

Liyang Yu

A Developer's Guide to the Semantic Web

 Springer

A Developer's Guide to the Semantic Web

Liyang Yu

A Developer's Guide to the Semantic Web

 Springer

Liyang Yu
Delta Air Lines, Inc.
Delta Blvd. 1030
Atlanta, GA 30354
USA
liyang910@yahoo.com

ISBN 978-3-642-15969-5 e-ISBN 978-3-642-15970-1
DOI 10.1007/978-3-642-15970-1
Springer Heidelberg Dordrecht London New York

ACM Computing Classification (1998): H.3.5, D.2, I.2

© Springer-Verlag Berlin Heidelberg 2011

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilm or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German Copyright Law of September 9, 1965, in its current version, and permission for use must always be obtained from Springer. Violations are liable to prosecution under the German Copyright Law.

The use of general descriptive names, registered names, trademarks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

Cover design: KuenkelLopka GmbH, Heidelberg

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

To my parents, Zaiyun Du my mother and Hanting Yu
my father

The truth is, they cannot read this dedication without someone translating it. However, this is never a problem for them, since there is something in this world that goes beyond the boundary of all languages and all cultures, and still remains the same to all human hearts. It lets my parents understand every word I have said here without the need of any translation at all.

It is the love they have been giving me. I will never be able to pay them back enough, and I can only wish that I will be their son in my next life, so I can continue to love them, and be loved.

Preface

Objectives of the Book

This book is all about the Semantic Web.

From its basics, the Semantic Web can be viewed as a collection of standards and technologies that allow machines to understand the meaning (semantics) of information on the Web. It represents a new vision about how the Web should be constructed so that its information can be processed automatically by machines on a large scale.

This exciting vision opens the possibility of numerous new applications on the Web. Since 2001, there have been many encouraging results in both academic world and real application world. A whole suite of standards, technologies, and related tools have been specified and developed around the concept of the Semantic Web.

However, such an extensive and rapid progress of the Semantic Web has presented a steep learning curve for those who are new to the Semantic Web. Understanding its related concepts, learning the core standards and key technical components, and finally reaching the point where one can put all these into real development work require a considerable amount of effort.

To facilitate this learning process, a comprehensive and easy-to-follow text is a must. This book, *A Developer's Guide to the Semantic Web*, serves this exact purpose. It provides an in-depth coverage on both the *What-Is* and *How-To* aspects of the Semantic Web. From this book, you will not only obtain a solid understanding about the Semantic Web but also learn how to combine all the pieces together to build new applications on the Semantic Web. More specifically,

- it offers a complete coverage of all the core standards and technical components of the Semantic Web. This coverage includes RDF, RDFS, OWL (both OWL 1 and OWL 2), and SPARQL (including features offered by SPARQL 1.1). Other related technologies are also covered, such as Turtle, microformats, RDFa, GRDDL, and SKOS;
- it provides an in-depth description of multiple well-known applications and projects in the area of the Semantic Web, such as FOAF, semantic Wiki, SearchMonkey by Yahoo!, Rich Snippets by Google, Open Linked Data Project, and DBpedia Project;

- it explains the key concepts, core standards, and technical components in the context of examples. In addition, the readers will be taken in a step-by-step fashion through the development of each example. Hopefully for the first time ever, such teaching method will ease the learning curve for those who have found the Semantic Web a daunting topic;
- it includes several complete programming projects, which bridge the gap between *What-Is* and *How-To*. These example applications are real coding projects and are developed from the scratch. In addition, the code generated by these projects can be easily reused in the readers' future development work.

Intended Readers

The book is written with the following readers in mind:

- software engineers and developers who are interested in learning the Semantic Web technology in general;
- Web application developers who have the desire and/or needs to study the Semantic Web and build Semantic Web applications;
- researchers working in research institutes who are interested in the Semantic Web research and development;
- undergraduate and graduate students from computer science departments, whose focus of work is in the area of the Semantic Web;
- practitioners in related engineering fields. For example, data mining engineers whose work involves organizing and processing a large amount of data by machines.

The prerequisites needed to understand this book include the following:

- working knowledge of Java programming language and
- basic understanding of the Web, including its main technical components such as URL, HTML, and XML.

Structure of the Book

This book is organized as follows:

- [Chapters 1–6](#) cover the basic concept, the core standards, and technical components of the Semantic Web. The goal of these chapters is to show you the *What-Is* aspect about the Semantic Web.

[Chapter 1](#) introduces the concept of the Semantic Web by using a simple example. With this example, the difference between the traditional Web and the Semantic

Web is clearly revealed. Further discussion in this chapter helps you to establish a solid understanding about the concept of the Semantic Web.

[Chapter 2](#) covers RDF in great detail to give you a sound technical foundation to further understand the Semantic Web. If you are not familiar with RDF, you should not skip this chapter, since everything else is built upon RDF. In addition, Turtle format is presented in this chapter, which will be used to understand the material presented in [Chap. 6](#).

[Chapter 3](#) goes on with other RDF-related technologies, including Microformats, RDFa, and GRDDL. If you prefer to get a full picture about the Semantic Web as quickly as possible, you can skip this chapter. However, the material presented in this chapter will be necessary in order to understand [Chap. 8](#).

[Chapter 4](#) presents RDF schema and also introduces the concept of ontology. You should not skip this chapter since [Chap. 5](#) is built upon this chapter. SKOS is also presented in this chapter; you can skip it if you are not working with any existing vocabularies in knowledge management field.

[Chapter 5](#) discusses OWL in great detail and covers both OWL 1 and OWL 2. This is one of the key chapters in this book and should not be skipped. Unless RDF schema can satisfy the needs of your application, you should spend enough time to understand OWL, which will give you the most updated information about latest ontology development language.

[Chapter 6](#) covers SPARQL. This is another chapter that you should carefully read. Working on the Semantic Web without using SPARQL is like working with database systems without knowing SQL. Note that SPARQL 1.1 is covered in this chapter as well. At the time of this writing, SPARQL 1.1 has not become a standard yet, so when you are reading this book, note the possible updates.

- [Chapters 7–11](#) provide an in-depth discussion of some well-known Semantic Web applications/projects in the real application world. These chapters serve as a transition from knowing *What-Is* to understanding *How-To* in the world of the Semantic Web.

[Chapter 7](#) presents FOAF (Friend of A Friend) project. The FOAF ontology is arguably the most widely used ontology at this point. The goal of this chapter is to introduce you to a real-world example in the social networking area. Since the modeling of this domain does not require any specific domain knowledge, it is easy to follow and you can therefore focus on appreciating the power of the Semantic Web. This chapter should not be skipped, not only because of the popularity of the FOAF ontology but also because this ontology has been used frequently in the later chapters as well.

[Chapter 8](#) presents Google's Rich Snippets and Yahoo!'s SearchMonkey; both are using RDFa and microformats as the main tools when adding semantic markups. These are important examples, not only because they are currently the major Semantic Web applications developed by leading players in the field but also because they show us the benefits of having the added semantics on the Web.

[Chapter 9](#) discusses the topic of Semantic Wiki, together with a real-world example. This chapter represents the type of Semantic Web applications built by using manual semantic markup. After reading this chapter, you should not only see the power of the added semantics but also start to understand those situations where manual semantic markup can be a successful solution.

[Chapter 10](#) presents DBpedia in great detail. DBpedia is a well-known project in the Semantic Web community, and a large number of real-world Semantic Web applications take advantage of the DBpedia datasets directly or indirectly. Also, DBpedia gives an example of automatic semantic markup. Together with [Chap. 9](#), where manual semantic markup is used, you have a chance to see both methods at work.

[Chapter 11](#) discusses the Linked Open Data project (LOD) as a real-world implementation example of the Web of Data concept. For the past several years, LOD has attracted tremendous attention from both the academic world and the real application world. In fact, DBpedia, as a huge dataset, stays in the center of the LOD cloud. Therefore, LOD together with DBpedia becomes a must for anyone who wants to do development work on the Semantic Web. More specifically, this chapter covers both the production and the consumption aspects of Linked Data; it also provides application examples that are built upon LOD cloud. In addition, this chapter explains how to access LOD programmatically, which should be very useful to your daily development work.

- [Chapters 12–15](#) are the section of *How-To*. After building a solid foundation for development work on the Semantic Web, this section presents three different running applications that are created from scratch. The methods, algorithms, and concrete classes presented in these chapters will be of immediate use to your future development work.

[Chapter 12](#) helps to build a foundation for your future development work on the Semantic Web. More specifically, it covers four major tool categories you should know, namely development frameworks, reasoners, ontology engineering tools, and other tools such as search engines for the Semantic Web. This chapter also discusses some related development methodology for the Semantic Web, such as the ontology-driven software development methodology. Furthermore, since ontology development is the key of this methodology, this chapter also presents an ontology development guide that you can use.

[Chapter 13](#) covers a popular development framework named Jena to prepare you for your future development work on the Semantic Web. More specifically, this chapter starts from how to set up Jena development environment and then presents a Hello World example to get you started. In what follows, this chapter covers the basic operation every Semantic Web application needs, such as creating RDF models, handling persistence, querying RDF dataset, and inferencing with ontology models. After reading this chapter, you will be well prepared for real development work.

Developing applications for the Semantic Web requires a set of complex skills, and this skill set lands itself on some basic techniques. In [Chap. 13](#), you have learned some basics. [Chapter 14](#) continues along the same path by building an agent that implements the Follow-Your-Nose algorithm on the Semantic Web. After all, most Semantic Web applications will have to be based on the Web, so moving or crawling from one dataset to another on the Web with some specific goals in mind is a routine task. Follow-Your-Nose method is one such basic technique. Besides implementing this algorithm, [Chap. 14](#) also introduces some useful operations, such as how to remotely access SPARQL endpoints.

[Chapter 15](#) presents two additional Semantic Web applications from scratch. The first application helps you to create an e-mail list that you can use to enhance the security of your e-mail system. The second one is a ShopBot that runs on the Semantic Web, and you can use it to find products that satisfy your own specific needs. These two projects are discussed in great detail, showing how applications on the Semantic Web are built. This includes RDF documents handling, ontology handling, inferencing based on ontologies, and SPARQL query handling, just to name a few.

Where to Get the Code

The source code for all the examples, application projects in this book can be downloaded from the author's personal Web site, www.liyangyu.com

Acknowledgment

My deepest gratitude goes to Dr. Weirong Ding, a remarkable person in my life, for supporting me in all the ways that one can ever wish to be supported. It is not nearly as possible to list all the supports she gave me, but her unreserved confidence in my knowledge and talents has always been a great encouragement for me to finish this book. Being the first reader of this book, she has always been extremely patient with many of my ideas and thoughts, and interestingly enough, her patience has made her a medical doctor who is also an expert of the Semantic Web. And to make the readers of this book become experts of the Semantic Web, I would like to share something she always says to me: “never give yourself excuses and always give 200% of yourself to reach what you love.”

I would like to thank Dr. Jian Jiang, a good friend of mine, for introducing me to the field of the Semantic Web and for many interesting and insightful discussions along the road of this book.

My gratitude is also due to Mr. Ralf Gerstner, senior editor at Springer. As a successful IT veteran himself, he has given me many valuable suggestions about the content and final organization of this book. The communication with him is always quite enjoyable: it is not only prompt and efficient, but also very insightful

and helpful. It is simply my very good luck to have a chance to work with an editor like Ralf.

Finally, I would like to express my love and gratitude to my beloved parents for their understanding and endless love. They always give me the freedom I need, and they accept my decisions even when they cannot fully understand them. In addition, I wanted to thank them for being able to successfully teach me how to think and speak clearly and logically when I was at a very young age, so I can have one more dream fulfilled today.

Atlanta, GA, USA
September 2010

Liyang Yu

Contents

1	A Web of Data: Toward the Idea of the Semantic Web	1
1.1	A Motivating Example: Data Integration on the Web	1
1.1.1	A Smart Data Integration Agent	2
1.1.2	Is Smart Data Integration Agent Possible?	7
1.1.3	The Idea of the Semantic Web	9
1.2	A More General Goal: A Web Understandable to Machines	9
1.2.1	How Do We Use the Web?	9
1.2.2	What Stops Us from Doing More?	12
1.2.3	Again, the Idea of the Semantic Web	14
1.3	The Semantic Web: A First Look	14
1.3.1	The Concept of the Semantic Web	14
1.3.2	The Semantic Web, Linked Data, and the Web of Data	15
1.3.3	Some Basic Things About the Semantic Web	17
	Reference	18
2	The Building Block for the Semantic Web: RDF	19
2.1	RDF Overview	19
2.1.1	RDF in Official Language	19
2.1.2	RDF in Plain English	21
2.2	The Abstract Model of RDF	25
2.2.1	The Big Picture	25
2.2.2	Statement	25
2.2.3	Resource and Its URI Name	27
2.2.4	Predicate and Its URI Name	31
2.2.5	RDF Triples: Knowledge That Machine Can Use	33
2.2.6	RDF Literals and Blank Node	35
2.2.7	A Summary So Far	41
2.3	RDF Serialization: RDF/XML Syntax	42
2.3.1	The Big Picture: RDF Vocabulary	42
2.3.2	Basic Syntax and Examples	43
2.3.3	Other RDF Capabilities and Examples	59

- 2.4 Other RDF Sterilization Formats 65
 - 2.4.1 Notation-3, Turtle, and N-Triples 65
 - 2.4.2 Turtle Language 66
- 2.5 Fundamental Rules of RDF 72
 - 2.5.1 Information Understandable by Machine 73
 - 2.5.2 Distributed Information Aggregation 75
 - 2.5.3 A Hypothetical Real-World Example 76
- 2.6 More About RDF 79
 - 2.6.1 Dublin Core: Example of Pre-defined RDF Vocabulary 79
 - 2.6.2 XML vs. RDF? 81
 - 2.6.3 Use an RDF Validator 84
- 2.7 Summary 85
- 3 Other RDF-Related Technologies: Microformats, RDFa, and GRDDL 87**
 - 3.1 Introduction: Why Do We Need These? 87
 - 3.2 Microformats 88
 - 3.2.1 Microformats: The Big Picture 88
 - 3.2.2 Microformats: Syntax and Examples 89
 - 3.2.3 Microformats and RDF 94
 - 3.3 RDFa 95
 - 3.3.1 RDFa: The Big Picture 95
 - 3.3.2 RDFa Attributes and RDFa Elements 96
 - 3.3.3 RDFa: Rules and Examples 97
 - 3.3.4 RDFa and RDF 104
 - 3.4 GRDDL 105
 - 3.4.1 GRDDL: The Big Picture 105
 - 3.4.2 Using GRDDL with Microformats 105
 - 3.4.3 Using GRDDL with RDFa 107
 - 3.5 Summary 107
- 4 RDFS and Ontology 109**
 - 4.1 RDFS Overview 109
 - 4.1.1 RDFS in Plain English 109
 - 4.1.2 RDFS in Official Language 110
 - 4.2 RDFS + RDF: One More Step Toward Machine Readable 111
 - 4.2.1 A Common Language to Share 111
 - 4.2.2 Machine Inferencing Based on RDFS 113
 - 4.3 RDFS Core Elements 114
 - 4.3.1 The Big Picture: RDFS Vocabulary 114
 - 4.3.2 Basic Syntax and Examples 114
 - 4.3.3 Summary So Far 132
 - 4.4 The Concept of Ontology 136
 - 4.4.1 What Is Ontology? 137
 - 4.4.2 The Benefits of Ontology 137
 - 4.5 Building the Bridge to Ontology: SKOS 138
 - 4.5.1 Knowledge Organization Systems (KOS) 138

- 4.5.2 Thesauri vs. Ontologies 140
- 4.5.3 Filling the Gap: SKOS 141
- 4.6 Another Look at Inferencing Based on RDF Schema 149
 - 4.6.1 RDFS Ontology-Based Reasoning: Simple, Yet Powerful 149
 - 4.6.2 Good, Better, and Best: More Is Needed 151
- 4.7 Summary 152
- 5 OWL: Web Ontology Language 155**
 - 5.1 OWL Overview 155
 - 5.1.1 OWL in Plain English 155
 - 5.1.2 OWL in Official Language: OWL 1 and OWL 2 156
 - 5.1.3 From OWL 1 to OWL 2 158
 - 5.2 OWL 1 and OWL 2: The Big Picture 158
 - 5.2.1 Basic Notions: Axiom, Entity, Expression, and IRI Names 159
 - 5.2.2 Basic Syntax Forms: Functional Style, RDF/XML Syntax, Manchester Syntax, and XML Syntax 160
 - 5.3 OWL 1 Web Ontology Language 161
 - 5.3.1 Defining Classes: The Basics 161
 - 5.3.2 Defining Classes: Localizing Global Properties 163
 - 5.3.3 Defining Classes: Using Set Operators 172
 - 5.3.4 Defining Classes: Using Enumeration, Equivalent, and Disjoint 175
 - 5.3.5 Our Camera Ontology So Far 177
 - 5.3.6 Define Properties: The Basics 179
 - 5.3.7 Defining Properties: Property Characteristics 184
 - 5.3.8 Camera Ontology Written Using OWL 1 192
 - 5.4 OWL 2 Web Ontology Language 196
 - 5.4.1 What Is New in OWL 2? 196
 - 5.4.2 New Constructs for Common Patterns 197
 - 5.4.3 Improved Expressiveness for Properties 200
 - 5.4.4 Extended Support for Datatypes 210
 - 5.4.5 Punning and Annotations 214
 - 5.4.6 Other OWL 2 Features 218
 - 5.4.7 OWL Constructs in Instance Documents 222
 - 5.4.8 OWL 2 Profiles 226
 - 5.4.9 Our Camera Ontology in OWL 2 233
 - 5.5 Summary 238
- 6 SPARQL: Querying the Semantic Web 241**
 - 6.1 SPARQL Overview 241
 - 6.1.1 SPARQL in Official Language 241
 - 6.1.2 SPARQL in Plain English 242

- 6.1.3 Other Related Concepts: RDF Data Store, RDF Database, and Triple Store 243
- 6.2 Set up Joseki SPARQL Endpoint 244
- 6.3 SPARQL Query Language 247
 - 6.3.1 The Big Picture 249
 - 6.3.2 SELECT Query 252
 - 6.3.3 CONSTRUCT Query 272
 - 6.3.4 DESCRIBE Query 274
 - 6.3.5 ASK Query 275
- 6.4 What Is Missing from SPARQL? 277
- 6.5 SPARQL 1.1 277
 - 6.5.1 Introduction: What Is New? 277
 - 6.5.2 SPARQL 1.1 Query 278
 - 6.5.3 SPARQL 1.1 Update 285
- 6.6 Summary 290
- 7 FOAF: Friend of a Friend 291**
 - 7.1 What Is FOAF and What It Does 291
 - 7.1.1 FOAF in Plain English 291
 - 7.1.2 FOAF in Official Language 292
 - 7.2 Core FOAF Vocabulary and Examples 293
 - 7.2.1 The Big Picture: FOAF Vocabulary 293
 - 7.2.2 Core Terms and Examples 294
 - 7.3 Create Your FOAF Document and Get into the Friend Circle 301
 - 7.3.1 How Does the Circle Work? 301
 - 7.3.2 Create Your FOAF Document 303
 - 7.3.3 Get into the Circle: Publish Your FOAF Document 305
 - 7.3.4 From Web Pages for Human Eyes to Web Pages for Machines 307
 - 7.4 Semantic Markup: a Connection Between the Two Worlds 308
 - 7.4.1 What Is Semantic Markup 308
 - 7.4.2 Semantic Markup: Procedure and Example 308
 - 7.4.3 Semantic Markup: Feasibility and Different Approaches 312
 - 7.5 Summary 314
- 8 Semantic Markup at Work: Rich Snippets and SearchMonkey 315**
 - 8.1 Introduction 315
 - 8.1.1 Prerequisite: How Does a Search Engine Work? 315
 - 8.1.2 Rich Snippets and SearchMonkey 318
 - 8.2 Rich Snippets by Google 319
 - 8.2.1 What Is Rich Snippets: An Example 319
 - 8.2.2 How Does It Work: Semantic Markup Using Microformats/RDFa 319
 - 8.2.3 Test It Out Yourself 322

- 8.3 SearchMonkey from Yahoo! 322
 - 8.3.1 What Is SearchMonkey: An Example 323
 - 8.3.2 How Does It Work: Semantic Markup Using
Microformats/RDFa 324
 - 8.3.3 Test It Out Yourself 329
- 8.4 Summary 330
- Reference 330
- 9 Semantic Wiki 331**
 - 9.1 Introduction: From Wiki to Semantic Wiki 331
 - 9.1.1 What Is a Wiki? 331
 - 9.1.2 From Wiki to Semantic Wiki 333
 - 9.2 Adding Semantics to Wiki Site 335
 - 9.2.1 Namespace and Category System 336
 - 9.2.2 Semantic Annotation in Semantic MediaWiki 339
 - 9.3 Using the Added Semantics 347
 - 9.3.1 Browsing 347
 - 9.3.2 Wiki Site Semantic Search 350
 - 9.3.3 Inferencing 356
 - 9.4 Where Is the Semantics? 359
 - 9.4.1 SWiVT: an Upper Ontology for Semantic Wiki 360
 - 9.4.2 Understanding OWL/RDF Exports 362
 - 9.4.3 Importing Ontology: a Bridge to Outside World 372
 - 9.5 The Power of the Semantic Web 375
 - 9.6 Use Semantic MediaWiki to Build Your Own Semantic Wiki 376
 - 9.7 Summary 376
- 10 DBpedia 379**
 - 10.1 Introduction to DBpedia 379
 - 10.1.1 From Manual Markup to Automatic
Generation of Annotation 379
 - 10.1.2 From Wikipedia to DBpedia 380
 - 10.1.3 The Look and Feel of DBpedia: Page Redirect 382
 - 10.2 Semantics in DBpedia 385
 - 10.2.1 Infobox Template 385
 - 10.2.2 Creating DBpedia Ontology 388
 - 10.2.3 Infobox Extraction Methods 394
 - 10.3 Accessing DBpedia Dataset 396
 - 10.3.1 Using SPARQL to Query DBpedia 397
 - 10.3.2 Direct Download of DBpedia Datasets 401
 - 10.3.3 Access DBpedia as Linked Data 406
 - 10.4 Summary 408
 - Reference 408
- 11 Linked Open Data 409**
 - 11.1 The Concept of Linked Data and Its Basic Rules 409
 - 11.1.1 The Concept of Linked Data 409

- 11.1.2 How Big Is the Web of Linked Data and the LOD Project 411
- 11.1.3 The Basic Rules of Linked Data 412
- 11.2 Publishing RDF Data on the Web 413
 - 11.2.1 Identifying Things with URIs 413
 - 11.2.2 Choosing Vocabularies for RDF Data 423
 - 11.2.3 Creating Links to Other RDF Data 427
 - 11.2.4 Serving Information as Linked Data 434
- 11.3 The Consumption of Linked Data 439
 - 11.3.1 Discover Specific Target on the Linked Data Web 441
 - 11.3.2 Accessing the Web of Linked Data 445
- 11.4 Linked Data Application 455
 - 11.4.1 Linked Data Application Example: Revyu 456
 - 11.4.2 Web 2.0 Mashups vs. Linked Data Mashups 463
- 11.5 Summary 465
- 12 Building the Foundation for Development on the Semantic Web 467**
 - 12.1 Development Tools for the Semantic Web 467
 - 12.1.1 Frameworks for the Semantic Web Applications 467
 - 12.1.2 Reasoners for the Semantic Web Applications 471
 - 12.1.3 Ontology Engineering Environments 474
 - 12.1.4 Other Tools: Search Engines for the Semantic Web 478
 - 12.1.5 Where to Find More? 478
 - 12.2 Semantic Web Application Development Methodology 478
 - 12.2.1 From Domain Models to Ontology-Driven Architecture 478
 - 12.2.2 An Ontology Development Methodology Proposed by Noy and McGuinness 484
 - 12.3 Summary 489
 - Reference 490
- 13 Jena: A Framework for Development on the Semantic Web 491**
 - 13.1 Jena: A Semantic Web Framework for Java 491
 - 13.1.1 What Is Jena and What It Can Do for Us? 491
 - 13.1.2 Getting Jena Package 492
 - 13.1.3 Using Jena in Your Projects 495
 - 13.2 Basic RDF Model Operations 501
 - 13.2.1 Creating an RDF Model 502
 - 13.2.2 Reading an RDF Model 507
 - 13.2.3 Understanding an RDF Model 509
 - 13.3 Handling Persistent RDF Models 515
 - 13.3.1 From In-memory Model to Persistent Model 515
 - 13.3.2 Setting Up MySQL 516
 - 13.3.3 Database-Backed RDF Models 517
 - 13.4 Inferencing Using Jena 524

- 13.4.1 Jena Inferencing Model 524
- 13.4.2 Jena Inferencing Examples 525
- 13.5 Summary 531
- 14 Follow Your Nose: A Basic Semantic Web Agent 533**
 - 14.1 The Principle of Follow-Your-Nose Method 533
 - 14.1.1 What Is Follow-Your-Nose Method? 533
 - 14.1.2 URI Declarations, Open Linked Data, and Follow-Your-Nose Method 535
 - 14.2 A Follow-Your-Nose Agent in Java 536
 - 14.2.1 Building the Agent 536
 - 14.2.2 Running the Agent 543
 - 14.2.3 More Clues for Follow Your Nose 545
 - 14.2.4 Can You Follow Your Nose on Traditional Web? 546
 - 14.3 A Better Implementation of Follow-Your-Nose Agent: Using SPARQL Queries 548
 - 14.3.1 In-memory SPARQL Operation 549
 - 14.3.2 Using SPARQL Endpoints Remotely 553
 - 14.4 Summary 556
- 15 More Application Examples on the Semantic Web 559**
 - 15.1 Building Your Circle of Trust: A FOAF Agent You Can Use 559
 - 15.1.1 Who Is on Your E-mail List? 559
 - 15.1.2 The Basic Idea 560
 - 15.1.3 Building the `EmailAddressCollector` Agent 563
 - 15.1.4 Can You Do the Same for Traditional Web? 572
 - 15.2 A ShopBot on the Semantic Web 573
 - 15.2.1 A ShopBot We Can Have 573
 - 15.2.2 A ShopBot We Really Want 574
 - 15.2.3 Building Our ShopBot 583
 - 15.2.4 Discussion: From Prototype to Reality 599
 - 15.3 Summary 600
- Index 601**

Chapter 1

A Web of Data: Toward the Idea of the Semantic Web

If you are reading this book, chance is you are a software engineer who makes a living by developing applications on the Web – or, more precisely, on the Web that we currently have.

And yes, there is another kind of Web. It is built on top of the current Web and called *the Semantic Web*. As a Web application developer, your career on the Semantic Web will be more exciting and fulfilling.

This book will prepare you well for your development work on the Semantic Web. This chapter will tell you exactly what the Semantic Web is, and why it is important for you to learn everything about it.

We will get started by presenting a simple example to illustrate the difference between the current Web and the Semantic Web. You can consider this example as a development assignment for you. Once you start to ponder the issues such as what exactly do we need to change on the current Web to finish this assignment, you are well on the way to see the basic picture about the Semantic Web.

With a basic understanding about the Semantic Web, we will continue to discuss how much more it can revolutionize the way we use the Web, and can further change the patterns we conduct our development work on the Web. We will then formally introduce the concept of the Semantic Web; hopefully this concept will seem to be much more intuitive to you.

This chapter will build a solid foundation for you to understand the rest of this book. When you have finished the whole book, come back and read this chapter again. You should be able to acquire a deeper appreciation of the idea of the Semantic Web. By then, I hope you have also formed your own opinion about the vision of the Semantic Web, and with what you have learned from this book, you are ready to start your own exciting journey of exploring more and creating more on the Semantic Web.

1.1 A Motivating Example: Data Integration on the Web

Data integration on the Web refers to the process of combining and aggregating information resources on the Web so they could be collectively useful to us. In

this section, we will use a concrete example to see how data integration can be implemented on the Web, and why it is so interesting to us.

1.1.1 A Smart Data Integration Agent

Here is our goal: for a given resource (could be a person, an idea, an event, or a product, such as a digital camera), we would like to know everything that has been said about it. More specifically, we would like to accomplish this goal by collecting as much information as possible about this resource; we will then understand it by making queries against the collected information.

To make this more tangible, let us use myself as the resource. In addition, assume we have already built a “smart” agent, which will walk around the Web and try to find everything about me on our behalf.

To get our smart agent started, we feed it with the URL of my personal home page as the starting point of its journey on the Web:

`http://www.liyangyu.com`

Now, our agent downloads this page and tries to collect information from this page. This is also the point where everything gets more interesting:

- If my Web page were a traditional Web document, our agent would not be able to collect anything that is much useful at all.

More specifically, the only thing our agent is able to understand on this page would be those HTML language constructs, such as `<p>`, `
`, `<href>`, `<table>` and ``. Besides telling a Web browser about how to present my Web page, these HTML constructs do not convey any useful information about the underlying resource. Therefore, other than these HTML tokens, to our agent, my whole Web page would simply represent a string of characters that look no different from any other Web document.

- However, let us assume my Web page is not a traditional Web document: besides the HTML constructs, it actually contains some “statements” that can be collected by our agent.

More specifically, all these statements follow the same simple structure, and each one of them represents one aspect of the given resource. For example, List 1.1 shows some example statements that have been collected:

List 1.1 Some of the statements collected by our smart agent from my personal Web page

```
ns0:LiyangYu ns0:name "Liyang Yu".
ns0:LiyangYu ns0:nickname "LaoYu".
ns0:LiyangYu ns0:author <ns0:_x>.
ns0:_x ns0:ISBN "978-1584889335".
ns0:_x ns0:publisher <http://www.crcpress.com>.
```

At this point, let us not worry about the issues such as how these statements are added to my Web page, and how our agent collects them. Let us simply assume when our agent visits my Web page, it can easily discover these statements.

Note that `ns0` represents a namespace, so that we know everything, with `ns0` as its prefix, is collected from the same Web page. `ns0:LiyangYu` represents a resource that is described by my Web page; in this case, this resource is me.

With this said, the first statement in List 1.1 can be read like this:

```
resource ns0:LiyangYu has a ns0:name whose value is Liyang Yu.
```

Or, like this:

```
resource ns0:LiyangYu has a ns0:name property, whose value is Liyang Yu.
```

The second way to read a given statement is perhaps more intuitive. With this in mind, each statement actually adds one *property-value* pair to the resource that is being described. For example, the second statement claims the `ns0:nickname` property of resource `ns0:LiyangYu` has a value given by `LaoYu`.

The third statement in List 1.1 is a little bit unusual. When specifying the value of `ns0:author` property for resource `ns0:LiyangYu`, instead of using a simple character string as its value (as the first two statements in List 1.1), it uses another resource, and this resource is identified by `ns0:_x`. To make this fact more obvious, `ns0:_x` is included by `<>`.

Statement 4 in List 1.1 specifies the value of `ns0:ISBN` property of resource `ns0:_x`, and the last statement in List 1.1 specifies the value of `ns0:publisher` property of the same resource. Note again that the value of this property is not a character string, but another resource identified by <http://www.crcpress.com>.

The statements in List 1.1 probably still make sense to our human eyes, no matter how ugly they look. The real interesting question is, how much does our agent understand these statements?

Not much at all. However, without too much understanding about these statements, our agent can indeed organize them into a graph format, as shown in Fig. 1.1 (note that more statements have been added to the graph).

After the graph shown in Fig. 1.1 is created, our agent declares its success on my personal Web site and moves on to the next one. It will repeat the same process again when it hits the next Web document.

Let us say the next Web site our agent hits is www.amazon.com. Similarly, if Amazon were still the Amazon today, our agent could not do much either. In fact, it can retrieve information about this ISBN number, 978-1584889335, by using Amazon Web Services.¹ For now, let us say our agent does not know how to do that.

However, again assume Amazon is a new Amazon already. Our agent can therefore collect lots of statements, which follow the same format as shown in List 1.1. Furthermore, among the statements that have been collected, some of them are shown in List 1.2.

¹<http://aws.amazon.com/>

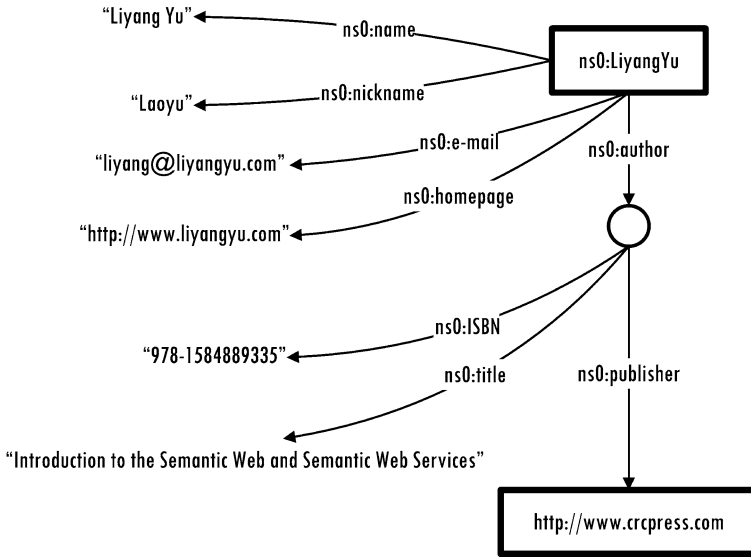


Fig. 1.1 A graph generated by our agent after visiting my person Web page

List 1.2 Statements collected by our agent from Amazon.com

```

ns1:book-1584889330 ns1:ISBN "978-1584889335" .
ns1:book-1584889330 ns1:price USD62.36 .
ns1:book-1584889330 ns1:customerReview "4.5 star" .

```

Note that, similar to namespace prefix ns0, ns1 represents another namespace prefix. And now, our agent can again organize these statements into a graph form as shown in Fig. 1.2 (note that more statements have been added to the graph).

For human eyes, one important fact is already quite obvious: ns0:_x, as a resource in Fig. 1.1 (the empty circle), represents exactly the same item denoted

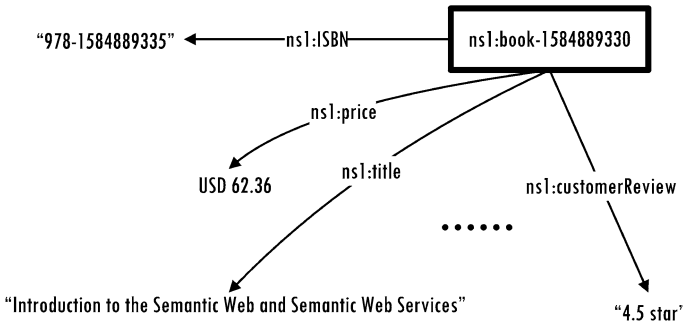


Fig. 1.2 The graph generated by our agent after visiting Amazon.com

by the resource named `ns1:book-1584889330` in Fig. 1.2. And once we made this connection, we start to see other facts easily. For example, a person who has a home page with its URL given by <http://www.liyangyu.com> has a book published and the current price of that book is US \$62.36 on Amazon. Obviously, this fact is not explicitly stated on either one of the Web sites, but our human minds have integrated the information from both www.liyangyu.com and www.amazon.com to reach this conclusion.

For our agent, similar data integration is not difficult either. In fact, our agent sees the ISBN number, 978-1584889335, showing up in both Figs. 1.1 and 1.2, it will therefore make a connect between these two appearances, as shown in Fig. 1.3. It will then automatically add the following new statement to its original statement collection:

```
ns0:_x sameAs ns1:book-1584889330.
```

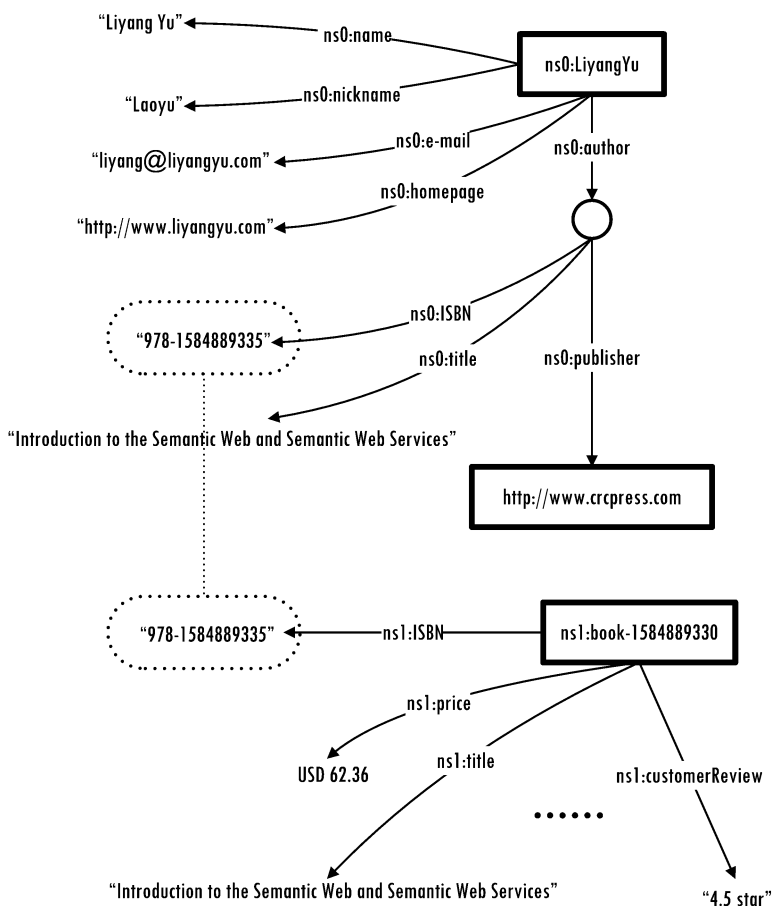


Fig. 1.3 Our agent can combine Figs. 1.1 and 1.2 automatically

And once this is done, for our agent, Figs. 1.1 and 1.2 are already “glued” together by overlapping the ns0:_x node with the ns1:book-1584889330 node. This gluing process is exactly the data integration process on the Web.

Now, without going into the details, it is not difficult to convince ourselves that our agent can answer lots of questions that we might have. For example, what is the price of the book written by a person whose home page is given by this URL, <http://www.liyangyu.com?>

This is indeed very encouraging. And this is not all.

Let us say now our agent hits www.linkedin.com. Similarly, if LinkedIn were still the LinkedIn today, our agent could not do much. However, again assume LinkedIn is a new LinkedIn and our agent is able to collect quite a few statements from this Web site. Some of them are shown in List 1.3.

List 1.3 Statements collected by our agent from LinkedIn

```
ns2:LiyangYu ns2:email "liyang910@yahoo.com".
ns2:LiyangYu ns2:workPlaceHomepage "http://www.delta.com".
ns2:LiyangYu ns2:connectedTo <ns2:Connie>.
```

The graph created by the agent is shown in Fig. 1.4 (note that more statements have been added to the graph).

For human readers, we know ns0:LiyangYu and ns2:LiyangYu represent exactly the same resource, because both these two resources have the same e-mail address. For our agent, just by comparing the two identities (ns0:LiyangYu vs. ns2:LiyangYu) does not ensure the fact that these two resources are the same.

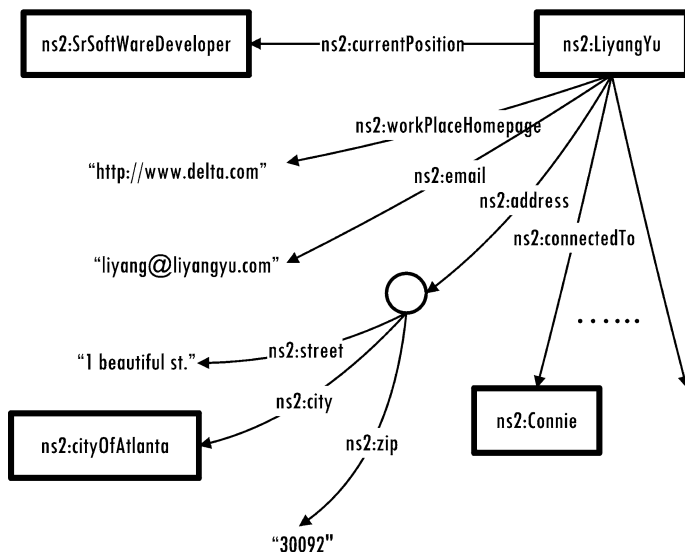


Fig. 1.4 A graph generated by our agent after visiting linkedIn.com

However, if we can “teach” our agent the following fact:

If the e-mail property of resource A has the same value as the e-mail property of resource B, then resources A and B are the same resource.

Then our agent will be able to automatically add the following new statement to its current statement collection:

```
ns0:LiyangYu sameAs ns2:LiyangYu.
```

With the creation of this new statement, our agent has in fact integrated Figs. 1.1 and 1.4 by overlapping node `ns0:LiyangYu` with node `ns2:LiyangYu`. Clearly, this integration process is exactly the same as the one where Figs. 1.1 and 1.2 are connected together by overlapping node `ns0:_x` with node `ns1:book-1584889330`. And now, Figs. 1.1, 1.2, and 1.4 are all connected.

At this point, our agent will be able to answer even more questions. The following are just some of them:

- What is Laoyu’s workplace home page?
- How much does it cost to buy Laoyu’s book?
- Which city does Liyang live in?

And clearly, to answer these questions, the agent has to depend on the integrated graph, not any single one. For example, to answer the first question, the agent has to go through the following link that runs across different graphs:

```
ns0:LiyangYu ns0:nickname "LaoYu".
ns0:LiyangYu sameAs ns2:LiyangYu.
ns2:LiyangYu ns2:workPlaceHomepage "http://www.delta.com".
```

Once the agent reaches the last statement, it can present an answer to us. You should be able to understand how the other questions are answered by mapping out the similar path as shown above.

Obviously, the set of questions that our agent is able to answer grows when it hits more Web documents. We can continue to move on to another Web site so as to add more statements to our agent’s collection. However, the idea is clear: automatic data integration on the Web can be quite powerful and can help us a lot when it comes to information discovery and retrieval.

1.1.2 Is Smart Data Integration Agent Possible?

The question is also clear: is it even possible to build a smart data integration agent as the one we have just discussed? Human do this kind of information integration on the Web on a daily basis, and now, we are in fact hoping to program a machine to do this for us.

To answer this question, we have to understand what exactly has to be there to make our agent possible.

Let us go back to the basics. Our agent, after all, is a piece of software that works on the Web. So to make it possible, we have to work on two parties: the Web and the agent.

Let us start with the Web. Recall that we have assumed our agent is able to collect some statements from various Web sites (see Lists 1.1, 1.2, and 1.3). Therefore, each Web site has to be different from its traditional form, so changes have to be made. Without going into the details, here are some changes we need to have on each Web site:

- Each statement collected by our agent represents a piece of knowledge. Therefore, there has to be a way (a model) to represent knowledge on the Web. Furthermore, this model of representing knowledge has to be easily and readily processed (understood) by machines.
- This model has to be accepted as a standard by all the Web sites. Otherwise, statements contained in different Web sites will not share a common pattern.
- There has to be a way to create these statements on each Web site. For example, they can be either manually added or automatically generated.
- The statements contained in different Web sites cannot be completely arbitrary. For example, they should be created by using some common terms and relationships, at least for a given domain. For instance, to describe a person, we have some common terms such as name, birthday, and home page.
- There has to be a way to define these common terms and relationships, which specifies some kind of agreement on these common terms and relationships. Different Web sites, when creating their statements, will use these terms and relationships.
- Perhaps there are more to be included.

With these changes on the Web, a new breed of Web will be available for our agent. And in order to take advantage of this new Web, our agent has to be changed as well. For example,

- Our agent has to be able to understand each statement that it collects. One way to accomplish this is by understanding the common terms and relationships that are used to create these statements.
- Our agent has to be able to conduct reasoning based on its understanding of the common terms and relationships. For example, knowing the fact that resources A and B have the same e-mail address and considering the knowledge expressed by the common terms and relationships, it should be able to conclude that A and B are in fact the same resource.
- Our agent should be able to process some common queries that are submitted against the statements it has collected. After all, without providing a query interface, the collected statements will not be of too much use to us.
- Perhaps there are more to be included as well.

Therefore, here is our conclusion: yes, our agent is possible, provided that we can implement all the above (and possibly more).

1.1.3 The Idea of the Semantic Web

At this point, the Semantic Web can be understood as follows: the Semantic Web provides the technologies and standards that we need to make our agent possible, including all the things we have listed in the previous section. It can be understood as a brand new layer built on top of the current Web, and it adds machine-understandable meanings (or “semantics”) to the current Web. Thus the name the *Semantic Web*.

The Semantic Web is certainly more than automatic data integration on a large scale. In the next section, we will position it in a more general setting. We will then summarize the concept of the Semantic Web, which will hopefully look more natural to you.

1.2 A More General Goal: A Web Understandable to Machines

1.2.1 How Do We Use the Web?

In its early days, the Web could be viewed as a set of Web sites which offered a collection of Web documents, and the goal was to get the content pushed out to its audiences. It acted like a one-way traffic: people read whatever was out there, with the goal of getting information they could use in a variety of ways.

