cool, label, "cool" ->

<- urn:notetaker:keyword:test, related, urn:notetaker:keyword:cool ->
```

The last fact is the sole "relatedness" between keywords. We could also store a fact that is the reverse of the last fact, since `test` is also related to

cool. If we do that, we'll have many extra facts for notes with many keywords. Here we'll just describe the minimum facts necessary to link the keywords.

Steps 5 and 6 of the modeling anticipate Chapter 14, Templates. Eventually, we'd like to extract these facts from an RDF document efficiently. This means adding extra structure to the basic facts. Scripts and templates will be able to exploit this structure. The only such structures that RDF provides are container tags, a little type information, and the capbility to add facts without restriction. Looking at the NoteTaker tool, scripts and templates will want to:

1. Find out if a note exists for a given URL, so that NoteTaker knows whether to display one or not.
2. Extract the details of a note, including keywords, using a URL. That is for display in the dialog box, for both the Edit and the Keywords panes.
3. Extract just the summary and details of a note using a URL. This is for the toolbar textboxes and for the content of the HTML-based note.
4. Extract a list of all the existing keywords. That is for the drop-down menu in the toolbar.
5. Somehow extract all the related keywords for a given keyword, and all their related keywords, and so on. That is for the user's information in the Keywords pane of the dialog box.

In addition to these fact extractions, we want to be able to add, remove, and update notes easily. Those tasks are generally easy, so we'll address only the preceding list. That list is effectively a set of queries.

Mozilla can find facts in an RDF document no matter how they are arranged, but some arrangement makes it faster, and some arrangements make the RDF easier to read. We'll do two things:

- Put all the URLs for existing notes into a <Bag> RDF container with URI urn:notetaker:notes. Queries 1, 2, and 3 can use this container to find a note and its content.
- Put all the URNs for existing keywords into a <Bag> RDF container with URI urn:notetaker:keywords. Query 4 can use this container to get its list.

Query 5 is quite difficult because it requires extensive exploration of the keyword-laden facts. We'll just say that there is no obvious help we can add at this stage. Whatever system handles query 5 will probably have to search the whole set of facts, or a large subset.

Because RDF documents are supposed to use <Bag> tags as fact subjects, we'd better add a topmost <Description> tag with resource urn:notetaker:root. Our two <Bag> tags can be property values of that tag.

That is all the modeling we need. In step 7, the last step, it is purely mechanical to turn the data model into an RDF document. A skeleton document based on our RDF container choices, and containing no notes, is shown in Listing 11.18.

Listing 11.18 Example facts for one NoteTaker note.

```
<?xml version="1.0"?>
<RDF xmlns="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
     xmnls:NT="http://www.mozilla.org/notetaker-rdf#>
  <Description about="urn:notetaker:root">
    <NT:notes>
      <Seq about="urn:notetaker:notes"/>
    </NT:notes>
    <NT:keywords>
      <Seq about="urn:notetaker:keywords"/>
    </NT:keywords>
  </Description>
</RDF>
```

If this document is populated with the note of Listing 11.18, then the final RDF document is shown in Listing 11.19.

Listing 11.19 Example NoteTaker database in RDF—one note only.

```
<?xml version="1.0"?>
<RDF xmlns="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
     xmnls:NT="http://www.mozilla.org/notetaker-rdf#>
  <Description about="urn:notetaker:root">
    <NT:notes>
      <Seq about="urn:notetaker:notes">
        <li resource="http://saturn/test.html"/>
      </Seq>
    </NT:notes>
    <NT:keywords>
      <Seq about="urn:notetaker:keywords">
         <NT:keyword resource="urn:notetaker:keyword:cool/>
         <NT:keyword resource="urn:notetaker:keyword:test/>
      </Seq>
    </NT:keywords>
  </Description>

<!-- one note -->
  <Description about="http://saturn/test.html">
    <NT:summary>My Summary<NT:summary/>
    <NT:details>My Details<NT:details/>
    <NT:top>100<NT:top/>
    <NT:left>90<NT:left/>
    <NT:width>80<NT:width/>
    <NT:height>70<NT:height/>
    <NT:keyword resource="urn:notetaker:keyword:test"/>
```

```
    <NT:keyword resource="urn:notetaker:keyword:cool"/>
  </Description>

<!-- values for each keyword -->
  <Description about="urn:notetaker:keyword:test label="test"/>
  <Description about="urn:notetaker:keyword:cool label="cool"/>