Today, the Web has become much more interactive. First off, more and more of what is now known as user-generated content has emerged on the Web, and a host of new Internet companies were created around this trend as well. More specifically, instead of only reading the content, people are now using the Web to create content and also interact with each other by using social networking sites over Web platforms. And certainly, even if we don't create any new content or participate in any social networking sites, we can still enjoy the Web a lot: we can chat with our friends, we can shop online, we can pay our bills online, and we can also watch a tennis game online, just to name a few.

Second, the life of a Web developer has changed a lot too. Instead of offering merely static Web contents, today's Web developer is capably of building Web sites that can execute complex business transactions, from paying bills online to booking a hotel room and airline tickets.

Third, more and more Web sites have started to publish structured content so that different business entities can share their content to attract and accomplish more transactions online. For example, Amazon and eBay both are publishing structured data via Web service standards from their databases, so other applications can be built on top of these contents.

With all these being said, let us summarize what we do on today's Web at a higher level, and this will eventually reveal some critical issues we are having on the Web. Note that our summary will be mainly related to how we consume the available information on the Web, and we are not going to include activities such as chatting with friends or watching a tennis game online – these are nice things you can do on the Web, but not much related to our purpose here.

Now, to put it simple, searching, information integration, and Web data mining are the three main activities we conduct using the Web.

1.2.1.1 Searching

This is probably the most common usage of the Web. The goal is to locate some specific information or resources on the Web. For instance, finding different recipes for making margarita and locating a local agent who might be able to help buying a house are all good examples of searching.

Quite often though, searching on the Web can be very frustrating. At the time of this writing, for instance, using a common search engine (Google, for example), let us search the word SOAP with the idea that SOAP is a W3C standard for Web services in our mind. Unfortunately, we will get about 63,200,000 listings back and will soon find this result hardly helpful: there are listings for dish detergents, facial soaps, and even soap operas! Only after sifting through multiple listings and reading through the linked pages are we able to find information about the W3C's SOAP specifications.

The reason for this situation is that search engines implement their search based on the core concept of “which documents contain the given keyword” – as long as a given document contains the keyword, it will be included in the candidate set and will be later presented back to the user as the search result. It is then up to the user to read and interpret the result to extrapolate any useful information.

1.2.1.2 Information Integration

We have seen an example of information integration in Sect. 1.1.1, and we have also created an imaginary agent to help us accomplish our goal. In this section, we will take a closer look at this common task on the Web.

Let us say you decide to try some Indian food for your weekend dining out. You first search the Web to find a list of restaurants specialized in Indian cuisine, you then pick one restaurant, and you write down the address. Now you open up a new browser and go to your favorite map utility to get the driving direction from your house to the restaurant. This process is a simple integration process: you first get some information (the address of the restaurant) and you use it to get more information (the direction), and they collectively help you to enjoy a nice dining out.

Another similar but more complex example is to create a holiday plan that pretty much every one of us has done. Today, it is safe to say that we all have to do this manually: search for a place that fits not only our interest, but also our budget, and then hotel, then flights, and finally cars. The information from these different steps will then be combined together to create a perfect plan for our vacation, hopefully.

Clearly, to conduct this kind of information integration manually is a somewhat tedious process. It will be more convenient if we can finish the process with more help from the machine. For example, we can specify what we need to some application, which will then help us out by conducting the necessary steps for us. For the vacation example, the application can even help us more: after creating our itinerary (including the flights, hotels, and cars), it can search the Web to add related information such as daily weather, local maps, driving directions, city guides.

Another good example of information integration is the application that makes use of Web services. As a developer who mainly works on Web applications, Web service should not be a new concept. For example, company A can provide a set of Web services via its Web site, company B can write java code (or whatever language of their choice) to consume these services so as to search company A's product database on the fly. For instance, when provided with several keywords that should appear in a book title, the service returns a list of books whose titles contain the given keywords.

Clearly, company A is providing structured data for another application to consume. It does not matter which language company A has used to build its Web services and what platform these services are running on; it does not matter either which language company B is using and what platform company B is on; as long as company B follows the related standards, this integration can happen smoothly and nicely.

In fact, our perfect vacation example can also be accomplished by using a set of Web services. More specifically, finding a vacation place, booking a hotel room, buying air tickets, and finally making a reservation at a car rental company can all be accomplished by consuming the right Web services.

On today's Web, this type of integration often involves different Web services. Just like the case where you want to have dinner in an Indian restaurant, you have to manually locate and integrate these services together (and normally, this integration is implemented at the development phase). It would be much quicker, cleaned, powerful, and more maintainable if we could have some application that helps us to find the appropriate services on the fly and also invoke these services dynamically to accomplish our goal.

1.2.1.3 Web Data Mining

Intuitively speaking, data mining is the non-trivial extraction of useful information from a large (and normally distributed) datasets or databases. Given the fact that the Web can be viewed as a huge distributed database, the concept of Web data mining refers to the activity of getting useful information from the Web.

Web data mining might not be as interesting as searching sounds to a casual user, but it could be very important and even be the daily work of those who work as analysts or developers for different companies and research institutes.

One example of Web data mining is as follows. Let us say that we are currently working as consultants for the air traffic control group at Hartsfield-Jackson Atlanta International Airport, reportedly the busiest airport in the nation. Management group

in the control tower wanted to understand how the weather condition may effect the take-off rate on the runways, with the take-off rate defined as the number of aircrafts that have taken off at a given hour. Intuitively, a severe weather condition will force the control tower to shut down the airport so the take-off rate will go down to zero, and a moderate weather condition will just make the take-off rate low.

For a task like this, we suggest that we gather as much historical data as possible and analyze them to find the pattern of the weather effect. And we are told that historical data (the take-off rates at different major airports for the past, say, 5 years) do exist, but they are published in different Web sites. In addition, the data we need on these Web sites are normally mingled together with other data that we do not need.

To handle this situation, we will develop a specific application that acts like a crawler: it will visit these Web sites one by one, and once it reaches a Web site, it will identify the data we need and only collect the needed information (historical take-off rates) for us. After it collects these rates, it will store them into the data format that we want. Once it finishes with one Web site, it will move on to the next one until it has visited all the Web sties that we are interested in.

Clearly, this application is a highly specialized piece of software that is normally developed on a case-by-case basis. Inspired by this example, you might want to code up your own agent, which will visit all the related Web sites to collect some specific stock information and report back to you, say, every 10 min. By doing so, you don't have to open up a browser every 10 min to check the stock prices, risking the possibility that your boss will catch you visiting these Web sites, yet you can still follow the latest changes happening in the stock market.

And this stock watcher application you have developed is yet another example of Web data mining. Again, it is a very specialized piece of software and you might have to re-code it if something important has changed on the Web sites that it routinely visits. And yes, it would be much nicer if the application could understand the meaning of the Web pages on the fly so you do not have to change your code so often.

By now, we have talked about the three major activities that you can do and normally do with the Web. You might be a casual visitor to the Web, or you might be a highly trained professional developer, but whatever you do with the Web will more or less fall into one of these three categories.

The next question then is, what are the common difficulties that we have experienced in these activities? Does any solution exist to these difficulties at all? What would we do if we had the magic power to change the way the Web is constructed so that we did not have to experience these difficulties at all?

Let us talk about this in the next section.

1.2.2 What Stops Us from Doing More?

Let us go back to the first main activity: search. Among the three major activities, search is probably the most popular one, and it is also interesting that this activity in fact shows the difficulty of the current Web in a most obvious way: whenever we

do a search, we want to get only relevant results; we want to minimize the human work that is required when trying to find the appropriate documents.

However, the conflict also starts from here: the current Web is entirely aimed at human readers and it is purely display oriented. In other words, the Web has been constructed in such a way that it is oblivious to the actual information content on any given Web site. Web browsers, Web servers, and even search engines do not actually distinguish weather forecasts from scientific papers and cannot even tell a personal home page from a major corporate Web site. Search engines are therefore forced to do keyword-matching only: as long as a given document contains the keyword(s), it will be included in the candidate set that is later presented to the user as the search result.

If we had the magic power, we would re-construct the whole Web such that computers not only can present the information that is contained in the Web documents, but can also understand the very information they are presenting so they can make intelligent decisions on our behalf. For example, search engines can filter the pages before they present them back to us, if not able to directly give us the answer back.

For the second activity, integration, the main difficulty is that there is too much manual work involved in the integration process. If the Web were constructed in such a way that the meaning of each Web document can be retrieved from a collection of statements, our agent would be able to understand each page, and information integration would have become amazingly fun and easy, as we have shown in Sect. 1.1.1.

Information integration implemented by using Web services may seem quite different at first glance. However, to automatically composite and invoke the necessary Web services, the first step is to *discover* them in a more efficient and automated manner. Currently, this type of integration is difficult to implement mainly because the discovery process of its components is far from efficient.

The reason, again, as you can guess, is that although all the services needed to be integrated do exist on the Web, the Web is, however, not programmed to understand and remember the meaning of any of these services. As far as the Web is concerned, all these components are created equal, and there is no way for us to teach our computers to understand the meaning of each component, at least on the current Web.

What about the last activity, namely, Web data mining? The truth is Web data mining applications are not scalable, and they have to be implemented at a very high price, if they are possible to be implemented at all.

More specifically, each Web data mining application is highly specialized and has to be specially developed for that application context. For a given project in a specific domain, only the development team knows the meaning of each data element in the data source and how these data elements should interact together to present some useful information. The developers have to program these meanings into the mining software before setting it off to work; there is no way to let the mining application learn and understand these meanings on the fly. In addition, the underlying decision tree has to be pre-programmed into the application as well.

Also, even for a given specific task, if the meaning of the data element changes (this can easily happen given the dynamic nature of the Web documents), the mining application has to be changed accordingly since it cannot learn the meaning of the data element dynamically. All these practical concerns have made Web data mining a very expensive task to do.

Now, if the Web were built to remember all the meanings of data elements, and in addition, if all these meanings could be understood by a computer, we would then program the mining software by following a completely different pattern. We can even build a generic framework for some specific domain so that once we have a mining task in that domain, we can reuse it all the time – Web data mining will not be as expensive as today.

Now we finally reached some interesting point. Summarizing the above discussion, we have come to an understanding of an important fact: our Web is constructed in a way that its documents only contain enough information for a machine to present them, not to understand them.

Now the question is the following: is it still possible to re-construct the Web by adding some information into the documents stored on the Web, so that machines can use this extra information to understand what a given document is really about?

The answer is yes, and by doing so, we in fact change the current (traditional) Web into something we call *the Semantic Web* – the main topic of this chapter and this whole book.

1.2.3 Again, the Idea of the Semantic Web

At this point, the Semantic Web can be understood as follows: the Semantic Web provides the technologies and standards that we need to make the following possible:

- adds machine-understandable meanings to the current Web, so that
- computers can understand the Web documents and therefore can automatically accomplish tasks that have been otherwise conducted manually, on a large scale.

With all the intuitive understanding of the Semantic Web, we are now ready for some formal definition of the Semantic Web, which will be the main topic of the next section.

1.3 The Semantic Web: A First Look

1.3.1 The Concept of the Semantic Web

First off, the word “semantics” is related to the word *syntax*. In most languages, syntax is how you say something, where *semantics* is the meaning behind what you have said. Therefore, the Semantic Web can be understood as “the Web of meanings,” which echoes what we have learned so far.

At the time of this writing, there is no formal definition of the Semantic Web yet. And it often means different things to different groups of individuals. Nevertheless, the term “Semantic Web” was originally coined by World Wide Web Consortium² (W3C) director Sir Tim Berners-Lee and formally introduced to the world by the May 2001 *Scientific American* article “The Semantic Web” (Berners-Lee et al. 2001):

The Semantic Web is an extension of the current Web in which information is given well-defined meaning, better enabling computers and people to work in cooperation.

There has been a dedicated team of people at W3C working to improve, extend, and standardize the idea of the Semantic Web. This is now called *W3C Semantic Web Activity*.³ According to this group, the Semantic Web can be understood as follows:

The Semantic Web provides a common framework that allows data to be shared and reused across application, enterprise, and community boundaries.

For us, and for the purpose of this book, we can understand the Semantic Web as the following:

The Semantic Web is a collection of technologies and standards that allow machines to understand the meaning (semantics) of information on the Web.

To see this, recall that Sect. 1.1.2 has described some requirements for both the current Web and agents, in order to bring the concept of the Semantic Web into reality. If we understand the Semantic Web as a collection of technologies and standards, Table 1.1 summarizes how those requirements summarized in Sect. 1.1.2 can be mapped to the Semantic Web’s major technologies and standards.

Table 1.1 may not make sense at this point, but all the related technologies and standards will be covered in detail in the upcoming chapters of this book. When you finish this book and come back to review Table 1.1, you should be able to understand it with much more ease.

1.3.2 The Semantic Web, Linked Data, and the Web of Data

Linked Data and the *Web of Data* are concepts that are closely related to the concept of the Semantic Web. We will take a brief look at these terms in this section. You will see more details about Linked Data and Web of Data in the later chapters.

The idea of Linked Data was originally proposed by Tim Berners-Lee, and his 2006 Linked Data principles⁴ is considered to be the official and formal introduction

²<http://www.w3.org/>

³<http://www.w3.org/2001/sw/>

⁴<http://www.w3.org/DesignIssues/LinkedData.html>

Table 1.1 Requirements summarized in Sect. 1.1.2 can be mapped to the Semantic Web's technologies and standards

Requirements	The Semantic Web's technologies and standards
Each statement collected by our agent represents a piece of knowledge. Therefore, there has to be a way (a model) to represent knowledge on the Web site. And furthermore, this model of representing knowledge has to be easily and readily processed (understood) by machines	Resource description framework (RDF)
This model has to be accepted as a standard by all the Web sites; therefore statements contained in different Web sites all looked "similar" to each other	Resource description framework (RDF)
There has to be a way to create these statements on each Web site, for example, they can be either manually added or automatically generated	Semantic markup, RDFa, Microformats
The statements contained in different Web sites cannot be too arbitrary. For example, they should be created by using some common terms and relationships, perhaps on the basis of a given domain. For instance, to describe a person, we have some common terms such as name, birthday, and home page	Domain-specific ontologies/vocabularies
There has to be a way to define these common terms and relationship, and there has to be some kind of agreement on these common terms and relationships. Different Web sites, when creating their statements, will use these terms and relationships	RDF Schema (RDFS), Web Ontology Language (OWL)
Our agent has to be able to understand each statement that it collects. One way to accomplish this is by understanding the common terms and relationships that are used to create these statements	Supporting tools for ontology processing
Our agent has to be able to conduct reasoning based on its understanding of the common terms and relationships. For example, knowing the fact that resources A and B have the same e-mail address and considering the knowledge expressed by the common terms and relationships, it should be able to decide that A and B are in fact the same resource	Reasoning based on ontologies
Our agent should be able to process some common queries that are submitted against the statements it has collected. After all, without providing a query interface, the collected statements will not be of too much use to us	SPARQL query language

of the concept itself. At its current stage, Linked Data is a W3C-backed movement that focuses on connecting datasets across the Web, and it can be viewed as a subset of the Semantic Web concept, which is all about adding meanings to the Web.

To understand the concept of Linked Data, think about the motivating example presented in Sect. 1.1.1. More specifically, our agent has visited several pages and has collected a list of statements from each Web site. These statements, as we know now, represent the added meaning to that particular Web site. This has given us the

impression that the added structure information for machines has to be associated with some hosting Web site and has to be either manually created or automatically generated.

In fact, there is no absolute need that machines and human beings have to share the same Web site. Therefore, it is also possible to *directly* publish some structured information online (a collection of machine-understandable statements, for example) without having them related to any Web site at all. Once this is done, these statements as a dataset are ready to be harvested and processed by applications.

Imaginally, such a dataset will not be of much use if it does not have any link to other datasets. Therefore, one important design principle is to make sure that each such dataset has outgoing links to other datasets.

Therefore, publishing structured data online and adding links among these datasets are the key aspects of the Linked Data concept. The Linked Data principles have specified the steps of accomplishing these key aspects. Without going into the details of Linked Data principles, understand that the term of Linked Data refers to a set of best practices for publishing and connecting structured data on the Web.

What is the relationship between Linked Data and the Semantic Web? Once you finish this book (Linked Data is covered in [Chap. 11](#)), you should be able to get a much better understanding. For now, let us simply summarize their relationship without much discussion:

- Linked Data is published by using Semantic Web technologies and standards.
- Similarly, Linked Data is linked together by using Semantic Web technologies and standards.
- Finally, the Semantic Web is the goal, and Linked Data provides the means to reach the goal.

Another important concept is the Web of Data. At this point, we can understand Web of Data as an interchangeable term for the Semantic Web. In fact the Semantic Web can be defined as a collection of standard technologies to realize a Web of Data. In other words, if Linked Data is realized by using the Semantic Web standards and technologies, the result would be a Web of Data.

1.3.3 Some Basic Things About the Semantic Web

Before we set off and get into the world of the Semantic Web, there are some useful information resources you should know about. Let us list them here in this section:

- <http://www.w3.org/2001/sw/>

This is the W3C Semantic Web activity's official Web site. It has quite a lot information including a short introduction, latest publications (articles and interviews), presentations, links to specifications, and links to different working groups.

- <http://www.w3.org/2001/sw/SW-FAQ>

This is the W3C Semantic Web frequently asked questions page. This is certainly very helpful to you if you have just started to learn the Semantic Web. Again, when you finish this whole book, come back to this FAQ, take yet another look, and you will find yourself having a much better understanding at that point.

- <http://www.w3.org/2001/sw/interest/>

This is the W3C Semantic Web interest group, which provides a public forum to discuss the use and development of the Semantic Web, with the goal to support developers. Therefore, this could be a useful resource for you when you start your own development work.

- <http://www.w3.org/2001/sw/aneews/>

This is the W3C Semantic Web activity news Web site. From here, you can follow the news related to the Semantic Web Activity, especially news and progress about specifications and specification proposals.

- http://www.w3.org/2001/sw/wiki/Main_Page

This is the W3C Semantic Web community Wiki page. It has quite a lot of information for anyone interested in the Semantic Web. It has links to a set of useful sites, such as events in the Semantic Web community, Semantic Web tools, people in the Semantic Web community, and popular ontologies, just to name a few. Make sure to check this page if you are looking for related resources during the course of your study of this book.

- <http://iswc.semanticweb.org/>

There are a number of different conferences in the world of the Semantic Web. Among these conferences, the International Semantic Web Conference (ISWC) is a major international forum at which research on all aspects of the Semantic Web is presented, and the above is their official Web site. The ISWC started in 2001 and has been the major conference ever since. Check out this conference more to follow the latest research activities.

With all these being said, we are now ready to start the book. Again, you can find a roadmap of the whole book in the Preface section, and you can download all the code examples for this book from www.liyangyu.com.

Reference

Berners-Lee T, Hendler J, Lassila O (2001) The Semantic Web. *Sci Am* 284(5):34–43

Chapter 2

The Building Block for the Semantic Web: RDF

This chapter is probably the most important chapter in this whole book: it covers RDF in detail, which is the building block for the Semantic Web. A solid understanding of RDF provides the key to the whole technical world that defines the foundation of the Semantic Web: once you have gained the understanding of RDF, all the rest of the technical components will become much easier to comprehend, and in fact, much more intuitive as well.

This chapter will cover all the main aspects of RDF, including its concept, its abstract model, its semantics, its language constructs, and its features, together with ample real-world examples. This chapter also introduces available tools you can use when creating or understanding RDF models. Make sure you understand this chapter well before you move on. In addition, use some patience when reading this chapter: some concepts and ideas may look unnecessarily complex at the first glance, but eventually, you will start to see the reasons behind them.

Let us get started.

2.1 RDF Overview

2.1.1 *RDF in Official Language*

RDF stands for *Resource Description Framework*, and it was originally created in early 1999 by W3C as a standard for encoding metadata. The name, Resource Description Framework, was formally introduced in the corresponding W3C specification document that outlines the standard.¹

As we have discussed earlier, the current Web is built for human consumption, and it is not machine understandable at all. It is therefore very difficult to automate anything on the Web, at least on a large scale. Furthermore, given the huge amount of information the Web contains, it is impossible to manage it manually either. A solution proposed by W3C is to use metadata to describe the data contained on

¹Resource Description Framework (RDF) model and syntax specification, a W3C Recommendation, 22 February 1999. <http://www.w3.org/TR/1999/REC-rdf-syntax-19990222/>

the Web, and the fact that this metadata itself is machine understandable enables automated processing of the related Web resources.

With the above consideration in mind, RDF was proposed in 1999 as a basic model and foundation for creating and processing metadata. Its goal is to define a mechanism for describing resources that makes no assumptions about a particular application domain (domain independent), and therefore can be used to describe information about any domain. The final result is that RDF concept and model can directly help to promote interoperability between applications that exchange machine-understandable information on the Web.

As we have discussed in [Chap. 1](#), the concept of the Semantic Web was formally introduced to the world in 2001, and the goal of the Semantic Web is to make the Web machine understandable. The obvious logical connection between the Semantic Web and RDF has greatly changed RDF: the scope of RDF has since then involved into something that is much greater. As we will see later throughout the book, RDF is not only used for encoding metadata about Web resources, but also used for describing *any* resources and their relations existing in the real world.

This much larger scope of RDF has been summarized in the updated RDF specifications published in 2004 by the RDF Core Working Group² as part of the W3C Semantic Web activity.³ These updated RDF specifications contain altogether six documents as shown in [Table 2.1](#). These six documents have since then jointly replaced the original Resource Description Framework specification (1999 Recommendation), and they together became the new RDF W3C Recommendation on 10 February 2004.

Table 2.1 RDF W3C recommendation, 10 February 2004

Specification	Recommendation
RDF Primer	10 February 2004
RDF Test Cases	10 February 2004
RDF Concept	10 February 2004
RDF Semantics	10 February 2004
RDF Schema	10 February 2004
RDF Syntax	10 February 2004

Based on these official documents, RDF can be defined as follows:

- RDF is a language for representing information about resources in the World Wide Web (*RDF Primer*).
- RDF is a framework for representing information on the Web (*RDF Concept*).
- RDF is a general-purpose language for representing information in the Web (*RDF Syntax*, *RDF Schema*).

²RDFCore Working Group, W3C Recommendations, <http://www.w3.org/2001/sw/RDFCore/>

³W3C Semantic Web activity, <http://www.w3.org/2001/sw/>

- RDF is an assertional language intended to be used to express propositions using precise formal vocabularies, particularly those specified using RDFS, for access and use over the World Wide Web, and is intended to provide a basic foundation for more advanced assertional languages with a similar purpose (*RDF Semantics*).

At this point, it is probably not easy to truly understand what RDF is, based on these official definitions. Let us keep these definitions in mind, and once you have finished this chapter, review these definitions and you should find yourself having a better understanding of them.

For now, let us move on to some more explanation in plain English about what exactly RDF is and why we need it. This explanation will be much easier to understand and will give you enough background and motivation to continue reading the rest of this chapter.

2.1.2 RDF in Plain English

Let us forget about RDF for a moment and consider those Web sites where we can find reviews of different products (such as Amazon.com, for example). Similarly, there are also Web sites that sponsor discussion forums where a group of people get together to discuss the pros and cons of a given product. The reviews published at these sites can be quite useful when you are trying to decide whether you should buy a specific product or not.

For example, I am a big fan of photography, and I have recently decided to upgrade my equipment – to buy a Nikon SLR (single lens reflex) camera so I will have more control over how a picture is taken and therefore have more chance to show my creative side. Note the digital version of SLR camera is called DSLR (digital single lens reflex).

However, pretty much all Nikon SLR models are quite expensive, so to spend money wisely, I have read quite a lot of reviews, with the goal of choosing the one particular model that fits my needs the best.

You must have had the same experience, probably with some other product. Also, you will likely agree with me that reading these reviews does take a lot of time. In addition, even after reading quite a lot of reviews, you are still not sure: could it be true that I have missed some reviews that could be very useful?

Now, imagine you are a quality engineer who works for Nikon. Your assignment is to read all these reviews and summarize what people have said about Nikon SLR cameras and report back to Nikon headquarter so the design department can make better designs based on these reviews.

Obviously, you can do your job by reading as many reviews as you can and manually create a summary report and submit it to your boss. However, it is not only tedious, but also quite demanding: you spend the whole morning reading, you have only covered a couple dozen reviews, with a couple hundreds more to go!



Fig. 2.1 Amazon’s review page for Nikon D300 SLR camera

One idea to solve this problem is to write an application that will read all these reviews for you and generate a report automatically, and all this will be done in a matter of couple of minutes. Better yet, you can run this application as often as you want, just to gather the latest reviews. This is a great idea with only one flaw: such an application is not easy to develop, since the reviews published online are intended for human eyes to consume, not for machines to read.

Now, in order to solve this problem once and for all so as to make sure you have a smooth and successful career path, you start to consider the following key issue:

Assuming all the review publishers are willing to accept and follow some standard when they publish their reviews, what standard would make it easier to develop such an application?

Note the words we used were *standard* and *easier*. Indeed, although writing such an application is difficult, there is in fact nothing stopping us from actually doing it, even on the given Web and without any standard. For example, screen scraping can be used to read reviews from Amazon.com’s review page, as shown in Fig. 2.1.

On this page, a screen-scraping agent can pick up the fact that 40 customers have assigned five stars to Nikon D300 (a DSLR camera by Nikon), and four attributes are currently used for reviewing this camera, and they are called *Ease of use*, *Features*, *Picture quality*, and *Portability*.

Once we have finished coding the agent that understands the reviews from Amazon.com, we can move on to the next review site. It is likely that we have to

add another new set of rules to our agent so it can understand the published reviews on that specific site. The same is true for the next site, so on and so forth.

There is indeed quite a lot of work, and obviously, it is not a scalable way to develop an application either. In addition, when it comes to the maintenance of this application, it could be more difficult than the initial development work. For instance, a small change on any given review site can easily break the logic that is used to understand that site, and you will find yourself constantly being busy changing and fixing the code.

And this is exactly why a standard is important: once we have a standard that all the review sites follow, it will be much easier to write an application to collect the distributed reviews and come up with a summary report.

Now, what exactly is this standard? Perhaps it is quite challenging to come up with a complete standard right away, but it might not be too difficult to specify some of the things we would want such a standard to have:

- It should be flexible enough to express any information anyone can think of.

Obviously, each reviewer has different things to say about a given Nikon camera, and whatever he/she wants to say, the standard has to provide a way to allow it. Perhaps the graph shown in Fig. 2.2 is a possible choice – any new information can be added to this graph freely: just grow it as you wish.

And to represent this graph as structured information is not as difficult as you think: the tabular notation shown in Table 2.2 is exactly equivalent to the graph shown in Fig. 2.2.

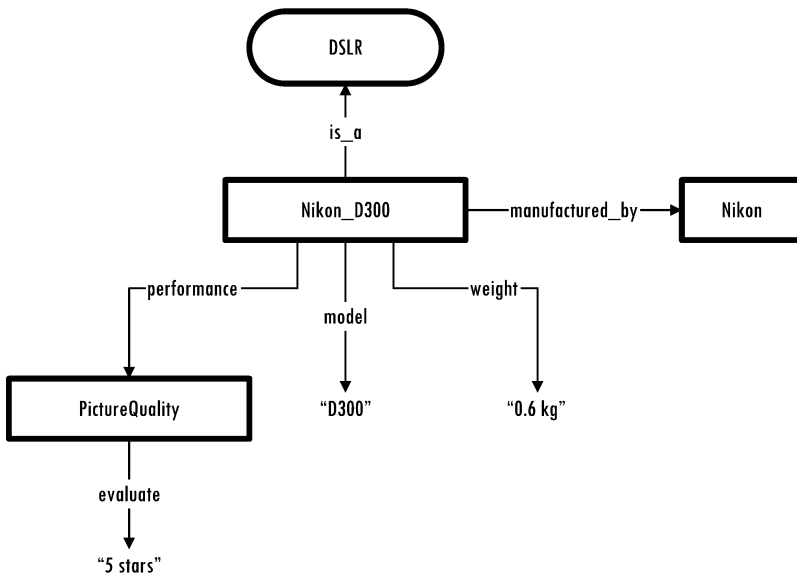


Fig. 2.2 A graph is flexible and can grow easily

Table 2.2 A tabular notation of the graph in Fig. 2.2

Start node	Edge label	End node
Nikon_D300	is_a	DSLR
Nikon_D300	manufactured_by	Nikon
Nikon_D300	performance	PictureQuality
Nikon_D300	model	“D300”
Nikon_D300	weight	“0.6 kg”
PictureQuality	evaluate	“5 stars”

More specifically, each row in the table represents one arrow in the graph, including the start node, the edge with the arrow, and the end node. The first column, therefore, has the name of the start node, the second column has the label of the edge, and the third column has the name of the end node. Clearly, no matter how the graph grows and no matter how complex it grows into, Table 2.2 will always be able to represent it correctly.

- It should provide a mechanism to connect the distributed information (knowledge) over the Web.

Now that every reviewer can publish his/her review freely and a given review can be represented by a graph as we have discussed above, the standard has to provide a way so that our application, when visiting each review graph, is able to decide precisely which product this review is talking about. After all, reviews created by reviewers are distributed all over the Web, and different reviewers can use different names for exactly the same product. For example, one reviewer can call it “Nikon D300,” the other reviewer can use “Nikon D-300,” and the next one simply names it “D300.” Our standard has to provide a way to eliminate this ambiguity so our application can process the reviews with certainty.

- You can think of more requirements?

Yes, there are probably more requirements you would like to add to this standard, but you have got the point. And, as you have guessed, W3C has long realized the need for such a standard, and the standard has been published and called RDF.

So, in plain English, we can define RDF as follows:

RDF is a standard published by W3C, and it can be used to represent distributed information/knowledge in a way that computer applications can use and process in a scalable manner.

At this point, the above definition about RDF is good enough for us to continue. With more and more understanding about RDF, the following will become more and more obvious to you:

- RDF is the basic building block for supporting the vision of the Semantic Web.
- RDF is for the Semantic Web what HTML has been for the Web.

And the reason of RDF being the building block for the Semantic Web lies in the fact that knowledge represented using RDF standard is structured, i.e., it is machine understandable. This further means that RDF allows interoperability among applications exchanging machine-understandable information on the Web, and this, as you can tell, is the fundamental idea of the Semantic Web.

2.2 The Abstract Model of RDF

In the previous section, we have mentioned the six documents composing the RDF specification (see Table 2.1). These documents all together describe different aspects of RDF. One fundamental concept of RDF is its abstract model that is used to represent knowledge about the world. In this section, we will learn this abstract model in detail.

2.2.1 *The Big Picture*

Before we get into the details, let us first take a look at the big picture of this abstract model, so it will be easier for you to understand the rest of its content.

The basic idea is straightforward: RDF uses this abstract model to decompose information/knowledge into small pieces, with some simple rules about the semantics (meaning) of each one of these pieces. The goal is to provide a general method that is simple and flexible enough to express any fact, yet structured enough that computer applications can operate with the expressed knowledge.

This abstract model has the following key components:

- statement
- subject and object resources
- predicate

And we will now discuss each one of these components, and we will then put them together to gain understanding of the abstract model as a whole.

2.2.2 *Statement*

As we have discussed, the key idea of RDF's abstract model is to break information into small pieces, and each small piece has clearly defined semantics so that machine can understand it and do useful things with it.

Now, using RDF's terminology, a given small piece of knowledge is called a *statement*, and the implementation of the above key idea can be expressed as the following rule:

Rule #1:

Knowledge (or information) is expressed as a list of statements, each statement takes the form of Subject-Predicate-Object, and this order should never be changed.

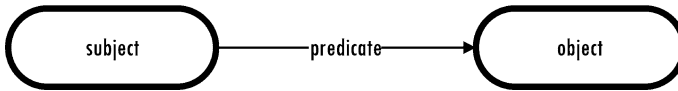


Fig. 2.3 Graph structure of RDF statement

Therefore, an RDF statement must have the following format:

```
subject predicate object
```

where the `subject` and `object` are names for two things in the world, with the `predicate` being the name of a relation that connects these two things. Figure 2.3 shows the graph structure of a statement.

Note that Fig. 2.3 shows a directed graph: the subject is contained in the oval on the left, the object is the oval on the right, and the predicate is the label on the arrow, which points from the subject to the object.

With this said, the information contained in Table 2.2 can be expressed as the following statements shown in List 2.1.

List 2.1 Expressing Table 2.2 as a collection of RDF statements

subject	predicate	object
Nikon_D300	is_a	DSLR
Nikon_D300	manufactured_by	Nikon
Nikon_D300	performance	PictureQuality
Nikon_D300	model	"D300"
Nikon_D300	weight	"0.6 kg"
PictureQuality	evaluate	"5 stars"

Note that since a statement always consists of three fixed components, it is also called a *triple*. Therefore, in the world of RDF, each statement or triple represents a single fact; a collection of statements or triples represents some given piece of information or knowledge; and a collection of statements is called an *RDF graph*.

Now, is this abstract model flexible enough to represent any knowledge? The answer is yes, as long as that given knowledge can be expressed as a labeled and directed graph as shown in Fig. 2.2. And clearly, any new fact can be easily added to an existing graph to make it more expressive. Furthermore, without any loss to its original meaning, any such graph can be represented by a tabular format as shown in Table 2.2, which can then be expressed as a collection of RDF statements as shown in List 2.1, representing a concrete implementation of the RDF abstract model.

For any given RDF statement, both its `subject` and `object` are simple names for things in the world, and they are said to *refer to* or *denote* these things. Note that these things can be anything, concrete or abstract. For example, the first statement in List 2.1 has both its `subject` and `object` referring to concrete things, whilst the third statement in List 2.1 has its `object` referring to `PictureQuality`, an abstract thing (concept).

In the world of RDF, the thing that a given `subject` or `object` denotes, be it concrete or abstract, is called *resource*. Therefore, a resource is anything that is being described by RDF statements.

With this said, both `subject` and `object` in a statement are all names for resources. The question now is how do we come up with these names? This turns out to be a very important aspect of the RDF abstract model. Let us discuss this in detail in the next section.

2.2.3 Resource and Its URI Name

Let us go back to List 2.1, which contains a list of statements about Nikon D300 camera as a resource in the real world. Imagine it is a review file created by one of the reviewers, and this review is intended to be published on the Web.

Now, once this review is put on the Web, the resource names in this review, such as `Nikon_D300`, will present a problem.

More specifically, it is quite possible that different reviewers may use different names to represent the same resource, namely, Nikon D300 camera in this case. For example, one might use `Nikon-D300` instead of `Nikon_D300`. Even such a small difference will become a big problem for an application that tries to aggregate the reviews from different reviewers: it does not know these two reviews are in fact evaluating the same resource.

On the flip side of the coin, it is also possible that two different documents may have used the same name to represent different resources. In other words, a single name has different meanings. Without even seeing any examples, we all understand this semantic ambiguity is exactly what we want to avoid in order for any application to work correctly on the Web.

The solution proposed by RDF's abstract model is summarized in Rule #2 as follows:

Rule #2:

The name of a resource must be global and should be identified by Uniform Resource Identifier (URI).

We are all familiar with URL (*Uniform Resource Locator*), and we have been using it all the time to locate a Web page we want to access. The reason why we can use URL to locate a Web resource is because it represents the network location of this given Web resource.

However, there is some subtle fact about URL that most of us are not familiar with: URL is often used to identify a Web resource that can be *directly* retrieved on the Web. For example, my personal home page has a URL as given by the following:

`http://www.liyangyu.com`

This URL is used not only to identify my home page, but also to retrieve it from the Web.

On the other hand, there are also lots of resources in the world that can be identified on the Web, but cannot be directly retrieved from the Web. For example, I myself as a person, can be identified on the Web, but cannot be directly retrieved from the Web. Similarly, a Nikon D300 camera can be identified on the Web, yet we cannot retrieve it from the Web. Therefore, for these resources, we cannot simply use URLs to represent them.

Fortunately, the Web provides a more general form of identifier for this purpose, and it is called the Uniform Resource Identifier (URI). In general, URLs can be understood as a particular kind of URI. Therefore, a URI can be created to identify anything that can be retrieved directly from the Web and also to represent anything that is not network accessible, such as a human being, a building, or even an abstract concept that does not physically exist, such as the picture quality of a given camera.

The reason why RDF's abstract model decides to use URIs to identify resources in the world should become obvious to you at this point. RDF model has to be extremely flexible since anyone can talk about anything at any time; it does not matter whether you can retrieve that resource on the Web or not. Furthermore, since any collection of RDF statements is intended to be published on the Web, using URIs to identify the subjects and objects in these statements is simply a natural fit.

Another benefit of using URIs to represent subject and object resources is related to their global uniqueness. Imagine we can collect all the URIs in the whole world, and let us call this collection the space of all names. Clearly, we can partition this whole name space into different sections simply by looking at their owners. For example, the organization W3C is the owner for all URIs that start with <http://www.w3c.org/>. And by convention, only W3C will create any new URI that starts with <http://www.w3c.org/>. This guarantees the global uniqueness of URIs and certainly prevents name clashes. If you create a URI using this convention, you can rest assured no one will use the same URI to denote something else.

All these said, how does a URI look like? In the world of RDF, by convention, there are two different types of URI we can use to identify a given resource, namely *hash URI* and *slash URI*. A slash URI is simply a normal URI that we are all familiar with; and a hash URI consists of the following components:

normal URI + # + fragment identifier

For example, to identify Nikon D300 as a resource on the Web, List 2.2 uses both the hash URI and the slash URI.

List 2.2 Use URI to identify Nikon D300 on the Web as a resource

```
http://www.liyangyu.com/camera/Nikon_D300
http://www.liyangyu.com/camera#Nikon_D300
```

The first URI in List 2.2 is a slash URI, and the second one is a hash URI. For this hash URI, its normal URI is given by <http://www.liyangyu.com/camera>, and its fragment identify is given by `Nikon_D300`.

Note that at times a hash URI is also called a *URI reference* or *URIref*. At the time of this writing, hash URI seems to be the name that is more and more widely used.

Now, an obvious question is, what is the difference between a hash URI and a slash URI? Or, when naming a given resource, should we use a hash URI or a slash URI?

In order to answer this question, we in fact have to answer another question first: if we type the URIs contained in List 2.2 (both the hash one and the slash one) into a Web browser, do we actually get anything back? Or, should we be expecting to get anything back at all?

Before the beginning of 2007, there was no expectation that actual content should be served at that location, the reason being URIs do not require the entities being identified to be actually retrievable from the Web. Therefore, the fact that URIs look like a Web address is totally incidental, they are merely verbose names for resources.

However, since early 2007, especially with the development of Linked Data project, dereferencing URIs in RDF models should return some content back, so that both human readers and applications can make use of the returned information.

You will see more about Linked Data project and understand more about URIs in [Chap. 11](#). For now, it is important to remember that URIs in RDF models should be dereferencable URIs. Therefore, if you mint a URI, you are actually required to put something at that address so that RDF clients can access that page and get some information back.

With this new requirement, the difference between a hash URI and a slash URI starts to become more significant. Since you are going to see all the details in [Chap. 11](#), let us simply state the conclusion here without too much explanation: it is easier to make sure a hash URI is also a dereferencable URI, since you can easily accomplish this without any content negotiation mechanism. However, to make a slash URI dereferencable, content negotiation is normally needed.

With all these said, for the rest of this chapter, we are going to use hash URI. Furthermore, if we do create a new URI, we will not worry about serving content at that location – you will learn how to do that in [Chap. 11](#).

Now, with the understanding that all the resources should be named by using URIs, we can revisit List 2.1 and rename all the resources there. List 2.3 shows the resources and their URI names.

List 2.3 Using URIs to name resources

Original name	URI name
Nikon_D300	http://www.liyangyu.com/camera#Nikon_D300
DSLR	http://www.liyangyu.com/camera#DSLR
Nikon	http://www.dbpedia.org/resource/Nikon
PictureQuality	http://www.liyangyu.com/camera#PictureQuality

Note that all the new URIs we have created contain the following domain:

<http://www.liyangyu.com/>

except the URI for Nikon, the manufacturer of the camera. And this URI looks like this:

```
http://www.dbpedia.org/resource/Nikon
```

In fact, we did not coin this URI, and it is an existing one. So why should we use an existing URI to represent Nikon? The reason is very simple: if a given resource has a URI that identifies it already, we should reuse this existing URI whenever we can. In our case, we happen to know the fact that the above URI created by DBpedia project⁴ (DBpedia is a well-known application in the world of the Semantic Web; you will see more details about it in [Chap. 10](#)) does represent Nikon, and it is indeed the same Nikon we are talking about. Therefore, we have decided to use it instead of inventing our own.

This does open up another whole set of questions. For example, is it good to always reuse URIs, or should we sometimes invent our own? If reuse is desirable, then for a given resource, how do we know if there exists some URI already? How do we find it? What if there are multiple URIs existing for this single resource?

At this point, we are not going into the details of the answers to these questions, since they are all covered in later chapters. For now, one thing important to remember is to always reuse URIs and only invent your own if you absolutely have to.

And as you can tell, for the rest of the resources in List 2.3, we have simply chosen to invent our own URIs, because the main goal here is to show you the concept of RDF abstract model. If we were to build a real project about reviewing cameras, we would have searched for existing URIs first (details presented in [Chap. 11](#)). For your information, the following is an existing URI that represents Nikon D300 camera. Again, this URI is minted by DBpedia project:

```
http://dbpedia.org/resource/Nikon_D300
```

Also note that both URIs created by DBpedia, i.e., the one representing Nikon and the one identifying Nikon D300 camera, are all slash URIs. The URIs that we have created in List 2.3 are all hash URIs.

Now, before we can re-write the statements listed in List 2.1, we do have one more issue to cover: if we use URIs to represent resources as required by RDF abstract model, all the resources will inevitably have fairly long names. This is not quite convenient and not quite readable either.

The solution to this issue is quite straightforward: a full URI is usually abbreviated by replacing it with its XML *qualified name* (*QName*). Recall in the XML world, a QName contains a *prefix* that maps to a namespace URI, followed by a colon, and then a *local name*. Using our case as an example, we can declare the two namespace prefixes as shown in List 2.4.

⁴<http://dbpedia.org/About>

List 2.4 Namespace prefixes for our example review

Prefix	Namespace
myCamera	http://www.liyangyu.com/camera#
dbpedia	http://www.dbpedia.org/resource/

And now, the following full URI

http://www.liyangyu.com/camera#Nikon_D300

can be written as

myCamera:Nikon_D300

and similarly, the full URI

http://www.dbpedia.org/resource/Nikon

can be written as

dbpedia:Nikon

As you will see later in this chapter, there are different serialization formats for RDF models, and the precise rules for abbreviation depend on the RDF serialization syntax being used. For now, this QName notation will be fine. And remember, namespaces process no significant meanings in RDF, they are merely a tool to abbreviate long URI names.

Now we can re-write the statements in List 2.1. After replacing the simple names we have used in List 2.1, the new statements are summarized in List 2.5.

List 2.5 RDF statements using URIs as resource names

subject	predicate	object
myCamera:Nikon_D300	is_a	myCamera:DSLR
myCamera:Nikon_D300	manufactured_by	dbpedia:Nikon
myCamera:Nikon_D300	performance	myCamera:PictureQuality
myCamera:Nikon_D300	model	"D300"
myCamera:Nikon_D300	weight	"0.6 kg"
myCamera:PictureQuality	evaluate	"5 stars"

Looking at List 2.5, you might start to think about the predicate column: do we have to use URI to name predicate as well? The answer is yes, and it is indeed very important to do so. Let us discuss this more in the next section.

2.2.4 Predicate and Its URI Name

In a given RDF statement, predicate denotes the relation between the subject and object. RDF abstract model requires the usage of URIs to identify predicates, rather than using strings (or words) such as “has” or “is_a” to identify predicates.

With this said, we can change rule #2 to make it more complete:

Rule #2:

The name of a resource must be global and should be identified by Uniform Resource Identifier (URI). The name of predicate must also be global and should be identified by URI as well.

Using URIs to identify predicates is important for a number of reasons. The first reason is similar to the reason why we should use URIs to name subjects and objects. For example, one group of reviewers who reviews cameras may use string `model` to indicate the fact that Nikon D300 has D300 as its model number, and another group of reviewers who mainly review television sets could also have used `model` to mean the specific model number of a given TV set. A given application that sees these `model` strings will have difficulty in distinguishing their meanings. On the other hand, if the predicates for the camera reviewers and TV reviewers are named, respectively, as follows:

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

it will then be clear to the application that these are distinct predicates.

Another benefit of using URIs to name predicates comes from the fact that this will enable the predicates to be treated as resources as well. This in fact has a far-reaching effect down the road. More specifically, if a given predicate is seen as a resource, we can then add RDF statements with this predicate's URI as subject, just as we do for any other resource. This means that additional information about the given predicate can be added. As we will see in later chapters, by adding this additional information, we can specify some useful fact about this predicate. For example, we can add the fact that this given predicate is the same as another predicate, or it is a sub-predicate of another predicate, or it is an inverse predicate of another predicate, and so on. This additional information turns out to be one of the main factors responsible for the reasoning power provided by RDF models, as you will see in later chapters.

The third benefit that will also become more obvious later is the fact that using URIs to name subjects, predicates, and objects in RDF statements promotes the development and use of shared vocabularies on the Web. Recall that we have been using the following URI to denote Nikon as a company that has manufactured Nikon D300:

```
http://www.dbpedia.org/resource/Nikon
```

Similarly, if we could find an existing URI that denotes `model` as a predicate, we could have used it instead of inventing our own. In other words, by discovering and using vocabularies already used by others to describe resources implies a shared understanding of those concepts, and that will eventually make the Web much more machine friendly. Again, we will discuss this more in the chapters yet to come.

Now, with all these said, let us name our predicates as shown in List 2.6.

List 2.6 Using URIs to name predicates

Original name	URI name
<code>is_a</code>	<code>http://www.liyangyu.com/camera#is_a</code>
<code>manufactured_by</code>	<code>http://www.liyangyu.com/camera#manufactured_by</code>
<code>performance</code>	<code>http://www.liyangyu.com/camera#performance</code>
<code>model</code>	<code>http://www.liyangyu.com/camera#model</code>

```
weight      http://www.liyangyu.com/camera#weight
evaluate    http://www.liyangyu.com/camera#evaluate
```

With these new predicate names, List 2.5 can be re-written. For example, the first statement can be written as the following:

```
subject:   myCamera:Nikon_D300
predicate: myCamera:is_a
object:    myCamera:DSLR
```

You can finish the rest of the statements in List 2.5 accordingly.

So far at this point, we have covered two basic rules about the abstract RDF model. Before we move on to other aspects of the abstract model, we would like to present a small example to show you the fact that these two rules have already taken you farther than you might have realized.

2.2.5 RDF Triples: Knowledge That Machine Can Use

Let us take a detour here, just to see how RDF statements (triples) can be used by machines. With the statements listed in List 2.5, let us ask the machine the following questions:

- What predicates did the reviewer use to describe Nikon D300?
- What performance measurements have been used for Nikon D300?

The first question can be expressed in the following RDF format:

```
question = new RDFStatement();
question.subject    = myCamera:Nikon_D300;
question.predicate = myCamera:*;
```

Note that `myCamera:*` is used as a wild card. The pseudo-code in List 2.7 can help the computer to get the question answered.

List 2.7 Pseudo-code to answer questions

```
// format my question
question = new RDFStatement();
question.subject    = myCamera:Nikon_D300;
question.predicate = myCamera:*;

// read all the review statements and store them in statement
array
RDFStatement[] reviewStatements = new RDFStatement[6];
reviewStatements[0].subject    = myCamera:Nikon_D300;
reviewStatements[0].predicate = myCamera:is_a;
reviewStatements[0].object     = myCamera:DSLR;
```

```

reviewStatements[1].subject    = myCamera:Nikon_D300;
reviewStatements[1].predicate = myCamera:manufactured_by;
reviewStatements[1].object    = dbpedia:Nikon;
reviewStatements[2].subject    = myCamera:Nikon_D300;
reviewStatements[2].predicate = myCamera:performance;
reviewStatements[2].object    = myCamera:PictureQuality;
reviewStatements[3].subject    = myCamera:Nikon_D300;
reviewStatements[3].predicate = myCamera:model;
reviewStatements[3].object    = "D300";
reviewStatements[4].subject    = myCamera:Nikon_D300;
reviewStatements[4].predicate = myCamera:weight;
reviewStatements[4].object    = "0.6 kg";
reviewStatements[5].subject    = myCamera:PictureQuality;
reviewStatements[5].predicate = myCamera:evaluate;
reviewStatements[5].object    = "5 stars";

// answer the question!
foreach s in reviewStatements[] {
    if ( (s.subject==question.subject || question.subject=='*') &&
        (s.predicate==question.predicate || question.predicate ==
         '*' ) ) {
        System.out.println(s.predicate.toString());
    }
};

```

Running this code will give us the following answer:

```

myCamera:is_a
myCamera:manufactured_by
myCamera:performance
myCamera:model
myCamera:weight

```

meaning that the reviewer has defined all the above predicates for Nikon D300.

Now to answer the second question, all you have to change is the question itself:

```

question = new RDFStatement();
question.subject    = myCamera:Nikon_D300;
question.predicate = myCamera:performance;

```

and also change the output line in List 2.7 to the following:

```

System.out.println(s.subject.toString());

```

And the answer will be returned to you:

```

myCamera:PictureQuality

```

meaning that the reviewer has used `myCamera:PictureQuality` as the performance measurement to evaluate Nikon D300.

In fact, try out some other questions, such as who is the manufacturer of Nikon D300 and what model number does it have. You will see the code does not have

to change much at all. And clearly, based on the knowledge presented in the RDF statements (Table 2.2), the machine can indeed conduct some useful work for us. It is also not hard for us to imagine some more interesting examples if we can add more RDF statements with more complex predicates and objects.

2.2.6 RDF Literals and Blank Node

We are not totally done with the abstract RDF model yet. In this section, we will describe two important components of abstract model: RDF literals and blank node. And first, let us summarize all the terminologies we have learned so far.

2.2.6.1 Basic Terminologies So Far

One difficulty about learning RDF comes from the fact that it has lots of terminologies and synonyms. To make our learning easier, let us summarize these terminologies and their synonyms in this section.

So far, we have learned the following:

- subject*: used to denote *resource* in the world, must be identified by URI, and also called *node* or *start node* in an RDF graph;
- object*: used to denote *resource* in the world, must be identified by URI, and also called *node* or *end node* in an RDF graph;
- predicate*: used to denote the relation between subject and object, must be identified by URI, also called *edge* in an RDF graph.

This summary needs to grow for sure. For example, you might have already noted a long time ago that the following statement does not completely follow the above summary:

subject	predicate	object
myCamera:Nikon_D300	myCamera:model	"D300"

since its object obviously takes a string as its value, instead of another resource. Also, the string value has nothing to do with URIs. In addition, there are two more similar statements in our list:

subject	predicate	object
myCamera:Nikon_D300	myCamera:weight	"0.6 kg"
myCamera:PictureQuality	myCamera:evaluate	"5 stars"

Before we explain all these issues, let us see something new first:

predicate: also called *property*, i.e., predicate and property are synonyms.

This is quite an intuitive change. To see this, consider the following statement:

subject	predicate	object
myCamera:Nikon_D300	myCamera:is_a	myCamera:DSLR

which can be read as follows:

resource `myCamera:Nikon_D300` and resource `myCamera:DSLR` are related by a predicate called `myCamera:is_a`.

Now, besides understanding predicate as a relation between the subject and object resource, we can also perceive it as putting some constraint on one of the attributes (properties) of the subject resource. In our case, the `myCamera:is_a` attribute (property) of the subject will take resource `myCamera:DSLR` as its value. With this said, the above statement can be read in a different way:

`myCamera:is_a` is a *property* of resource `myCamera:Nikon_D300` and resource `myCamera:DSLR` is the *value* of this property.

Now we can change the names of the components in an RDF statement to make it more consistent with the above reading:

resource	property	value
myCamera:Nikon_D300	myCamera:is_a	myCamera:DSLR

and with this said, Fig. 2.3 is completely equivalent to Fig. 2.4.

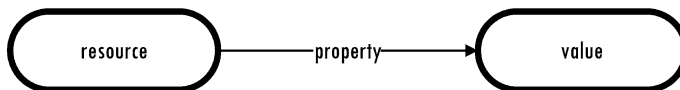


Fig. 2.4 Graph structure of RDF statement (equivalent to Fig. 2.3)

And now,

object: also called property *value*, and both literal strings and resources can be used as property value. If a resource is used as its value, this resource may or may not be identified by a URI. If it is not represented by a URI, it is called a *blank node*.

Note that the object in one statement can become the subject in another statement (such as `myCamera:PictureQuality`, for example). Therefore, a blank node object in one statement can become a blank node subject in another statement.

To summarize what we have learned:

subject: can be URI named resource, or a blank node;
object: also called property value, can be URI named resource, literal or blank node;
predicate: also called property, must be URI named resource.

And now we understand why we can have statements that use string values as their objects. Let us move on to learn more about literals and blank nodes; they are all important concepts in abstract RDF model.

2.2.6.2 Literal Values

RDF literals are simple raw text data, and they can be used as property values. As we have seen in List 2.7, "D300," "0.6 kg," and "5 stars" are all examples of literal values. Other common examples include people's names and book ISBN numbers.

A literal value can be optionally localized by attaching a language tag, indicating in which language the raw text is written, for example, "Dr."@en, the literal value Dr. with an English language tag, or "Dott."@it, the same with an Italian language tag.

A literal value can also be optionally typed by using a URI that indicates a datatype, and this datatype information can be used by RDF document parser to understand how to interpret the raw text. The datatype URI can be any URI, but quite often you will see that those datatypes defined in XML Schema are being used.

To add a datatype to a literal value, put the literal value in quotes and then use two carets, followed by the datatype URI. List 2.8 shows some examples of using both the language tag and datatype URIs.

List 2.8 Examples of using language tags and datatypes on RDF literal values

```
"D300"  
"D300"@en  
"D300"@it  
"D300"^^<http://www.w3.org/2001/XMLSchema#string>
```

In List 2.8, the first line uses simple raw text without any language tag and any datatype, it is therefore an un-typed literal value without any language tag. Lines 2 and 3 are also un-typed literal values, but they do have language tags. Line 4 is a typed literal value, and its full datatype URI is also written out.

Note that an un-typed literal, regardless of whether it has a language tag or not, is completely different from a typed literal. Therefore, the literal value on line 1 and the literal value on line 4 are considered two different things and have nothing related to each other at all. In fact, all the four literal values in List 2.8 are not related; therefore the four statements in List 2.9 are completely different, and no one can be inferred from the others.

List 2.9 Completely different statements (all the property values are different)

```
resource: myCamera:Nikon_D300  
property: myCamera:model  
value: "D300"  
  
resource: myCamera:Nikon_D300  
property: myCamera:model  
value: "D300"@en
```

```

resource: myCamera:Nikon_D300
property: myCamera:model
value: "D300"@it

resource: myCamera:Nikon_D300
property: myCamera:model
value: "D300"^^<http://www.w3.org/2001/XMLSchema#string>

```

For a typed literal, the purpose of its datatype URI is to tell the parser or an application how to map the raw text string to values. It is therefore possible that two typed literals that appear different can be mapped to the same value. For example, the two statements in List 2.10 are equivalent.

List 2.10 The two statements are identical

```

resource: myCamera:Nikon_D300
property: myCamera:weight
value: "0.6"^^<http://www.w3.org/2001/XMLSchema#float>

resource: myCamera:Nikon_D300
property: myCamera:weight
value: "0.60"^^<http://www.w3.org/2001/XMLSchema#float>

```

We will discuss more about datatypes and typed literals in later sections. But before we move on, here is one more thing to remember: literals are only used as object values; they can never be used as subjects.

2.2.6.3 Blank Nodes

A *blank node* is a node (denotes either a subject or an object) that does not have a URI as its identifier, i.e., a nameless node. It in fact happens quite often in RDF models and is also called an *anonymous node* or a *bnode*. List 2.11 shows one example of a blank node.

List 2.11 A blank node example

resource	property	value
myCamera:Nikon_D300	myCamera:reviewed_by	_:anon0
_:anon0	foaf:givenname	"liyang"
_:anon0	foaf:family_name	"yu"

First off, `foaf:givenname` and `foaf:family_name` are just QNames, and they have used a new namespace, namely, `foaf`, that you have not seen yet. At this point, understand that both `foaf:givenname` and `foaf:family_name` are simply abbreviated URIs that represent properties. And obviously, these two properties are used to denote a person's first and last names.

Now, the three statements in List 2.11 have expressed the following fact:

this Nikon D300 camera (`myCamera:Nikon_D300`) is reviewed by (`myCamera:reviewed_by`) some specific resource in the world. This resource has a property called `foaf:givename` whose value is `liyng`; it also has a property called `foaf:family_name` whose value is `yu`.

And obviously, the blank node here represents this specific resource. Note that when we say a node is a blank node, we refer to the fact that it does not have a URI as its name. However, in real RDF documents, it will most likely be assigned a local identifier so that it could be referred within the same document scope. In our example, this local identifier is given by `_:anon0`.

By now, we all know that a list of RDF statements can be represented by an RDF graph (and vice versa). For example, Fig. 2.5 shows the graph generated by representing the statement in List 2.5.

Now, if we add the statements in List 2.11 to the graph shown in Fig. 2.5, we get Fig. 2.6.

As you can tell, the local name of the blank node is not included in the graph, and it is now a real blank node – probably that is why the name was created in the first place.

The main benefit of using blank nodes is the fact that blank node provides a way to model the so-called *n-ary (n-way) relationship* in RDF models.

To see this, first understand that RDF only models *binary* relationships. For example, the following statement

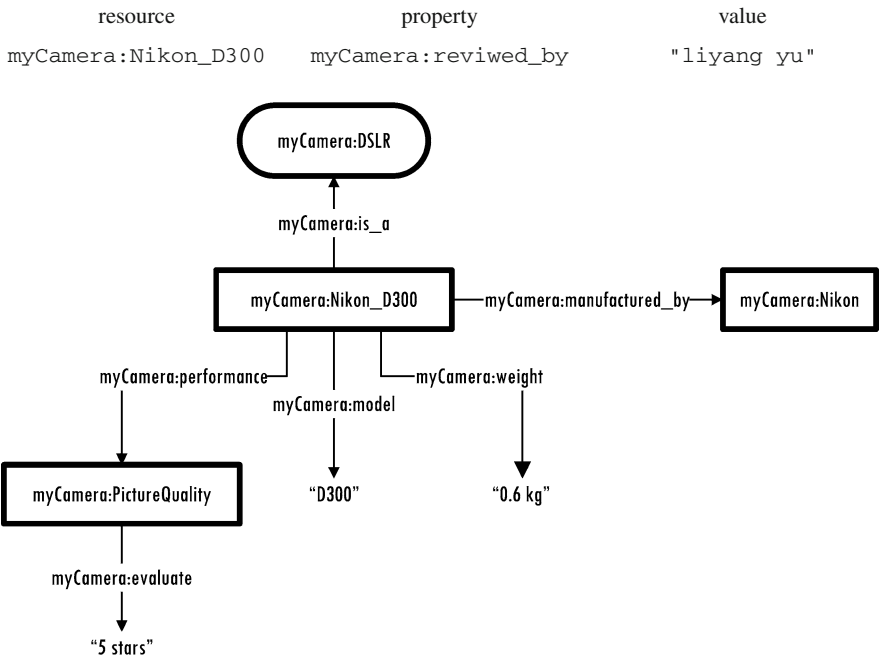


Fig. 2.5 Graph representation of the statements in List 2.5

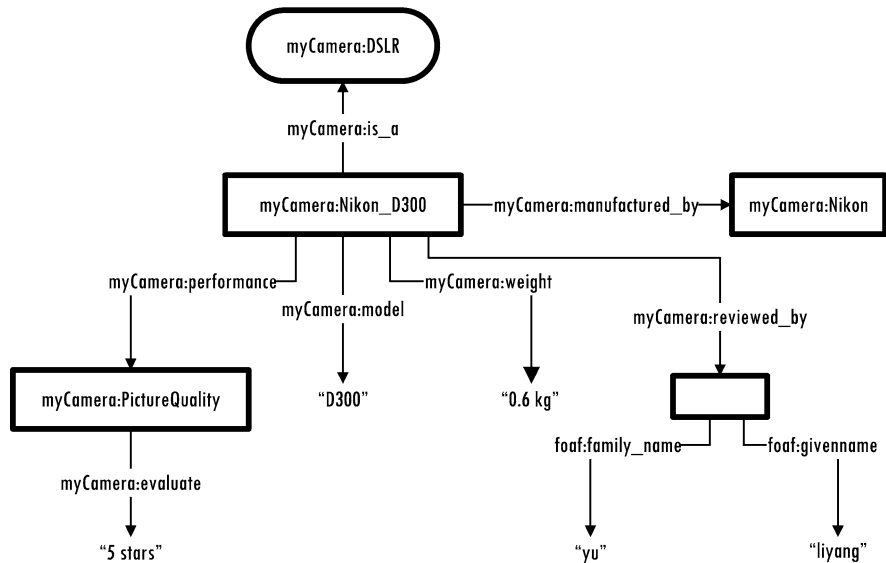


Fig. 2.6 Graph representation of the statements in List 2.5 together with List 2.11

represents a binary relationship, i.e., the relationship between a camera and the literal string that represents its reviewer. Now, there could be another reviewer who has the same name. In order to eliminate this ambiguity, we decide that we will add more details to the reviewer. This time, not only will we spell out the first name and the last name (as in List 2.11), but we will also add an e-mail address of the reviewer, so we can be quite certain whom we are referring to.

However, by doing so, the camera is no longer related to a single literal string; instead, it is related to a collection of components (a last name, a first name, and an e-mail address). In other words, the original binary relationship has now become an *n*-ary relationship (*n* = 3, to be more precise). So how does RDF model this *n*-way relationship?

The solution is to create another resource to represent this collection of components, and the original subject keeps its binary relationship to this newly created resource. Meanwhile, each one of the components in this collection can become a separate property of the new resource, as shown in List 2.12.

List 2.12 Modeling a three-way relationship between camera and reviewer

resource	property	value
myCamera:Nikon_D300	myCamera:reviewed_by	new_resource_URI
new_resource_URI	foaf:givenname	"liyang"
new_resource_URI	foaf:family_name	"yu"
new_resource_URI	foaf:mbox	<mailto:liyang910@yahoo.com>

Again, foaf:mbox is just another QName that represents e-mail address property (more about foaf namespace in later chapters). Also, new_resource_URI is

the new URI we have created and it represents the collection of the three components. The important fact is that we have now successfully modeled a three-way relationship between a given camera and its reviewer.

As you can easily imagine, there will be lots of similar scenarios like this in the real world, where we will have to model n -ary relationships. Clearly, for each such n -ary relationship, there will be a new URI invented, which means we have to invent numerous URIs such as `new_resource_URI`. However, most of these new URIs will never be referred from outside the graph; it is therefore not necessary for them to have URIs at all. This is exactly the concept of blank node, and this is how blank node can help us to model a given n -ary relationship.

Again, as we have mentioned, most RDF processors will automatically assign a local node identifier to a blank node, which is needed when the RDF statements are written out. In addition, other statements within the same document can make reference to this blank node if necessary. Of course, a blank node is not accessible from outside the graph, and it will not be considered when data aggregation is performed.

Before we move on, here is one more thing to remember: blank nodes can only be used as subjects or objects; they cannot be used as properties.

2.2.7 A Summary So Far

Up to this point, we have covered the basic components of abstract RDF model. Before we move on, the following is a summary of what we have learned so far:

- RDF offers an abstract model and framework that tells us how to decompose information/knowledge into small pieces.
- One such small piece of information/knowledge is represented as a statement which has the form (subject, predicate, object). A statement is also called a triple.
- A given RDF model can be expressed either as a graph or as a collection of statements or triples.
- Each statement maps to one edge in the graph. Therefore, the subject and object of a given statement are also called nodes, and its predicate is also called edge.
- Subjects and objects denote resources in the real world. Predicates denote the relationship between subjects and objects.
- Predicates are also called properties, and objects are also called property values. Therefore, a statement also has the form (resource, property, propertyValue).
- URIs are used to name resources and properties. For a given resource or property, if there is an existing URI to name it, you should reuse it instead of inventing your own.
- An RDF statement can only model a binary relationship. To model an n -ary relationship, intermediate resources are introduced and blank nodes are quite often used.
- An object can take either a simple literal or another resource as its value. If a literal is used as its value, the literal can be typed or un-typed, and it can also have an optional language tag.

If you are comfortable with the above, move on. Otherwise, review the material here in this section, and make sure you understand it completely.

2.3 RDF Serialization: RDF/XML Syntax

The RDF data model we have covered so far provides an abstract and conceptual framework for describing resources in a way that machine can process. The next step is to define some serialization syntax for creating and reading concrete RDF models, so applications can start to write and share RDF documents.

The W3C specifications define an XML syntax for this purpose. It is called RDF/XML and is used to represent an RDF graph as an XML document. Note that this is not the only serialization syntax that is being used. For example, Notation 3 (or N3) as a non-XML serialization format is also introduced by W3C and is widely used among the Semantic Web developers. This section will concentrate on RDF/XML syntax only, and other formats will be discussed in later sections.

2.3.1 The Big Picture: RDF Vocabulary

As we have discussed, RDF uses URIs instead of words to name resources and properties. In addition, RDF refers to a set of URIs (often created for a specific purpose) as a *vocabulary*. Furthermore, all the URIs in such a vocabulary normally share a common leading string, which is used as the common prefix in these URIs' QNames. This prefix will often become the namespace prefix for this vocabulary, and the URIs in this vocabulary will be formed by appending individual local names to the end of this common leading string.

In order to define RDF/XML serialization syntax, a set of URIs are created and are given specific meanings by RDF. This group of URIs becomes RDF's own vocabulary of terms, and it is called the *RDF vocabulary*. More specifically, the URIs in this RDF vocabulary all share the following lead strings:

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

By convention, this URI prefix string is associated with namespace prefix `rdf:` and is typically used in XML with the prefix `rdf`. For this reason, this vocabulary is also referred to as the `rdf:` vocabulary.

The terms in `rdf:` vocabulary are listed in List 2.13. Understanding the syntax of RDF/XML means to understand the meaning of these terms and how to use them when creating a concrete RDF model in XML format.

List 2.13 Terms in RDF vocabulary

Syntax names:

```
rdf:RDF, rdf:Description, rdf:ID, rdf:about, rdf:parseType,
rdf:resource, rdf:li, rdf:nodeID, rdf:datatype
```

Class names:

`rdf:Seq`, `rdf:Bag`, `rdf:Alt`, `rdf:Statement`, `rdf:Property`,
`rdf:XMLLiteral`, `rdf:List`

Property names:

`rdf:subject`, `rdf:predicate`, `rdf:object`, `rdf:type`,
`rdf:value`, `rdf:first`, `rdf:rest` `_n` (where `n` is a decimal integer greater than
zero with no leading zeros).

Resource names:

`rdf:nil`

From now on, `rdf:name` will be used to indicate a term from the RDF vocabulary, and its URI can be formed by concatenating the RDF namespace URI and name itself. For example, the URI of `rdf:type` is given as below:

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

2.3.2 Basic Syntax and Examples

As we have discussed, RDF/XML is the normative syntax for writing RDF models. In this section, we will describe RDF/XML syntax, and most of the example statements we are going to use come from List 2.5.

2.3.2.1 `rdf:RDF`, `rdf:Description`, `rdf:about`, and `rdf:resource`

Now, let us start with the first statement in List 2.5:

subject	predicate	object
<code>myCamera:Nikon_D300</code>	<code>myCamera:is_a</code>	<code>myCamera:DSLR</code>

List 2.14 shows the RDF/XML presentation of an RDF model which contains only this single statement:

List 2.14 RDF/XML presentation of the first statement in List 2.5

```

1: <?xml version="1.0"?>
2: <rdf:RDF
3:   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
4:   xmlns:myCamera="http://www.liyangyu.com/camera#">
5:   <rdf:Description
6:     rdf:about="http://www.liyangyu.com/camera#Nikon_D300">
7:     <myCamera:is_a
8:       rdf:resource="http://www.liyangyu.com/camera#DSLR"/>
9:   </rdf:Description>
10: </rdf:RDF>

```

Since this is our very first RDF model expressed in XML format, let us explain it carefully.

Line 1 should look familiar. It says this document is in XML format; it also indicates which version of XML this document is in. Line 2 creates an `rdf:RDF` element, indicating this XML document is intended to represent an RDF model, which ends at the end tag, `</rdf:RDF>`. In general, whenever you want to create an XML document representing an RDF model, `rdf:RDF` should always be the root element of your XML document.

Line 2 also includes an XML namespace declaration by using an `xmlns` attribute, which specifies that prefix `rdf:` is used to represent the RDF namespace URI reference, i.e., <http://www.w3.org/1999/02/22-rdf-syntax-ns#>. Based on the discussion in Sect. 2.3.1, we know that any tag with the form of `rdf:name` will be a term from the RDF vocabulary given in List 2.13. For instance, term `rdf:Description` (on line 4) is taken from the RDF vocabulary, and its URI name should be constructed by concatenating RDF namespace URI reference and local name. Therefore, its URI name is given by the following:

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

Line 3 adds a new `xmlns` attribute which declares another XML namespace. It specifies that prefix `myCamera:` should be used to represent namespace URI given by <http://www.liyangyu.com/camera#>. Any term that has the name `myCamera:name` is therefore a term taken from this namespace.

At this point, the opening `<rdf:RDF>` tag is closed, indicated by the “>” sign at the end of line 3. In general, this is a typical routine for all RDF/XML documents, with the only difference being more or less namespace declarations in different RDF documents.

Now, any statement in a given RDF model is a description of a resource in the real world, with the resource being the subject of the statement. The term, `rdf:Description`, translates this fact into RDF/XML syntax. It indicates the start of a description of a resource, and it uses the `rdf:about` attribute to specify the resource that is being described, as shown in line 4.

In general, this kind of XML node in a given RDF/XML document is called a *resource XML node*. In this example, it represents a subject of a statement. You can understand line 4 as the following:

```
<rdf:Description rdf:about = "URI of the statement's subject">
```

Now, given the fact that tag `rdf:Description` indicates the start of a statement, `</rdf:Description>` must signify the end of a statement. Indeed, line 6 shows the end of our statement.

With this being said, line 5 has to specify the property and property value of the statement. It does so by using a `myCamera:is_a` tag to represent the property. Since the property value in this case is another resource, `rdf:resource` is used to identify it by referring its URI.

Note that line 5 is nested within the `rdf:Description` element; therefore, the property and property value specified by line 5 apply to the resource specified by the `rdf:about` attribute of the `rdf:Description` element.

In general, the node created by line 5 is called a *property XML node*. Clearly, each property XML node represents a single statement. Note that a given property node is always contained within a resource XML node, which represents the subject of the statement.

Now, after all the above discussion, lines 4–6 can be viewed as the following:

```
4:   <rdf:Description rdf:about="URI of the statement's subject">
5:     <predicateURI rdf:resource="URI of the statement's object" />
6:   </rdf:Description>
```

and can be read like this:

This is a description about a resource named `myCamera:Nikon_D300`, which is an instance of another resource, namely, `myCamera:DSLR`.

At this point, we have finished our first RDF/XML document which has only one statement. We will keep adding statements into this document until we have covered all the RDF vocabulary features.

2.3.2.2 `rdf:type` and Typed Nodes

Now, take a look at the statement in List 2.14. In order to express the knowledge that Nikon D300 is a digital SLR, we had to invent a property called `myCamera:is_a`. It is not hard to imagine that this kind of requirement is quite common in other applications as well. For example, we will want to express the fact that a certain resource is a person, another resource is a book, so on and so forth. It then seems reasonable for RDF vocabulary to provide some term just for this purpose, so a given application does not have to invent its own.

In RDF vocabulary, `rdf:type` exists to identify the type of a given resource. List 2.15 shows the term `rdf:type` in use.

List 2.15 Using `rdf:type` to specify the type of a given resource

```
1: <?xml version="1.0"?>
2: <rdf:RDF
2a:   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
3:   xmlns:myCamera="http://www.liyangyu.com/camera#">
4:   <rdf:Description
4a:     rdf:about="http://www.liyangyu.com/camera#Nikon_D300">
5:     <rdf:type
5a:       rdf:resource="http://www.liyangyu.com/camera#DSLR" />
6:   </rdf:Description>
7: </rdf:RDF>
```

This is obviously a better choice: instead of inventing our own home-made property (`myCamera:is_a`), we are now using a common term from the RDF vocabulary. Figure 2.7 shows the graph representation of the statement in List 2.15.

The subject node in Fig. 2.7 is often called a *typed node* in a graph, or *typed node element* in RDF documents. Assigning a type to a resource has far-reaching implication than you might have realized now. As we will see in our later sections

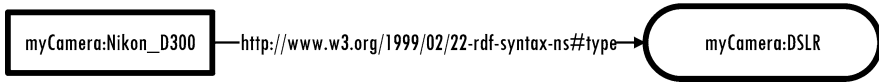


Fig. 2.7 Graph representation of the statement in List 2.15

and chapters, it is one of the reasons why we claim RDF model represents structured information that machine can understand.

In fact, once we have the term `rdf:type` at our disposal, we can often write the statement in List 2.15 in a simpler format without using `rdf:Description`. List 2.16 shows the detail.

List 2.16 A simpler form of List 2.15

```
1: <?xml version="1.0"?>
2: <rdf:RDF
3:     xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
4:     xmlns:myCamera="http://www.liyangyu.com/camera#">
5:     <myCamera:DSLR
6:         rdf:about="http://www.liyangyu.com/camera#Nikon_D300">
7:     </myCamera:DSLR>
8: </rdf:RDF>
```

List 2.16 is equivalent to List 2.15. In fact, most RDF parsers will change List 2.16 back to List 2.15 when they operate on the document. In addition, some developers do believe the format in List 2.15 is clearer.

Now, let us take the rest of the statements from List 2.5 and add them to our RDF/XML document. List 2.17 shows the document after we have added the next statement.

List 2.17 Adding one more statement from List 2.5 to List 2.15

```
1: <?xml version="1.0"?>
2: <rdf:RDF
3:     xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
4:     xmlns:myCamera="http://www.liyangyu.com/camera#">
5:     <rdf:Description
6:         rdf:about="http://www.liyangyu.com/camera#Nikon_D300">
7:         <rdf:type
8:             rdf:resource="http://www.liyangyu.com/camera#DSLR"/>
9:         </rdf:Description>
10:     <rdf:Description
11:         rdf:about="http://www.liyangyu.com/camera#Nikon_D300">
12:         <myCamera:manufactured_by
13:             rdf:resource="http://www.dbpedia.org/resource/Nikon"/>
14:     </rdf:Description>
15: </rdf:RDF>
```



```

11: </rdf:Description>
12:
13: </rdf:RDF>

```

The new statement added is expressed in lines 9–11. With the understanding of the first statement (lines 5–7), this new statement does not require too much explanation. However, we can make this a little bit more concise: since the two statements have the same subject, they can be combined together, as shown in List 2.18.

List 2.18 A simpler form of List 2.17

```

1: <?xml version="1.0"?>
2: <rdf:RDF
3:   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
4:   xmlns:myCamera="http://www.liyangyu.com/camera#">
5:   <rdf:Description
6:     rdf:about="http://www.liyangyu.com/camera#Nikon_D300">
7:     <rdf:type
8:       rdf:resource="http://www.liyangyu.com/camera#DSLR" />
9:     <myCamera:manufactured_by
10:      rdf:resource="http://www.dbpedia.org/resource/Nikon" />
11:   </rdf:Description>
12:
13: </rdf:RDF>

```

Now, moving on to the rest of the statements from List 2.5 does require some new knowledge, which will be explained in the next section.

2.3.2.3 Using Resource as Property Value

The next statement uses a resource called `myCamera:PictureQuality` as the value of its `myCamera:performance` property, which is not something totally new at this point. The two statements in the current RDF document (lines 6 and 7, List 2.18) are all using resources as their objects. However, there is a little bit more about this `myCamera:PictureQuality` resource: it itself has a property that needs to be described, as shown by the last statement in List 2.5.

List 2.19 shows one way to implement this.

List 2.19 Example of using resource as property value

```

1: <?xml version="1.0"?>
2: <rdf:RDF
3:   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
4:   xmlns:myCamera="http://www.liyangyu.com/camera#">
5:   <rdf:Description

```

```

5a:      rdf:about="http://www.liyangyu.com/camera#Nikon_D300">
6:      <rdf:type
7:      <myCamera:manufactured_by
7a:      rdf:resource="http://www.dbpedia.org/resource/Nikon"/>
8:      <myCamera:performance rdf:resource=
8a:      "http://www.liyangyu.com/camera#PictureQuality"/>
9:      </rdf:Description>
10:
11:      <rdf:Description
11a:      rdf:about="http://www.liyangyu.com/camera#PictureQuality">
12:      <myCamera:evaluate>5 stars</myCamera:evaluate>
13:      </rdf:Description>
14:
15: </rdf:RDF>

```

This approach first uses an `rdf:resource` attribute on `myCamera:performance` property, and this attribute points to the URI of the resource that is used at the object of this property (line 8). This object resource is further described separately by using a new `rdf:Description` node at the top level of the document (lines 11–13).

Another way to represent resource as property value is to simply put the description of the object resource into the property XML node that uses this resource as the object value, as shown in List 2.20.

List 2.20 Another format when using resource as property value

```

1: <?xml version="1.0"?>
2: <rdf:RDF
3:      xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
4:      xmlns:myCamera="http://www.liyangyu.com/camera#">
5:      <rdf:Description
6:      <rdf:type
7:      <myCamera:manufactured_by
7a:      rdf:resource="http://www.dbpedia.org/resource/Nikon"/>
8:      <myCamera:performance>
9:      <rdf:Description rdf:about=
9a:      "http://www.liyangyu.com/camera#PictureQuality">
10:      <myCamera:evaluate>5 stars</myCamera:evaluate>
11:      </rdf:Description>
12:      </myCamera:performance>
13: </rdf:Description>
14:
15: </rdf:RDF>

```

Clearly, lines 9–11 map to lines 11–13 in List 2.19. In fact, this pattern can be used recursively until all the resources have been described. More specifically, if `myCamera:PictureQuality` as a resource uses another resource as its property value (instead of literal value “5 stars” as shown in line 10), that resource can again be described inside the corresponding property XML node, so on and so forth.

2.3.2.4 Using Un-typed Literals as Property Values, `rdf:value` and `rdf:parseType`

We move on to the next statement in List 2.5, where a literal string is used as the value of `myCamera:model` property. Again, this is not new. We have learned how to use a literal value as the object of a property XML node (line 10, List 2.20). Specially, the value is simply put inside the XML element.

At this point, List 2.21 shows the document that includes all the statements from List 2.5 so far.

List 2.21 RDF/XML document that includes all the statements from List 2.5

```

1: <?xml version="1.0"?>
2: <rdf:RDF
2a:   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
3:   xmlns:myCamera="http://www.liyangyu.com/camera#">
4:
5:   <rdf:Description
5a:     rdf:about="http://www.liyangyu.com/camera#Nikon_D300">
6:     <rdf:type
6a:       rdf:resource="http://www.liyangyu.com/camera#DSLR"/>
7:     <myCamera:manufactured_by
7a:       rdf:resource="http://www.dbpedia.org/resource/Nikon"/>
8:     <myCamera:performance>
9:       <rdf:Description rdf:about=
9a:         "http://www.liyangyu.com/camera#PictureQuality">
10:        <myCamera:evaluate>5 stars</myCamera:evaluate>
11:      </rdf:Description>
12:    </myCamera:performance>
13:    <myCamera:model>D300</myCamera:model>
14:    <myCamera:weight>0.6 kg</myCamera:weight>
15:  </rdf:Description>
16:
17:</rdf:RDF>

```

Lines 13 and 14 show how literal values are used. For example, line 14 tells us property `myCamera:weight` has a literal value of 0.6 kg.

However, given the fact that the Web is such a global resource itself, it might not be a good idea to use a literal value such as 0.6 kg. When we do this, we in fact assume that anyone who accesses this property will be able to understand

the unit that is being used, which may not be a safe assumption to make. A better or safer solution is to explicitly express the value and the unit in separate property values. In other words, the value of `myCamera:weight` property would need to have two components: the literal for the decimal value and an indication of the unit of measurement (kg). Note in this situation that the decimal value itself can be viewed as the main value of `myCamera:weight` property, whilst the unit component exists just to provide additional contextual information that qualifies the main value.

To implement this solution, we need to model such a qualified property as new structured value. More specifically, a totally separate resource should be used to represent this structured value as a whole. This new resource should have properties representing the individual components of the structured value. In our example, it should have two properties: one for the decimal value, the other for the unit. This new resource will then be used as the object value of the original statement.

RDF vocabulary provides a pre-defined `rdf:value` property just for this use case. List 2.22 shows how to use it.

List 2.22 Using `rdf:value` to represent literal value

```

1: <?xml version="1.0" ?>
2: <rdf:RDF
3:     xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
4:     xmlns:uom="http://www.example.org/units#"
5:     xmlns:myCamera="http://www.liyangyu.com/camera#">
6:   <rdf:Description
7:     rdf:about="http://www.liyangyu.com/camera#Nikon_D300">
8:     <rdf:type
9:       rdf:resource="http://www.liyangyu.com/camera#DSLR"/>
10:    <myCamera:manufactured_by
11:      rdf:resource="http://www.dbpedia.org/resource/Nikon"/>
12:    <myCamera:performance>
13:      <rdf:Description rdf:about=
14:        "http://www.liyangyu.com/camera#PictureQuality">
15:        <myCamera:evaluate>5 stars</myCamera:evaluate>
16:      </rdf:Description>
17:    </myCamera:performance>
18:    <myCamera:model>D300</myCamera:model>
19:    <myCamera:weight>
20:      <rdf:Description>
21:        <rdf:value>0.6</rdf:value>
22:        <uom:units
23:          rdf:resource="http://www.example.org/units#kg"/>
24:        </rdf:Description>
25:      </myCamera:weight>

```

```

21: </rdf:Description>
22:
23: </rdf:RDF>

```

Now, property `myCamera:weight` is using a resource (lines 16–19) as its value. This resource, as we discussed earlier, has two properties. The first property is the pre-defined `rdf:value` property; its value is 0.6 (line 17). The other one is the `uom:units` property defined in the `uom` namespace (line 3). The value of this property is another resource, and <http://www.example.org/units#kg> is the URI of this resource.

Another interesting part of List 2.22 is the name of the resource given by lines 16–19. Note that in line 16, `<rdf:Description>` tag does not have anything like `rdf:about` attribute. Therefore, this resource is an *anonymous* resource (we have discussed the concept of anonymous resource in Sect. 2.2.6.3).

Why is the resource used by `myCamera:weight` property made to be anonymous? Since its purpose is to provide a context for the other two properties to exist, and other RDF documents will have no need to use or add any new details to this resource, there is simply no need to give this resource an identifier.

In RDF models, there is an easier way to implicitly create a blank node. It is considered to be a shorthand method provided by RDF. This involves the usage of `rdf:parseType` keyword from the RDF vocabulary, as shown in List 2.23.

List 2.23 Using `rdf:parseType` to represent literal value

```

1: <?xml version="1.0" ?>
2: <rdf:RDF
3:     xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
4:     xmlns:uom="http://www.example.org/units#"
5:     xmlns:myCamera="http://www.liyangyu.com/camera#">
6:   <rdf:Description
7:     rdf:about="http://www.liyangyu.com/camera#Nikon_D300">
8:     <rdf:type
9:       rdf:resource="http://www.liyangyu.com/camera#DSLR"/>
10:    <myCamera:manufactured_by
11:      rdf:resource="http://www.dbpedia.org/resource/Nikon"/>
12:    <myCamera:performance>
13:      <rdf:Description rdf:about=
14:        "http://www.liyangyu.com/camera#PictureQuality">
15:        <myCamera:evaluate>5 stars</myCamera:evaluate>
16:      </rdf:Description>
17:    </myCamera:performance>
18:    <myCamera:model>D300</myCamera:model>
19:    <myCamera:weight rdf:parseType="Resource">
20:      <rdf:value>0.6</rdf:value>
21:      <uom:units

```

```

17a:         rdf:resource="http://www.example.org/units#kg"/>
18:     </myCamera:weight>
19: </rdf:Description>
20:
21: </rdf:RDF>

```

List 2.23 is identical to List 2.22. `rdf:parseType="Resource"` in line 15 is used as the attribute of the `myCamera:weight` element. It indicates to the RDF parser that the contents of the `myCamera:weight` element (lines 16 and 17) should be interpreted as the description of a new resource (a blank node) and should be treated as the value of property `myCamera:weight`. Without seeing a nested `rdf:Description` tag, the RDF parser creates a blank node as the value of the `myCamera:weight` property and then uses the enclosed two elements as the properties of that blank node. Obviously, this is exactly what we wish the parser to accomplish.

2.3.2.5 Using Typed Literal Values and `rdf:datatype`

We have mentioned typed literal values, but have not had a chance to use them yet in our RDF document. Let us take a look at typed literals in this section.

Line 16 of List 2.23 uses `0.6` as the value of the `rdf:value` property. Here, `0.6` is a plain un-typed literal, and only we know that the intention is to treat it as a decimal number; there is no information in List 2.23 that can explicitly indicate that. However, sometimes, it is important for the RDF parser or the application to know how to explain the plain value.

The solution is to use the `rdf:datatype` keyword from RDF vocabulary. Note that RDF/XML syntax does not provide any datatype system of its own, such as datatypes for integers, real numbers, strings, and dates. It instead borrows an external datatype system, and currently, it is the XML Schema datatypes. The reason is also very simple: since XML enjoys such a great success, its schema datatypes would most likely be interoperable among different software agents.

Now let us use `rdf:datatype` to clearly indicate that the value `0.6` should be treated as a decimal value, as shown in List 2.24.

List 2.24 Example of using `rdf:datatype`

```

1: <?xml version="1.0"?>
2: <rdf:RDF
2a:     xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
3:     xmlns:uom="http://www.example.org/units#"
4:     xmlns:myCamera="http://www.liyangyu.com/camera#">
5:
6:   <rdf:Description
6a:     rdf:about="http://www.liyangyu.com/camera#Nikon_D300">
7:     <rdf:type

```

```

7a:         rdf:resource="http://www.liyangyu.com/camera#DSLR"/>
8:     <myCamera:manufactured_by
8a:         rdf:resource="http://www.dbpedia.org/resource/Nikon"/>
9:     <myCamera:performance>
10:         <rdf:Description rdf:about=
10a:             "http://www.liyangyu.com/camera#PictureQuality">
11:             <myCamera:evaluate>5 stars</myCamera:evaluate>
12:         </rdf:Description>
13:     </myCamera:performance>
14:     <myCamera:model
14a:         rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
15:         D300</myCamera:model>
16:     <myCamera:weight rdf:parseType="Resource">
17:         <rdf:value rdf:datatype=
17a:             "http://www.w3.org/2001/XMLSchema#decimal">
18:             0.6</rdf:value>
19:         <uom:units
19a:             rdf:resource="http://www.example.org/units#kg"/>
20:     </myCamera:weight>
21: </rdf:Description>
22:
23: </rdf:RDF>

```

As shown at line 17 in List 2.24, property `rdf:value` now has an attribute named `rdf:datatype` whose value is the URI of the datatype. In our example, this URI is <http://www.w3.org/2001/XMLSchema#decimal>. The result is the value of the `rdf:value` property, namely, 0.6, will be treated as a decimal value as defined in the XML Schema datatypes.

Note that there is no absolute need to use `rdf:value` in the above example. A user-defined property name can be used instead of `rdf:value` and the `rdf:datatype` attribute can still be used together with that user-defined property. Line 14 shows one example: it specifies literal D300 should be interpreted as a string. In fact, RDF does not associate any special meaning with `rdf:value`; it is simply provided as a convenience for use in the cases as described by our example.

Also note that since <http://www.w3.org/2001/XMLSchema#decimal> is used as an attribute value, it has to be written out, rather than using any shorthand abbreviation. However, this makes the line quite long and might hurt readability in some cases. To improve the readability, some RDF documents would use XML entities.

More specifically, an XML entity can associate a name with a string of characters and this name can be referenced anywhere in the XML document. When XML processors reach such a name, they will replace the name with the character string which normally represents the real content. Since we can make the name really short, this provides us with the ability to abbreviate the long URI.

To declare the entity, we can do the following:

```
<!DOCTYPE
rdf:RDF [<!ENTITY xsd "http://www.w3.org/2001/XMLSchema#">]>
```

A reference name `xsd` is defined here to be associated with the namespace URI which contains the XML Schema datatypes. Anywhere in the RDF document we can use `&xsd;` (note the “;” which is necessary) to represent the above URI. Using this abbreviation, we have the following more readable version as shown in List 2.25.

List 2.25 A more readable version of List 2.24

```
1: <?xml version="1.0"?>
2: <!DOCTYPE
2a: rdf:RDF [<!ENTITY xsd "http://www.w3.org/2001/XMLSchema#">]>
3:
4: <rdf:RDF
4a:     xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
5:     xmlns:uom="http://www.example.org/units#"
6:     xmlns:myCamera="http://www.liyangyu.com/camera#">
7:
8:   <rdf:Description
8a:     rdf:about="http://www.liyangyu.com/camera#Nikon_D300">
9:     <rdf:type
9a:       rdf:resource="http://www.liyangyu.com/camera#DSLR"/>
10:     <myCamera:manufactured_by
10a:      rdf:resource="http://www.dbpedia.org/resource/Nikon"/>
11:     <myCamera:performance>
12:       <rdf:Description rdf:about=
12a:         "http://www.liyangyu.com/camera#PictureQuality">
13:         <myCamera:evaluate>5 stars</myCamera:evaluate>
14:       </rdf:Description>
15:     </myCamera:performance>
16:     <myCamera:model
16a:       rdf:datatype="&xsd:string">D300</myCamera:model>
17:     <myCamera:weight rdf:parseType="Resource">
18:       <rdf:value rdf:datatype="&xsd;decimal">0.6</rdf:value>
19:       <uom:units
19a:        rdf:resource="http://www.example.org/units#kg"/>
20:     </myCamera:weight>
21:   </rdf:Description>
22:
23: </rdf:RDF>
```


2.3.2.6 `rdf:nodeID` and More About Anonymous Resources

In Sect. 2.3.2.3 we have talked about blank node. For example, in List 2.22, lines 16–19 represent a blank node. As you can see, that blank node is embedded inside the XML property node, `myCamera:weight`, and is used as the property value of this node.