<!-- all related keyword pairings go here; one so far -->
  <Description about="urn:notetaker:keyword:test">
    <NT:related resource="urn:notetaker:keyword:cool">
  </Description>
</RDF>
```

Some of the `<NT:keyword>` tags are duplicated in this file. There's always a balance between keeping data simple and keeping systems that query that data simple. In our case we've chosen a little duplication of data so that most of the data querying work (described in Chapter 14, Templates) is straightforward.

Thus we are now finished with the data model for NoteTaker's notes. The file format that we'll eventually use to store those notes is set at the same time. The only reason we're using RDF is because NoteTaker is a client-only tool (no server), and the amounts of data are likely to be small.

11.7 Debug Corner: Dumping RDF

RDF data are processed silently so there's not much feedback to be had. A few tricks exist.

RDF documents usually contain whitespace that hint at the purpose of the facts stated. You can strip that whitespace out and view such a document as plain, hierarchical XML. To do this, just change the file extension to `.xml` (or use a shortcut or a link) and load it directly into a Navigator window.

There is no need to restart Mozilla every time you read an RDF document, or even every time you modify that document by hand. The document is managed by the Mozilla cache. If you modify the fact store extensively (and get lost), then you should restart Mozilla entirely.

The RDF diagram (as in Figure 11.5) for a given document is not easy to produce automatically. It is possible to write a graph-drawing routine for a graph of known, familiar, or simple structure. Such a graph can be displayed by an SVG-enabled version of Mozilla—you can use DOM operations on an SVG document to create such a graph dynamically. What you can't easily do is write a general-purpose function that analyzes and arranges an unknown RDF graph in a guaranteed readable way—there are theoretical constraints. It is a hard task and not worth attempting on limited time. The RDF page at the W3C (*www.w3.org*) contains a massive list of RDF software tools.

If you are not sure of the state of your fact store, then the easy way to inspect it is to write it out. The code in Listing 11.20 achieves this end.

Listing 11.20 `downloads.rdf` file after a single complete download.

```
// preparation ..
var Cc = Components.classes;
var Ci = Components.interfaces;
var comp  = Cc["@mozilla.org/rdf/rdf-service;1"]
var iface = Ci.nsIRDFService;

var svc = comp.getService(iface);
var ds  = svc.GetDataSource("file:///C|/tmp/test.rdf");
var rds = ds.QueryInterface(Ci.nsIRDFRemoteDataSource);

// .. normal processing of the fact store here ..

function dumpRDF()
{
    // one change must be made before a store is writable
    var sub  = svc.GetResource("urn:debug:subject");
    var pred = svc.GetResource("DebugProp");
    var obj  = svc.GetResource("urn:debug:object");
    ds.Assert(sub, pred, obj, true);

    rds.Flush(); // write it out
}
```

This code does nothing more than create a data source for the file `C:/tmp/test.rdf`. The function `dumpRDF()` can be called anytime after the source has finished loading, which can be detected with the `rds.loaded` flag or with an observer object. The `dumpRDF()` function just adds a fact to the fact store and writes the whole fact store out. One fact is added so that the data source knows that the file is out of date. If the written file is subsequently viewed, the extra fact appears like this:

```
<RDF:Description about="urn:debug:subject">
    <DebugProp resource="urn:debug:object"/>
  </RDF:Description>
```

`DebugProp` and `debug` are plain strings and have no special meaning.

11.8 SUMMARY

Fact-based systems are a little different from normal data processing. There's plenty of new concepts to pick up: fact, tuple, triple, subject, object, predicate, fact store. When those are finished, there's RDF-specific terminology as well: description, resource, property, value, container, URL, URN. RDF is a not-so simple application of XML, but at least it has a core of sensible design.

In addition to standards- and theory-based concept, Mozilla has a number of concrete, heavyweight internal structures that can pump information around. These are different from simple event-processing systems because the processing content takes time. Channels and data sources are heavyweight structures that are intimately tied to RDF.

Inside Mozilla, RDF is used all over the place. There are many data sources, components, and interfaces that an application programmer can take advantage of, and a fair chunk of the finished browser application uses RDF to store state permanently. If only RDF processing were a little faster, it might be useable as a general message exchange format, but at the moment it's best for nonperformance critical tasks.

After digesting RDF, something lighter would be most welcome. In the next chapter, Mozilla's overlay and chrome systems are examined. It requires only XML and XUL tags. Just to keep the discussion honest, the RDF system underneath is examined as well.