This kind of embedded blank node works well most of the time, but it does have one disadvantage: it cannot be referenced from any other part of the same document. In some cases, we do have the need to make reference to a blank node within the same document.

To solve this problem, RDF/XML syntax provides another way to represent a blank node: use the so-called blank node identifier. The idea is to assign a *blank node identifier* to a given blank node, so it can be referenced within this particular RDF document and still remains unknown outside the scope of the document.

This blank node identifier method uses the RDF keyword `rdf:nodeID`. More specifically, a statement using a blank node as its subject value should use an `rdf:Description` element together with an `rdf:nodeID` attribute instead of an `rdf:about` or `rdf:ID` (discussed in later section) attribute. By the same token, a statement using a blank node as its object should also use a property element with an `rdf:nodeID` attribute instead of an `rdf:Resource` attribute. List 2.26 shows the details.

List 2.26 Use `rdf:nodeID` to name a blank node

```

1: <?xml version="1.0"?>
2: <rdf:RDF
3a:   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
4:   xmlns:uom="http://www.example.org/units#"
5:   xmlns:myCamera="http://www.liyangyu.com/camera#">
6:   <rdf:Description
7a:     rdf:about="http://www.liyangyu.com/camera#Nikon_D300">
8:     <rdf:type
9a:       rdf:resource="http://www.liyangyu.com/camera#DSLR"/>
10:    <myCamera:manufactured_by
11a:     rdf:resource="http://www.dbpedia.org/resource/Nikon"/>
12:    <myCamera:performance>
13:      <rdf:Description rdf:about="
14a:        "http://www.liyangyu.com/camera#PictureQuality">
15:        <myCamera:evaluate>5 stars</myCamera:evaluate>
16:      </rdf:Description>
17:    </myCamera:performance>
18:    <myCamera:model>D300</myCamera:model>

```

```

15:   <myCamera:weight rdf:nodeID = "youNameThisNode"/>
16: </rdf:Description>
17:
18: <rdf:Description rdf:nodeID = "youNameThisNode">
19:   <rdf:value>0.6</rdf:value>
20:   <uom:units
20a:     rdf:resource="http://www.example.org/units#kg"/>
21: </rdf:Description>
22:
23: </rdf:RDF>

```

Note that the blank node in List 2.22 (lines 16–19) has been given a local identifier called `youNameThisNode`, and the resource named `youNameThisNode` is then described in lines 18–21. We, on purpose, name this identifier to be `youNameThisNode`, just to show you the fact that you can name this node whatever you want to. The real benefit is that this resource now has a local identifier, so it can be referenced from other places within the same document. Although in this particular case, it's not referenced by any other resource except for being the object of property `myCamera:weight`, you should be able to imagine the cases where a blank node could be referenced multiple times.

Blank node is very useful in RDF, and we will see more examples of using blank node in later sections. In addition, note that `rdf:nodeID` is case sensitive. For example, an RDF parser will flag an error if you have mistakenly written it as `rdf:nodeId`. In fact, every single term in RDF vocabulary is case sensitive, so make sure they are right.

2.3.2.7 `rdf:ID`, `xml:base`, and RDF/XML Abbreviation

By far, you probably have already realized one thing about RDF/XML syntax: it is quite verbose and quite long. In this section, we will discuss the things you can do to make it shorter.

We have seen RDF/XML abbreviation already in previous section. For example, compare List 2.17 with List 2.18. In List 2.18, multiple properties are nested within the `rdf:Description` element that identifies the subject, and in List 2.17, each property requires a separate statement, and these statements all share the same subject.

Another abbreviation we have seen is to use `ENTITY` declaration (together with `DOCTYPE` declaration at the beginning of a given RDF/XML document). List 2.25 has presented one such example.

The last abbreviation we have seen involves the so-called *long form* of RDF/XML syntax. More specifically, List 2.15 uses the `rdf:Description` together with `rdf:about` combination to describe a resource, and this form is called the long form. On the other hand, List 2.16 is an abbreviation of this long form, and they are equivalent to each other. Most RDF parsers will translate the abbreviated form into the long form first before any processing is done.

A new abbreviation of the long form that we have not seen yet is to use the `rdf:ID` term from the RDF vocabulary, as shown in List 2.27 (note that since we only want to show the use of `rdf:ID`, we did not include other properties as described in List 2.26).

List 2.27 Example of using `rdf:ID`

```

1: <?xml version="1.0"?>
2: <rdf:RDF
2a:     xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
3:     xmlns:myCamera="http://www.liyangyu.com/camera#">
4:
5:     <rdf:Description rdf:ID="Nikon_D300">
6:         <rdf:type
6a:             rdf:resource="http://www.liyangyu.com/camera#DSLR"/>
7:         <myCamera:manufactured_by
7a:             rdf:resource="http://www.dbpedia.org/resource/Nikon"/>
8:     </rdf:Description>
9:
10: </rdf:RDF>

```

Compare List 2.27 with List 2.18, you can see the difference. Instead of using `rdf:about`, RDF keyword `rdf:ID` is used to identify the resource that is being described by this RDF document (line 5).

This does make the statement shorter; at least there is no long URI needed for the resource. However, to use `rdf:ID`, we have to be very careful. More specifically, `rdf:ID` only specifies a fragment identifier; the complete URI of the subject is obtained by concatenating the following three pieces together:

in-scope base URI + “#” + `rdf:ID` value

Since the in-scope base URI is not explicitly stated in the RDF document (more on this later), it is then provided by the RDF parser based on the location of the file. In this example, since <http://www.liyangyu.com/rdf/review.rdf> is the location, <http://www.liyangyu.com/rdf/review.rdf#Nikon-D300> is then used as the URI of the subject.

Clearly, using `rdf:ID` results in a relative URI for the subject, and the URI changes if the location of the RDF document changes. This seems to be contradicting to the very meaning of URI: it is the unique and global identifier of a resource, and how can it change based on the location of some file then?

The solution is to explicitly state the in-scope base URI. Specifically, we can add the `xml:base` attribute in the RDF document to control which base is used to resolve the `rdf:ID` value. Once an RDF parser sees the `xml:base` attribute, it will generate the URI by using the following mechanism:

`xml:base` + “#” + `rdf:ID` value

List 2.28 shows the details (line 4).

List 2.28 Example of using `xml:base`

```

1: <?xml version="1.0"?>
2: <rdf:RDF
3:     xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
4:     xmlns:myCamera="http://www.liyangyu.com/camera#"
5:     xml:base="http://www.liyangyu.com/camera#"
6:     <rdf:Description rdf:ID="Nikon_D300">
7:         <rdf:type
8:             rdf:resource="http://www.liyangyu.com/camera#DSLR"/>
9:         <myCamera:manufactured_by
10:            rdf:resource="http://www.dbpedia.org/resource/Nikon"/>
11:     </rdf:Description>
12: </rdf:RDF>

```

`rdf:ID` (together with `xml:base`) can also be used in the short form (see List 2.16), as shown in List 2.29.

List 2.29 Example of using `xml:base` with the short form

```

1: <?xml version="1.0"?>
2: <rdf:RDF
3:     xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
4:     xmlns:myCamera="http://www.liyangyu.com/camera#"
5:     xml:base="http://www.liyangyu.com/camera#"
6:     <myCamera:DSLR rdf:ID="Nikon_D300">
7:         <myCamera:manufactured_by
8:             rdf:resource="http://www.dbpedia.org/resource/Nikon"/>
9:     </myCamera:DSLR>
10: </rdf:RDF>

```

In both Lists 2.28 and 2.29, the subject will have the following URI:

`http://www.liyangyu.com/camera#Nikon_D300`

which is what we wanted, and it will not change when the location of the RDF document changes.

As a summary, Lists 2.15, 2.16, 2.28, and 2.29 are all equivalent forms. However, it might be a good idea to use `rdf:about` instead of `rdf:ID`, since it provides an absolute URI for the resource. Also, that URI is taken verbatim as the subject, which certainly avoids all the potential confusions.

At this point, we have covered the most frequently used RDF/XML syntax, which you certainly need in order to understand the rest of the book. We will discuss some other capabilities provided by RDF/XML syntax in the next few sections to complete the description of the whole RDF picture.

2.3.3 Other RDF Capabilities and Examples

RDF/XML syntax also provides some additional capabilities, such as representing a group of resources and making statements about statements. In this section, we will take a brief look at these capabilities.

2.3.3.1 RDF Containers: `rdf:Bag`, `rdf:Seq`, `rdf:Alt`, and `rdf:li`

Let us say that a Nikon D300 camera can be reviewed based on the following criteria (it is certainly over-simplified to apply only three measurements when it comes to review a camera, but it is good enough to make our point clear):

- effective pixels;
- image sensor format; and
- picture quality.

How do we express this fact in RDF?

RDF/XML syntax models this situation by the concept of *container*. A container is a resource that contains things, and each one of these things is called a *member* in the container. A member can be represented by either a resource or a literal.

The following three types of containers are provided by RDF/XML syntax using a pre-defined container vocabulary:

- `rdf:Bag`
- `rdf:Seq`
- `rdf:Alt`

A resource can have type `rdf:Bag`. In this case, the resource represents a group of resources or literals, the order of these members is not significant, and there could be duplicated members as well. For example, the review criteria presented above can be modeled by using `rdf:Bag`.

An `rdf:Seq` type resource is the same as an `rdf:Bag` resource, except the order of its member is significant. For instance, if we want to show which criterion is more important than the others, we will have to represent them using `rdf:Seq`.

`rdf:Alt` is also a container. However, items in this container are alternatives. For example, it can be used to describe a list of alternative stores where you can find a Nikon D300 camera.

Let us take a look at the example shown in List 2.30.

List 2.30 Example of using `rdf:Bag`

```
1: <?xml version="1.0"?>
2: <!DOCTYPE rdf:RDF
2a:     [<!ENTITY myCamera "http://www.liyangyu.com/camera#">]>
3:
```

```

4: <rdf:RDF
4a:     xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
5:     xmlns:myCamera="http://www.liyangyu.com/camera#">
6:
7:   <rdf:Description
7a:     rdf:about="http://www.liyangyu.com/camera#Nikon_D300">
8:     <myCamera:hasReviewCriteria>
9:       <rdf:Description>
10:        <rdf:type rdf:resource=
10a:          "http://www.w3.org/1999/02/22-rdf-syntax-ns#Bag"/>
11:        <rdf:li rdf:resource="&myCamera;EffectivePixel"/>
12:        <rdf:li rdf:resource="&myCamera;ImageSensorFormat"/>
13:        <rdf:li rdf:resource="&myCamera;PictureQuality"/>
14:      </rdf:Description>
15:    </myCamera:hasReviewCriteria>
16:  </rdf:Description>
17:
18: </rdf:RDF>

```

To express the fact that a Nikon D300 camera can be reviewed based on a given set of criteria, a property called `myCamera:hasReviewCriteria` has been assigned to Nikon D300 (line 8), and this property's value is a resource whose type is `rdf:Bag` (line 10). Furthermore, `rdf:li` is used to identify the members of this container resource, as shown in lines 11–13. Note that lines 7–16 represent one single statement, with the container resource represented by a blank node.

Figure 2.8 shows the corresponding graph representation of List 2.30.

Note that `rdf:li` is a property provided by RDF/XML syntax for us to use, so we do not have to explicitly number each membership property. Under the hood, a given RDF parser will normally generate properties such as `rdf:_1`, `rdf:_2`, and `rdf:_3` (as shown in Fig. 2.8) to replace `rdf:li`. In this case, since the members are contained in an `rdf:Bag`, these numbers should be ignored by the applications creating or processing this graph. Note that RDF models do not regulate the processing of List 2.30; it is up to the applications to handle it in the way that it is intended to.

The example of `rdf:Seq`, including the RDF/XML syntax and the graph representation, is exactly the same as List 2.30, except that the container type will be changed to `rdf:Seq`. Again, note that properties such as `rdf:_1`, `rdf:_2`, and `rdf:_3` will be generated by RDF parser to replace `rdf:li`, and it is up to the applications to correctly interpret the sequence.

The syntax and graph representation of `rdf:Alt` are also exactly the same except that you need to use `rdf:Alt` as the type of the container resource. And again, it is up to the application to understand that only one member should be taken, and it should be identified by `rdf:_1`.

As a summary, these three types of containers are pre-defined by RDF/XML syntax for you to use. You should, however, use them according to their “intended

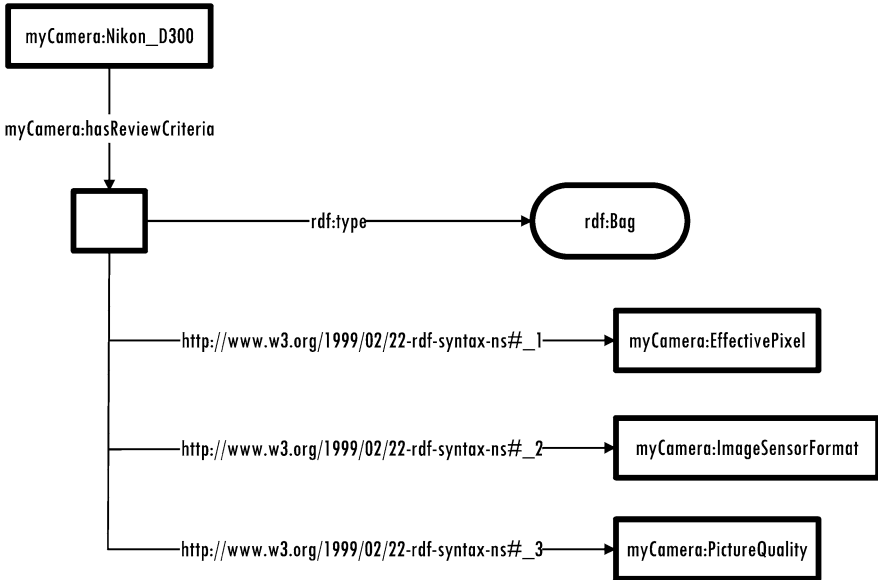


Fig. 2.8 Graph representation of the statements in List 2.30

usage”; RDF/XML itself does not provide any check at all. In fact, this container vocabulary is created with the goal to help make data representation and processing more interoperable; applications are not required to use them. They can choose their own way to describe groups of resources if they prefer.

2.3.3.2 RDF Collections: `rdf:first`, `rdf:rest`, `rdf:nil`, and `rdf:List`

In the last section, we discussed the container class. The problem with an RDF container is that it is not closed: a container includes the identified resources as its members, it never excludes other resources to be members. Therefore, it could be true that some other RDF documents may add additional members to the same container.

To solve this problem, RDF uses a pre-defined *collection vocabulary* to describe a group that contains only the specified resources as members. Its vocabulary includes the following keywords:

- `rdf:first`
- `rdf:rest`
- `rdf:List`
- `rdf:nil`

To express the fact that “*only* effective pixels, image sensor format, and picture quality can be used as criteria to review a given Nikon D300 camera,” the above keywords can be used as shown in Fig. 2.9.

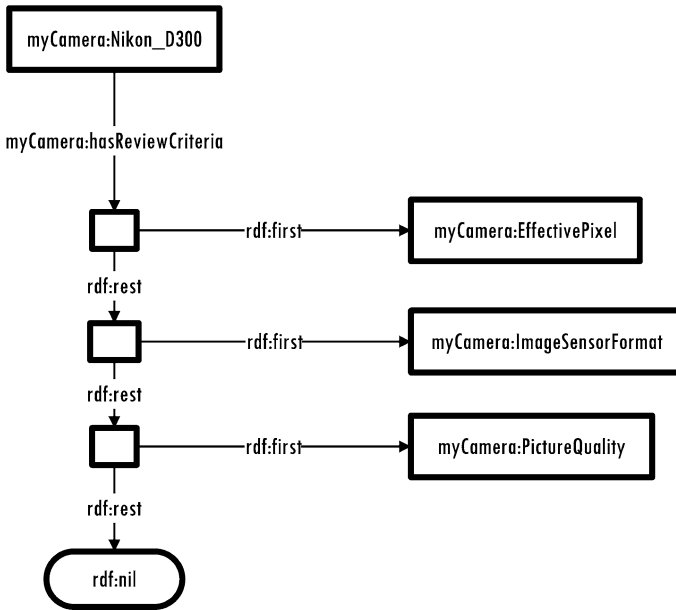


Fig. 2.9 RDF collection vocabulary

Clearly, the members of a given container are all linked together by repeatedly using `rdf:first`, `rdf:rest`, until the end (indicated by `rdf:nil`, a resource that is of type `rdf:List`). Note how the blank nodes are used in this structure (Fig. 2.9). Obviously, there is no way to add any new members into this container, since other RDF documents will not be able to access the blank nodes here. This is how RDF/XML syntax can guarantee the underlying container is closed.

Since ideally every closed container should follow the same pattern as shown in Fig. 2.9, RDF/XML decides to provide a special notation to make it easier to describe a close container. More specifically, there is no need to explicitly use `rdf:first`, `rdf:rest`, and `rdf:nil` keywords; all we need to do is to use the attribute `rdf:parseType` with its value set to be `Collection`, as shown in List 2.31.

List 2.31 Example of using RDF collection

```

1: <?xml version="1.0"?>
2: <!DOCTYPE rdf:RDF
2a:     [<!ENTITY myCamera "http://www.liyangyu.com/camera#">]>
3:
4: <rdf:RDF
4a:     xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
5:     xmlns:myCamera="http://www.liyangyu.com/camera#">
6:

```



```

7:   <rdf:Description
7a:       rdf:about="http://www.liyangyu.com/camera#Nikon_D300">
8:     <myCamera:hasReviewCriteria rdf:parseType="Collection">
9:       <rdf:Description rdf:about="&myCamera;EffectivePixel"/>
10:      <rdf:Description
10a:         rdf:about="&myCamera;ImageSensorFormat"/>
11:      <rdf:Description rdf:about="&myCamera;PictureQuality"/>
12:     </myCamera:hasReviewCriteria>
13: </rdf:Description>
14:
15: </rdf:RDF>

```

An RDF parser which sees List 2.31 will then automatically generate the structure shown in Fig. 2.9.

Note that it is possible, however, to manually use `rdf:first`, `rdf:rest`, and `rdf:nil` keywords to construct a close container, without using the notation shown in List 2.31. If you decide to do so, it is your responsibility to make sure you have created the pattern as shown in Fig. 2.9, and you have to use blank nodes so no other RDF document can access the list and modify it. Therefore, the best solution is indeed to use this special notation offered by RDF/XML syntax.

2.3.3.3 RDF Reification: `rdf:statement`, `rdf:subject`, `rdf:predicate`, and `rdf:object`

At this point, we have covered most of the terms in RDF vocabulary. In this section, we will discuss the remaining terms, more specifically, `rdf:statement`, `rdf:subject`, `rdf:predicate`, and `rdf:object`.

In fact, these four terms make up the built-in vocabulary used for describing RDF statements. Therefore, if we need to describe RDF statements using RDF, this vocabulary provides the terms we would like to use.

For example, for a given RDF statement, we might want to record information such as when this statement is created and who has created it. A description of a statement using this vocabulary is often called a *reification* of the statement, and accordingly, this vocabulary is also called *RDF reification vocabulary*.

Let us take a look at one example. The following statement from List 2.5

```
myCamera:Nikon_D300 myCamera:manufactured_by dbpedia:Nikon
```

states the fact that the Nikon D300 camera is manufactured by Nikon Corporation. A reification of this statement is shown in List 2.32.

List 2.32 Reification example

```

myCamera:statement_01  rdf:type           rdf:Statement
myCamera:statement_01  rdf:subject       myCamera:Nikon_D300
myCamera:statement_01  rdf:predicate     myCamera:manufactured_by
myCamera:statement_01  rdf:object         dbpedia:Nikon

```

In List 2.32, `myCamera:statement_01` is an URI that is assigned to the statement that is being described, i.e.,

```
myCamera:Nikon_D300 myCamera:manufactured_by dbpedia:Nikon
```

And the first statement in List 2.32 says the resource identified by `myCamera:statement_01` is an RDF statement. The second statement says that the subject of this RDF statement is identified by resource `myCamera:Nikon_D300`. The third statement says the predicate of this RDF statement is given by `myCamera:manufactured_by`, and the last statement says the object of the statement refers to the resource identified by `dbpedia:Nikon`.

Obviously, this reification example has used four statements to describe the original statement. This usage pattern is often referred to as the conventional use of the RDF reification vocabulary. Since it always involves four statements, it is also called a *reification quad*.

Now, to record provenance information about the original statement, we can simply add additional statement to this quad, as shown in List 2.33.

List 2.33 Adding provenance information using reification

```
myCamera:statement_01 rdf:type          rdf:Statement
myCamera:statement_01 rdf:subject      myCamera:Nikon_D300
myCamera:statement_01 rdf:predicate    myCamera:manufactured_by
myCamera:statement_01 rdf:object       dbpedia:Nikon
myCamera:statement_01 dc:creator      http://www.liyangyu.com#liyang
```

As you can see, the last statement in List 2.33 is added to show the creator of the original statement, and <http://www.liyangyu.com#liyang> is the URI identifying this creator.

Note that `dc:creator` is another existing URI (just like `dbpedia:Nikon` is an existing URI representing Nikon Corporation) taken from a vocabulary called Dublin Core. We will discuss Dublin Core in more detail in the next section. For now, understand that `dc:creator` represents the creator of a given document is good enough.

You can certainly add more statement into List 2.33 to record more provenance information about the original statement, such as the date when the original statement was created.

The usage of reification vocabulary is fairly straightforward. However, it does require some caution when using it. Recall that we have assigned an URI to the original statement (`myCamera:statement_01`), so it can be represented as a resource, and new RDF statements can be created to describe it. However, this kind of logic connection only exists in our mind. The URI is completely arbitrary, and there is no built-in mechanism in RDF to understand that this URI is created to represent a particular statement in a given RDF graph.

As a result, it is up to the RDF application to handle this, and it has to be done with care. For example, given the statements in List 2.34, an RDF application

may try to match `rdf:subject`, `rdf:predicate`, and `rdf:object` taken from List 2.33 to a statement so as to decide whether the reification in List 2.33 is used on this particular statement. However, there could be multiple statements in different RDF models, and all these statements will be matched successfully, and it is therefore hard to decide exactly which one is the candidate. For example, different camera reviewers can make the same statement in their reviews (in RDF format), and our RDF application built on all these reviews will find multiple matches. Therefore, for a given statement, we cannot simply depend on matching its `rdf:subject`, `rdf:predicate`, and `rdf:object` components. Most likely, more application-specific assumptions may have to be made to make this work.

In addition, note that other applications receiving these RDF documents may not share the same application-specific understanding, and therefore may not be able to interpret these statements correctly.

With all these being said, RDF reification is still useful and remains an important topic, mainly because it provides one way to add provenance information, which is important to handle the issue of trust on the Web. For now, understand it, and in your own development work, use it with care.

2.4 Other RDF Sterilization Formats

2.4.1 Notation-3, Turtle, and N-Triples

By now, there is probably one important aspect of RDF that we have not emphasized enough: RDF is an abstract data model, and RDF standard itself does not specify its representation. The recommended and perhaps the most popular representation of an RDF model is the XML serialization format (noted as RDF/XML), as we have seen so far.

However, RDF/XML is not designed for human eyes. For instance, it is hard to read and can be quite long as well. There are indeed other RDF serialization formats, such as *Notation-3* (or *N3*), *Turtle*, and *N-Triples*.

Notation-3 is a non-XML serialization of RDF model and is designed with human readability in mind. It is therefore much more compact and readable than XML/RDF format.

Since Notation-3 does have several features that are not necessary for serialization of RDF models (such as its support for RDF-based rules), Turtle is created as a simplified and RDF-only subset of Notation-3. In addition, N-Triples is another simpler format than both Notation-3 and Turtle, and therefore offers another alternative to developers.

In this section, we will focus mainly on Turtle format because of its popularity among developers. In addition, as you will see in [Chap. 6](#), SPARQL has borrowed almost everything from Turtle to form its own query language. Therefore, understanding Turtle will make us comfortable with the syntax used in SPARQL query language as well.

2.4.2 Turtle Language

Formally speaking, Turtle represents *Terse RDF Triple Language*. It is a text-based syntax for serialization of RDF model. You can find a complete discussion about Turtle in

<http://www.w3.org/TeamSubmission/turtle/>

And you should know the following about Turtle in general:

- The URI that identifies the Turtle language is given by

<http://www.w3.org/2008/turtle#turtle>

- The XML (Namespace name, local name) pair that identifies Turtle language is as follows:

<http://www.w3.org/2008/turtle#,turtle>

- The suggested namespace prefix is `ttl`, and a Turtle document should use `ttl` as the file extension.

2.4.2.1 Basic Language Feature

Now, let us take a brief look at Turtle language. First off, a Turtle document is a collection of RDF statements, and each statement has a format that is called a triple:

`<subject> <predicate> <object>.`

Note that

- each statement has to end with a period;
- subject must be represented by a URI;
- predicate must be represented by a URI;
- object can be either a URI or a literal;
- a URI must be surrounded in `<>` brackets, which are used to delineate a given URI.

A given literal may have a language or a datatype URI as its suffix, but it is not allowed to have both. If it is given a language suffix, the suffix is created by a `@` together with the language tag. For example,

`"this is in English"@en`

If it is given a datatype suffix, `^^` is used:

`"10"^^<http://www.w3.org/2001/XMLSchema#decimal>`

`"foo"^^<http://example.org/mydatatype/sometype>`

Note that a literal does not have to be appended by a datatype URI or language tag. For example, these two literals are perfectly legal:

`"10"`

`"foo"`

With all these said, List 2.34 shows some triple examples in Turtle format (note the period at the end of each statement).

List 2.34 Triple examples in Turtle format

```
<http://www.liyangyu.com/foaf.rdf#liyang>
<http://xmlns.com/foaf/0.1/name> "liyang yu".

<http://www.liyangyu.com/foaf.rdf#liyang>
<http://xmlns.com/foaf/0.1/interest>
<http://dbpedia.org/resource/Semantic_Web>.
```

And this is the main idea for Turtle. However, there are lots of abbreviations and shortcuts that can make the RDF Turtle documents much more compact and still readable. Let us discuss these features next.

2.4.2.2 Abbreviations and Shortcuts: Namespace Prefix, Default Prefix, and @base

Obviously, full URIs are long and somewhat unreadable. To make them shorter and also more readable, we can define a namespace prefix so we don't have to write the long common part of the URI over and over. The general format for defining namespace prefix is given as below:

```
@prefix pref: <uri>.
```

where *pref* is the shortcut for *uri*. For example,

```
@prefix foaf: <http://xmlns.com/foaf/0.1/>.
@prefix liyang: <http://www.liyangyu.com/foaf.rdf#>.
```

and now the two statements in List 2.34 can be re-written as in List 2.35.

List 2.35 Statements in List 2.34 are re-written using namespace prefix

```
@prefix foaf: <http://xmlns.com/foaf/0.1/>.
@prefix liyang: <http://www.liyangyu.com/foaf.rdf#>.

liyang:liyang foaf:name "liyang yu".
liyang:liyang foaf:interest
    <http://dbpedia.org/resource/Semantic_Web>.
```

These are obviously much more readable and compact as well.

Another way to abbreviate namespace is to create a default namespace prefix, acting as the “main” namespace for a Turtle document. For example, if we are creating or working on a FOAF document (more about FOAF in [Chap. 7](#)), making

FOAF namespace as the default (main) namespace is a good choice. To create a default namespace, we can use the same general form, but without a `pref` string:

```
@prefix : <uri>.
```

for instance,

```
@prefix : <http://xmlns.com/foaf/0.1/>.
```

will set `<http://xmlns.com/foaf/0.1/>` as the default namespace, and List 2.35 will be changed to List 2.36.

List 2.36 Statements in List 2.35 are re-written using default namespace prefix

```
@prefix : <http://xmlns.com/foaf/0.1/>.
```

```
@prefix liyang : <http://www.liyangyu.com/foaf.rdf#>.
```

```
liyang:liyang :name "liyang yu".
```

```
liyang:liyang :interest
```

```
  <http://dbpedia.org/resource/Semantic_Web>.
```

In other words, any URI identifier starting with `:` will be in the default namespace.

Note in some document, `@base` directive is also used to allow abbreviation of URIs. It could be confusing if you are not familiar with this since it somewhat feels like default namespace prefix, but in fact it is not. Let us talk about this a little bit more.

The key point to remember about `@base` is this: whenever it appears in a document, it defines the base URI against which all relative URIs are going to be resolved. Let us take a look at List 2.37.

List 2.37 Example of using `@base`

```
1: <subj0> <pred0> <obj0>.
```

```
2: @base <http://liyangyu.com/ns0/>.
```

```
3: <subj1> <http://liyangyu.com/ns0/pred1> <obj1>.
```

```
4: @base <foo/>.
```

```
5: <subj2> <pred2> <obj2>.
```

```
6: @predix : <bar#>.
```

```
7: :subj3 :pred3 :obj3.
```

```
8: @predix : <http://liyangyu.com/ns1/>.
```

```
9: :subj4 :pred4 :obj4.
```

How should this be resolved? Clearly, line 1 is a triple that all of its components are using relative URIs; therefore, all these URIs should be resolved against the current `@base` value. Since there is no explicit definition of `@base` yet, the location of this document will be treated as the current base. Assuming this document locates at `http://liyangyu.com/data/`, line 1 will resolve as the following:

```
<http://liyangyu.com/data/subj0>
```

```
<http://liyangyu.com/data/pred0>
```

```
<http://liyangyu.com/data/obj0>.
```

Since line 2 has specified a new base value, line 3 will be resolved as the following:

```
<http://liyangyu.com/ns0/subj1>
<http://liyangyu.com/ns0/pred1>
<http://liyangyu.com/ns0/obj1>.
```

Note `pred1` does not need to resolve, since it has an absolute URI.

Now, line 4 again uses `@base` to define a relative URI, which will be resolved against the current base; in other words, line 4 is equivalent to the following:

```
@base <http://liyangyu.com/ns0/foo/>.
```

therefore, line 5 will then be resolved using this new base URI:

```
<http://liyangyu.com/ns0/foo/subj2>
<http://liyangyu.com/ns0/foo/pred2>
<http://liyangyu.com/ns0/foo/obj2>.
```

Line 6 defines a default namespace prefix:

```
@prefix : <bar#>.
```

and since it is again a relative URI, it has to be resolved against the current base first. Therefore, this default namespace will have the following resolved URI:

```
@prefix : <http://liyangyu.com/ns0/foo/bar#>.
```

Therefore, the triple on line 7 will be resolved to this:

```
<http://liyangyu.com/ns0/foo/bar#subj3>
<http://liyangyu.com/ns0/foo/bar#pred3>
<http://liyangyu.com/ns0/foo/bar#obj3>.
```

Finally, line 8 defines another default namespace, and since it is an absolute URI already, it does not have to be resolved against the current base, and line 9 is resolved to this:

```
<http://liyangyu.com/ns1/subj4>
<http://liyangyu.com/ns1/pred4>
<http://liyangyu.com/ns1/obj4>.
```

This should have cleared up the confusion around `@base` directive and default namespace prefix, and this has also completed the discussion about URI abbreviation. Let us talk about some other frequently used abbreviations.

2.4.2.3 Abbreviations and Shortcuts: Token `a`, Comma, and Semicolons

Token `a` in Turtle is always equivalent to the following URI:

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

therefore,

```
liyang:liyang rdf:type foaf:Person.
```

can be written as follows:

```
liyang:liyang a foaf:Person.
```

Both commas and semicolons can be used to make a given document shorter. More specifically, if two or more statements with the same subject and predicate are made, we can combine the statements and separate different objects by one or more commas. For example, consider List 2.38.

List 2.38 A Turtle document that has two statements with the same subject and predicate

```
@prefix foaf: <http://xmlns.com/foaf/0.1/>.
@prefix liyang: <http://www.liyangyu.com/foaf.rdf#>.

liyang:liyang foaf:name "liyang yu".
liyang:liyang foaf:interest
    <http://dbpedia.org/resource/Semantic_Web>.
liyang:liyang foaf:interest <http://semantic-mediawiki.org/>.
```

It can be changed to List 2.39 which is equivalent yet has a shorter form.

List 2.39 Combine the two statements in List 2.38

```
@prefix foaf: <http://xmlns.com/foaf/0.1/>.
@prefix liyang: <http://www.liyangyu.com/foaf.rdf#>.

liyang:liyang foaf:name "liyang yu".
liyang:liyang foaf:interest
    <http://dbpedia.org/resource/Semantic_Web>,
    <http://semantic-mediawiki.org>.
```

If we have the same subject but different predicates in more than one statements, we can use semicolons to make them shorter. For example, List 2.39 can be further re-written as shown in List 2.40.

List 2.40 Using ; to re-write List 2.39

```
@prefix foaf: <http://xmlns.com/foaf/0.1/>.
@prefix liyang: <http://www.liyangyu.com/foaf.rdf#>.

liyang:liyang foaf:name "liyang yu" ;
    foaf:interest <http://www.foaf-project.org/>,
    <http://semantic-mediawiki.org>.
```


2.4.2.4 Turtle Blank Nodes

Last but not the least, let us discuss blank nodes. Some literature does not recommend using blank nodes, but in some cases, they could be very handy to use. In Turtle, a blank node is denoted by `[]` and you can use it as either the subject or the object. For example, List 2.41 says “there exists a person named liyang yu”:

List 2.41 Using blank node as the subject

```
@prefix foaf: <http://xmlns.com/foaf/0.1/>.

[] a foaf:Person;
   foaf:name "liyang yu" .
```

In List 2.41, blank node is used as the subject. If you decide to serialize this model using RDF/XML format, you will get the document shown in List 2.42.

List 2.42 Express the statement in List 2.41 using RDF/XML format

```
<?xml version="1.0"?>
<rdf:RDF xmlns:foaf="http://xmlns.com/foaf/0.1/"
         xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
  <foaf:Person>
    <foaf:name>liyang yu</foaf:name>
  </foaf:Person>
</rdf:RDF>
```

It will have the following underlying triples:

```
_:bnode0 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
         <http://xmlns.com/foaf/0.1/Person>.
_:bnode0 <http://xmlns.com/foaf/0.1/name> "liyang yu".
```

We can also use blank node to represent an object. For example, the Turtle statement in List 2.43 says “Liyang is a person and he knows another person named Connie”:

List 2.43 Use blank node as the object

```
@prefix foaf: <http://xmlns.com/foaf/0.1/>.
@prefix liyang: <http://www.liyangyu.com/foaf.rdf#>.

liyang:liyang a foaf:Person;
              foaf:knows [
                a foaf:Person;
                foaf:name "connie".
              ] .
```

Again, in RDF/XML format, the statements in List 2.43 will look like the ones shown in List 2.44.

List 2.44 Express the statement in List 2.43 using RDF/XML format

```

<?xml version="1.0"?>
<rdf:RDF xmlns:foaf="http://xmlns.com/foaf/0.1/"
        xmlns:liyang="http://www.liyangyu.com/foaf.rdf#"
        xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
<foaf:Person rdf:about="http://www.liyangyu.com/foaf.rdf#liyang">
  <foaf:knows>
    <foaf:Person>
      <foaf:name>connie</foaf:name>
    </foaf:Person>
  </foaf:knows>
</foaf:Person>

</rdf:RDF>

```

Underlying triples are also listed here:

```

<http://www.liyangyu.com/foaf.rdf#liyang>
<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
<http://xmlns.com/foaf/0.1/Person>.

_:bnode0 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
  <http://xmlns.com/foaf/0.1/Person>.
_:bnode0 <http://xmlns.com/foaf/0.1/name> "connie".

<http://www.liyangyu.com/foaf.rdf#liyang>
<http://xmlns.com/foaf/0.1/knows> _:bnode0.

```

You can tell how compact the Turtle format is!

2.5 Fundamental Rules of RDF

Since we have covered most of the contents about RDF, it is time for us to summarize the basic rules of RDF. There are altogether three basic rules, and they are critically related to some of the most important aspects of the Semantic Web. At this point, these closely related aspects are as follows:

1. RDF represents and models information and knowledge in a way that machine can understand.
2. Distributed RDF graphs can be aggregated to facilitate new information discovery.

In this section, we will examine the three basic RDF rules. The goal is to establish a sound understanding of why these basic RDF rules provide the foundation to the above aspects of the Semantic Web.

2.5.1 Information Understandable by Machine

Let us start from Rule 1. We have seen this rule already, where it was presented to describe the abstract RDF model. Here we will look at it again from a different perspective: it plays an important role when making machines understand the knowledge expressed in RDF statements. Here is this rule again:

Rule #1:

Knowledge (or information) is expressed as a list of statements, each statement takes the form of Subject-Predicate-Object, and this order should never be changed.

Before we get into the details on this part, let us take a look at this triple pattern once more time.

Since the value of a property can be a literal or a resource, a given RDF statement can take the form of alternating sequence of resource–property, as shown in List 2.45.

List 2.45 The pattern of RDF statement

```

1: <rdf:Description rdf:resources="#resource-0">
2:   <someNameSpace:property-0>
3:     <rdf:Description rdf:resource="#resource-1">
4:       <someNameSpace:property-1>
5:         <rdf:Description rdf:resource="#resource-2">
6:           <someNameSpace:property-2>
7:             ...
8:           </someNameSpace:property-2>
9:         </rdf:Description>
10:       </someNameSpace:property-1>
11:     </rdf:Description>
12:   </someNameSpace:property-0>
13: </rdf:Description>

```

In List 2.45, #resource-0 has a property named property-0; its value is another resource described using lines 3–11 (#resource-1). Furthermore, #resource-1 has a property named property-1 whose value is yet another resource described using lines 5–9. This pattern can go on and on; however, the Resource-Property-Value structure is never changed.

Why is this order so important? Because if we follow this order when we create RDF statements, an RDF-related application will be able to understand the meaning of these statements. To see this, let us study the example shown in List 2.46.

List 2.46 One simple statement about Nikon D300

```

1: <?xml version="1.0"?>
2:

```

```

3: <rdf:RDF
3a:   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
4:   xmlns:myCamera="http://www.liyangyu.com/camera#">
5:
6:   <rdf:Description
6a:     rdf:about="http://www.liyangyu.com/camera#Nikon_D300">
7:     <myCamera:effectivePixel>12.1M</myCamera:effectivePixel>
8:   </rdf:Description>
9:
10: </rdf:RDF>

```

List 2.46 is equivalent to the following RDF statement:

```
myCamera:Nikon-D300 myCamera:effectivePixel 12.1M
```

We, as human reader, understand its meaning. For a given application, the above triple looks more like this:

```
$#!6^:af#@dy $#!6^:3pyu9a 12.1M
```

However, the application does understand the structure of an RDF statement, so the following is true as far as the application is concerned:

```

$#!6^:af#@dy is the subject
$#!6^:3pyu9a is the property
12.1 M is the value

```

And now, here is the interesting part: the application also has a vocabulary it can access, and the following fact is stated in this vocabulary:

```
property $#!6^:3pyu9a is used exclusively on resource whose type is
$#!6^:Af5%
```

We will see what exactly is this vocabulary (in fact, it is called RDF Schema), and we will also find out how to express the above fact by using this vocabulary in [Chap. 4](#). For now, let us just assume the above fact is well expressed in the vocabulary.

Now all these said, the application, without really associating any special meaning to the above statement, can draw the following conclusion:

```
resource $#!6^:af#@dy is an instance of resource $#!6^:Af5%
```

When the application shows the above conclusion to the screen, for human eyes, that conclusion looks like the following:

```
Nikon-D300 is an instance of DSLR
```

which makes perfect sense!

The key point here is a given application cannot actually associate any special meanings to the RDF statements. However, with the fix structure of statement and some extra work (the vocabulary, for instance), the logical pieces of meaning can be

mechanically maneuvered by the given application. It therefore can act as if it *does* understand these statements. In fact, in [Chaps. 4 and 5](#), once we understand more about RDF Schema and OWL, we will see more examples of this exciting inference power.

2.5.2 Distributed Information Aggregation

The second and third rules are important for distributed information aggregation. Here is again Rule #2:

Rule #2:

The name of a resource must be global and should be identified by Uniform Resource Identifier (URI). The name of predicate must also be global and should be identified by URI as well.

And Rule #3 is given below:

Rule #3:

I can talk about any resource at my will, and if I chose to use an existing URI to identify the resource I am talking about, then the following is true:

- *The resource I am talking about and the resource already identified by this existing URI are exactly the same thing or concept.*
- *Everything I have said about this resource is considered to be additional knowledge about that resource.*

These two rules together provide the foundation for distributed information aggregation. At this point, they seem to be trivial and almost like a given already. However, they are the key idea behind the Linked Open Data project (see [Chap. 11](#)), and they are the starting point for new knowledge discovery. We will see lots of these exciting facts in the later chapters. For now, a simple comparison of the traditional Web and the “Web of RDF documents” may give you a better understanding of their importance.

Recall the situation in the current Web. One of the things about the Internet that is quite attractive to all of us is the fact that you can talk about anything you want, and you can publish anything you want. When you do this, you can also link your document to any other pages you would like to.

For example, assume on my own Web site (www.liyangyu.com), I have offered a review about Nikon D300, and I also linked my page to some digital camera review site. Someone else perhaps did the same and has a link to the same digital camera review site as well. What will this do to this review site? Not much at all, except that some search engines will realize the fact that quite a few pages have link to it, and the rank of this site should be adjusted to be a little bit more important. But this is pretty much all of it; the final result is still the same: the Web is a huge distributed information storage place, from which getting information is normally pretty hard.

On the other hand, on the “Web of RDF documents,” things can be quite different. For example, based on the above rule, all the RDF documents containing a resource identified by the same known URI can be connected together. This connection is implemented based on this URI which has a well-defined meaning. Even though these RDF documents are most likely distributed everywhere on the Web, however, each one of them presents some knowledge about that resource, and adding them together can produce some very powerful result.

More specifically, when I publish my review of D300, all I need to do is to use a URI to represent this resource. Anyone else wants to review the same camera has to use the same URI. These reviews can then be automatically aggregated to produce the summary one might want to have. An example along this path will be discussed in the next section.

One last point before we move on. It is clear to us now that only named resource can be aggregated. Therefore, anonymous resource cannot be aggregated. The reason is simple: if a resource in a document is anonymous, an aggregation tool will not be able to tell if this resource is talking about some resource already been defined and described. This is probably one disadvantage of using anonymous resources.

2.5.3 A Hypothetical Real-World Example

It is now a good time to go back to our original question: as a quality engineer who is working for Nikon, my assignment is to read all these reviews and summarize what people have said about Nikon SLR cameras. I will have to report back to Nikon’s design department, so they can make better designs based on these reviews.

And as we have discussed, we need a standard so that we can develop an application that will read all these reviews and generate a report automatically.

Now, with the RDF standard being in place, how should I proceed with this task? The following steps present one possible solution I can use:

Step 1. Create a group of URIs to represent Nikon’s digital camera products.

At this point, you should understand why this step is necessary. The following are some possible choices for these URIs:

```
http://www.liyangyu.com/camera#Nikon_D300  
http://www.liyangyu.com/camera#Nikon_D90  
http://www.liyangyu.com/camera#Nikon_D60
```

Obviously, we should be re-using URIs as much as we can. For example, the following URIs taken from DBpedia are good choices:

```
http://dbpedia.org/resource/Nikon_D300  
http://dbpedia.org/resource/Nikon_D90  
http://dbpedia.org/resource/Nikon_D60
```

However, for this hypothetical example, we are fine with making up new URIs.

Step 2. Provide a basic collection of terms that one can use to review a camera.

This step is also a critical step, and we will see a lot more about this step in later chapters. For now, we can understand this step like this: with only the URIs to represent different cameras, reviewers themselves are not able to share much of their knowledge and common language about cameras.

To make a common language among the reviewers, we can provide some basic terms for them to use when reviewing cameras. The following are just two example terms at this point:

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

and we can add more terms and collect these terms together to create a vocabulary for the reviewers to use.

Recall that RDF model should be flexible enough that anyone can say anything about a resource. What if some reviewer wants to say something about a camera, and the term he/she wants to use is not included in our vocabulary? The solution is simple: he/she can simply download the vocabulary, add that term, and then upload the vocabulary for all the reviewers to use, as simple as this.

Now, a key question arises. Assume I have already developed an automatic tool that can help me to read all these reviews and generate a summary report. If the vocabulary is undergoing constant update, do I have to change my application constantly as well?

The answer is no. Probably it is not easy to see the reason at this point, but this is exactly where the flexibility is. More specifically, with a set of common URIs and a shared vocabulary, distributed RDF graphs can be created by different sources on the Web, and applications operating on these RDF models are extremely robust to the change of the shared vocabulary.

You will see this more clearly in later chapters. For now, understand this is a concrete implementation of one of the design goals of RDF standard: it has to be flexible enough that anyone can say anything about a given resource.

Step 3. Make sure the reviewers will use the given set of URIs and the common vocabulary when they publish their reviews on the Web.

This is probably the most difficult step: each reviewer has to learn RDF and has to use the given URIs to represent cameras. In addition, they have to use the given vocabulary as well, although they do have the flexibility of growing the vocabulary as discussed above.

The issue of how to make sure they will accept this solution is beyond the scope of this book – it is not related to the technology itself. Rather, it is about the acceptance of the technology.

With this said, we will simply assume the reviewers will happily accept our solution. To convince yourself about this assumption, think about the very reason of being a reviewer. For any reviewer, the goal is to make sure his/her voice is heard by both the consumers and the producers of a given product. And if this reviewer is

not publishing his/her review in RDF document by using the given URIs and vocabulary, his/her review will never be collected, therefore he/she will not have a chance to make a difference about that product at all.

Step 4. Build the application itself and use it to collect reviews and generate reports.

This is in fact the easy part. This application will first act like a crawler that will visit some popular review sites to collect all the RDF documents. Once the documents are collected, all the statements in these RDF documents will be grouped based on their subjects, i.e., those statements that have the same subject will be grouped together regardless of which RDF document they are originally from, and this is exactly what data aggregation is.

Clearly, one such group represents all the reviews for a given camera, if the URI that represents that camera is used as the subject. Once this is done, a report about this camera can be generated by querying the statements in this group.

Let us take a look at a small example. Imagine the application has collected the statements shown in List 2.5 already. In addition, it has also collected the statements shown in List 2.47 from another reviewer.

List 2.47 Statements about Nikon D300 from another reviewer

subject	predicate	object
myCamera:Nikon_D300	myCamera:effectivePixel	"12.1M"
myCamera:Nikon_D300	myCamera:shutterrange	"30s - 1/8000s"
myCamera:Nikon_D300	myCamera:wb	"auto"

Clearly, the statements from List 2.5 and the statements from List 2.47 are all about the same Nikon D300 camera, so these statements can be aggregated together into a single group.

Now repeat the same procedure as described above. Obviously, more and more statements about Nikon D300 will be collected from different reviewers and will be added to the same statement group. It is not hard to imagine this group will contain quite a large number of statements once our application has visited enough review sites.

Once the application stops its crawling on the Web, we can implement different queries against the collected statements in this group. To see how this can be done, take a look at the example code (List 2.7) presented in Sect. 2.2.5. The only difference now is the fact that we have many more statements than the simple test case discussed in Sect. 2.2.5. Therefore, more interesting results can be expected. Clearly, you can implement different queries, but the basic idea remains the same.

As a side note, recall we claimed that any new terms added by reviewers would not disturb the application itself. To see this, consider the query *what properties did the reviewers use to describe Nikon D300?* This query is important to Nikon's design department, since it shows the things that consumers would care about for a given camera. As you can tell, to implement this query, a simple pattern match is done as shown in List 2.7, and only the subject has to be matched; the property part is what we want to collect for this query. Obviously, the reviewers can add new terms (properties) and these added new terms will not require any change to the code.

Finally, it is interesting to think about this question: exactly what do all the reviewers have to agree upon to make this possible?

Surprisingly, the only two things all the reviewers have to agree upon are as follows:

- Reviewers have to agree to use RDF.
- Reviewers have to agree to use the given URIs instead of inventing their own.

What about the basic vocabulary that reviewers use to review cameras? We don't have to reach an agreement on that at all – one can add new properties without disturbing the application, as we have just discussed. Furthermore, adding a new term does not require any agreement from other reviewers either. We do provide an initial version of the vocabulary; however, it is merely a starting point for reviewers to use, not something that everyone has to agree upon.

In addition, the pseudo-code in List 2.7 does not need to know anything about the nature of the data in the statements in order to make use of it. Imagine even when we change to another application domain, the pseudo-code in List 2.7 will not change much at all.

To summarize our point: with the help from RDF standard, we can indeed create an application that can help us to finish our job with much more ease.

2.6 More About RDF

At this point, you have gained fairly solid understanding about RDF. Before we move on to the next chapter, we have several more issues to cover here, and some of them are probably on your mind already for quite a while.

2.6.1 *Dublin Core: Example of Pre-defined RDF Vocabulary*

In this chapter, we have used terms from Dublin Core vocabulary without formally introducing it. Chance is you will see terms from Dublin Core vocabulary in different RDF documents quite often. So in this section, let us focus on Dublin Core vocabulary.

To put it simply, Dublin Core is a set of pre-defined URIs representing different properties of a given document. Since they are widely used in RDF documents, they can also be understood as another set of pre-defined RDF vocabulary.

Dublin Core was developed in the *March 1995 Metadata Workshop* sponsored by the Online Computer Library Center (OCLC) and the National Center for Supercomputing Applications (NCSA). The workshop itself was held in Dublin, Ohio, hence the name Dublin Core. Currently, it is maintained by the Dublin Core metadata Initiative⁵ project.

⁵<http://dublincore.org/>

Table 2.3 Element examples in Dublin Core Metadata Scheme

Element name	Element description
Creator	This element represents the person or organization responsible for creating the content of the resource, e.g., authors in the case of written documents
Publisher	This element represents the entity responsible for making the resource available in its present form. It can be a publishing house, a university department, etc
Contributor	This element represents the person or organization not specified in a <code>Creator</code> element who has made significant intellectual contributions to the resource but whose contribution is secondary to any person or organization specified in a <code>Creator</code> element, e.g., editor, transcriber, illustrator
Title	This element represents the name given to the resource, usually by the <code>Creator</code>
Subject	This element represents the topic of the resource. Normally this will be expressed as keywords or phrases that describe the subject or content of the resource
Date	This element represents the date associated with the creation or availability of the resource
Identifier	This element is a string or number that uniquely identifies the resource. Examples include URLs, Purls, and ISBN, or other formal names
Description	This element is a free text description of the content of the resource. It can be in flexible format, including abstracts or other content descriptions
Language	This element represents the language used by the document
Format	This element identifies the data format of the document. This information can be used to identify the software that might be needed to display or operate the resource, e.g., postscript, HTML, text, jpeg, XML

Dublin Core has 15 elements called the Dublin Core metadata element set (DCMES). It is proposed as the minimum number of metadata elements required to facilitate the discovery of document-like objects in a networked environment such as the Internet. Table 2.3 shows some of these terms.

Generally speaking, if we are using RDF to describe a document, or maybe part of our RDF document is to describe a document, we should use Dublin Core predicates as much as we can. For example, `Title` predicate and `Creator` predicate are all good choices.

Note that the URIs in Dublin Core vocabulary all have the following lead strings:

```
http://www.purl.org/metadata/dublin-core#
```

By convention, this URI prefix string is associated with namespace prefix `dc:` and is typically used in XML with the prefix `dc`.

For example, List 2.48 is a simple RDF description about my personal Web page. The two statements use Dublin Core terms to indicate the creator of this Web site and the date this site was created (lines 8 and 9).

List 2.48 Example of using Dublin Core terms

```

1: <?xml version="1.0"?>
2: <!DOCTYPE rdf:RDF
3:
4: <rdf:RDF
5:     xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
6:     xmlns:dc="http://www.purl.org/metadata/dublin-core#">
7:   <rdf:Description rdf:about="http://www.liyangyu.com">
8:     <dc:creator>Liyang Yu</dc:creator>
9:     <dc:date rdf:datatype="&xsd:date">2006-09-10</dc:date>
10:   </rdf:Description>
11:
12: </rdf:RDF>

```

We can certainly add more if we want to describe more information. But you see how easy it is to use it: you just need to specify the Dublin Core namespace and use it anywhere you want in your document.

2.6.2 XML vs. RDF?

The relationship between XML and RDF can be described quite simply: RDF and XML are not much related at all.

RDF, as you have seen, is a standard for describing things in the real world. More importantly, these descriptions can be processed by machines on a large scale. To serialize an RDF abstract model, different serialization formats are available. Among these formats, RDF/XML is recommended by W3C and used in most documents. Therefore, the only connection between RDF and XML is the fact that RDF uses the XML syntax and its namespace concept.

Given this relationship between XML and RDF, perhaps a better question to ask is why XML cannot accomplish what RDF has accomplished?

There are several reasons behind this. First of all, XML provides very limited semantics, and even for this limited semantics, it is quite ambiguous. This fact is nicely summarized as follows:

XML is only the first step to ensuring that computers can communicate freely. XML is an alphabet for computers and as everyone traveling in Europe knows, knowing the alphabet doesn't mean you can speak Italian or French. – *Business Week*, March 18th 2002

The key point here is XML is by far the best format to share data on the Web and exchange information between different platforms and applications. However, it does not have enough restrictions to successfully express semantics.

Let us look at one example. How do we use XML to express the following knowledge: “the author of *A Developer’s Guide to the Semantic Web* is Liyang Yu”? Using XML, you have several ways to do this. See List 2.49.

List 2.49 Ambiguity of XML document

```

<!-- form 1 -->
<author>
  <firstName>Liyang</firstName>
  <lastName>Yu</lastName>
  <book>
    <title>A Developer's Guide to the Semantic Web</title>
  </book>
</author>

<!-- form 2 -->
<author>
  <name>Liyang Yu</name>
  <book>
    <title>A Developer's Guide to the Semantic Web</title>
  </book>
</author>

<!-- form 3 -->
<author>
  <name>Liyang Yu</name>
  <book>A Developer's Guide to the Semantic Web</book>
</author>

```

Clearly, there is no agreement on the structure one can use. This makes an automatic agent which intends to work on a large scale become virtually impossible, if not prohibitively expensive.

On the other hand, using RDF to express the same idea is very straightforward, and it leaves no space for any ambiguity, as shown in List 2.50.

List 2.50 Use RDF document to express the fact described in List 2.49

```

1: <rdf:RDF
1a:   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
2:   xmlns:dc="http://www.purl.org/metadata/dublin-core#">
3:
4: <rdf:Description
4a:   rdf:about="http://www.liyangyu.com/book#SemanticWeb">
5:   <dc:title>A Developer's Guide to the Semantic Web</dc:title>
6:   <dc:creator>Liyang Yu</dc:creator>
7: </rdf:Description>
8:
9: </rdf:RDF>

```

The only thing you can change in List 2.50 is the URI that represents the book (line 4). For example, you have to mint one if it does not already exist. Any RDF

application can easily characterize this structure and understand which part of the structure is the subject, the property, and the value of that property.

Second, parsing XML statements heavily depends on the tree structure, which is not quite scalable on a global basis. To be more specific, you can easily make up some XML document so that the representation of this document in machine's memory depends on the data structures such as tree and character strings. In general, these data structures can be quite hard to handle, especially when the amount is large.

RDF statement presents a very simple data structure – a directly labeled graph which has long been a very well understood data structure in the field of computer science. It is also quite scalable for large dataset. The nodes of the graph are the resources or literals, the edges are the properties, and the labels are URIs of nodes and edges. You can certainly change the graph into a collection of triples (subject–predicate–object), which fits into the framework of relational database very well. All these are quite attractive compared to XML documents.

The third reason, which is even more important, is that using RDF format promotes the development and usage of standardized vocabularies (or, ontologies, as you will see in the later chapters). The more you understand about the Semantic Web, the more you will appreciate the importance of these vocabularies. The following are some of the benefits of using standard vocabularies:

- Without a shared vocabulary, it is always possible that the same word can mean different concepts and different words can refer to the same concept.
- Without a shared vocabulary, distributed information will likely remain isolated. An application that is capable of processing this distributed information on a global scale will be very hard to build.
- Without a shared vocabulary, machine inferencing will be difficult to implement. Therefore new knowledge discovered will be difficult to do.
- There are much more, as we will see in the later chapters.

At this point, the above might not seem quite clear and convincing. However, as your understanding about the Semantic Web grows, they will become more obvious to you.

As a conclusion, XML is unequalled as an information exchange format over the Internet. But by itself, it simply does not provide what we need for the construction of the Semantic Web.

If you are still not convinced, do this small experiment. Take the hypothetical example we have discussed earlier, pretend there is no RDF standard at all. In other words, replace all the RDF documents with XML documents, see how many more constraints you need to artificially impose to make it work, and how many more case-specific code you need to write. You will see the benefit of RDF abstract model quite easily.

2.6.3 Use an RDF Validator

One last thing before we move on to the next chapter: use an RDF validator.

As you have seen by now, RDF/XML syntax can be quite convoluted and error-prone, especially when you are creating RDF documents by hand. One good idea is to use a validator whenever you can.

There are a number of available validators; you can choose anyone you like. For example, I have been using the RDF validator provided by W3C for quite a while. This validator can be accessed from the location

<http://www.w3.org/RDF/Validator/>

Figure 2.10 shows its current look-and-feel.



Fig. 2.10 RDF validator provided by W3C

To use this validator, simply paste the RDF document into the document window, and click `Parse RDF` button. You can also ask for an RDF graph by making the corresponding selection using the `Triples and/or Graph` drop-down list. You can further specify the graph format in the `Graph format` drop-down list, as shown in Fig. 2.10.

If there is indeed any error in your document, the validator will flag it by telling you the line and column from where the error occurs. You can always make changes to your RDF document and submit it again, until you have a valid RDF document.

2.7 Summary

In this chapter, we have learned RDF, the building block for the Semantic Web.

The first thing we should understand from this chapter is the RDF abstract model. More specifically, this abstract model includes the following main points:

- It provides a framework for us to represent knowledge in a way that can be processed by machines.
- It involves important concepts such as resource, statement (triple), subject, object, predicate, and RDF graph.
- It has fundamental rules that one should follow when using RDF model to represent structured information on the Web. These rules include that the structure of a statement has to be in the form of subject–predicate–object, and URIs should be used to identify subject, predicate, and object.

In order for us to create and operate with concrete RDF documents, this chapter also covers the two major RDF serialization formats, including RDF/XML syntax and Turtle language. We should have learned the following:

- the concept of RDF vocabulary, and why this vocabulary is important when it comes to RDF serialization;
- understand the main features of RDF/XML syntax, including all the language constructs (terms from the RDF vocabulary) that can be used to represent an RDF model;
- understand the main features of Turtle language, and how to use it to represent an RDF model.

This chapter also discusses the reason why RDF is the choice for expressing knowledge that machines can understand. Examples are used to show the power of RDF, and a detailed discussion about distributed information aggregation using RDF is also included. We should have learned the following main points:

- what exactly it means when we claim RDF graphs can be understood by machine;
- why the fundamental rules about RDF are important in order for machine to understand and operate with RDF graphs;
- why URI reuse is important for distributed information aggregation.

Finally, this chapter discusses some related issues about RDF. This includes the following:

- Dublin Core, as an example of another pre-defined RDF vocabulary;
- the relationship between XML and RDF; and
- tools and support you can use when working with concrete RDF models.

At this point, make sure you have established a clear understanding about all these main points included in this summary. If not, review the material in this chapter before you move on.

Chapter 3

Other RDF-Related Technologies: Microformats, RDFa, and GRDDL

3.1 Introduction: Why Do We Need These?

So far at this point, we have learned the concept of the Semantic Web, and we have learned RDF. Let us think about these two for a moment.

Recall that the vision of the Semantic Web is to add meaning into the current Web so machines can understand its contents. Based on what we have learned about RDF, we understand that RDF can be used to express the meaning of a Web document in a machine-processable way. More specifically, for a given Web document, we can create a set of RDF triples to describe its meaning and somehow indicate to the machine that these RDF statements are created for the machine to understand this document.

Although we are not quite there yet, it is not hard for us to understand the feasibility of this idea. In fact, it is called *semantic markup* as we will see in later chapters.

However, there is one obvious flaw with this approach: it is simply too complex for most of us. More specifically, to finish this markup process, we have to first create a collection of RDF statements to describe the meaning of a Web document, then put them into a separate file, and finally, we have to somehow link the original Web document to this RDF file. Is there a simpler way of doing all these?

The answer is yes, and that is to use microformats or RDFa. They are simpler since microformats or RDFa constructs can be *directly* embedded into XHTML to convey the meaning of the document itself, instead of collecting them into separated documents.

This in fact plays an important role in the grand plan for the Semantic Web, since a single given Web page is now readable not only by human eyes, but also by machines. A given application which understands microformats or RDFa can perform tasks that are much more complex than those performed by the applications that are built solely based on screen scraping. In fact, in [Chap. 8](#), we will see two Semantic Web applications created by Yahoo! and Google, respectively, and they are the direct results of microformats and RDFa.

To understand how GRDDL (pronounced “*griddle*”) fits into the picture, think about the semantic information an XHTML page contains when it is embedded

with microformats or RDFa constructs. It will be quite useful if we can obtain RDF statements from this XHTML page automatically. GRDDL is a tool that can help us to accomplish this. Once we can do this, the RDF statements harvested from these XHTML pages can be aggregated together to create even more powerful applications.

And these are the reasons why we need microformats, RDFa and GRDDL. If you skip this chapter for now, you can still continue learning the core technology components of the Semantic Web. However, you need to understand this chapter in order to fully understand [Chap. 8](#).

3.2 Microformats

3.2.1 Microformats: The Big Picture

To put it simple, microformats are a way to embed specific semantic data into the HTML content that we have today, so when a given application accesses this content, it will be able to tell what this content is about.

We are all familiar with HTML pages that represent people, so let us start from here. Let us say we would like to use microformats to add some semantic data about people. To do so, we need the so-called `hCard` microformat, which offers a group of constructs you can use to mark up the content:

- a root class called `vcard`;
- a collection of properties, such as `fn` (formatted name) and `n` (name), and quite a few others.

We will see more details about `hCard` microformat in the next section. For now, understand that `hCard` microformat can be used to mark up the page content where a person is described. In fact, `hCard` microformat not only is used for people, but can also be used to mark up the content about companies, organizations and places, as we will see in the next section.

Now, what if we would like to mark up some other content? For example, some event described in a Web document? In this case, we will need to use the `hCalendar` microformat, which also provides a group of constructs we can use to mark up the related content:

- a root class called `vcalendar`;
- a collection of properties, such as `dtstart`, `summary`, `location`, and quite a few others.

By the same token, if we would like to mark up a page content that contains a person's resume, we then need to use the `hResume` microformat. What about `hRecipe` microformat? Obviously, it is used for adding markups to a page content where a cooking recipe is described.

By now, the big picture about microformats is clear, and we can define microformats as follows:

Microformats are a collection of individual microformats, with each one of them representing a specific domain (such as person, event, location) that can be described by a Web content page. Each one of these microformats provides a way of adding semantic markups to these Web pages, so that the added information can be extracted and processed by software applications.

With this definition in mind, it is understandable that the microformats collection is always growing: there are existing microformats that cover a number of domains, and for the domains that have not been covered yet, new microformats are created to cover them.

For example, hCard microformat and hCalendar microformat are stable microformats; hResume microformat and hRecipe microformat are still in draft states. In fact, there is a microformats community that is actively working on new microformats. You can always find the latest news from their official Web site,¹ including a list of stable microformats and a list of draft ones that are under discussion.

Finally, note that microformats are not a W3C standard or recommendation. They are offered by an open community and are open standards originally licensed under Creative Commons Attribution. They have been placed into the public domain since 29 December 2007.

3.2.2 Microformats: Syntax and Examples

In this section, we will take a closer look at how to use microformats to mark up a given Web document. As we have discussed earlier, microformats are a collection of individual microformats, and to present each one of them in this chapter is not only impossible but also unnecessary. In fact, understanding one of such microformats will be enough; the rest of them are quite similar when it comes to actually using them to mark up a page.

With this said, we will focus on hCard microformat in this section. The reason being that at the time of this writing, hCard microformat is considered to be one of the most popular and well-established microformats. We will begin with an overview of hCard microformat, followed by some necessary HTML knowledge, and as usual, we will learn hCard by examples.

3.2.2.1 From vCard to hCard Microformat

hCard microformat has its root in vCard and can be viewed as a vCard representation in HTML, hence the letter *h* in hCard (HTML vCard). It is therefore helpful to have a basic understanding about vCard.

¹<http://microformats.org>

Table 3.1 Example properties contained in vCard standard

Property name	Property description	Semantic
N	Name	The name of the person, place, or thing associated with the vCard object
FN	Formatted name	The formatted name string associated with the vCard object
TEL	Telephone	Phone number string for the associated vCard object
EMAIL	E-mail	E-mail address associated with the vCard object
URL	URL	A URL that can be used to get online information about the vCard object

vCard is a file format standard that specifies how basic information about a person or an organization should be presented, including name, address, phone numbers, e-mail addresses and URLs. This standard was originally proposed in 1995 by the Versit Consortium, which had Apple, AT&T Technologies, IBM and Siemens as its members. In late 1996, this standard was passed on to the Internet Mail Consortium, and since then it has been used widely in address book applications to facilitate the exchange and backup of contact information.

To this date, this standard has been given quite a few extensions, but its basic idea remains the same: vCard has defined a collection of properties to represent a person or an organization. Table 3.1 shows some of these properties.

Since this standard was formed before the advent of XML, the syntax is just simple text that contains property–value pairs. For example, my own vCard object can be expressed as shown in List 3.1.

List 3.1 My vCard object

```
BEGIN:VCARD
FN:Liyang Yu
N:Yu;Liyang;;;
URL:http://www.liyangyu.com
END:VCARD
```

First off, note this vCard object has a `BEGIN:VCARD` and `END:VCARD` element, which marks the scope of the object. Inside the object, the `FN` property has a value of `Liyang Yu`, which is used as the display name. The `N` property represents the structured name, in the order of first, last, middle names, prefixes and suffixes, separated by semicolons. This can be parsed by a given application so as to understand each component in the person’s name. Finally, `URL` is the URL of the Web site that provides more information about the vCard object.

With the understanding about vCard standard, it is much easier to understand hCard microformat, since it is built directly on the vCard standard. More specifically, the properties supported by the vCard standard are mapped directly to the properties and sub-properties contained in hCard microformat, as shown in Table 3.2.

Table 3.2 Examples of mapping vCard properties to hCard properties

vCard property	hCard properties and sub-properties
FN	fn
N	n with sub-properties: family-name, given-name, additional-name, honorific-prefix, honorific-suffix
EMAIL	email with sub-properties: type, value
URL	url

Note Table 3.2 does not include all the property mappings, and you can find the complete mappings from microformats' official Web site (see Sect. 3.2.1). As a high-level summary, hCard properties can be grouped into six categories:

- Personal information properties: these include properties such as `fn`, `n`, `nickname`.
- Address properties: these include properties such as `adr`, with sub-properties such as `street-address`, `region` and `postal-code`.
- Telecommunication properties: these include properties such as `email`, `tel`, and `url`.
- Geographical properties: these include properties such as `geo`, with sub-properties such as `latitude` and `longitude`.
- Organization properties: these include properties such as `logo`, `org`, with sub-properties such as `organization-name` and `organization-unit`.
- Annotation properties: these include properties such as `title`, `note`, and `role`.

With the above mapping in place, the next issue is to represent a vCard object (contained within `BEGIN:VCARD` and `END:VCARD`) in hCard microformat. To do so, hCard microformat uses a root class called `vcard`, and in HTML content, an element with a class name of `vcard` is itself called an hCard.

Now, we are ready to take a look at some examples to understand how exactly we can use hCard microformat to mark up some page content.

3.2.2.2 Using hCard Microformat to Mark Up Page Content

Let us start with a very simple example. Suppose that in one Web page, we have some HTML code as shown in List 3.2.

List 3.2 Example HTML code without hCard microformat markup

```
... <!-- other HTML code -->
<div>
  <a href="http://www.liyangyu.com/">Liyang Yu</a>
</div>
... <!-- other HTML code -->
```

Obviously, for our human eyes, we understand that the above link is pointing to a Web site which describes a person named Liyang Yu. However, any application that sees this code does not really understand that, except for showing a link on the screen as follows:

Liyang Yu

Now let us use `hCard` microformat to add some semantic information to this link. The basic rules when doing markup can be summarized as follows:

- use `vcard` as the class name for the element that needs to be marked up, and this element now becomes a `hCard` object, and
- the properties of an `hCard` object are represented by elements inside the `hCard` object. An element with class name taken from a property name represents the value of that property. If a given property has sub-properties, the values of these sub-properties are represented by elements inside the element for that given property.

Based on these rules, List 3.3 shows one possible markup implemented by using `hCard` microformat.

List 3.3 `hCard` microformat markup added to List 3.2

```
... <!-- other HTML code -->
<div class="vcard">
  <div class="fn">Liyang Yu</div>
  <div class="n">
    <div class="given-name">Liyang</div>
    <div class="family-name">Yu</div>
  </div>
  <div class="url">http://www.liyangyu.com</div>
</div>
... <!-- other HTML code -->
```

This markup is not hard to follow. For example, the root class has a name given by `vcard`, and the property names are used as class names inside it. And certainly, this simple markup is able to make a lot of difference to an application: any application that understands `hCard` microformat will be able to understand the fact that this is a description of a person, with the last name, first name and URL given.

If you open up List 3.3 using a browser, you will see it is a little bit different from the original look-and-feel. Instead of a clickable name, it actually shows the full name, first name, last name and the URL separately. So let us make some changes to our initial markup, without losing the semantics, of course.

First off, a frequently used trick when implementing markup for HTML code comes from the fact that `class` (also including `rel` and `rev` attributes) attribute in HTML can actually take a space-separated list of values. Therefore, we can combine `fn` and `n` to reach something as shown in List 3.4.

List 3.4 An improved version of List 3.3

```
... <!-- other HTML code -->
<div class="vcard">
  <div class="n fn">
    <div class="given-name">Liyang</div>
    <div class="family-name">Yu</div>
  </div>
  <div class="url">http://www.liyangyu.com</div>
</div>
... <!-- other HTML code -->
```

This is certainly some improvement: at least we don't have to encode the name twice. However, if you open up List 3.4 in a browser, it still does not show the original look. To go back to its original look, at least we need to make use of element `<a>` together with its `href` attribute.

In fact, microformats do not force the content publishers to use specific elements; we can choose any element and use it together with the `class` attribute. Therefore, List 3.5 will be our best choice.

List 3.5 Final hCard microformat markup for List 3.2

```
... <!-- other HTML code -->
<div class="vcard">
  <a class="n fn url" href="http://www.liyangyu.com">
    <span class="given-name">Liyang</span>
    <span class="family-name">Yu</span>
  </a>
</div>
... <!-- other HTML code -->
```

And this is it: if you open up List 3.5 from a Web browser, you get exactly the original look-and-feel. And certainly, any application that understands hCard microformat will be able to understand what a human eye can see: this is a link to a Web page that describes a person, whose last name is Yu and first name is Liyang.

List 3.6 is another example of using hCard microformat. It is more complex and certainly more interesting. We present it here so you can get more understanding about using hCard microformat to mark up content files.

List 3.6 A more complex hCard microformat markup example

```
<div id="hcard-liyang-yu" class="vcard">
  <a class="n fn url" href="http://www.liyangyu.com">
    <span class="given-name">Liyang</span>
    <span class="family-name">Yu</span>
  </a>
  <div class="org">Delta Air Lines</div>
```

```

<div class="tel">
  <span class="type">work</span>
  <span class="value">404.773.8994</span>
</div>
<div class="adr">
  <div class="street-address">1030 Delta Blvd.</div>
  <span class="locality">Atlanta</span>,
  <span class="region">GA</span>
  <span class="postal-code">30354</span>
  <div class="country-name">USA</div>
</div>
<a class="email" href="mailto:liyang.yu@delta.com">
  liyang.yu@delta.com
</a>

</div>

```

And List 3.7 shows the result rendered by a Web browser.

List 3.7 Rendering result of List 3.6

```

Liyang Yu
Delta Air Lines
work 404.773.8994
1030 Delta Blvd.
Atlanta, GA 30354
USA


```

3.2.3 Microformats and RDF

At this point, we have learned hCard microformat. With what you have learned here, it is not hard for you to explore other microformats on your own.

In this section, we will first summarize the benefits offered by microformats, and more importantly, we will also take a look at the relationship between microformats and RDF.

3.2.3.1 What Is So Good About Microformats?

First off, microformats do not require any new standards; instead, they leverage existing standards. For example, microformats reuse HTML tags as much as possible, since almost all the HTML tags allow `class` attributes to be used.

Second, the learning curve is minimum for content publishers. They continue to mark up their Web documents as they normally would. The only difference is that they are now invited to make their documents more semantically rich by using `class` attributes with standardized properties values, such as those from hCard microformat as we have discussed.

Third, the added semantic markup has no impact on the document's presentation, if it is done right.

Lastly, and perhaps the most important one, is the fact that this small change in the markup process does bring a significant change to the whole Web world. The added semantic richness can be utilized by different applications, since applications can start to understand at least part of the document on the Web now. We will see some exciting applications in [Chap. 8](#), and it is also possible that at the time you are reading this book, more applications built upon microformats have become available to us.

With this said, how is microformats related to RDF? Do we still need RDF at all? Let us answer these questions in the next section.

3.2.3.2 Microformats and RDF

Obviously, the primary advantage microformats offer over RDF is the fact that we can embed metadata directly in the XHTML documents. This not only reduces the amount of markup we need to write, but also provides one single content page for both human readers and machines. The other advantage of microformats is that microformats have a simple and intuitive syntax, and therefore do not need much of a learning curve compared to RDF.

However, microformats were not designed to cover the same scope as RDF was, and they simply do not work on the same exact level. To be more specific, the following are something offered by RDF, but not by microformats (note that at this point, you may not be able to fully appreciate all the items in the list, but after you read more of this book, you will be able to):

- RDF does not depend on pre-defined “formats,” and it has the ability to utilize, share, and even create any number of vocabularies.
- With the help from these vocabularies, RDF statements can participate in reasoning process and new facts can be discovered by machines.
- Resources in RDF statements are represented as URIs, allowing a Linked Data Web to be created.
- RDF itself is infinitely extensible and open-ended.

You can continue to grow this list once you learn more about microformats and RDF from this book and your real development work. However, understanding microformats is also a must, and this will at least enable you to pick up the right tool for the right situation.

3.3 RDFa

3.3.1 RDFa: *The Big Picture*

With what we have learned so far, the big picture of RDFa is quite simple to understand: it is just another way to directly add semantic data into XHTML pages. Unlike microformats which reuse the existing `class` attribute on most HTML tags, RDFa

provides a set of new attributes that can be used to carry the added markup data. Therefore, in order to use RDFa to embed the markup data within the Web documents, some attribute-level extensions to XHTML have to be made. In fact, this is also the reason for the name: RDFa means RDF in HTML attributes.

Note that unlike microformats, RDFa is a W3C standard. More specifically, it became a W3C standard on 14 October 2008, and you can find the main standard document on W3C official Web site.² Based on this document, RDFa is officially defined as follows:

RDFa is a specification for attributes to express structured data in any markup language.

Another W3C RDFa document, *RDFa for HTML Authors*,³ has provided the following definition of RDFa:

RDFa is a thin layer of markup you can add to your web pages that make them understandable for machines as well as people. By adding it, browsers, search engines, and other software can understand more about the pages, and in so doing offer more services or better results for the user.

And once you have finished Sect. 3.3, you should be able to understand both these definitions better.

3.3.2 RDFa Attributes and RDFa Elements

First off, attributes introduced by RDFa have names. For example, `property` is one such attribute. Obviously, when we make reference to this attribute, we say `attribute.property`. In order to avoid repeating the word `attribute` too often, `attribute.property` is often written as `@property`. You will see this a lot if you read about RDFa. And in what follows, we will write `@attributeName` to represent one attribute whose name is given by `attributeName`.

The following attributes are used by RDFa at the time of this writing:

```
about
content
datatype
href
property
rel
resource
rev
role
src
typeof
```

²<http://www.w3.org/TR/2008/REC-rdfa-syntax-20081014/>

³<http://www.w3.org/MarkUp/2009/rdfa-for-html-authors>

Some of them are more often used than the others, as we will discuss in later sections. Before we get into the detail, let us first understand to what XHTML elements these attributes can be used.

The rule is very simple: you can use these attributes to just about any element. For example, you can use them on `div` element, on `p` element, or even on `h2` (or `h3`, etc.) element. In real practice, there are some elements that are more frequently used with these attributes.

The first such element is the `span` element. It is a popular choice for RDFa simply because you can insert it anywhere in the body of an XHTML document. `link` and `meta` elements are also popular choices, since you can use them to add RDFa markups to the `head` element of a HTML document. This is in fact one of the reasons why RDFa is gaining popularity: these elements have been used to add metadata to the `head` element for years; therefore, any RDFa-aware software can extract useful metadata from them with only minor modifications needed.

The last frequently used element when it comes to add RDFa markup into the content is the `a` linking element. With what you have learned about RDF from [Chap. 2](#), it is not hard for you to see the reason here: a linking element actually expresses a relationship between one resource (the one where it is stored) and another (the resource it links to). In fact, as you will see in the examples, we can always use `@rel` on a `link` element to add more information about the relationship, and this information serves as the predicate of a triple stored in that `a` element.

3.3.3 RDFa: Rules and Examples

In this section we will explain how to use RDFa to mark up a given content page, and we will also summarize the related rules when using the RDFa attributes. We will not cover all the RDFa attributes as listed in [Sect. 3.3.2](#), but what you will learn here should be able to get you far into the world of RDFa if you so desire.

3.3.3.1 RDFa Rules

Before we set off to study the usage of each RDFa attribute, let us understand its basic rules first. Note that at this point, these rules may seem unclear to you, but you will start to understand them better when we start to see more examples.

As we have learned in [Chap. 2](#), any given RDF statement has three components: subject, predicate and object. It turns out that RDFa attributes are closely related to these components:

- Attributes `rel`, `rev` and `property` are used to represent predicates.
- For attribute `rel`, its subject is the value of `about` attribute, and its object is the value of `href` attribute.
- For attribute `rev`, its subject and object are reversed compared to `rel`: its subject is the value of `href` attribute, and its object is the value of `about` attribute.
- For attribute `property`, its subject is the value of `about` attribute, and its object is the value of `content` attribute.

Table 3.3 RDFa attributes as different components of an RDF statement

Object values	Subject attribute	Predicate attribute	Object
Literal strings	about	property	Value of content attribute
Resource (identified by URI)	about	rel	Value of href attribute

Now recall the fact that we always have to be careful about the object of a given RDF statement: its object can either take a literal string as its value or use another resource (identified by a URI) as its value. How is this taking effect when it comes to RDFa? Table 3.3 summarizes the rules.

Based on Table 3.3, if the object of an RDF statement takes a literal string as its value, this literal string will be the value of `content` attribute. Furthermore, the subject of that statement is identified by the value of `about` attribute, and the predicate of that statement is given by the value of `property` attribute. If the object of an RDF statement takes a resource (identified by a URI) as its value, the URI will be the value of `href` attribute. Furthermore, the subject of that statement is identified by the value of `about` attribute, and the predicate of that statement is given by the value of `rel` attribute.

Let us see some examples along this line. Assume I have posted an article about the Semantic Web on my Web site. In that post, I have some simple HTML code as shown in List 3.8.

List 3.8 Some simple HTML code in my article about the Semantic Web

```
<div>
  <h2>This article is about the Semantic Web and written
  by Liyang.</h2>
</div>
```

This can be easily understood by a human reader of the article. First, it says this article is about the Semantic Web; second, it says the author of this article is Liyang. Now I would like to use RDFa to add some semantic markup, so that machine can see these two facts. One way to do this is shown in List 3.9.

List 3.9 Use RDFa to mark up the content HTML code in List 3.8

```
<div xmlns:dc="http://purl.org/dc/elements/1.1/">
  <p>This article is about <span about="http://www.liyangyu.
  com/article/theSemanticWeb.html" rel="dc:subject" href="http:
  //dbpe-dia.org/resource/Semantic_Web"/>the Semantic Web and
  written by <span about="http://www.liyangyu.com/article/the
  SemanticWeb.html" property="dc:creator" content="Liyang"/>
  Liyang.</p>
</div>
```

Recall that `dc` represents Dublin Core vocabulary namespace (review [Chap. 2](#) for more understanding about Dublin Core). We can pick up the RDFa markup segments from List 3.9 and show them in List 3.10.

List 3.10 RDFa markup text taken from List 3.9

```
<span about="http://www.liyangyu.com/article/theSemanticWeb.html"
rel="dc:subject"
href="http://dbpedia.org/resource/Semantic_Web"/>

<span about="http://www.liyangyu.com/article/theSemanticWeb.html"
property="dc:creator" content="Liyang"/>
```

Clearly, in the first `span` segment, the object is a resource identified by a URI. Therefore, `@rel` and `@href` have to be used as shown in List 3.10. Note that http://dbpedia.org/resource/Semantic_Web is used as the URI identifying the object resource. This is an URI created by DBpedia project (will be discussed in [Chap. 10](#)) to represent the concept of the Semantic Web. Here we are reusing this URI instead of inventing our own. To see more details about reusing URIs, review [Chap. 2](#).

On the other hand, in the second `span` segment, the object is represented by a literal string. Therefore, `@property` and `@content` have to be used.

The last rule we need to discuss here is about attribute `about`. At this point, we understand attribute `about` is used to represent the subject of the RDF statement. But for a given XHTML content marked up by RDFa, how does an RDFa-aware application exactly identify the subject of the markup? This can be summarized as follows:

- If attribute `about` is used explicitly, then the value represented by `about` is the subject.
- If an RDFa-aware application does not find `about` attribute, it will assume that the `about` attribute on the nearest ancestor element represents the subject.
- If an RDFa-aware application searches through all the ancestors of the element with RDFa markup information and does not find an `about` attribute, then the subject is an empty string and will effectively indicate the current document.

These rules about subject are in fact quite intuitive, especially the last one, given the fact that lots of a document's markup information will be typically about the document itself.

With all the understanding about RDFa rules, we can now move on to the example of RDFa markup.

3.3.3.2 RDFa Examples

In this section, we will use examples to show how semantic markup information can be added by using RDFa attributes. Note that we will be able to cover only a subset

of ways to add RDFa metadata in an XHTML document; it is, however, enough to get your far if you decide to explore more on yourself.

A common usage of RDFa attributes is to add *inline* semantic information. This is in fact the original motivation that led to the creation of RDFa: how to take human-readable Web page content and make it machine readable. List 3.9 is a good example of this inline markup. You can compare List 3.8 with List 3.9; List 3.8 is the original page content that is written for human eyes, and List 3.9 is what we have after inline RDFa markup. Note that the presentation rendered by any Web browser does not alter at all.

Another example is to mark up the HTML code shown in List 3.2. It is a good exercise for us since we have already marked up List 3.2 using hCard microformats, and using RDFa to mark up the same HTML content shows the difference between the two.

List 3.11 shows the RDFa markup of List 3.2. It accomplishes the same goal as shown in List 3.5. It tells an RDFa-aware application the following fact: this is a link to the home page of a person, whose first name is Liyang and last name is Yu.

List 3.11 RDFa markup for the HTML code shown in List 3.2

```
... <!-- other HTML code -->
<div xmlns:foaf="http://xmlns.com/foaf/0.1/">
  <a about="http://www.liyangyu.com#liyang"
     rel="foaf:homepage"
     href="http://www.liyangyu.com/">Liyang Yu</a>
  <span property="foaf:firstName" content="Liyang"/>
  <span property="foaf:lastName" content="Yu"/>
</div>
... <!-- other HTML code -->
```

Again, if you open up the above with a Web browser, you see the same output as given by List 3.2. With what we have learned so far, understanding List 3.11 should not be difficult at all.

Note that FOAF vocabulary is used for RDFa to mark up the content; we have covered FOAF briefly in Chap. 2 and you will see a detailed discussion about FOAF in Chap. 7. For now, just remember FOAF is a vocabulary, with a collection of words that one can use to describe people and their basic information.

This is in fact an important difference between microformats and RDFa. More specifically, when using microformats to mark up a given document, the possible values for the properties are pre-defined. For example, if hCard microformat is used, only hCard properties and sub-properties can be used in the markup (see List 3.5 for example). However, this is not true for RDFa markup: you can in fact use anything as the values for the attributes. For example, List 3.11 could have been written as the one shown in List 3.12.

List 3.12 Another version of List 3.11

```

... <!-- other HTML code -->
<div xmlns:yu="http://www.liyangyu.com/yu">
  <a about="http://www.liyangyu.com#liyang"
    rel="yu:myHomepage"
    href="http://www.liyangyu.com/">Liyang Yu</a>
  <span property="yu:myFirstName" content="Liyang"/>
  <span property="yu:myLastName" content="Yu"/>
</div>
... <!-- other HTML code -->

```

However, this is not a desirable solution at all. In order for any RDFa-aware application to understand the markup in List 3.12, that application has to understand your vocabulary first. And clearly, if all the Web publishers went ahead to invent their own keywords, the world of available keywords would have become quite messy. Therefore, it is always the best choice to use words from a well-recognized vocabulary when it comes to mark up your page. Again, FOAF vocabulary is one such well-accepted vocabulary, and if you use it in your markup (as shown in List 3.11), chance is any application that understands RDFa will be able to understand FOAF as well.

In fact, this flexibility of the possible values of RDFa attributes is quite useful for many markup requirements. For example, assume in my Web site, I have the following HTML snippet as shown in List 3.13.

List 3.13 HTML code about my friend, Dr. Ding

```

... <!-- other HTML code -->
<div>
<p>My friend, Dr.Ding, also likes to play tennis.</p>
</div>
... <!-- other HTML code -->

```

And I would like to mark up the code in List 3.13 so that the machine will understand these facts: first, I have a friend whose name is Dr. Ding; second, Dr. Ding likes to play tennis.

You can certainly try to use microformats to reach the goal; however, RDFa seems to be quite easy to use, as shown in List 3.14.

List 3.14 RDFa markup of List 3.13

```

... <!-- other HTML code -->
<div xmlns:foaf="http://xmlns.com/foaf/0.1/">
<p>My friend, <span about="http://www.liyangyu.com#liyang"
rel="foaf:knows" href="http://www.example.org#ding">Dr.Ding
</span>, also likes to play <span about="http://www.example.
org#ding" rel="foaf:interest" href="http://dbpedia.org/
resource/Tennis">tennis. </span></p>

```

```
<span about="http://www.example.org#ding" property="foaf:
title" content="Dr." /> <span about="http://www.ex-ample.org#
ding" proerty="foaf:lastName" content="Ding" />
</div>
... <!-- other HTML code -->
```

Again, note that <http://dbpedia.org/resource/Tennis> is used as the URI identifying tennis as a sport. This is also a URI created by DBpedia project, as you will see in later chapters. We are reusing this URI since it is always good to reuse existing ones. On the other hand, <http://www.example.org#ding> is a URI that we invented to represent Dr. Ding, since there is no URI for this person yet.

An application which understands RDFa will generate the RDF statements as shown in List 3.15 from List 3.14 (expressed in Turtle format).

List 3.15 RDF statements generated from the RDFa markup in List 3.14

```
@prefix foaf: <http://xmlns.com/foaf/0.1/>.

<http://www.liyangyu.com#liyang>
foaf:knows <http://www.example.org#ding>.
<http://www.example.org#ding>
foaf:interest <http://dbpedia.org/resource/Tennis>.
<http://www.example.org#ding> foaf:title "Dr.".
<http://www.example.org#ding> foaf:lastName "Ding".
```

So far, all the examples we have seen are about inline markup. Sometimes, RDFa semantic markup can also be added about the containing document without explicitly using attribute `about`. Since this is a quite common use case of RDFa, let us take a look at one such example.

List 3.16 shows the markup that can be added to the document header.

List 3.16 RDFa markup about the containing document

```
<html xmlns:dc="http://purl.org/dc/elements/1.1/">
  <head>
    <meta property="dc:title" content="Liyang Yu's Homepage"/>
    <meta property="dc:creator" content="Liyang Yu"/>
  </head>
  <body>
<!-- body of the page -->
```

Clearly, there is no `about` attribute used. Based on the RDFa rules we have discussed earlier, when no subject is specified, an RDFa-aware application assumes an empty string as the subject, which represents the document itself.

At this point, we have covered the following RDFa attributes: `about`, `content`, `href`, `property` and `rel`. These are all frequently used attributes, and understanding these can get you quite far already.

The last attribute we would like to discuss here is attribute `typeof`. It is quite important and useful since it presents a case where a blank node is created. Let us take a look at one example.

Assume on my home page, I have the following HTML code to identify myself as shown in List 3.17.

List 3.17 HTML code that identifies myself

```
<div>
  <p>Liyang Yu</p>
  <p>E-mail: <a
    href="mailto:liyang910@yahoo.com">liyang910@yahoo.com</a>
</div>
```

We would now like to use RDFa to mark up this part so the machine will understand that this whole `div` element is about a person, whose name is Liyang Yu and whose e-mail address is `liyang910@yahoo.com`.

List 3.18 shows this markup.

List 3.18 RDFa markup of the HTML code shown in List 3.17

```
<div typeof="foaf:Person"
  xmlns:foaf="http://xmlns.com/foaf/0.1/">
  <p property="foaf:name">Liyang Yu</p>
  <p>E-mail: <a rel="foaf:mbox"
    href="mailto:liyang910@yahoo.com">liyang910@yahoo.com</a>
</div>
```

Note the usage of attribute `typeof`. More specifically, this RDFa attribute is designed to be used when we need to declare a new data item with a certain type. In this example, this type is the `foaf:Person` type. For now, just understand `foaf:Person` is another keyword from the FOAF vocabulary, and it represents human being as a class called `Person`. Again, you will see more about FOAF vocabulary in a later chapter.

Now, when `typeof` is used as one attribute on the `div` element, the whole `div` element represents a data item whose type is `foaf:Person`. Therefore, once reading this line, any RDFa-aware application will be able to understand this `div` element is about a person. In addition, `foaf:name` and `foaf:mbox` are used with `@property` and `@rel`, respectively, to accomplish our goal to make the machine understand this information, as you should be familiar by now.

Note we did not specify attribute `about` like we have done in the earlier examples. So what would be the subject for these properties then? In fact, attribute `typeof` on the enclosing `div` does the trick: it implicitly sets the subject of the properties marked up within that `div`. In other words, the name and e-mail address are associated with a new node of type `foaf:Person`. Obviously, this new node does not have a given URI to represent itself; it is therefore a blank node. Again, this is a trick you will see quite often if you are working with RDFa markup, so make sure you are comfortable with it.

The last question before we move on is, if this new node is a blank node, how do we use it when it comes to data aggregation? For example, the markup information here could be quite important; it could be some supplement information about a resource we are interested in. However, without a URI identifying it, how do we relate this information to the correct resource at all?

In this case, the answer is yes. In fact, we can indeed relate this markup information to another resource that exists outside the scope of this document. The secret lies in the `foaf:mbox` property: as you will see in [Chap. 5](#), this property is an inverse functional property, and that is how we know which resource should be the subject of this markup information, even though the subject itself is represented by a blank node.

3.3.4 *RDFa and RDF*

3.3.4.1 What Is So Good About RDFa?

In [Sect. 3.2.3.1](#), we have discussed the benefit offered by microformats. In fact, all are still true for RDFa, and we can add one more here: RDFa is useful because microformats only exist as a collection of centralized vocabularies. More specifically, what if we want to mark up a Web page about a resource, for which there is no microformat available to use? In that case, RDFa is always a better choice, since you can in fact use any vocabulary for your RDFa markup.

In this chapter, we only see Dublin Core vocabulary and FOAF vocabulary. However, as you will see after you finish more chapters, there are quite a lot of vocabularies out there, covering different domains, and all are available to you when using RDFa to mark up a given page. In fact, you can even invent your own if it is necessary (again, more on this later).

3.3.4.2 *RDFa and RDF*

At this point in the book, RDFa and RDF can be understood as the same thing. To put it simple, RDFa is just a way of expressing RDF triples inside given XHTML pages.

However, RDFa does makes it much easier for people to express semantic information in conjunction with a normal Web page. For instance, while there are many ways to express RDF (such as in serialized XML files that live next to standard Web pages), RDFa helps machines and humans read exactly the same content. This is one of the major motivations for the creation of RDFa.

It might be a good idea to come back to this topic after you have finished the whole book. By then, you will have a better understanding of the whole picture. For example, having a HTML representation and a separate RDF/XML representation (or N3 and Turtle, etc.) is still a good solution for many cases, where HTTP content negotiation is often used to decide which format should be returned to the client (details in [Chap. 11](#)).

3.4 GRDDL

3.4.1 GRDDL: The Big Picture

As we have discussed in Sect. 3.1, GRDDL (Gleaning Resource Descriptions from Dialects of Languages) is a way (a markup format, to be more precise) that enables users to obtain RDF triples out of XML documents (called *XML dialects*), in particular XHTML documents. The following GRDDL terminologies are important for us to understand GRDDL:

- *GRDDL-aware agent*: a software agent that is able to recognize the GRDDL transformations and run these transformations to extract RDF.
- *GRDDL Transformation*: an algorithm for getting RDF from a source document.

GRDDL became a W3C Recommendation on 11 September 2007.⁴ In this standard document, GRDDL is defined as the following:

GRDDL is a mechanism for Gleaning Resource Descriptions from Dialects of Languages. The GRDDL specification introduces markup based on existing standards for declaring that an XML document includes data compatible with RDF and for linking to algorithms (typically represented in XSLT), for extracting this data from the document.

You can also find more information about GRDDL from the official Web site of W3C GRDDL Working Group.⁵

In this section, we will take a quick look at GRDDL and introduce the markup formats needed for extracting markup information created by using microformats and RDFa. What you will learn from here will give you enough background if you decide to go further into GRDDL.

The last words before we move on: do not bury your semantic markup data in (X)HTML pages. Instead, when you publish a document that contains markup data, do reference GRDDL profiles and/or transformations for their extraction. You will see how to do this in the next two sections.

3.4.2 Using GRDDL with Microformats

There are a number of ways to reference GRDDL in a document where microformats markup data are added. Referencing GRDDL transformations directly in the head of the HTML document is probably the easiest implementation: only two markup lines are needed.

More specifically, the first thing is to add a `profile` attribute to the `head` element to indicate the fact that this document contains GRDDL metadata. List 3.19 shows how to do this.

⁴<http://www.w3.org/TR/2007/REC-grddl-20070911/>

⁵<http://www.w3.org/2001/sw/grddl-wg/>

List 3.19 Adding `profile` attribute for GRDDL transformation

```
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head profile="http://www.w3.org/2003/g/data-view">
  <title>Liyang Yu's Homepage</title>
</head>
<body>
<!-- body of the page -->
```

In HTML, `profile` attribute in `head` element is used to link a given document to a description of the metadata schema that the document uses. The URI for GRDDL is given by the following,

```
http://www.w3.org/2003/g/data-view
```

And by including this URI as shown in List 3.19, we declare that the metadata in the markup can be interpreted using GRDDL.

The second step is to add a `link` element containing the reference to the appropriate transformation. More specifically, recall the fact that microformats is a collection of individual microformats such as `hCard` microformat and `hCalendar` microformat. Therefore, when working with markup data added by using microformats, it is always necessary to name the specific GRDDL transformation.

Let us assume the document in List 3.19 contains `hCard` microformat markups. Therefore, the `link` element has to contain the reference to the specific transformation for converting HTML containing `hCard` patterns into RDF. This is shown in List 3.20.

List 3.20 Adding `link` element for GRDDL transformation (`hCard` microformat)

```
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head profile="http://www.w3.org/2003/g/data-view">
  <title>Liyang Yu's Homepage</title>
  <link rel="transformation"
        href="http://www.w3.org/2006/vcard/hcard2rdf.xsl"/>
</head>
<body>
<!-- body of the page -->
```

These two steps are all there is to it: the `profile` URI tells a GRDDL-aware application to look for a `link` element whose `rel` attribute contains the token `transformation`. Once the agent finds this element, the agent should use the value of `href` attribute on that element to decide how to extract the `hCard` microformat markup data as RDF triples from the enclosing document.

What if hCalendar microformat markup has been used in the document? If that is the case, we should use the following transformation as the value of href attribute:

```
http://www.w3.org/2002/12/cal/glean-hcal.xml
```

3.4.3 Using GRDDL with RDFa

With what we have learned from Sect. 3.4.2, it is now quite easy to use GRDDL with RDFa. The first step is still the same, i.e., we need to add a profile attribute to the head element, as shown in List 3.19. For the second step, as you have guessed, we will have to switch the transformation itself, as shown in List 3.21.

List 3.21 Adding link element for GRDDL transformation (RDFa)

```
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head profile="http://www.w3.org/2003/g/data-view">
  <title>Liyang Yu's Homepage</title>
  <link rel="transformation"
        href="http://www.w3.org/2001/sw/grddl-wg/td/RDFa2RDFXML.xsl"/>
</head>
<body>
<!-- body of the page -->
```

3.5 Summary

This chapter covers the technical details of both microformats and RDFa. GRDDL as a popular markup format which automatically converts microformats and RDFa markup information into RDF triples is also included.

From this chapter, you should have learned the following main points:

- the concepts of microformats and RDFa, and how they fit into the whole idea of the Semantic Web;
- the language constructs of both microformats and RDFa, and how to mark up a given (X)HTML page by using these constructs;
- the advantages and limitations of both microformats and RDFa, their relationships to RDF;
- the concept of GRDDL, how it fits into the idea of the Semantic Web, and how to use GRDDL to automatically extract markup data from (X)HTML pages.

With all these said, the final goal is for you to understand these technical components and also be able to pick up the right one for a given development assignment.

Chapter 4

RDFS and Ontology

Even after you have read the previous two chapters carefully, you probably still have lots of questions that remain unanswered on your mind. This chapter is a natural continuation of those two chapters, especially [Chap. 2](#). After reading this chapter, you will be able to find answers to most of your questions.

This chapter will cover all the main aspects of RDFS, including its concept, its semantics, its language constructs and features, and certainly, real-world examples. It will also formally introduce the concept of ontology, together with a description about SKOS. Again, make sure you understand the content in this chapter, since what you will learn here is important for you to continue on with [Chap. 5](#).

4.1 RDFS Overview

4.1.1 RDFS in Plain English

Unlike the previous chapter, we will start this chapter by discussing RDFS in plain English. In this section, our goal is to answer the following two questions:

- Why do we need RDFS?
- What is RDFS?

Let us go back to [Chap. 2](#) by taking another look at List 2.25, the RDF/XML representation of List 2.5. At least the following questions may come to your mind:

- Line 9 of List 2.25 says `myCamera:Nikon_D300` is an instance (by using predicate `rdf:type`) of the resource identified by URI `myCamera:DSLR`, but where is this `myCamera:DSLR` resource defined? What does it look like?
- If we use object-oriented concepts, `myCamera:DSLR` can be understood as a class. Now, if `myCamera:DSLR` represents a class, are there any other classes that are defined as its super classes or sub-classes?
- The rest of List 2.25 uses several properties (such as `myCamera:model` and `myCamera:weight`) to describe `myCamera:Nikon_D300`. Are there any other properties that we can use to describe `myCamera:Nikon_D300`? How do we know these properties exist for us to use at the first place?

You can ask more questions like these. The last question, in particular, raises an important issue: when we describe a real-world resource such as `myCamera:Nikon_D300`, what are the things (predicates) we can use to describe it? If we all say something about it, and furthermore, if we all go on to invent our own things to say about it, there will be no common language shared among us. And in that case, any given application cannot go too much further beyond simply aggregating the distributed RDF models.

A common language or shared vocabulary seems to be the key there. More specifically, if properties such as `myCamera:model` and `myCamera:weight` are used to describe a camera, that is because somewhere, in some document, someone has defined that these are indeed the predicates we can use to describe it. There are possibly more terms defined for us to use, and it is our choice which predicates to use when publishing our own descriptions. Therefore, this common language can make sure one important thing for us: everything we say about a given resource, we have a reason to say it.

Clearly at this stage, what seems to be missing for our RDF documents is such a common language, or, a vocabulary, where classes, sub-classes, properties, and also relations between these classes and properties are defined.

As you might have guessed, RDFS is such a language we can use to define a vocabulary, which can then be used to structure the RDF documents we create.

Vocabulary is not something totally new to us at this point; we have used the word vocabulary in previous chapters for a number of times. For example, all the RDF terms we have covered in [Chap. 2](#) are elements from RDF vocabulary, and Dublin Core is another vocabulary. We have used both to create RDF documents in the previous two chapters. Now, for a given specific application domain (such as photography), we may need some application-specific vocabularies, and this is where we find the use of RDFS.

Therefore, in plain English, we can define RDFS as follows:

RDFS is a language one can use to create a vocabulary (often the created vocabulary is domain-specific), so when distributed RDF documents are created in this domain, terms from this vocabulary can be used. Therefore, everything we say, we have a reason to say it.

At this point, we understand how RDFS fits into the world of RDF. We will also see how it works together with RDF to create more structured and machine-understandable documents on the Web in the coming sections of this chapter. For now, let us move on to see the official definition of RDFS.

4.1.2 RDFS in Official Language

RDFS stands for RDF Schema. Various abbreviations such as RDF(S), RDF-S, or RDF/S can be used, and these are all referring to the same RDF Schema.

As a W3C standard, its initial version¹ was originally published by W3C in April 1998. With the change of RDF standards (see [Chap. 2](#)), W3C released the final

¹<http://www.w3.org/TR/1998/WD-rdf-schema-19980409/>

RDFS Recommendation² in February 2004, and it is included in the six documents published as the updated RDF specifications as shown in [Table 2.1](#).

Based on this official document, RDFS can be defined as follows:

RDFS is a recommendation from W3C and it is an extensible knowledge representation language that one can use to create a vocabulary for describing classes, sub-classes and properties of RDF resources.

With this definition, RDFS can be understood as RDF's vocabulary description language. As a standard, RDFS provides language constructs that can be used to describe classes, properties within a specific application domain. For example, what is a DSLR class, and what is property `model`, and how could it be used to describe a resource. Note that the language constructs in RDFS are themselves classes and properties; in other words, RDFS provides standard classes and properties that can be used to describe classes and properties in a specific domain.

To further help you understand what is RDFS, let us move on to some more explanations and examples in the next section. Before we can discuss its language features and constructs, a solid understanding about it is always needed.

4.2 RDFS + RDF: One More Step Toward Machine Readable

In this section, we will discuss more about the reason why RDFS is needed. This will not only enhance your understanding about RDFS, but also help you to put the pieces together to understand the world of the Semantic Web.

4.2.1 A Common Language to Share

The first important fact about RDFS is that RDFS can be used to define a vocabulary, a common language everyone can use. The goal? Everything we say, we have a reason to say it.

Let us take a look at one example. [Figure 4.1](#) shows a small vocabulary in the world of photography.

Note in this tiny vocabulary, an oval box is used to represent a specific resource type, and the arrow from one oval box to another oval box means that the first oval box is a sub-type of the second oval box. The properties that one can use to describe a given resource type are included in `[]` and is placed besides that specific oval box.

With all these said, this simple vocabulary tells us the following fact:

*We have a resource called **Camera**, and **Digital** and **Film** are its two sub-resources. Also, resource **Digital** has two sub-resources, **DSLR** and **PointAndShoot**. Resource **Camera** can be described by properties called **model** and **manufactured_by**, and resource **Digital** can be described by a property called **effectivePixel**.*

²<http://www.w3.org/TR/rdf-schema/>

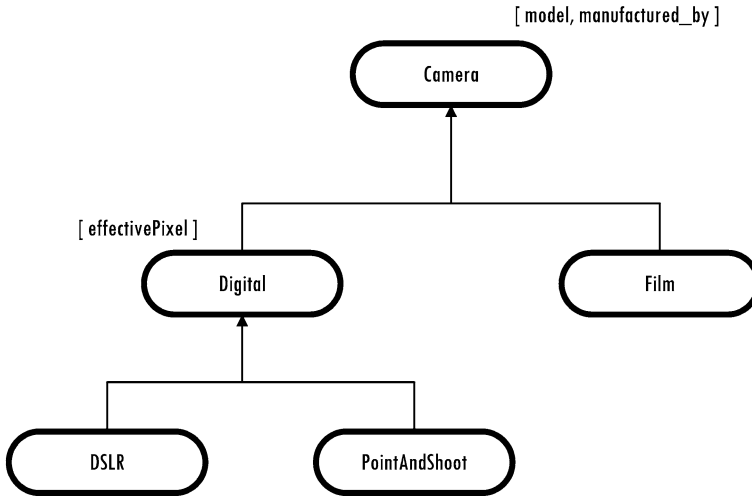


Fig. 4.1 A small vocabulary for the domain of photography

Again, DSLR is short for digital single lens reflex; it is a type of camera that is more often used by professional photographers and tends to be expensive as well. On the other hand, a *point-and-shoot* camera, also called a compact camera, is often used by non-professionals, and it normally offers functionalities such as auto-focus and auto-exposure setting.

Now, if we want to describe Nikon D300 as a DSLR, we know what we can say about it. List 4.1 shows one such example.

List 4.1 A simple description about Nikon D300

```

1: <?xml version="1.0"?>
2: <rdf:RDF
2a:     xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
3:     xmlns:myCamera="http://www.liyangyu.com/camera#">
4:
5:   <rdf:Description
5a:     rdf:about="http://www.liyangyu.com/camera#Nikon_D300">
6:     <rdf:type
6a:       rdf:resource="http://www.liyangyu.com/camera#DSLR"/>
7:     <myCamera:model>Nikon D300</myCamera:model>
8:     <myCamera:manufactured_by
      rdf:resource="http://www.dbpedia.org/resource/Nikon"/>
9:     <myCamera:effectivePixel>12.3</myCamera:effectivePixel>
10:  </rdf:Description>
11:
12: </rdf:RDF>
  
```

As we see from Fig. 4.1, resource `Camera` can be described by properties named `manufactured_by` and `model`. Why can we use them to describe `Nikon D300`, an instance of `DSLR`, not `Camera` (lines 7 and 8)? The reason is really simple: any property that can be used to describe the base type can also be used to describe any sub-type of this base type. Again, anything we say here, we have a reason to say.

On the other hand, we will not be able to use a term if that term is not defined in the vocabulary. If we have to do so, we will then need to grow the vocabulary accordingly.

Now, imagine someone else from the same application domain has come up with another RDF document describing the same camera (or another camera). Whatever the resource being described might be, all these documents now share the same terms. Note that when we say the same terms are shared, it is not that all the documents will use exactly the same terms to describe resource – one document might use different properties compared to the other document, but all the properties available to use are included in the given vocabulary. The result is that any application that “knows” this vocabulary will be able to process these documents with ease. This is an obvious benefit of having a common vocabulary.

Another important benefit of having a vocabulary defined is to facilitate machine understanding, as discussed in the next section.

4.2.2 Machine Inferencing Based on RDFS

A vocabulary created by using RDFS can facilitate inferencing on the RDF documents which make use of this vocabulary. To see this, let us go back to List 4.1 to understand what inferences machine can make.

The inferencing for this case is based on line 6, which says the resource identified by http://www.liyangyu.com/camera#Nikon_D300 is a `DSLR`. Given the vocabulary in Fig. 4.1, the following inferences can be made:

- resource http://www.liyangyu.com/camera#Nikon_D300 is a `Digital camera`, and
- resource http://www.liyangyu.com/camera#Nikon_D300 is a `Camera`.

This is all done by the machine, and these inferred conclusions can be critical information for many applications. In fact, a lot more inferencing can be done when a vocabulary is defined, and we will see more examples during the course of this chapter.

At this point, you have seen all the important aspects of RDFS, especially why we need it. The rest is to understand its syntax, which we will cover in coming sections. And the good news is RDFS itself can be written in RDF/XML format, and any vocabulary created using RDFS can also be written in RDF/XML format, so it is not totally new.

4.3 RDFS Core Elements

In this section, we will cover the syntax of RDFS, together with examples. Our goal is to build a camera vocabulary by using RDFS terms, with the one shown in Fig. 4.1 as our starting point.

4.3.1 *The Big Picture: RDFS Vocabulary*

First off, as we have discussed, RDFS is a collection of terms we can use to define classes and properties for a specific application domain. Just like RDF terms and Dublin Core terms, all these RDFS terms are identified by pre-defined URIs and all these URIs share the following leading string:

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

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

Second, all these RDFS terms can be divided into the following groups based on their purposes:

- **classes**
This group includes RDFS terms that can be used to define classes. More specifically, the following terms are included here: `rdfs:Resource`, `rdfs:Class`, `rdfs:Literal`, `rdfs:Datatype`.
- **properties**
This group includes RDFS terms that can be used to define properties, and the following terms are included: `rdfs:range`, `rdfs:domain`, `rdfs:subClassOf`, `rdfs:subPropertyOf`, `rdfs:label` and `rdfs:comment`.
- **utilities**
As its name suggests, this group of RDFS terms are used for miscellaneous purposes as we will see later in this chapter. For now, understand that this group contains the following terms: `rdfs:seeAlso` and `rdfs:isDefinedBy`.

4.3.2 *Basic Syntax and Examples*

4.3.2.1 Defining Classes

First off, `rdfs:Resource` represents the root class; every other class defined using RDFS terms will be sub-class of this class. In practice, this term is rarely used, it mainly acts as a logic root to hold everything together: all things described by RDF are instances of class `rdfs:Resource`.

To define a class in a vocabulary, `rdfs:Class` is used. For our camera vocabulary, List 4.2 shows the definition of the `Camera` class.

List 4.2 Definition of the Camera class

```

1: <?xml version="1.0"?>
2: <rdf:RDF
2a:     xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
3:     xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
4:     xmlns:myCamera="http://www.liyangyu.com/camera#">
5:
6: <rdf:Description
6a:     rdf:about="http://www.liyangyu.com/camera#Camera">
7:   <rdf:type
7a:     rdf:resource="http://www.w3.org/2000/01/rdf-schema#Class"/>
8: </rdf:Description>
9:
10: </rdf:RDF>

```

Let us understand List 4.2 line by line. First of all, everything is defined between `<rdf:RDF>` and `</rdf:RDF>`, indicating this document is either an RDF document (as we have seen in [Chap. 2](#)) or an RDF schema document (as seen here). Lines 2–4 have defined several namespaces, and the new one here is the `rdfs` namespace (line 3), which includes all the pre-defined terms in RDF Schema. Line 4 defines the namespace for our camera vocabulary.

Now, the key lines are lines 6–8. Line 6 defines a new resource by using the term `rdf:Description` from RDF vocabulary, and this new resource has the following URI:

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

Line 7 specifies the type property of this resource by using RDF term `rdf:type`, and its value is another resource (indicated by using RDF term `rdf:resource`), which has the following URI:

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

Obviously, this URI is a pre-defined term in RDFS vocabulary and its QName is given by `rdfs:Class`. Now, we have defined a new class and we can read it as follows:

Here we declare: this resource,
<http://www.liyangyu.com/camera#Camera>, *is a class.*

Note that `Camera` class is by default a sub-class of `rdfs:Resource`, the root class of all classes. In addition, pay attention not to mix together these two terms: `rdfs:Resource` and `rdf:resource`. `rdfs:Resource` is a class defined in RDFS as we have discussed above, and `rdf:resource` is simply an XML attribute that goes together with a specific property element (in List 4.2, it is used together with `rdf:type` property element) to indicate the fact that the property's value is another resource. Also, `rdf:resource` is case sensitive, i.e., cannot be written as `rdf:Resource`. If you do so, your validator will raise a red flag at it for sure.

Sometimes, you will see `rdf:ID` is used instead of `rdf:about`. For example, List 4.3 is equivalent to List 4.2, and List 4.3 does use `rdf:ID`.

List 4.3 Use `rdf:ID` to define `Camera` class

```

1: <?xml version="1.0"?>
2: <rdf:RDF
3a:   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
3:   xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
4:   xmlns:myCamera="http://www.liyangyu.com/camera#"
5:     xml:base="http://www.liyangyu.com/camera#"
6:
7:   <rdf:Description rdf:ID="Camera">
8:     <rdf:type
8a:   rdf:resource="http://www.w3.org/2000/01/rdf-schema#Class"/>
9:   </rdf:Description>
10:
11: </rdf:RDF>

```

Note the usage of line 5. It is always a good practice (almost necessary) to use `xml:base` together with `rdf:ID`, as we have discussed in [Chap. 2](#). Since using `rdf:ID` does make the line shorter, from now on, we will be using `rdf:ID` more.

In fact, there is a short form you can use, which is equivalent to both Lists 4.2 and 4.3. This short form is shown in List 4.4.

List 4.4 A short form that is equivalent to Lists 4.2 and 4.3

```

1: <?xml version="1.0"?>
2: <rdf:RDF
3a:   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
3:   xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
4:   xmlns:myCamera="http://www.liyangyu.com/camera#"
5:     xml:base="http://www.liyangyu.com/camera#"
6:
7:   <rdfs:Class rdf:ID="Camera">
8:   </rdfs:Class>
9:
10: </rdf:RDF>

```

This short form not only looks cleaner, but is also more intuitive: `rdfs:Class` is used to define a class and `rdf:ID` is used to provide a name for the class being defined (lines 7 and 8). And of course, if you prefer to use `rdf:about` instead of `rdf:ID`, List 4.4 will become List 4.5, which again is equivalent to both Lists 4.2 and 4.3.

List 4.5 A short form using `rdf:about`

```

1: <?xml version="1.0"?>
2: <rdf:RDF
3:     xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
4:     xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
5:     xmlns:myCamera="http://www.liyangyu.com/camera#">
6:   <rdfs:Class rdf:about="http://www.liyangyu.com/camera#Camera">
7:     </rdfs:Class>
8:
9: </rdf:RDF>

```

Note in List 4.5, `xml:base` is not needed anymore.

To define more classes, we can simply add more class definitions by using `rdfs:Class` as shown in List 4.6.

List 4.6 Adding more class definitions into the vocabulary

```

1: <?xml version="1.0"?>
2: <rdf:RDF
3:     xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
4:     xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
5:     xmlns:myCamera="http://www.liyangyu.com/camera#"
6:     xml:base="http://www.liyangyu.com/camera#">
7:   <rdfs:Class rdf:about="http://www.liyangyu.com/camera#Camera">
8:     </rdfs:Class>
9:
10:  <rdfs:Class rdf:about="http://www.liyangyu.com/camera#Lens">
11:    </rdfs:Class>
12:
13:  <rdfs:Class rdf:about="http://www.liyangyu.com/camera#Body">
14:    </rdfs:Class>
15:
16:  <rdfs:Class
16a:    rdf:about="http://www.liyangyu.com/camera#ValueRange">
17:    </rdfs:Class>
18:
19: </rdf:RDF>

```

Note that in List 4.6, we have defined class `Lens`, `Body`, and `ValueRange`; the reason for having these classes will become clear in later sections of this chapter.

So much for the top-level classes at this point. Let us move on to sub-classes. To define sub-classes, we need to use `rdfs:subClassOf` property defined in RDF Schema. List 4.7 shows the details.

List 4.7 Sub-class definitions are added

```

1: <?xml version="1.0"?>
2: <rdf:RDF
2a:     xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
3:     xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
4:     xmlns:myCamera="http://www.liyangyu.com/camera#"
5:     xml:base="http://www.liyangyu.com/camera#">
6:
7: <rdfs:Class rdf:about="http://www.liyangyu.com/camera#Camera">
8: </rdfs:Class>
9:
10: <rdfs:Class rdf:about="http://www.liyangyu.com/camera#Lens">
11: </rdfs:Class>
12:
13: <rdfs:Class rdf:about="http://www.liyangyu.com/camera#Body">
14: </rdfs:Class>
15:
16: <rdfs:Class
16a:     rdf:about="http://www.liyangyu.com/camera#ValueRange">
17: </rdfs:Class>
18:
19: <rdfs:Class
19a:     rdf:about="http://www.liyangyu.com/camera#Digital">
20:   <rdfs:subClassOf rdf:resource="#Camera"/>
21: </rdfs:Class>
22:
23: <rdfs:Class rdf:about="http://www.liyangyu.com/camera#Film">
24:   <rdfs:subClassOf rdf:resource="#Camera"/>
25: </rdfs:Class>
26:
27: <rdfs:Class rdf:about="http://www.liyangyu.com/camera#DSLR">
28:   <rdfs:subClassOf rdf:resource="#Digital"/>
29: </rdfs:Class>
30:
31: <rdfs:Class
31a:   rdf:about="http://www.liyangyu.com/camera#PointAndShoot">
32:   <rdfs:subClassOf rdf:resource="#Digital"/>
33: </rdfs:Class>
34:
35: <rdfs:Class
35a:   rdf:about="http://www.liyangyu.com/camera#Photographer">
36:   <rdfs:subClassOf
36a:     rdf:resource="http://xmlns.com/foaf/0.1/Person"/>
37: </rdfs:Class>

```

```
38:
39: </rdf:RDF>
```

Lines 19–37 define some sub-classes that are used in our vocabulary. First off, note how the base class is identified in the `rdfs:subClassOf` property. For instance, line 19 defines a class, `Digital`, and line 20 uses the `rdfs:subClassOf` property to specify the base class of `Digital` is `Camera`. The way `Camera` is identified is as follows:

```
<rdfs:subClassOf rdf:resource="#Camera"/>
```

This is perfectly fine in this case since when an RDF parser sees `#Camera`, it assumes that class `Camera` must have been defined in the same document (which is true here). To get the URI of class `Camera`, it concatenates `xml:base` and this name together to get the following:

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

This is clearly the right URI for this class, and this is also the reason why we need to add line 5 to specify the base URI to use when concatenation is done. Of course, you can always do the following to specify the full URI of the base class:

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

And this is often used when the base class is defined in some other document. In fact, lines 35–37 provide a perfect example, where class `Photographer` is being defined as a sub-class of `Person`. Since the base class `Person` is not defined in this vocabulary, we use its full URI to identify this class, as shown by line 36. We will see class `Person` later in this book; it is a key class created by the popular FOAF project, which will also be presented in [Chap. 7](#).

The rest of the sub-class definitions can be understood similarly. For now, we have defined the following sub-classes: `Digital`, `Film`, `DSLR`, `PointAndShoot` and `Photographer`.

Another important fact about `rdfs:subClassOf` property is that you can use it multiple times when defining a class. If you do so, all the base classes introduced by `rdfs:subClassOf` will be ANDed together to create the new class. For instance, let us say you have already defined a class called `Journalist`; you can now define a new class called `Photojournalist` as follows:

```
<rdfs:Class
  rdf:about="http://www.liyangyu.com/camera#Photojournalist">
  <rdfs:subClassOf rdf:resource="#Photographer"/>
  <rdfs:subClassOf rdf:resource="#Journalist"/>
</rdfs:Class>
```


This means class `Photojournalist` is a sub-class of *both* `Photographer` class and `Journalist` class. Therefore, any instance of `Photojournalist` is an instance of `Photographer` and `Journalist` at the same time.

4.3.2.2 Defining Properties

At this point, we have defined all the classes we need for our camera vocabulary. Figure 4.2 summarizes all these classes and their relationships.

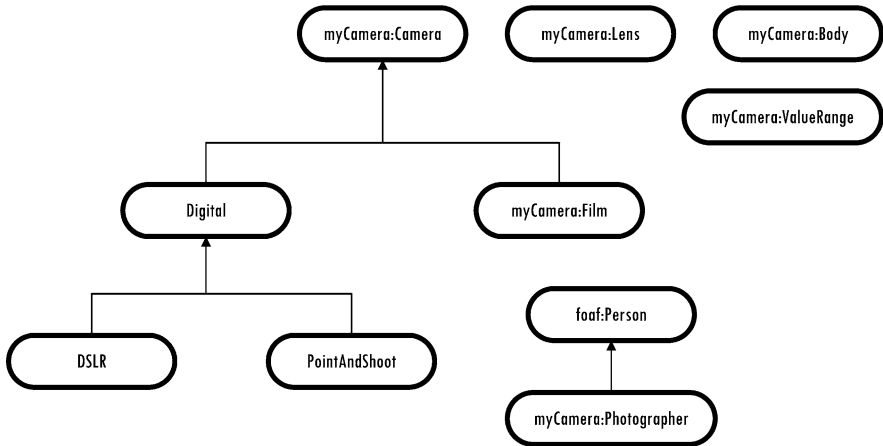


Fig. 4.2 Classes defined for our camera ontology

Note that all the classes in Fig. 4.2 are “floating” around: except for the base-class and sub-class relationship, there seems to be no other bounds among them. In fact, the bounds, or the relationships among these classes, will be expressed by properties. Let us now move on to define these properties.

To define a property, `rdf:Property` type is used, and `rdf:about` in this case specifies the URI of the property. Furthermore, `rdfs:domain` and `rdfs:range` together indicate how the property should be used. Let us take a look at List 4.8.

List 4.8 Define property `owned_by`

```

1: <?xml version="1.0"?>
2: <rdf:RDF
2a:   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
3:   xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
4:   xmlns:myCamera="http://www.liyangyu.com/camera#"
5:   xml:base="http://www.liyangyu.com/camera#"
6:
7:
... // classes, sub-classes definitions as shown in List 4.6

```

```

38:
39: <rdf:Property
41a:     rdf:about="http://www.liyangyu.com/camera#owned_by">
40:     <rdfs:domain rdf:resource="#DSLR" />
41:     <rdfs:range rdf:resource="#Photographer" />
42: </rdf:Property>
43:
44: </rdf:RDF>

```

As shown in List 4.8, lines 39–42 define the property called `owned_by`. We can read this as follows:

We define a property called `owned_by`. It can only be used to describe the characteristics of class `DSLR`, and its possible values can only be instances of class `Photographer`

or equivalently,

```

subject: DSLR
predicate: owned_by
object: Photographer

```

The new RDFS terms here are `rdfs:domain` and `rdfs:range`. More specifically, property `rdfs:domain` is used to specify which class the property being defined can be used with. It is optional, so you can declare property `owned_by` like this:

```

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

```

This means property `owned_by` can be used to describe any class. For instance, you can say something like “a Person is `owned_by` a Photographer.” In most cases, this is not what we want, and the definition with `rdfs:domain` as shown in List 4.8 is much better. It says that `owned_by` can only be used on the instances of class `DSLR`.

Note that when defining a property, multiple `rdfs:domain` properties can be specified. In that case, we are indicating that the property can be used with a resource that is an instance of *every* class defined by `rdfs:domain` property. For example,

```

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

```

This says property `owned_by` can only be used with something that is a `DSLR` camera *and* a `PointAndShoot` camera at the same time. In fact, a `DSLR` camera can be used as a point-and-shoot camera, so the above definition does hold.

As for `rdfs:range`, all the above discussion is true. First of all, it is optional, like the following:

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

This says property `owned_by` can be used with `DSLR` class, but its value can be anything. Therefore, in our RDF document, we can add a statement that says a `DSLR` camera is owned by another `DSLR` camera, which certainly does not make much sense. Therefore, most likely, we will need to use at least one `rdfs:range` property when defining a property.

We can also use multiple `rdfs:range` properties such as the following (assume we have already defined a class call `Journalist`):

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

This says property `owned_by` can be used to depict `DSLRs`, and its value has to be someone who is a `Photographer` and `Journalist` at the same time. In other words, this someone has to be a `photojournalist`.

With all these said, we can continue to define other properties used in our camera vocabulary, and this is shown in List 4.9.

List 4.9 Camera vocabulary with properties defined

```
1: <?xml version="1.0"?>
2: <rdf:RDF
2a:   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
3:   xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
4:   xmlns:myCamera="http://www.liyangyu.com/camera#"
5:   xml:base="http://www.liyangyu.com/camera#">
6:
7:
... // classes/sub-classes definitions as shown in List 4.6
38:
39: <rdf:Property
39a:   rdf:about="http://www.liyangyu.com/camera#owned_by">
40:   <rdfs:domain rdf:resource="#DSLR"/>
41:   <rdfs:range rdf:resource="#Photographer"/>
42: </rdf:Property>
43:
```

```
44: <rdf:Property
44a:   rdf:about="http://www.liyangyu.com/camera#manufactured_by">
45:   <rdfs:domain rdf:resource="#Camera" />
46: </rdf:Property>
47:
48: <rdf:Property
48a:   rdf:about="http://www.liyangyu.com/camera#body">
49:   <rdfs:domain rdf:resource="#Camera" />
50:   <rdfs:range rdf:resource="#Body" />
51: </rdf:Property>
52:
53: <rdf:Property
53a:   rdf:about="http://www.liyangyu.com/camera#lens">
54:   <rdfs:domain rdf:resource="#Camera" />
55:   <rdfs:range rdf:resource="#Lens" />
56: </rdf:Property>
57:
58: <rdf:Property
58a:   rdf:about="http://www.liyangyu.com/camera#model">
59:   <rdfs:domain rdf:resource="#Camera" />
60:   <rdfs:range
60a:     rdf:resource="http://www.w3.org/2001/XMLSchema#string" />
61: </rdf:Property>
62:
63: <rdf:Property
63a:   rdf:about="http://www.liyangyu.com/camera#effectivePixel">
64:   <rdfs:domain rdf:resource="#Digital" />
65:   <rdfs:range
65a:     rdf:resource="http://www.w3.org/2001/XMLSchema#decimal" />
66: </rdf:Property>
67:
68: <rdf:Property
68a:   rdf:about="http://www.liyangyu.com/camera#shutterSpeed">
69:   <rdfs:domain rdf:resource="#Body" />
70:   <rdfs:range rdf:resource="#ValueRange" />
71: </rdf:Property>
72:
73: <rdf:Property
73a:   rdf:about="http://www.liyangyu.com/camera#focalLength">
74:   <rdfs:domain rdf:resource="#Lens" />
75:   <rdfs:range
75a:     rdf:resource="http://www.w3.org/2001/XMLSchema#
       string" />
76: </rdf:Property>
```

```

77:
78: <rdf:Property
78a:     rdf:about="http://www.liyangyu.com/camera#aperture">
79:     <rdfs:domain rdf:resource="#Lens"/>
80:     <rdfs:range rdf:resource="#ValueRange"/>
81: </rdf:Property>
82:
83: <rdf:Property
83a:     rdf:about="http://www.liyangyu.com/camera#minValue">
84:     <rdfs:domain rdf:resource="#ValueRange"/>
85:     <rdfs:range
85a:     rdf:resource="http://www.w3.org/2001/XMLSchema#float"/>
86: </rdf:Property>
87:
88: <rdf:Property
88a:     rdf:about="http://www.liyangyu.com/camera#maxValue">
89:     <rdfs:domain rdf:resource="#ValueRange"/>
90:     <rdfs:range
90a:     rdf:resource="http://www.w3.org/2001/XMLSchema#float"/>
91: </rdf:Property>
92:
93: </rdf:RDF>

```

As shown in List 4.9, we have defined four properties related to `Camera` class. Property `body` can be used on a `Camera` instance, and it takes a `Body` instance as its value (lines 48–51). `lens` property is defined similarly (lines 53–56): it can be used on a `Camera` instance, and it takes a `Lens` instance as its value. Together these two properties specify the fact that any given camera will always have a body and lens, which is quite intuitive indeed.

Another property that is shared by all cameras is the `model` property, as shown in lines 58–61. Finally, note the definition of property `manufactured_by`, which does not have property `rdfs:range` defined (lines 44–46). As we have discussed, it is almost always better to have `rdfs:range`, but for simplicity, we are not going to have it here.

Property `effectivePixel` is only applicable to digital cameras, therefore its `rdfs:domain` property points to `Digital` class as seen in lines 63–66.

We know that for any given camera body and its lens, there are three parameters that are often used to specify its performance: shutter speed (a parameter that can be adjusted on the camera's body), focal length of the lens, and aperture of the lens. Therefore, we need to define all these properties for our camera vocabulary to be useful.

Property `focalLength` is defined in lines 73–76. It is used on instances of `Lens` class, and it takes a string as its value, such as 50 mm. Note that for zoom lens, i.e.,

lens with changeable focal length, this definition will not be enough, since there has to be a way to specify the range of the changeable focal lengths. For now, let us assume we only consider non-zoom lens, and our definition will be fine.

Another parameter for lens is aperture, which indeed has a range. For instance, 2.8–22 can be the typical range of aperture values for a given lens. Taking this into account, we have defined property `aperture` as shown in lines 78–81. It is used on instances of `Lens` class and its value should take an instance of `ValueRange` class. Note that the same method is used for shutter speed parameter: property `shutterSpeed` is used on `Body` class, and its value also takes an instance of `ValueRange` class (lines 68–71).

Finally, we need to include range information in `ValueRange` class. To implement this, lines 83–91 define two more properties `minValue` and `maxValue`. These two properties will be used on instances of `ValueRange` class, and by using these two properties, we will be able to model the fact that some parameters take a range of values instead of a single value.

Up to this point, we have added the related properties into our camera vocabulary. Together with the class definitions shown in Fig. 4.2, this now becomes a complete vocabulary, and we can also update Fig. 4.2 and change it to Fig. 4.3.

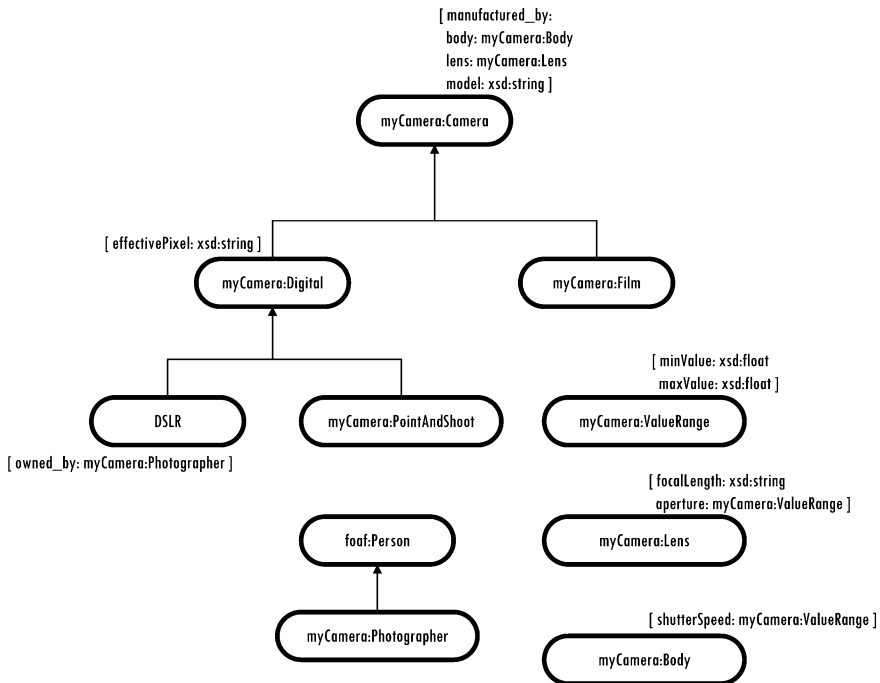


Fig. 4.3 Our camera ontology so far

Again, in Fig. 4.3, the properties that can be used to describe a given resource type are included in [] and is placed besides that specific oval box. The value range of that property is also included. For example, property `myCamera:body` can be used on `myCamera:Camera` class, and its value can be an instance of type `myCamera:Body`. Note that if there are no constraints on the values a given property can assume, there will be no value specified for that property. Note that property `myCamera:manufactured_by` is one such example.

In fact, properties not only describe the relationship among classes, but are also the more interesting part in a vocabulary: they are the key factors when it comes to reasoning based on vocabularies. Let us discuss this more in the next few sections.

4.3.2.3 More About Properties

First off, properties are inheritable from base classes to sub-classes. More specifically, remember class `Digital` has a property called `effectivePixel`; also it has two sub-classes, namely, `DSLR` and `PointAndShoot`. Then do these sub-classes also have the property `effectivePixel`? In other words, can we use `effectivePixel` to describe a `DSLR` instance? The answer is yes, since a sub-class always inherits properties from its base class.

Therefore, classes `DSLR` and `PointAndShoot` both have a property called `effectivePixel`.

In fact, take one step further, a class always inherits properties from *all* its base classes. For instance, we can use `model` property on class `Camera`, and since `Camera` is also a base class of `DSLR` (although not a direct base class), we can then `model` property on class `DSLR` as well.

The second important issue about property is the sub-property. We can define a property to be a sub-property of another property, and this is done by using `rdfs:subPropertyOf`. For example, the `model` property describes the “name” of a camera. However, the manufacturer could sell the same model using different model names. For instance, the same camera sold in North America could have a different model name than the one sold in Asia. Therefore, we can define another property, say, `officialModel`, to be a sub-property of `model`:

```
<rdfs:Property
  rdf:about="http://www.liyangyu.com/camera#officialModel">
  <rdfs:subPropertyOf rdf:resource="#model"/>
</rdfs:Property>
```

This declares the property `officialModel` as a specialization of property `model`. Property `officialModel` inherits `rdfs:domain` and `rdfs:range` values from its base property `model`. However, you can narrow the domain and/or the range as you wish.

We can also use multiple `rdfs:subPropertyOf` when defining a property. If we do so, we are declaring that the property being defined has to be a sub-property of *each* of the base properties.

The third issue about property is that we have been using the abbreviated form to define properties. It is important to know this since you might see the long form in other documents. List 4.10 shows the long form one can use to define a property.

List 4.10 Use long form to define property `owned_by`

```
<rdf:Description
  rdf:about="http://www.liyangyu.com/camera#owned_by">
  <rdf:type rdf:resource=
    "http://www.w3.org/1999/02/22-rdf-syntax-ns#Property" />
  <rdfs:domain rdf:resource="#DSLR" />
  <rdfs:range rdf:resource="#Photographer" />
</rdf:Description>
```

The fourth issue we would like to mention might also be something you have realized already: the separation of class definitions and property definitions in our vocabulary. Those who are used to the object-oriented world might find this fact uncomfortably strange.

For instance, if we are using any object-oriented language (such as Java or C++), we may define a class called `DigitalCamera`, and we will then encapsulate several properties to describe a digital camera. These properties will be defined at the same time when we define the class, and they are defined in the class scope as its member variables. Normally, these properties are not directly visible to the outside world.

For RDF Schema, it is quite a different story. We define a class, and very often we also indicate its relationships to other classes. However, this is it: we never declare its member variables, i.e., the properties it may have. A class is just an entity which may have relationships to other entities. What are inside this entity, i.e., its member variables/properties, are simply unknown.

The truth is we declare properties separately and associate the properties with classes if we wish to do so. Properties are never owned by any class; they are never local to any class either. If we do not associate a given property to any class, this property is simply independent and it can be used to describe any class.

What is the reason behind this? What is the advantage of separating the class definition and property definition? Before you read on, think about it, you should be able to figure out the answer by now.

The answer is Rule #3 that we discussed in [Chap. 2](#). Let me put it here again:

Rule #3:

I can talk about any resource at my will, and if I chose to use an existing URI to identify the resource I am talking about, then the following is true:

- *the resource I am talking about and the resource already identified by this existing URI are exactly the same thing or concept;*
- *everything I have said about this resource is considered to be additional knowledge about that resource.*

And more specifically, the separation of the class definition and property definition is just an implementation of this rule. The final result is that the application we build will have more power to automatically process the distributed information, together with a stronger inferencing engine.

To see this, think about the case where someone else would like to add some new properties into our camera vocabulary and then publish RDF documents which use these newly added properties. The camera reviewers example in [Chap. 2](#) fits into this example perfectly. For example, those reviewers will have an initial vocabulary they can use to publish their reviews, and they also enjoy the freedom to come up with new terms to describe a given camera.

Adding new properties to an existing vocabulary can be understood as an implementation of Rule #3 as well: anyone, anywhere, and at any time can talk about a resource by adding more properties to it.

And here is an important fact: adding new properties will not disturb any existing application, and no change is needed to any existing application each time a new property is added. The reason behind this fact is the separation of class definitions and property definitions. If the definition of class were not separate from the definition of property, this would not have been accomplished.

The final point about property is related to an important programming trick that you should know. Let us modify `owned_by` property as follows:

```
<rdf:Property rdf:ID="owned_by">
  <rdfs:domain rdf:resource="#Digital"/>
  <rdfs:domain rdf:resource="#Film"/>
  <rdfs:range rdf:resource="#Photographer"/>
</rdf:Property>
```

If we define `owned_by` property like this, we are saying `owned_by` is to be used with instances that are *both* digital cameras and film cameras at the same time. Clearly, such a camera has not been invented yet. Actually, what we wanted to express here is the fact that a photographer can own a digital camera or a film camera or both. How do we accomplish this?

Given the fact that a sub-class will inherit all the properties associated with its base class, we can associate `owned_by` property with the base class:

```
<rdf:Property rdf:ID="owned_by">
  <rdfs:domain rdf:resource="#Camera"/>
  <rdfs:range rdf:resource="#Photographer"/>
</rdf:Property>
```

Since both `Digital` and `Film` are sub-classes of `Camera`, they all inherit property `owned_by`. Now we can use `owned_by` property with `Digital` class *or* `Film` class, and this has solved our problem.

Before we move on to the next section, here is one last thing we need to be cautious about: `Class` is in the `rdfs` namespace and `Property` is in the `rdf` namespace, and it is not a typo in the above lists.

4.3.2.4 RDFS Datatypes

As we discussed earlier, property `rdfs:range` is used to specify the possible values of a property being declared. In some cases, the property being defined can simply have *plain* or *untyped* string as its value, represented by `rdfs:Literal` class contained in RDFS vocabulary. For example, property `model` could have been defined as follows, and it could then use any string as its value:

```
<rdf:Property
  rdf:about="http://www.liyangyu.com/camera#model">
  <rdfs:domain rdf:resource="#Camera"/>
  <rdfs:range
    rdf:resource="http://www.w3.org/2001/01/rdf-schema#Literal"/>
</rdf:Property>
```

However, using `rdfs:Literal` is not a recommended solution for most cases. A better idea is to always provide *typed* values if you can. For example, we have specified the valid value for the `model` property has to be strings specified by the XML Schema, as shown in List 4.9, lines 58–61. More specifically, the full URI of this datatype is given by the following:

`http://www.w3.org/2001/XMLSchema#string`

and we can use this URI directly in our schema without explicitly indicating that it represents a datatype (as we have done in List 4.9). However, it is always useful to clearly declare that a given URI represents a datatype, as shown here:

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

The next example will show that using `rdfs:Datatype` is not only a good practice, but also necessary in some cases. For instance, the following could be another definition of `effectivePixel` property:

```
<rdf:Property
  rdf:about="http://www.liyangyu.com/camera#effectivePixel">
  <rdfs:domain rdf:resource="#Digital"/>
  <rdfs:range
    rdf:resource="http://www.liyangyu.com/camera#MegaPixel"/>
</rdf:Property>
<rdfs:Datatype
  rdf:about="http://www.liyangyu.com/camera#MegaPixel">
  <rdfs:subClassOf
```

```

    rdf:resource="http://www.w3.org/2001/XMLSchema#decimal"/>
</rdfs:Datatype>

```

When an RDF Schema parser reaches the above code, it first concludes the property `effectivePixel`'s value should come from a resource with the following URI:

```

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

```

And once it reaches the next couple of lines, it realizes this URI is in fact identifying an `rdfs:Datatype` instance, which has a base class given by this URI, <http://www.w3.org/2001/XMLSchema#decimal>. The parser then concludes that `effectivePixel` should always use a typed literal as its value.

Note that when `rdfs:Datatype` is used in our RDF Schema document to indicate a datatype, the corresponding RDF instance statements should then use `rdf:datatype` property as follows:

```

<model rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
    Nikon_D300
</model>
<effectivePixel
    rdf:datatype="http://www.liyangyu.com/camera#MegaPixel">
    12.3
</effectivePixel>

```

A related topic here is the usage of `rdfs:XMLLiteral`. Remember, in most cases, its usage should be avoided. To make our discussion complete, let us briefly talk about the reason here.

First understand that `rdfs:XMLLiteral` denotes a well-formed XML string, and it is always used together with `rdf:parseType="Literal"`. For instance, if you used `rdfs:XMLLiteral` in an RDF Schema document to define some property, the RDF statements which describe an instance of this property will have to use `rdf:parseType="Literal"`. Let us see an example.

Suppose we have defined a new property called `features` as follows:

```

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

```

An example RDF statement could be as this:

```

<features rdf:parseType="Literal">
    Nikon D300 is good!, also, ...
</features>

```

Note the usage of `rdf:parseType="Literal"`, which indicates the value here is a well-formed XML content.

Now, note that although the content is a well-formed XML content, it does not have the `resource/property/value` structure in general. And as you have

already learned, this structure is one of the main reasons why a given application can understand the content. Therefore, if we use XML paragraph as the value of some property, we have to accept the fact that no tools will be able to understand its meaning well. So, avoid using `XMLLiteral` if you can.

4.3.2.5 RDFS Utility Vocabulary

Up to this point, we have covered the most important classes and properties in RDF Schema. In this section, we will take a look at some utility classes and properties defined in RDFS vocabulary, and as you will see in the later chapters, some of these terms are quite useful. As a summary, the following terms will be covered in this section:

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

`rdfs:seeAlso` is a property that can be used on any resource, and it indicates another resource may provide additional information about the given resource. For example, List 4.11 shows one RDF document that uses this property.

List 4.11 Example of using `rdfs:seeAlso`

```
1: <?xml version="1.0"?>
2: <rdf:RDF
3a:   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
4:   xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
5:   xmlns:myCamera="http://www.liyangyu.com/camera#">
6:   <rdf:Description
7a:     rdf:about="http://www.liyangyu.com/camera#Nikon_D300">
8:     <rdf:type
9a:       rdf:resource="http://www.liyangyu.com/camera#DSLR"/>
10:     <rdfs:seeAlso
11:       rdf:resource="http://dbpedia.org/resource/Nikon_D300"/>
12:   </rdf:Description>
13: </rdf:RDF>
```

Line 8 says this: to understand more about the resource identified by this URI

http://www.liyangyu.com/camera#Nikon_D300

you can take a look at the resource identified at this URI:

http://dbpedia.org/resource/Nikon_D300

Note that `rdfs:seeAlso` has no formal semantics defined. In real application, it only implies the fact that these two URIs are somehow related to each other; it is then up to the application to decide how to handle this situation.

For our case, recall the above URI is created by DBpedia to represent exactly the same resource, namely, Nikon D300 camera. Therefore, these two URIs are considered to be URI aliases, and an application can act accordingly. For example, the application can retrieve an RDF document from the second URI and collect more information from this new document – a typical example of information aggregation based on URI aliases. As you will see in later chapters, this is also one of the key concepts in the world of Linked Data.

`rdfs:isDefinedBy` is quite similar to `rdfs:seeAlso`, and it is actually an `rdfs:subPropertyOf` of `rdfs:seeAlso`. It is intended to specify the primary source of information about a given resource. For example, the following statement:

```
subject rdfs:isDefinedBy object
```

says that the `subject` resource is defined by the `object` resource, and more specifically, this `object` resource is supposed to be an original or authoritative description of the resource.

The last two properties you may encounter in documents are `rdfs:label` and `rdfs:comment`. `rdfs:label` is used to provide a class/property name for human eyes, and similarly, `rdfs:comment` provides a human-readable description of the property/class being defined. One example is shown in List 4.12.

List 4.12 Example of using `rdfs:label` and `rdfs:comment`

```
1: <rdf:Property rdf:ID="officialModel">
2:   <rdfs:subPropertyOf rdf:resource="#model"/>
3:   <rdfs:label xml:lang="EN">officialModelName</rdfs:label>
4:   <rdfs:comment xml:lang="EN">
4a:     this is the official name of the camera.
4b:     the manufacturer may use different names when
4c:     the camera is sold in different regions/countries.
5:   </rdfs:comment>
6: </rdf:Property>
```

And their usage is quite straightforward and does not require much of an explanation.

4.3.3 Summary So Far

4.3.3.1 Our Camera Vocabulary

At this point, we have finished our discussion about RDFS core terms, and our final product is a simple camera vocabulary defined by using RDFS terms. List 4.13 shows the complete vocabulary. Compared to Lists 4.9, 4.13 includes all the datatype information. Note that Fig. 4.3 does not change and is still the graphical representation of our camera ontology.

List 4.13 Our camera vocabulary

```

1: <?xml version="1.0"?>
2: <rdf:RDF
2a:   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
3:   xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
4:   xmlns:myCamera="http://www.liyangyu.com/camera#"
5:   xml:base="http://www.liyangyu.com/camera#">
6:
7: <rdfs:Class rdf:about="http://www.liyangyu.com/camera#Camera">
8: </rdfs:Class>
9:
10: <rdfs:Class rdf:about="http://www.liyangyu.com/camera#Lens">
11: </rdfs:Class>
12:
13: <rdfs:Class rdf:about="http://www.liyangyu.com/camera#Body">
14: </rdfs:Class>
15:
16: <rdfs:Class
16a:   rdf:about="http://www.liyangyu.com/camera#ValueRange">
17: </rdfs:Class>
18:
19: <rdfs:Class
19a:   rdf:about="http://www.liyangyu.com/camera#Digital">
20:   <rdfs:subClassOf rdf:resource="#Camera"/>
21: </rdfs:Class>
22:
23: <rdfs:Class rdf:about="http://www.liyangyu.com/camera#Film">
24:   <rdfs:subClassOf rdf:resource="#Camera"/>
25: </rdfs:Class>
26:
27: <rdfs:Class rdf:about="http://www.liyangyu.com/camera#DSLR">
28:   <rdfs:subClassOf rdf:resource="#Digital"/>
29: </rdfs:Class>
30:
31: <rdfs:Class
31a:   rdf:about="http://www.liyangyu.com/camera#PointAndShoot">
32:   <rdfs:subClassOf rdf:resource="#Digital"/>
33: </rdfs:Class>
34:
35: <rdfs:Class
35a:   rdf:about="http://www.liyangyu.com/camera#Photographer">
36:   <rdfs:subClassOf
36a:     rdf:resource="http://xmlns.com/foaf/0.1/Person"/>
37: </rdfs:Class>

```

```

38:
39: <rdf:Property
39a:     rdf:about="http://www.liyangyu.com/camera#owned_by">
40:     <rdfs:domain rdf:resource="#DSLR" />
41:     <rdfs:range rdf:resource="#Photographer" />
42: </rdf:Property>
43:
44: <rdf:Property
44a:     rdf:about="http://www.liyangyu.com/camera#manufactured_by">
45:     <rdfs:domain rdf:resource="#Camera" />
46: </rdf:Property>
47:
48: <rdf:Property
48a:     rdf:about="http://www.liyangyu.com/camera#body">
49:     <rdfs:domain rdf:resource="#Camera" />
50:     <rdfs:range rdf:resource="#Body" />
51: </rdf:Property>
52:
53: <rdf:Property
53a:     rdf:about="http://www.liyangyu.com/camera#lens">
54:     <rdfs:domain rdf:resource="#Camera" />
55:     <rdfs:range rdf:resource="#Lens" />
56: </rdf:Property>
57:
58: <rdf:Property
58a:     rdf:about="http://www.liyangyu.com/camera#model">
59:     <rdfs:domain rdf:resource="#Camera" />
60:     <rdfs:range
60a:         rdf:resource="http://www.w3.org/2001/XMLSchema#string" />
61: </rdf:Property>
62: <rdfs:Datatype
62a:     rdf:about="http://www.w3.org/2001/XMLSchema#string" />
63:
64: <rdf:Property
64a:     rdf:about="http://www.liyangyu.com/camera#effectivePixel">
65:     <rdfs:domain rdf:resource="#Digital" />
66:     <rdfs:range
66a:         rdf:resource="http://www.liyangyu.com/camera#MegaPixel" />
67: </rdf:Property>
68: <rdfs:Datatype
68a:     rdf:about="http://www.liyangyu.com/camera#MegaPixel">
69:     <rdfs:subClassOf
69a:         rdf:resource="http://www.w3.org/2001/XMLSchema#decimal" />
70: </rdfs:Datatype>
71:

```

```

72: <rdf:Property
72a:     rdf:about="http://www.liyangyu.com/camera#shutterSpeed">
73:     <rdfs:domain rdf:resource="#Body"/>
74:     <rdfs:range rdf:resource="#ValueRange"/>
75: </rdf:Property>
76:
77: <rdf:Property
77a:     rdf:about="http://www.liyangyu.com/camera#focalLength">
78:     <rdfs:domain rdf:resource="#Lens"/>
79:     <rdfs:range
79a:     rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
80: </rdf:Property>
81: <rdfs:Datatype
81a:     rdf:about="http://www.w3.org/2001/XMLSchema#string"/>
82:
83: <rdf:Property
83a:     rdf:about="http://www.liyangyu.com/camera#aperture">
84:     <rdfs:domain rdf:resource="#Lens"/>
85:     <rdfs:range rdf:resource="#ValueRange"/>
86: </rdf:Property>
87:
88: <rdf:Property
88a:     rdf:about="http://www.liyangyu.com/camera#minValue">
89:     <rdfs:domain rdf:resource="#ValueRange"/>
90:     <rdfs:range
90a:     rdf:resource="http://www.w3.org/2001/XMLSchema#float"/>
91: </rdf:Property>
92: <rdfs:Datatype
92a:     rdf:about="http://www.w3.org/2001/XMLSchema#float"/>
93:
94: <rdf:Property
94a:     rdf:about="http://www.liyangyu.com/camera#maxValue">
95:     <rdfs:domain rdf:resource="#ValueRange"/>
96:     <rdfs:range
96a:     rdf:resource="http://www.w3.org/2001/XMLSchema#float"/>
97: </rdf:Property>
98: <rdfs:Datatype
98a:     rdf:about="http://www.w3.org/2001/XMLSchema#float"/>
99:
100: </rdf:RDF>

```

At the beginning of the chapter, we have said that vocabulary like this can help machines to make inferences, based on the knowledge expressed in the vocabulary. We will discuss this inferencing power in a later section. For now, let us understand this first: how is the knowledge expressed in the vocabulary?

4.3.3.2 Where Is the Knowledge?

So far into this chapter, we have created a simply camera vocabulary by using some pre-defined classes and properties from RDF Schema. So how is the knowledge encoded in this vocabulary?

And here is the answer: in a given vocabulary, the meaning of a term is expressed and understood by defining the following:

- all the properties that can be used on it and
- the types of those objects that can be used as the values of these properties.

For example, let us take a look at the term `Camera`. As far as any application is concerned, a `Camera` is something like this:

- It is a class.
- We can use property `manufactured_by` on it; any resource can be the value of this property.
- We can use property `body` on it, with a `Body` instance as this property's value.
- We can use property `lens` on it, with a `Lens` instance as this property's value.
- We can use property `model` on it, with an XML string as this property's value.

And similarly, for any application, a `Digital` camera is something like this:

- It is a class.
- We can use property `manufactured_by` on it; any resource can be the value of this property.
- We can use property `body` on it, with a `Body` instance as this property's value.
- We can use property `lens` on it, with a `Lens` instance as this property's value.
- We can use property `model` on it, with an XML string as this property's value.
- We can use property `effectivePixel` on it, with an XML decimal as this property's value.

You can come up with the meaning of the word `DSLR` just as above.

How can the knowledge be used and understood by applications? Before we move on to this topic, let us take a look at a new concept: ontology.

4.4 The Concept of Ontology

Ontology plays a critical role for the Semantic Web, and it is necessary to understand ontology in order to fully appreciate the idea of the Semantic Web. Its concept, however, seems quite abstract and hard to grasp from the beginning. It does take a while to get used to, but the more you know it, the more you see the value of it.

4.4.1 What Is Ontology?

First off, understand we have already built an ontology: List 4.13 is in fact a tiny ontology in the domain of photography.

There are many definitions of ontology; perhaps each single one of these definitions starts from a different angle of view. And some of these definitions can be confusing as well. For example, the most popular definition of ontology is “ontology is a formalization of a conceptualization!”

For us, in the world of the Semantic Web, the definition presented in W3C’s OWL Use Cases and Requirements Documents³ is good enough (you will know all about OWL in the next chapter):

An ontology formally defines a common set of terms that are used to describe and represent a domain . . . An ontology defines the terms used to describe and represent an area of knowledge.

There are several things needed to be made clear from this definition. First of all, ontology is domain specific, and it is used to describe and represent *an area of knowledge*. A domain is simply a specific subject area or area of knowledge, such as the area of photography, medicine, real estate, and education.

Second, ontology contains terms and the relationships among these terms. Terms are often called classes, or concepts, and these words are interchangeable. The relationships between these classes can be expressed by using a hierarchical structure: super classes represent higher level concepts and sub-classes represent finer concepts. The finer concepts have all the attributes and features that the higher concepts have.

Third, besides the above relationships among the classes, there is another level of relationship expressed by using a special group of terms: properties. These property terms describe various features and attributes of the concepts, and they can also be used to associate different classes together. Therefore, the relationships among classes are not only super class or sub-class relationships, but also relationships expressed in terms of properties.

By having the terms and the relationships among these terms clearly defined, ontology encodes the knowledge of the domain in such a way that the knowledge can be understood by a computer. This is the basic idea of ontology.

4.4.2 The Benefits of Ontology

We can summarize the benefits of ontology as follows (and you should be able to come up with most of the items in this list):

- It provides a common and shared understanding/definition about certain key concepts in the domain.
- It offers the terms one can use when creating RDF documents in the domain.

³“OWL Web Ontology Language Use Cases and Requirements,” <http://www.w3.org/TR/webont-req/>

- It provides a way to reuse domain knowledge.
- It makes the domain assumptions explicit.
- Together with ontology description languages (such as RDFS and OWL, which we will learn in the next chapter), it provides a way to encode knowledge and semantics such that the machine can understand.
- It makes automatic large-scale machine processing become possible.

When you have made more progress with this book, you will get more understanding about these benefits, and you will be able to add more as well.

It is now a good time to discuss some related concepts and introduce an important vocabulary, SKOS, which can be very useful when it comes to development work on the Semantic Web.

4.5 Building the Bridge to Ontology: SKOS

We have discussed the concept of ontology in the previous section. In this section, we will take a small detour to understand SKOS, a model and vocabulary that is used to bridge the world of knowledge organization systems (KOS) and the Semantic Web.

If you are doing development work on the Semantic Web, it is likely that you will have a chance to see or use SKOS. In addition, understanding SKOS will enhance your understanding about ontology, and it will also give you a chance to appreciate more the benefit of having ontologies on the Semantic Web.

4.5.1 Knowledge Organization Systems (KOS)

If you have experience working with those so-called KOSs, you may be familiar with some well-understood knowledge organizing schemes such as taxonomies, thesauri, subject headers, and other types of controlled vocabulary. These schemes are not all the same, but they all allow the organization of concepts into concept schemes where it is also possible to indicate relationships between the terms contained in the scheme.

You have probably already started to consider the relationships between these schemes and ontologies. What is the difference between these schemes and ontologies? Is there a way to build a bridge between these two so we can express knowledge organization systems in a machine-understandable way, within the framework of the Semantic Web?

To understand these interesting questions, let us first get more understanding about some basic schemes that are widely used in a variety of knowledge organization systems. We will concentrate on two of these schemes, namely taxonomy and thesaurus. When we have some good understanding about these schemes, we can move on to study the relationships between these schemes and ontologies.

First off, understand that KOS is a general term that refers to, among other things, a set of elements, often structured and controlled, that can be used for describing

objects, indexing objects, browsing collections, etc. KOSs are commonly found in cultural heritage institutions such as libraries and museums. They can also be used in other scientific areas, examples include biology and chemistry, where naming and classifying are important.

More specifically, taxonomies, thesauri are all typical examples of KOSs.

- Taxonomy

Based on its Greek roots, taxonomy is the science of classification. Originally, it referred only to the classifying of organisms. Now, it is often used in a more general setting, referring to the classification of things or concepts, as well the schemes underlying such a classification. In addition, taxonomy normally has some hierarchical relationships embedded in its classifications.

Table 4.1 shows a small example of taxonomy of American cities, categorized according to a hierarchy of regions and states in the United States. Note that just a few cities are included to show the example.

Table 4.1 A small example of taxonomy of American cities

Region	State	City
Southwest	California	San Francisco
		Los Angeles
	Arizona	Tucson
		Phoenix
Midwest	Indiana	Ft. Wayne
		West Lafayette
	Illinois	Chicago
		Milwaukee

- Thesaurus

Thesaurus can be understood as an extension to taxonomy: it takes taxonomy as described above, allowing subjects to be arranged in a hierarchy and in addition, it adds the ability to allow other statements be made about the subjects. Table 4.2 shows some of the examples.

The following is a small example, which can help us to put the above together:

```
Tennis
  RT Courts
  BT Sports
Sports
  BT Activity
  NT Tennis
  NT Football
  NT Basketball
```

Table 4.2 A thesaurus allows statements to be made about the subjects

Thesaurus term	Meaning
BT	Short for “broader term,” refers to the term above the current one in the hierarchy and must have a wider or less specific meaning
NT	Short for “narrower term,” an inverse property of BT. In fact, a taxonomy is a thesaurus that only uses the BT/NT properties to build a hierarchy. Therefore, every thesaurus contains a taxonomy
SN	Short for “scope note,” and it is a string attached to the term explaining its meaning within the thesaurus
USE	Refers to another term that is to be preferred instead of this term, implying that the terms are synonymous. For example, if we have a term named “Resource Description Framework,” we can put a USE property referring to another term named RDF. This means that we have the term “Resource Description Framework,” but “RDF” means the same thing, and we encourage the use of term “RDF” instead of this one
UF	An inverse property of USE
TT	Short for “top term,” refers to the topmost ancestor of this current term
RT	Short for “related term,” refers to a term that is related to this term, without being a synonym of it or a broader/narrower term

Now that we understand both taxonomy and thesaurus as examples of KOSs, the question is why we need these schemes? How do they help us in real life?

KOSs can be useful in many ways. The following is just to name a few:

- They can make search more robust (instead of simple keywords matching, related words, for example, can also be considered).
- They can help to build more intelligent browsing interfaces (following the hierarchical structure, and explore broader/narrower terms, etc.).
- They can help us to formally organize our knowledge for a given domain, therefore promote reuse of the knowledge, and also facilitate data interoperability.

With all these said, let us continue on to understand how KOSs are related to ontologies, and why we are interested in KOSs in the world of Semantic Web.

4.5.2 *Thesauri vs. Ontologies*

To understand how KOSs are related to ontologies, we use the example of thesauri vs. ontologies. There are quite a few KOSs in the application world; concentrating on one of them will make our discussion a lot easier. In addition, taxonomies are just special thesauri, therefore, comparing thesauri with ontologies does include taxonomies as well.

- KOSs are used for knowledge organization, whilst ontologies are used for knowledge representation.

Compared to ontologies, KOSs’ descriptive capability is simply far too weak, which is also the reason why KOSs cannot be used to represent knowledge. More

specifically, the broader/narrower relationship used to build the hierarchy is essentially the only one relationship offered by a taxonomy. A thesaurus extends this with the BT/RT and UF/USE relationships, and the SN property, which allows them to better describe the terms. However, the descriptive power offered by these language constructs are still very limited.

- KOSs are semantically much less rigorous than ontologies, and no formal reasoning can be conducted by just having KOSs.

As we will learn later, ontologies are based upon description logic, therefore logical inferencing can be conducted. However, in KOSs, relationships between concepts are semantically weak. For example, ontologies can specify a *is-a* relationship, while in thesauri, the hierarchical relation can represent anything from *is-a* to *part-of*, depending on the interpretations rooted from the domain and application.

With all these said, KOSs cannot match up with ontologies when it comes to fully represent the knowledge in the ways that the Semantic Web requires. However, there are indeed needs to port KOSs to the Semantic Web. Some of the reasons can be summarized as follows:

- porting KOSs into the Semantic Web so these schemes are machine readable and can be exploited in a much more effective and intelligent way;
- porting KOSs into the shared space offered by the Semantic Web will promote reuse of these schemes, and further promote interoperability;
- porting KOSs into the Semantic Web allows KOSs to leverage all the new ideas and technologies originated from the Semantic Web. For example, part of the implementation of porting KOSs to the Semantic Web means to have each single concept represented by a URI, and therefore uniquely identified on the Web. Furthermore, “similar” concepts contained in different KOS schemes can be linked together, which will then form a distributed, heterogeneous global concept scheme. Obviously, this global scheme can be used as the foundation for new applications that allow meaningful navigation between KOSs.

You will come up with other benefits when you have more experience with the Semantic Web and KOSs. For now, the key question is, how to port these existing KOSs to the Semantic Web so that machine can understand them? This gives rise to SKOS, a vocabulary built specifically for this purpose, as we will discuss in the next few sections.

4.5.3 *Filling the Gap: SKOS*

4.5.3.1 What Is SKOS?

SKOS, short for simple knowledge organization systems, is an RDF vocabulary for representing KOSs, such as taxonomies, thesauri, classification schemes, and

subject heading lists. It is used to port existing KOSs into the shared space of the Semantic Web; therefore they can be published on the Web and they can be machine readable and exchanged between software applications.

SKOS is developed by W3C Semantic Web Development Working Group (SWDVG) and has an official Web site⁴ which contains all the information related to SKOS. It has become a W3C standard on 18 August 2009. This standard includes the following specifications:

- SKOS Reference W3C Recommendation;
- SKOS Primer W3C Working Group Note;
- SKOS Use Cases and Requirements W3C Working Group Note; and
- SKOS RDF files.⁵

Recall the Dublin Core vocabulary we have discussed in [Chap. 2](#): whenever we would like to use RDF statements to describe a document, we should use the terms from Dublin Core vocabulary. SKOS is the vocabulary we should use when we try to publish a given KOS into the shared space of the Semantic Web.

Note that the URIs in SKOS vocabulary all have the following lead strings:

```
http://www.w3.org/2004/02/skos/core#
```

By convention, this URI prefix string is associated with namespace prefix `skos:` and is typically used in different sterilization formats with the prefix `skos`.

4.5.3.2 SKOS Core Constructs

In this section, we will discuss the core constructs of SKOS, which will include the following:

- Conceptual resources should be identified by URIs and can be explicated as concepts.
- Concepts can be labeled with lexical strings in one or more natural languages.
- Concepts can be documented with different types of notes.
- Concepts can be semantically related to each other in informal hierarchies.
- Concepts can be aggregated into concept schemes.

These SKOS features are not all that are offered by the SKOS model, but will be enough for representing most KOSs on the Semantic Web. For the rest of this section, we will use Turtle for our examples, and the following namespaces will be needed. We now list these namespaces here so they will not be included in every single example:

⁴<http://www.w3.org/2004/02/skos/>

⁵<http://www.w3.org/2004/02/skos/vocabs>

```

@prefix skos: <http://www.w3.org/2004/02/skos/core#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix dc: <http://purl.org/dc/elements/1.1/>.
@prefix ex: <http://www.example.com/> .
@prefix ex1: <http://www.example.com/1/> .
@prefix ex2: <http://www.example.com/2/> .

```

Concept is a fundamental element in any given KOS. SKOS introduces the class `skos:Concept`, so that we can use it to state the fact that a given resource is a concept. To do so, we first create (or reuse) a URI to uniquely identify the concept; we then use one RDF statement to assert that the resource, identified by this URI, is of type `skos:Concept`.

For example, the following RDF statement says tennis is a `skos:Concept`:

```
<http://dbpedia.org/resource/Tennis> rdf:type skos:Concept.
```

Instead of creating a URI to represent tennis as a concept, we reuse the URI for tennis created by DBpedia (we will see more about DBpedia project in [Chap. 10](#)). Clearly, using SKOS to publish concept schemes makes it easy to reference the concepts in resource descriptions on the Semantic Web. In this particular example, for the resource <http://dbpedia.org/resource/Tennis>, besides everything that has been said about it, we know it is also a `skos:Concept`.

The first thing to know about `skos:Concept` is that SKOS allows us to use labels on a given concept. Three label properties are provided: `skos:prefLabel`, `skos:altLabel` and `skos:hiddenLabel`. They are all sub-properties of the `rdfs:label` property, and they are all used to link a `skos:Concept` to an RDF plain literal, which is formally defined as a character string combined with an optional language tag. More specifically,

- `skos:prefLabel` property is used to assign a preferred lexical label to a concept.

This preferred lexical label should contain terms used as descriptors in indexing systems and is normally used in a KOS to unambiguously represent the underlying concept. Therefore it is recommended that no two concepts in the same KOS be given the same preferred lexical label for any given language tag.

- `skos:altLabel` property is used when synonyms, near-synonyms, or abbreviations need to be represented.
- `skos:hiddenLabel` property is used mainly for indexing and/or searching capabilities.

For example, the character string as the value of this property will be accessible to applications performing text-based indexing and searching operations, but will

not be visible otherwise. A good example is to include misspelled variants of the preferred label.

List 4.14 shows how these properties are used for the concept tennis.

List 4.14 Different label properties used for tennis concept

```
<http://dbpedia.org/resource/Tennis> rdf:type skos:Concept;
    skos:prefLabel "tennis"@en;
    skos:altLabel "Lawn_Tennis"@en;
    skos:hiddenLabel "Tennis"@en.
```

The second characterizations of concepts are the human-readable documentation properties defined for a given concept. `skos:scopeNote`, `skos:definition`, `skos:example` and `skos:historyNote` are examples of these properties. And all these properties are sub-properties of `skos:note` property. These properties are all quite straightforward and do not require much of an explanation.

For example, `skos:definition` property is used to provide a complete explanation of the intended meaning of a concept. Note that the organization of these properties, with `skos:note` as their root, offers a straightforward way to retrieve all the documentation associated with one single concept. For instance, to find all the documentation for a concept, all we need to find is all the sub-property values of the `skos:note` property.

At this point, List 4.15 is the latest definition of our tennis concept.

List 4.15 Use `skos:definition` in our tennis concept

```
<http://dbpedia.org/resource/Tennis> rdf:type skos:Concept;
    skos:prefLabel "tennis"@en;
    skos:altLabel "Lawn_Tennis"@en;
    skos:hiddenLabel "Tennis"@en;
    skos:definition "Tennis is a sport usually played
between two players or between two teams of two players each.
Each player uses a racket that is strung to strike a hollow
rubber ball covered with felt past a net into the
opponent's court."@en.
```

Now, let us take a look at some semantic relationships that can be specified when defining a concept using SKOS. For a given KOS, the meaning of a concept is defined not just by the natural-language words in its labels, but also by its relationships to other concepts in the same KOS. To map these relationships to a machine-readable level, three standard properties, `skos:broader`, `skos:narrower` and `skos:related`, are offered by SKOS vocabulary. More specifically,

- `skos:broader` and `skos:narrower` together are used for representing the hierarchical structure of the KOS, which can be either a is-a relationship (similar to a

class and sub-class relationship) or a part-of relationship (one concept represents a resource that is a part of the resource represented by another concept).

For example, List 4.16 shows the usage of `skos:broader`.

List 4.16 Use `skos:broader` in our tennis concept

```
<http://dbpedia.org/resource/Tennis> rdf:type skos:Concept;
    skos:prefLabel "tennis"@en;
    skos:altLabel "Lawn_Tennis"@en;
    skos:hiddenLabel "Tenis"@en;
    skos:broader <http://dbpedia.org/resource/Racquet_sport>.
```

Based on List 4.16, http://dbpedia.org/resource/Racquet_sport is another concept that is broader in meaning. Again, the URI of this new concept is taken from DBpedia, another example of URI reuse.

Note that `skos:broader` property does not explicitly indicate its direction, and it should be read as “has broader concept.” In other words, the subject of a `skos:broader` statement is the more specific concept, and the object is the more general one.

Also note that `skos:broader` and `skos:narrower` are each other’s inverse property. You will see more about inverse property in later chapters, but for now, understand that if an inferencing engine reads List 4.16, it will be able to add the following inferred statement automatically:

```
<http://dbpedia.org/resource/Racquet_sport> skos:narrower
<http://dbpedia.org/resource/Tennis>.
```

meaning that the subject has a narrower concept identified by the object.

Note that the SKOS vocabulary does not specify `skos:broader` and `skos:narrower` as transitive properties.

For example, <http://dbpedia.org/resource/Tennis>, as a concept, has http://dbpedia.org/resource/Racquet_sport as its broader concept. And this later concept itself has <http://dbpedia.org/resource/Sport> as a broader concept.

Therefore, <http://dbpedia.org/resource/Tennis> should have another broader concept called <http://dbpedia.org/resource/Sport>. This chain of transitivity does make sense, but we can also find example where such transitivity does not make sense. Therefore, `skos:broader` and `skos:narrower` are not formally considered as transitive properties.

- `skos:related` is used for non-hierarchical links, but for associative relationship between two concepts.

List 4.17 shows one example of using `skos:related`.

List 4.17 Use `skos:related` in our tennis concept

```
<http://dbpedia.org/resource/Tennis> rdf:type skos:Concept;
    skos:prefLabel "tennis"@en;
    skos:altLabel "Lawn_Tennis"@en;
    skos:hiddenLabel "Tenis"@en;
    skos:broader <http://dbpedia.org/resource/Racquet_sport>;
    skos:related
<http://dbpedia.org/reource/International_Tennis_Federation>.
```

List 4.17 claims that <http://dbpedia.org/resource/Tennis> is related to another concept given by the following URI:

```
http://dbpedia.org/resource/International_Tennis_Federation
```

Understand that `skos:related` is a symmetric property (you will see symmetric property in later chapters). Therefore, an inferencing engine will be able to add the following statement based on List 4.17:

```
<http://dbpedia.org/reource/International_Tennis_Federation>
skos:related
<http://dbpedia.org/resource/Tennis>.
```

Again, note that the SKOS vocabulary does not specify `skos:related` to be transitive property, as is the case for `skos:broader` and `skos:narrower`.

At this point, we have covered those related terms in SKOS vocabulary so we understand how to define a concept, label a concept, add documentation notes about a concept, and also, how to specify semantic relationships about a concept. Obviously, for a given KOS, there will be multiple concepts and these concepts are logically contained together by the same KOS to form a vocabulary. SKOS offers `skos:ConceptScheme` class and other related terms to model this aspect of a vocabulary. Let us take a look at these constructs.

First off, the following shows how to define a concept scheme that represents a vocabulary:

```
ex:myTennisVocabulary rdf:type skos:ConceptScheme;
    dc:creator ex:liyangYu.
```

This declares a vocabulary (concept scheme) named `myTennisVocabulary`. And by using `skos:inScheme` property, we can add our tennis concept into this vocabulary, as shown in List 4.18.

List 4.18 Use `skos:ConceptScheme` and `skos:inScheme` to build vocabulary

```
ex:myTennisVocabulary rdf:type skos:ConceptScheme;
    dc:creator ex:liyangYu.
<http://dbpedia.org/resource/Tennis> rdf:type skos:Concept;
    skos:inScheme ex:myTennisVocabulary;
    skos:prefLabel "tennis"@en;
```

```

    skos:altLabel "Lawn_Tennis"@en;
    skos:hiddenLabel "Tennis"@en;
    skos:broader <http://dbpedia.org/resource/Racquet_sport>;
    skos:related
<http://dbpedia.org/reource/International_Tennis_Federation>.

```

We can now add more concepts (and labels for concepts, relationships between concepts, etc.) as we wish, just like what we have done in List 4.18, until we have covered all the concepts and relationships in a given KOS. This way, we can create a vocabulary that represents a given KOS. The final result is that the given KOS has now been converted to a machine-readable RDF document and can be shared and reused on the Semantic Web. This process is called *mapping* a KOS onto the Semantic Web.

`skos:hasTopConcept` is another very useful property provided by the SKOS vocabulary. This can be used to provide an “entry point” that we can use to access the machine-readable KOS. List 4.19 shows how.

List 4.19 Use `skos:hasTopConcept` to provide an entry point of the vocabulary

```

ex:myTennisVocabulary rdf:type skos:ConceptScheme;
    skos:hasTopConcept <http://dbpedia.org/resource/Tennis>;
    dc:creator ex:liyangYu.

<http://dbpedia.org/resource/Tennis> rdf:type skos:Concept;
    skos:inScheme ex:myTennisVocabulary;
    skos:prefLabel "tennis"@en;
    skos:altLabel "Lawn_Tennis"@en;
    skos:hiddenLabel "Tennis"@en;
    skos:broader <http://dbpedia.org/resource/Racquet_sport>;
    skos:related
<http://dbpedia.org/reource/International_Tennis_Federation>.

```

Now, an application can query the value of `skos:hasTopConcept` property and use the returned concept and its `skos:broader` and `skos:narrower` properties to explore the whole vocabulary. Note that multiple `skos:hasTopConcept` properties can be defined for a given concept scheme.

4.5.3.3 Interlinking Concepts by Using SKOS

At this point, we understand that we can use SKOS to map a traditional KOS onto the Semantic Web. The key difference between a traditional KOS and its corresponding Semantic Web version is that the latter is machine readable. When we claim a given KOS is machine readable, we mean the following facts:

- every SKOS concept is identified by a URI, and
- everything is expressed using RDF statements, which can be processed by machines.

The fact that every concept is uniquely identified by a URI makes it possible to state that two concepts from different schemes have some semantic relations. With the help of these interlinking concepts, applications such as information retrieval packages can start to make use of several KOSs at the same time. In fact, linking concepts contained in different KOSs is considered to be a key benefit of publishing KOSs on the Semantic Web.

SKOS vocabulary provides the following terms one can use to build the interlinks between concepts:

- `skos:exactMatch` and `skos:closeMatch`
- `skos:broadMatch`, `skos:narrowMatch` and `skos:relatedMatch`

Using property `skos:closeMatch` means that the two concepts are close enough in meanings and they can be used interchangeably in applications that are built upon the two schemes containing these two concepts. For example, List 4.20 shows how this property is used.

List 4.20 Use `skos:closeMatch` to link concept in another vocabulary

```
ex:myTennisVocabulary rdf:type skos:ConceptScheme;
  skos:hasTopConcept <http://dbpedia.org/resource/Tennis>;
  dc:creator ex:liyangYu.
<http://dbpedia.org/resource/Tennis> rdf:type skos:Concept;
  skos:inScheme ex:myTennisVocabulary;
  skos:prefLabel "tennis"@en;
  skos:altLabel "Lawn_Tennis"@en;
  skos:hiddenLabel "Tennis"@en;
  skos:closeMatch ex2:Tennis;
  skos:broader <http://dbpedia.org/resource/Racquet_sport>;
  skos:related
  <http://dbpedia.org/reource/International_Tennis_Federation>.
```

This says, among other things, that <http://dbpedia.org/resource/Tennis> concept is a close match to another concept named `ex2:Tennis`.

Note that `skos:closeMatch` is not transitive, which is also the main difference between `skos:closeMatch` and property `skos:exactMatch`. More specifically, `skos:exactMatch` is a sub-property of `skos:closeMatch`, and it indicates the two concepts have equivalent meanings. Therefore, any application that makes use of this property can expect an even stronger link between schemes. In addition, `skos:exactMatch` is indeed declared as transitive property, as you might have guessed.

For `skos:broadMatch`, `skos:narrowMatch` and `skos:relatedMatch`, their usage is quite straightforward and does not need much of an explanation. Also,

- `skos:broadMatch` is a sub-property of `skos:broader`;
- `skos:narrowMatch` is a sub-property of `skos:narrower`; and
- `skos:relatedMatch` is a sub-property of `skos:related`.

With these said, for example, a statement which asserts a `skos:broadMatch` between two concepts will be treated as a statement that declares a `skos:broader` between these two concepts.

At this point, we have finished the discussion of the SKOS vocabulary. We have not covered everything about it, but what we have learned here will be enough to get you started. As a summary, you can use the SKOS vocabulary to map a given KOS to the Semantic Web and change it to machine readable, therefore bridge the world of taxonomies and thesauri and other controlled vocabularies to the world of the Semantic Web.

4.6 Another Look at Inferencing Based on RDF Schema

We have discussed the benefits offered by ontologies on the Semantic Web. In order to convince ourselves, let us take another look at our camera ontology to see how it can make machine more intelligent. In addition, we will not only see more reasoning power provided by our camera ontology, but also find things that can be improved – this will point to another new building block called OWL, which be presented in the next chapter in details.

4.6.1 RDFS Ontology-Based Reasoning: Simple, Yet Powerful

Early this chapter (Sect. 4.2.2), we have used an example to show you how reasoning is done by using the camera ontology (we called it camera vocabulary back then). In this section, we present this reasoning ability in a more formal way, together with the extra reasoning examples that we did not cover in the previous sections.

More specifically, with the help of the camera ontology, a given application can accomplish reasoning in the following ways:

- Understand a resource’s class type by reading the property’s `rdfs:domain` tag.

When we define a property **P**, we normally use `rdfs:domain` to specify exactly which class this property **P** can be used to describe; let us use **C** to denote this class. Now for a given resource identified by a specific URI, if our application detects property **P** is indeed used to describe this resource, our application can then conclude the resource represented by this particular URI must be an instance of class **C**. We have example for this type of reasoning presented in Sect. 4.2.2, as you have seen.

- Understand a resource’s class type by reading the property’s `rdfs:range` tag.

When we define a property \mathbf{P} , we normally use `rdfs:range` to specify exactly what are the possible values this property can assume. More specifically, this value can be a typed or un-typed literal and can also be an instance of a given class \mathbf{C} . Now when parsing a resource, if our application detects that property \mathbf{P} is used to describe this resource, and the value of \mathbf{P} of this resource is represented by a specific URI pointing to another resource, our application can then conclude the resource represented by this particular URI must be an instance of class \mathbf{C} .

To see how this works, take a look at the simple RDF document presented in List 4.21.

List 4.21 A simple RDF document using camera ontology

```

1: <?xml version="1.0"?>
2: <rdf:RDF
3:   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
4:   xmlns:myCamera="http://www.liyangyu.com/camera#">
5: <rdf:Description
6:   rdf:about="http://www.liyangyu.com/camera#Nikon_D300">
7:   <myCamera:lens rdf:resource=
8:     "http://dbpedia.org/resource/Nikon_17-35mm_f/2.8D_ED-
9:     IF_AF-S_Zoom-Nikkor"/>
10: </rdf:Description>
11: </rdf:RDF>

```

This is a very simple RDF document: it only uses one property, namely, `myCamera:lens` to describe the given resource (line 6). However, based on the definition of this property (lines 53–56, List 4.13), an application is able to make the following reasoning:

`http://dbpedia.org/resource/Nikon_17-35mm_f/2.8D_ED-IF_AF-S_Zoom-Nikkor` is an instance of class `myCamera:Lens`.

Again, note that we have used an existing URI from DBpedia to represent a Nikon zoom lens, the reason being the same: we should reuse URI as much as we can.

In fact, our application, based on the definition of property `myCamera:lens`, can also make the following reasoning:

`http://www.liyangyu.com/camera#Nikon_D300` is an instance of class `myCamera:Camera`.

- Understand a resource's super class type by following the class hierarchy described in the ontology.

This can be viewed as extension to the above two reasoning scenarios. In both of the above cases, the final result is that the class type of some resource has been successfully identified. Now our application can scan the class hierarchy defined in the ontology; if the identified class has one or more super classes defined in the ontology, our application can then conclude that this particular resource is not only an instance of the identified class, but also an instance of all the super classes.

- Understand more about the resource by using the `rdfs:subPropertyOf` tag.

Let us use an example to illustrate this reasoning. Suppose we have defined the following property:

```
<rdf:Property rdf:ID="parent">
<rdfs:domain rdf:resource="#Person"/>
<rdfs:range rdf:resource="#Person"/>
</rdf:Property>
<rdf:Property rdf:ID="mother">
<rdfs:subClassOf rdf:resource="#parent"/>
</rdf:Property>
```

This defines two properties, namely, `parent` and `mother`, with `mother` being a sub-property of `parent`. Assume we have a resource in the RDF statement document:

```
<Person rdf:ID="Liyang">
<mother>
<Person rdf:resource="#Zaiyun"/>
</mother>
</Person>
```

When parsing this statement, an application realizes the fact that `Liyang`'s `mother` is `Zaiyun`. One step further, since `mother` is a sub-property of `parent`, it then concludes that `Liyang`'s `parent` is also `Zaiyun`. This can be very useful in some cases.

The above are the four main ways a given application can make inferences based on the given ontology, together with the instance document. These are indeed simple yet very powerful already.

4.6.2 *Good, Better, and Best: More Is Needed*

RDF Schema is quite impressive indeed: you can use its terms to define ontologies, and having these ontologies defined, our application can conduct reasoning on the run. However, there are still something missing when you use RDFS vocabulary to define ontologies.

For example, what if we have two classes representing the same concept? For example, we have a `DSLRL` class in our camera ontology, and we know `DSLRL` represents digital single lens reflex, and it will be quite useful if we can define another class named `DigitalSingleLensReflex` and also indicate in our ontology that these two classes represent exactly the same concept in life. However, using RDF Schema, it is not possible to accomplish this.

Another example is there is no cardinality constraint available using RDF Schema. For example, `effectivePixel` is a property that is used to describe the image size of a digital camera. For one particular camera, there should be only one `effectivePixel` value. However, in our RDF document, we can use multiple `effectivePixel` properties on a single digital camera instance!

Therefore, there is indeed a need to extend RDF Schema to allow for the expression of more complex relationships among classes and of more precise constraints on specific classes and properties. In other words, we need a more advanced language which will be able to do the following:

- to express relationships among classes defined in different documents across the Web;
- to construct new classes by unions, intersections, and complements of other existing classes;
- to add constraints on the number and type for properties of classes;
- to determine if all members of a class will have a particular property, or if only some of them might;
- and more.

This new language is called OWL and it is the main topic of the next chapter, read on.

4.7 Summary

In this chapter, we have learned RDFS, another important building block for the Semantic Web.

The first thing we should understand from this chapter is how RDFS fits into the whole concept of the Semantic Web. More specifically, this includes the following main points:

- It provides a collection of terms (RDFS vocabulary) that one can use to build ontologies.
- With the ontologies, RDF documents can be created by using sharing knowledge and common terms, i.e., whatever we say, we have a reason to say it.

In order for us to create ontologies by using RDFS, this chapter also covers the main language features of RDFS. We should have learned the following:

- the concept of RDFS vocabulary, and how it is related to other vocabularies, such as RDF vocabulary, Dublin Core vocabulary;
- understand the key terms contained in RDFS vocabulary, and how to use them to develop domain-specific ontologies.

This chapter also discusses the concept of ontologies, and further introduces another vocabulary called SKOS. We should have learned the following main points:

- the concept of ontology, and the reason of having ontologies for the Semantic Web;
- the concept of SKOS, and how to use SKOS vocabulary to map an existing KOS onto the Semantic Web, and certainly, the benefit of doing so.

Finally, this chapter shows the reasoning power provided by ontologies. This includes the following:

- The meaning of a given term is expressed by the properties that can be used on this term and the values these properties can assume.
- Machine can understand such meanings, and four different ways of reasoning can be implemented by machines based on this understanding.
- RDF Schema can be improved in a number of different ways.

In the next chapter, we will present OWL, essentially a much more advanced version of RDFS, and you will have more chances to see how machine can understand the meanings by conducting useful reasoning on the fly.

Chapter 5

OWL: Web Ontology Language

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

5.1 OWL Overview

5.1.1 OWL in Plain English

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

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

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

In plain English, we can define OWL as follows:

OWL = RDF Schema + new constructs for better expressiveness

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

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

5.1.2 OWL in Official Language: OWL 1 and OWL 2

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

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

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

OWL started as a research-based revision of the DAML+OIL Web ontology language. W3C created the Web Ontology Working Group² in November 2001, and the first working drafts of the abstract syntax, reference and synopsis were published in July 2002. The OWL documents became a formal W3C Recommendation on 10 February 2004. The recommendation includes the following documents³:

- OWL Web Ontology Language Overview
- OWL Web Ontology Language Guide
- OWL Web Ontology Language Reference
- OWL Web Ontology Language Semantics and Abstract Syntax

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

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

³<http://www.w3.org/2004/OWL/#specs>

- OWL Web Ontology Language Test Cases
- OWL Web Ontology Language Use Cases and Requirements

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

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

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

The 2005 OWL Experiences and Directions Workshop⁴ has created a list of new features to be provided by OWL 1.1. The actual development of these new features was then undertaken by an informal group of language users and developers. The deliverables of their work was submitted to W3C as a member submission, and at the same time, a new W3C OWL Working Group⁵ was officially formed in September 2007.

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

On 27 October 2009, OWL 2 has become a W3C standard,⁶ which has the following core specifications:

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

These core specifications are part of the W3C OWL 2 Recommendations, and they are mainly useful for ontology tool designers. For example, in order to implement a OWL 2 validator or a reasoner that understands OWL 2, one has to be familiar with

⁴<http://www.mindswap.org/2005/OWLWorkshop/>

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

⁶<http://www.w3.org/TR/2009/REC-owl2-overview-20091027/>

these specifications. For developers like us, this chapter will help you to learn how to use OWL 2 to develop your own ontology documents.

With the understanding of the history behind OWL, let us take a look at its official definition. W3C's OWL 2 Primer⁷ has given a good definition about OWL:

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

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

5.1.3 From OWL 1 to OWL 2

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

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

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

5.2 OWL 1 and OWL 2: The Big Picture

Form here to the rest of this chapter, we will cover the syntax of OWL, together with examples. Our goal is to re-write our camera vocabulary developed in last chapter, and by doing so, we will cover most of the OWL features.

Similar to RDFS, OWL can be viewed as a collection of terms we can use to define classes and properties for a specific application domain. These pre-defined

⁷<http://www.w3.org/TR/2009/REC-owl2-primer-20091027/>

OWL terms all have the following URI as their leading string (applicable to both OWL 1 and OWL 2):

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

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

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

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

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

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

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

The last concept we would like to mention here is the *IRI* names; you will encounter this concept when reading OWL 2 related literatures.

As we know at this point, URIs are the standard mechanism for identifying resources on the Web. For the vision of the Semantic Web, we have been using URIs to represent classes, properties, and individuals, as shown in [Chaps. 2](#) and [4](#). This URI system fits well into the Semantic Web for the following two main reasons:

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

The first point here should be fairly clear (refer to [Chap. 2](#) for details), and the second point will be covered in detail in [Chap. 11](#).

IRIs stands for *Internationalized Resource Identifiers*, and they are just like URIs except that they can make use of the whole range of Unicode characters. As a comparison, URIs are limited to the ASCII subset of the characters, which only

has 127 characters. In addition, the ASCII subset itself is based on the needs of English-speaking users, which presents some difficulty for non-English users. And these considerations have been the motivation of IRIs.

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

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

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

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

- Functional-Style syntax

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

Once the structural specification is complete, it is necessary to move toward some concrete syntaxes. The first step of doing so is the Functional-Style syntax. This syntax is designed for translating the structural specification to various other syntaxes, and it is often used by OWL tool designers. In general, it is not intended to be used as an exchange syntax, and as OWL users, we will not be seeing or using this syntax often.

- RDF/XML syntax

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

- Manchester syntax

The Manchester syntax provides a textual based representation of OWL ontologies that is easy to read and write. The motivation behind Manchester syntax was to design a syntax that could be used for editing class expressions in tools such as

Protégé and alike. Since it is quite successful in these tools, it has been extended to represent a complete ontology.

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

- OWL/XML

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

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

5.3 OWL 1 Web Ontology Language

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

5.3.1 *Defining Classes: The Basics*

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

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

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

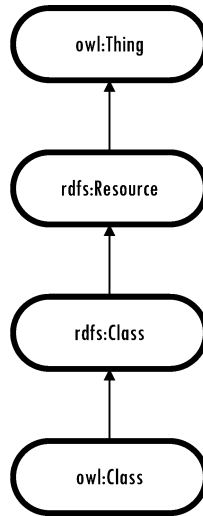


Fig. 5.1 Relationship between top classes

And the following is an equivalent format:

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

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

List 5.1 Class definitions for our camera ontology using OWL 1

```

1: <?xml version="1.0"?>
2: <rdf:RDF
3:   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
4:   xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
5:   xmlns:owl="http://www.w3.org/2002/07/owl#"
6:   xmlns:myCamera="http://www.liyangyu.com/camera#"
7:   xml:base="http://www.liyangyu.com/camera#">
8:   <owl:Class rdf:about="http://www.liyangyu.com/camera#Camera">
9:     </owl:Class>
10:
11:   <owl:Class rdf:about="http://www.liyangyu.com/camera#Lens">
12:     </owl:Class>
13:
14:   <owl:Class rdf:about="http://www.liyangyu.com/camera#Body">
15:     </owl:Class>
16:
17:   <owl:Class
17a:     rdf:about="http://www.liyangyu.com/camera#ValueRange">
18:     </owl:Class>
19:

```

```

20: <owl:Class
20a:   rdf:about="http://www.liyangyu.com/camera#Digital">
21:   <rdfs:subClassOf rdf:resource="#Camera"/>
22: </owl:Class>
23:
24: <owl:Class rdf:about="http://www.liyangyu.com/camera#Film">
25:   <rdfs:subClassOf rdf:resource="#Camera"/>
26: </owl:Class>
27:
28: <owl:Class rdf:about="http://www.liyangyu.com/camera#DSLR">
29:   <rdfs:subClassOf rdf:resource="#Digital"/>
30: </owl:Class>
31:
32: <owl:Class
32a:   rdf:about="http://www.liyangyu.com/camera#PointAndShoot">
33:   <rdfs:subClassOf rdf:resource="#Digital"/>
34: </owl:Class>
35:
36: <owl:Class
36a:   rdf:about="http://www.liyangyu.com/camera#Photographer">
37:   <rdfs:subClassOf
37a:     rdf:resource="http://xmlns.com/foaf/0.1/Person"/>
38: </owl:Class>
39:
40:</rdf:RDF>

```

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

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

5.3.2 Defining Classes: Localizing Global Properties

In [Chap. 4](#), we have defined properties by using RDFS terms. For example, recall the definition of `owned_by` property:

```

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

```

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

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

5.3.2.1 Value Constraints: `owl:allValuesFrom`

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

Let us say we now want to express the following fact: `DSLR`, especially an expensive one, is normally used by professional photographers. For example, only the body of some high-end digital SLR can cost as much as \$5000.00.

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

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

```

1: <?xml version="1.0"?>
2: <rdf:RDF
3:     xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
4:     xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
5:     xmlns:owl="http://www.w3.org/2002/07/owl#"
6:     xmlns:myCamera="http://www.liyangyu.com/camera#"
7:     xml:base="http://www.liyangyu.com/camera#">
8:
9:     ... same as List 4.1
10:
11:
12:
13:
14:
15:
16:
17:
18:
19:
20:
21:
22:
23:
24:
25:
26:
27:
28:
29:
30:
31:
32:
33:
34:
35:
36:
37:
38:
39:
40: <owl:Class
41:     rdf:about="http://www.liyangyu.com/camera#Professional">
42:     <rdfs:subClassOf rdf:resource="#Photographer"/>
43: </owl:Class>
44:
45:
46:
47:
48: <owl:Class
49:     rdf:about="http://www.liyangyu.com/camera#Amateur">
50:     <rdfs:subClassOf rdf:resource="#Photographer"/>
51: </owl:Class>
52:
53:
54:
55:
56:
57:
58:
59:
60:
61:
62:
63:
64:
65:
66:
67:
68:
69:
70:
71:
72:
73:
74:
75:
76:
77:
78:
79:
80:
81:
82:
83:
84:
85:
86:
87:
88:
89:
90:
91:
92:
93:
94:
95:
96:
97:
98:
99:
100: </rdf:RDF>

```

Does this ontology successfully express our idea? Not really. Since `owned_by` has `DSLR` as its `rdfs:domain` and `Photographer` as its `rdfs:value`, and given the fact that `ExpensiveDSLR` is a sub-class of `DSLR`, `Professional` and `Amateur`

are both sub-classes of `Photographer`, these new sub-classes all inherit the `owned_by` property. Therefore, we can indeed say something like this:

```
ExpensiveDSLR owned_by Professional
```

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

```
ExpensiveDSLR owned_by Amateur
```

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

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

```
1: <owl:Class
1a:   rdf:about="http://www.liyangyu.com/camera#ExpensiveDSLR">
2:   <rdfs:subClassOf rdf:resource="#DSLR"/>
3:   <rdfs:subClassOf>
4:     <owl:Restriction>
5:       <owl:onProperty rdf:resource="#owned_by"/>
6:       <owl:allValuesFrom rdf:resource="#Professional"/>
7:     </owl:Restriction>
8:   </rdfs:subClassOf>
9: </owl:Class>
```

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

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

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

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

With all these said, List 5.3 should be easier to understand. Lines 4–7 use `owl:Restriction` to define an anonymous class, the constraint is applied on `owned_by` property (this is specified by using `owl:onProperty` property in line 5), and the values for `owned_by` property has to all come from instances of class

Professional (this is specified by using `owl:allValuesFrom` property in line 6). Therefore, lines 4–7 can be read as follows:

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

With this, List 5.3 can be read as follows:

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

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

Here is a definition of class `ExpensiveDSLR`, it is a sub-class of `DSLR` and it has a property named `owned_by`, and only instance of class `Professional` can be the value for this property.

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

5.3.2.2 Enhanced Reasoning Power 1

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

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

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

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

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

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

```

    <myCamera:owned_by
      rdf:resource="http://www.liyangyu.com/people#Connie"/>
</myCamera:ExpensiveDSLR>

```

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

```

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

```

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

5.3.2.3 Value Constraints: `owl:someValuesFrom`

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

List 5.4 Use `owl:someValuesFrom` to define `ExpensiveDSLR` class

```

1: <owl:Class
1a:   rdf:about="http://www.liyangyu.com/camera#ExpensiveDSLR">
2:   <rdfs:subClassOf rdf:resource="#DSLR"/>
3:   <rdfs:subClassOf>
4:     <owl:Restriction>
5:       <owl:onProperty rdf:resource="#owned_by"/>
6:       <owl:someValuesFrom rdf:resource="#Professional"/>
7:     </owl:Restriction>
8:   </rdfs:subClassOf>
9: </owl:Class>

```

This can be read like this:

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

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

5.3.2.4 Enhanced Reasoning Power 2

Our application sees the following instance document:

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

The application will be able to add the following facts:

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

If our application sees the following instance document:

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

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

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

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

5.3.2.5 Value Constraints: `owl:hasValue`

Another way OWL 1 uses to localize a global property in the context of a given class is to use `owl:hasValue`. So far, we have defined `ExpensiveDSLR` as being a DSLR that is owned by a professional photographer (List 5.3), or owned by at least one professional photographer (List 5.4). These definitions are fine, but they are not straightforward. In fact, we can use a more direct approach to define what it means to be an expensive DSLR.

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

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


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

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

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

```

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

```

This defines class `ExpensiveDSLRL` as follows:

A class called `ExpensiveDSLRL` is defined. It is a sub-class of `DSLRL`, and every instance of `ExpensiveDSLRL` has a `cost` property whose value is `expensive`.

Meanwhile, instances of `DSLRL` or `PointAndShoot` can take whatever `cost` value they want (i.e., `expensive` or `inexpensive`), indicating the fact that they can be `expensive` or `inexpensive`. This is exactly what we want.

It is now a good time to take a look at the difference among these three properties. More specifically, whenever we decide to use `owl:allValuesFrom`, it is equivalent to declare that “all the values of this property must be of this type, but it is all right if there are no values at all.” Therefore, the property instance does not even have to appear. On the other hand, using `owl:someValuesFrom` is equivalent to say “there must be some values for this property, and at least one of these values has to be of this type. It is okay if there are other values of other types.” Clearly, using a `owl:someValuesFrom` restriction on a property implies this property has to appear at least once, whereas an `owl:allValuesFrom` restriction does not require the property to show up at all.

Finally, `owl:hasValue` says “regardless of how many values a class has for a particular property, at least one of them must be equal to the value that you specify.” It is therefore very much the same as `owl:someValuesFrom` except it is more specific because it requires a particular instance instead of a class.

5.3.2.6 Enhanced Reasoning Power 3

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

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

The application will be able to add the following facts:

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

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

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

Another way to define class by adding restrictions on properties is to constrain the cardinality of a property based on the class on which it is intended to use. In this section, we will add cardinality constraints to some of our existing class definitions in our camera ontology. By doing so, not only will we learn how to use the cardinality constraints, but our camera ontology will also become more accurate.

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

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

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

This defines class `Digital` as follows:

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

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

What about `model` property? Clearly, each camera should have at least one `model` value, but it can have multiple `model` values: as we have discussed, the exact same camera, when sold in Asia or North America, can indeed have different `model` values. To take this constraint into account, we can modify the definition of `Camera` class as shown in List 5.7.

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

```

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

```

And you can use `owl:minCardinality` together with `owl:maxCardinality` to specify a range, as shown in List 5.8, which says that a camera should have at least one `model` value, but cannot have more than 3.

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

```

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

```

```

10:     </owl:maxCardinality>
11:   </owl:Restriction>
12: </rdfs:subClassOf>
13: </owl:Class>

```

5.3.2.8 Enhanced Reasoning Power 4

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

```

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

```

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

```

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

```

The application will be able to add the following fact:

```

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

```

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

```

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

```

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

5.3.3 Defining Classes: Using Set Operators

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

5.3.3.1 Set Operators

The first operator is `owl:intersectionOf`. Recall the definition of `ExpensiveDSLR` presented in List 5.5; we can re-write this definition as shown in List 5.9.

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

```

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

```

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

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

Or, we can simply read List 5.9 as this:

A class called `ExpensiveDSLR` is defined. It is a `DSLR` that `cost` is `expensive`.

So what is the difference between Lists 5.5 and 5.9? Note that List 5.5 uses multiple `owl:subClassOf` terms, and in OWL 1, this means qualified individuals should all come from a *subset* of the final intersection of all the classes specified by the multiple `owl:subClassOf` terms. On the other hand, List 5.9 means that qualified individuals should all come from the final intersection of the classes included in the class collection (line 2 of List 5.9). So there is indeed some subtle difference between these two definitions. However, as far as reasoning is concerned, these two definitions will produce the same inferred facts.

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

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

```

1: <owl:Class
1a:   rdf:about="http://www.liyangyu.com/camera#CameraCollection">
2:   <owl:unionOf rdf:parseType="Collection">

```

```

3:      <owl:Class rdf:about="#Digital"/>
4:      <owl:Class rdf:about="#Film"/>
5:    </owl:unionOf>
6: </owl:Class>

```

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

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

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

```

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

```

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

5.3.3.2 Enhanced Reasoning Power 5

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

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

- If a class `C0` is the `owl:intersectionOf` a list of classes `C1`, `C2`, and `C3`, then `C0` is sub-class of each one of `C1`, `C2`, and `C3`.
- If a class `A` is the `owl:intersectionOf` a list of classes and class `B` is the `owl:intersectionOf` another list of classes, class `A` is a sub-class of class `B` if every constituent class of `A` is a sub-class of some constituent class of `B`.
- If a class `C0` is the `owl:unionOf` a list of classes `C1`, `C2`, and `C3`, then each one of `C1`, `C2`, and `C3` is a sub-class of `C0`.
- If a class `A` is the `owl:unionOf` a list of classes and class `B` is the `owl:unionOf` another list of classes, class `A` is a sub-class of class `B` if every constituent class of `B` is a super class of some constituent class of `A`.
- If a class `A` is `owl:complementOf` a class `B`, then all the sub-classes of `A` will be `owl:disjointWith` class `B`.

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

5.3.4 Defining Classes: Using Enumeration, Equivalent, and Disjoint

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

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

And we will discuss the details in this section.

5.3.4.1 Enumeration, Equivalent, and Disjoint

Defining classes by enumeration could be quite useful for many cases. To see why, let us recall the methods we have used when defining the `ExpensiveDSLR` class. So far we have defined the class `ExpensiveDSLR` by saying that it has to be owned by a professional photographer or its `cost` property has to take the value `expensive`, etc. All these methods are a *descriptive* way to define a class: as long as an instance satisfies all the conditions, it is a member of the defined class.

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

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

```

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

```

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

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

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

```

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

```

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

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

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

```

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

```

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

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

```

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

```

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

Finally, OWL 1 also provides a way to define the fact that the two classes are not related in any way. For instance, in our camera ontology, we have defined

DSLR and PointAndShoot as sub-classes of Digital. To make things simpler and without worrying the fact that a DSLR camera in many cases can be simply used as a PointAndShoot camera, we can define DSLR to be disjoint from the PointAndShoot class, as shown in List 5.16.

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

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

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

5.3.4.2 Enhanced Reasoning Power 6

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

- If a class C0 is `owl:oneOf` a list of classes C1, C2, and C3, then each of C1, C2, and C3 has `rdf:type` given by C0.
- If a class A is `owl:equivalentClass` to class B, then an `owl:sameAs` relationship will be asserted between these two classes.
- If a class A is `owl:disjointWith` class B, then any sub-class of A will be `owl:disjointWith` with class B.

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

5.3.5 Our Camera Ontology So Far

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

List 5.17 Our current camera ontology, with class definitions only

```
1: <?xml version="1.0"?>
2: <!DOCTYPE rdf:RDF [
3:   <!ENTITY owl "http://www.w3.org/2002/07/owl#" >
4:   <!ENTITY xsd "http://www.w3.org/2001/XMLSchema#" >
```

```

5:      <!ENTITY rdfs "http://www.w3.org/2000/01/rdf-schema#" >
6:      <!ENTITY myCamera "http://www.liyangyu.com/camera#" >
7: ]>
8:
9: <rdf:RDF
10a:   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
11:   xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
12:   xmlns:owl="http://www.w3.org/2002/07/owl#"
13:   xmlns:myCamera="http://www.liyangyu.com/camera#"
14:   xml:base="http://www.liyangyu.com/camera#">
15: <owl:Class rdf:about="&myCamera;Camera">
16:   <rdfs:subClassOf>
17:     <owl:Restriction>
18:       <owl:onProperty rdf:resource="&myCamera;model" />
19:       <owl:minCardinality
19a:         rdf:datatype="&xsd;nonNegativeInteger">
20:         1
21:       </owl:minCardinality>
22:     </owl:Restriction>
23:   </rdfs:subClassOf>
24: </owl:Class>
25:
26: <owl:Class rdf:about="&myCamera;Lens">
27: </owl:Class>
28:
29: <owl:Class rdf:about="&myCamera;Body">
30: </owl:Class>
31:
32: <owl:Class rdf:about="&myCamera;ValueRange">
33: </owl:Class>
34:
35: <owl:Class rdf:about="&myCamera;Digital">
36:   <rdfs:subClassOf rdf:resource="&myCamera;Camera" />
37:   <rdfs:subClassOf>
38:     <owl:Restriction>
39:       <owl:onProperty
39a:         rdf:resource="&myCamera;effectivePixel" />
40:       <owl:cardinality
40a:         rdf:datatype="&xsd;nonNegativeInteger">
41:         1
42:       </owl:cardinality>
43:     </owl:Restriction>
44:   </rdfs:subClassOf>
45: </owl:Class>
46:
47: <owl:Class rdf:about="&myCamera;Film">
48:   <rdfs:subClassOf rdf:resource="&myCamera;Camera" />
49: </owl:Class>
50:
51: <owl:Class rdf:about="&myCamera;DSLR">
52:   <rdfs:subClassOf rdf:resource="&myCamera;Digital" />

```

```

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

```

5.3.6 Define Properties: The Basics

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

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

```

rdfs:domain
rdfs:range
rdfs:subPropertyOf

```

With only three terms, a given application already shows impressive reasoning power. And more importantly, as we have seen in [Chap. 4](#), most of the reasoning power comes from the understanding of the properties by the application.

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

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

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

- `owl:ObjectProperty` is used to connect a resource to another resource;
- `owl:DatatypeProperty` is used to connect a resource to an `rdfs:Literal` (un-typed) or an XML Schema built-in datatype (typed) value.

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

List 5.18 Definitions of `owned_by` and `model` property, as shown in [List 4.13](#)

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

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

In OWL 1, these definitions will look like the ones shown in [List 5.19](#).

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

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

```

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

```

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

List 5.20 Our camera ontology defined in OWL 1

```

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

```

```
31:
32: <owl:Class rdf:about="&myCamera;ValueRange">
33: </owl:Class>
34:
35: <owl:Class rdf:about="&myCamera;Digital">
36:   <rdfs:subClassOf rdf:resource="&myCamera;Camera" />
37:   <rdfs:subClassOf>
38:     <owl:Restriction>
39:       <owl:onProperty
39a:         rdf:resource="&myCamera;effectivePixel" />
40:       <owl:cardinality
40a:         rdf:datatype="&xsd;nonNegativeInteger">
41:         1
42:       </owl:cardinality>
43:     </owl:Restriction>
44:   </rdfs:subClassOf>
45: </owl:Class>
46:
47: <owl:Class rdf:about="&myCamera;Film">
48:   <rdfs:subClassOf rdf:resource="&myCamera;Camera" />
49: </owl:Class>
50:
51: <owl:Class rdf:about="&myCamera;DSLR">
52:   <rdfs:subClassOf rdf:resource="&myCamera;Digital" />
53: </owl:Class>
54:
55: <owl:Class rdf:about="&myCamera;PointAndShoot">
56:   <rdfs:subClassOf rdf:resource="&myCamera;Digital" />
57: </owl:Class>
58:
59: <owl:Class rdf:about="&myCamera;Photographer">
60:   <rdfs:subClassOf
60a:     rdf:resource="http://xmlns.com/foaf/0.1/Person" />
61: </owl:Class>
62:
63: <owl:Class rdf:about="&myCamera;Professional">
64:   <rdfs:subClassOf rdf:resource="&myCamera;Photographer" />
65: </owl:Class>
66:
67: <owl:Class rdf:about="&myCamera;Amateur">
68:   <owl:intersectionOf rdf:parseType="Collection">
69:     <owl:Class
69a:       rdf:about="http://xmlns.com/foaf/0.1/Person" />
70:     <owl:Class>
71:       <owl:complementOf
71a:         rdf:resource="&myCamera;Professional" />
72:       </owl:Class>
73:     </owl:intersectionOf>
74:   </owl:Class>
75:
76: <owl:Class rdf:about="&myCamera;ExpensiveDSLR">
```

```
77:   <rdfs:subClassOf rdf:resource="&myCamera;DSLR" />
78:   <rdfs:subClassOf>
79:     <owl:Restriction>
80:       <owl:onProperty rdf:resource="&myCamera;owned_by" />
81:       <owl:someValuesFrom
81a:         rdf:resource="&myCamera;Professional" />
82:     </owl:Restriction>
83:   </rdfs:subClassOf>
84: </owl:Class>
85:
86: <owl:ObjectProperty rdf:about="&myCamera;owned_by">
87:   <rdfs:domain rdf:resource="&myCamera;DSLR" />
88:   <rdfs:range rdf:resource="&myCamera;Photographer" />
89: </owl:ObjectProperty>
90:
91: <owl:ObjectProperty rdf:about="&myCamera;manufactured_by">
92:   <rdfs:domain rdf:resource="&myCamera;Camera" />
93: </owl:ObjectProperty>
94:
95: <owl:ObjectProperty rdf:about="&myCamera;body">
96:   <rdfs:domain rdf:resource="&myCamera;Camera" />
97:   <rdfs:range rdf:resource="&myCamera;Body" />
98: </owl:ObjectProperty>
99:
100: <owl:ObjectProperty rdf:about="&myCamera;lens">
101:   <rdfs:domain rdf:resource="&myCamera;Camera" />
102:   <rdfs:range rdf:resource="&myCamera;Lens" />
103: </owl:ObjectProperty>
104:
105: <owl:DatatypeProperty rdf:about="&myCamera;model">
106:   <rdfs:domain rdf:resource="&myCamera;Camera" />
107:   <rdfs:range rdf:resource="&xsd:string" />
108: </owl:DatatypeProperty>
109: <rdfs:Datatype rdf:about="&xsd:string" />
110:
111: <owl:ObjectProperty rdf:about="&myCamera;effectivePixel">
112:   <rdfs:domain rdf:resource="&myCamera;Digital" />
113:   <rdfs:range rdf:resource="&myCamera;MegaPixel" />
114: </owl:ObjectProperty>
115: <rdfs:Datatype rdf:about="&myCamera;MegaPixel">
116:   <rdfs:subClassOf rdf:resource="&xsd;decimal" />
117: </rdfs:Datatype>
118:
119: <owl:ObjectProperty rdf:about="&myCamera;shutterSpeed">
120:   <rdfs:domain rdf:resource="&myCamera;Body" />
121:   <rdfs:range rdf:resource="&myCamera;ValueRange" />
122: </owl:ObjectProperty>
123:
124: <owl:DatatypeProperty rdf:about="&myCamera;focalLength">
125:   <rdfs:domain rdf:resource="&myCamera;Lens" />
126:   <rdfs:range rdf:resource="&xsd:string" />
```

```

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

```

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

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

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

5.3.7 Defining Properties: Property Characteristics

5.3.7.1 Symmetric Properties

A symmetric property describes the situation where if a resource R1 is connected to resource R2 by property P, then resource R2 is also connected to resource R1 by the same property. For instance, we can define a property `friend_with` for `Photographer` class, and if photographer A is `friend_with` photographer B, photographer B is certainly `friend_with` photographer A. This is shown in List 5.21.

List 5.21 Example of symmetric property

```

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

```

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

Note that List 5.21 does have a simpler form, as shown in List 5.22.

List 5.22 Example of symmetric property using a simpler form

```

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

```

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

Note that `owl:SymmetricProperty` is a sub-class of `owl:ObjectProperty`. Therefore, `rdfs:range` of a symmetric property can only be a resource and cannot be a literal or datatype.

5.3.7.2 Enhanced Reasoning Power 7

Our application sees the following instance document:

```

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

```

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

```

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

```

```

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

```

5.3.7.3 Transitive Properties

A transitive property describes the situation where if a resource R_1 is connected to resource R_2 by property P , and resource R_2 is connected to resource R_3 by the same property, then resource R_1 is also connected to resource R_3 by property P .

This can be a very useful feature in some cases. For example, photography is a fairly expensive hobby for most of us, and which camera to buy depends on which one can offer a better ratio of quality over price. Therefore, even if a given camera is very expensive, since it can provide excellent quality and performance, the ratio could be still high. On the other hand, a point-and-shoot camera has a very appealing price but it may not offer you that much room to discover your creative side, therefore may not have a high ratio at all.

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

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

List 5.23 Example of transitive property

```

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

```

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

5.3.7.4 Enhanced Reasoning Power 8

Our application collects the following statement from one instance document:

```

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

```

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

```

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

```

```

        "http://www.liyangyu.com/camera#Nikon_D40"/>
    </betterQPRatio>
</DSLR>

```

our application will add the following statement:

```

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

```

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

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

5.3.7.5 Functional Properties

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

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

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

List 5.24 Example of functional property

```

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

```

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

List 5.25 Definitions of Digital class and effectivePixel property

```

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

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

```

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

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

```

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

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

```

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

Note that `owl:FunctionalProperty` is a sub-class of `rdf:Property`. Therefore, `rdfs:range` of a functional property can be a resource, a literal, or a datatype.

5.3.7.6 Enhanced Reasoning Power 9

Our application collects the following statement from one instance document:

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

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

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

our application will add the following statement:

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

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

Note that the URI <http://dbpedia.org/resource/Nikon>, coined by DBpedia, represents Nikon Corporation. Similarly, the following URI is created by freebase⁸ to represent the same company,

<http://www.freebase.com/view/en/nikon>

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

5.3.7.7 Inverse Property

An inverse property describes the situation where if a resource `R1` is connected to resource `R2` by property `P`, then the inverse property of `P` will connect resource `R2` to resource `R1`.

⁸<http://www.freebase.com/>

A good example in our camera ontology is the property `owned_by`. Clearly, if a camera is `owned_by` a Photographer, then we can define an inverse property of `owned_by`, say, `own`, to indicate that the Photographer own the camera. This example is given in List 5.27.

List 5.27 Example of inverse property

```

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

```

Note that compared to the definition of property `owned_by`, property `own`'s values for `rdfs:domain` and `rdfs:range` are flipped from that in `owned_by`.

Note the fact that `owl:inverseOf` is a property, not a class (recall that `owl:FunctionalProperty` is a sub-class of `rdf:Property`). Therefore, it cannot be used to connect any `rdfs:domain` to any `rdfs:range`; it is only used as a constraint when some other property is being defined, as shown in List 5.27.

5.3.7.8 Enhanced Reasoning Power 10

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

```

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

```

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

```

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

```

5.3.7.9 Inverse Functional Property

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

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

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

List 5.28 Example of inverse functional property

```

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

```

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

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

```

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

```

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

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

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

Finally, note that `owl:InverseFunctionalProperty` is a sub-class of `rdf:Property`. Therefore, `rdfs:range` of an inverse functional property can be a resource, a literal, or a datatype.

5.3.7.10 Enhanced Reasoning Power 11

Our application collects the following statement from one instance document:

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

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

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

our application will add the following statement:

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

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

5.3.8 Camera Ontology Written Using OWL 1

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

List 5.30 Camera ontology written in OWL 1

```
1: <?xml version="1.0"?>
2: <!DOCTYPE rdf:RDF [
3:   <!ENTITY owl "http://www.w3.org/2002/07/owl#" >
4:   <!ENTITY xsd "http://www.w3.org/2001/XMLSchema#" >
5:   <!ENTITY rdfs "http://www.w3.org/2000/01/rdf-schema#" >
```



```

6:      <!ENTITY myCamera "http://www.liyangyu.com/camera#" >
7: ]>
8:
9: <rdf:RDF
9a:   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
10:     xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
11:     xmlns:owl="http://www.w3.org/2002/07/owl#"
12:     xmlns:myCamera="http://www.liyangyu.com/camera#"
13:     xml:base="http://www.liyangyu.com/camera#">
14:
15:   <owl:Class rdf:about="&myCamera;Camera">
16:     <rdfs:subClassOf>
17:       <owl:Restriction>
18:         <owl:onProperty rdf:resource="&myCamera;model" />
19:         <owl:minCardinality
20:           rdf:datatype="&xsd;nonNegativeInteger">
21:           1
22:         </owl:minCardinality>
23:       </owl:Restriction>
24:     </rdfs:subClassOf>
25:   </owl:Class>
26:
27:   <owl:Class rdf:about="&myCamera;Lens">
28:   </owl:Class>
29:
30:   <owl:Class rdf:about="&myCamera;Body">
31:   </owl:Class>
32:
33:   <owl:Class rdf:about="&myCamera;ValueRange">
34:   </owl:Class>
35:
36:   <owl:Class rdf:about="&myCamera;Digital">
37:     <rdfs:subClassOf rdf:resource="&myCamera;Camera" />
38:     <rdfs:subClassOf>
39:       <owl:Restriction>
40:         <owl:onProperty
40a:           rdf:resource="&myCamera;effectivePixel" />
41:         <owl:cardinality
42:           rdf:datatype="&xsd;nonNegativeInteger">
43:           1
44:         </owl:cardinality>
45:       </owl:Restriction>
46:     </rdfs:subClassOf>
47:   </owl:Class>
48:
49:   <owl:Class rdf:about="&myCamera;Film">
50:     <rdfs:subClassOf rdf:resource="&myCamera;Camera" />
51:   </owl:Class>
52:
53:   <owl:Class rdf:about="&myCamera;DSLR">
54:     <rdfs:subClassOf rdf:resource="&myCamera;Digital" />

```

```

55: </owl:Class>
56:
57: <owl:Class rdf:about="&myCamera;PointAndShoot">
58:   <rdfs:subClassOf rdf:resource="&myCamera;Digital" />
59: </owl:Class>
60:
61: <owl:Class rdf:about="&myCamera;Photographer">
62:   <rdfs:subClassOf
62a:     rdf:resource="http://xmlns.com/foaf/0.1/Person" />
63: </owl:Class>
64:
65: <owl:Class rdf:about="&myCamera;Professional">
66:   <rdfs:subClassOf rdf:resource="&myCamera;Photographer" />
67: </owl:Class>
68:
69: <owl:Class rdf:about="&myCamera;Amateur">
70:   <owl:intersectionOf rdf:parseType="Collection">
71:     <owl:Class
71a:       rdf:about="http://xmlns.com/foaf/0.1/Person" />
72:     <owl:Class>
73:       <owl:complementOf
73a:         rdf:resource="&myCamera;Professional" />
74:       </owl:Class>
75:     </owl:intersectionOf>
76:   </owl:Class>
77:
78: <owl:Class rdf:about="&myCamera;ExpensiveDSLR">
79:   <rdfs:subClassOf rdf:resource="&myCamera;DSLR" />
80:   <rdfs:subClassOf>
81:     <owl:Restriction>
82:       <owl:onProperty rdf:resource="&myCamera;owned_by" />
83:       <owl:someValuesFrom
83a:         rdf:resource="&myCamera;Professional" />
84:       </owl:Restriction>
85:     </rdfs:subClassOf>
86:   </owl:Class>
87:
88: <owl:ObjectProperty rdf:about="&myCamera;owned_by">
89:   <rdfs:domain rdf:resource="&myCamera;DSLR" />
90:   <rdfs:range rdf:resource="&myCamera;Photographer" />
91: </owl:ObjectProperty>
92:
93: <owl:ObjectProperty rdf:about="&myCamera;manufactured_by">
94:   <rdf:type rdf:resource="&owl;FunctionalProperty" />
95:   <rdfs:domain rdf:resource="&myCamera;Camera" />
96: </owl:ObjectProperty>
97:
98: <owl:ObjectProperty rdf:about="&myCamera;body">
99:   <rdfs:domain rdf:resource="&myCamera;Camera" />
100:  <rdfs:range rdf:resource="&myCamera;Body" />
101: </owl:ObjectProperty>

```

```
102:
103: <owl:ObjectProperty rdf:about="&myCamera;lens">
104:   <rdfs:domain rdf:resource="&myCamera;Camera"/>
105:   <rdfs:range rdf:resource="&myCamera;Lens"/>
106: </owl:ObjectProperty>
107:
108: <owl:DatatypeProperty rdf:about="&myCamera;model">
109:   <rdfs:domain rdf:resource="&myCamera;Camera"/>
110:   <rdfs:range rdf:resource="&xsd:string"/>
111: </owl:DatatypeProperty>
112: <rdfs:Datatype rdf:about="&xsd:string"/>
113:
114: <owl:ObjectProperty rdf:about="&myCamera;effectivePixel">
115:   <rdfs:domain rdf:resource="&myCamera;Digital"/>
116:   <rdfs:range rdf:resource="&myCamera;MegaPixel"/>
117: </owl:ObjectProperty>
118: <rdfs:Datatype rdf:about="&myCamera;MegaPixel">
119:   <rdfs:subClassOf rdf:resource="&xsd;decimal"/>
120: </rdfs:Datatype>
121:
122: <owl:ObjectProperty rdf:about="&myCamera;shutterSpeed">
123:   <rdfs:domain rdf:resource="&myCamera;Body"/>
124:   <rdfs:range rdf:resource="&myCamera;ValueRange"/>
125: </owl:ObjectProperty>
126:
127: <owl:DatatypeProperty rdf:about="&myCamera;focalLength">
128:   <rdfs:domain rdf:resource="&myCamera;Lens"/>
129:   <rdfs:range rdf:resource="&xsd:string"/>
130: </owl:DatatypeProperty>
131: <rdfs:Datatype rdf:about="&xsd:string"/>
132:
133: <owl:ObjectProperty rdf:about="&myCamera;aperture">
134:   <rdfs:domain rdf:resource="&myCamera;Lens"/>
135:   <rdfs:range rdf:resource="&myCamera;ValueRange"/>
136: </owl:ObjectProperty>
137:
138: <owl:DatatypeProperty rdf:about="&myCamera;minValue">
139:   <rdfs:domain rdf:resource="&myCamera;ValueRange"/>
140:   <rdfs:range rdf:resource="&xsd;float"/>
141: </owl:DatatypeProperty>
142: <rdfs:Datatype rdf:about="&xsd;float"/>
143:
144: <owl:DatatypeProperty rdf:about="&myCamera;maxValue">
145:   <rdfs:domain rdf:resource="&myCamera;ValueRange"/>
146:   <rdfs:range rdf:resource="&xsd;float"/>
147: </owl:DatatypeProperty>
148: <rdfs:Datatype rdf:about="&xsd;float"/>
149:
150: <owl:ObjectProperty rdf:about="&myCamera;own">
151:   <owl:inverseOf rdf:resource="&myCamera;owned_by"/>
152:   <rdfs:domain rdf:resource="&myCamera;Photographer"/>
```

```

153:   <rdfs:range rdf:resource="&myCamera;DSLR" />
154: </owl:ObjectProperty>
155:
156: <owl:DatatypeProperty rdf:about="&myCamera;reviewerID">
157:   <rdf:type rdf:resource="&owl;FunctionalProperty" />
158: <rdf:type rdf:resource="&owl;InverseFunctionalProperty" />
159:   <rdfs:domain rdf:resource="&myCamera;Photographer" />
160:   <rdfs:range rdf:resource="&xsd:string" />
161: </owl:DatatypeProperty>
162: <rdfs:Datatype rdf:about="&xsd:string" />
163:
164: </rdf:RDF>

```

5.4 OWL 2 Web Ontology Language

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

5.4.1 What Is New in OWL 2?

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

1. Syntactic sugar to make some common statements easier to construct

The reason why these features are called syntactic sugar is because these new constructs do not alter the reasoning process built upon the ontology that uses these constructs, they are there to make the language easier to use. You will see more details in the next few sections.
2. New constructs that improve expressiveness

These features will indeed increase the expressiveness. Examples include a collection of new properties, such as reflexive property, irreflexive property, and asymmetric property, just to name a few. Also, the new qualified cardinality constraints will greatly enhance the expressiveness of the language, so will the new features such as property chains and keys.
3. Extended support for datatypes

This includes more built-in datatypes being offered by OWL 2. In addition, OWL 2 allows user to define their own datatypes when creating ontologies. As you will see in later sections, these features can be very powerful to use.
4. Simple metamodeling capabilities and extended annotation capabilities

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

5. New sub-languages: the profiles

Another feature offered by OWL 2 is its sub-languages, namely, OWL 2 EL, OWL 2 QL and OWL 2 PL. These language profiles offer different level of tradeoff between expressiveness and efficiency, and therefore offer more choices to the users.

5.4.2 New Constructs for Common Patterns

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

5.4.2.1 Common Pattern: Disjointness

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

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

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

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

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

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

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

List 5.32 Example of using `owl:AllDisjointClasses`

```

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

```

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

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

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

If we do this by using only OWL 1 terms, we can first use `owl:unionOf` to include all the three classes, and then we can use three pair-wise disjoint statements to make the distinction clear. However, this solution is not as concise as the one shown in List 5.33, which uses OWL 2's `owl:disjointUnionOf` operator.

List 5.33 Example of using `owl:disjointUnionOf` operator

```

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

```

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

5.4.2.2 Common Pattern: Negative Assertions

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

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

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

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

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

List 5.34 Example of using `owl:NegativePropertyAssertion`

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

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

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

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

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

List 5.35 Example of using `owl:NegativeDataPropertyAssertion`

```
1: <owl:NegativeDataPropertyAssertion>
2:   <owl:sourceIndividual
2a:     rdf:resource="http://dbpedia.org/resource/Nikon_D300"/>
3:   <owl:assertionProperty rdf:resource="effectivePixel"/>
4:   <owl:targetValue
4a:     rdf:datatype="http://www.liyangyu.com/camera#MegaPixel">
5:     10
6:   </owl:targetValue>
7: </owl:NegativeDataPropertyAssertion>
```

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

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

5.4.3 *Improved Expressiveness for Properties*

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

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

5.4.3.1 **Property Self-Restriction**

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

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

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

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

List 5.36 Example of `owl:hasSelf`

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

It expresses the idea that all threads can create threads.

5.4.3.2 Property Self-Restriction: Enhanced Reasoning Power 12

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

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

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

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

5.4.3.3 Property Cardinality Restrictions

Let us go back to List 5.30 and take a look at the definition of `Professional` class. Now, instead of simply saying it is a sub-class of `Photographer`, we would like to say that any object of `Professional` photographer should own at least one DSLR camera. This can be done by using terms from OWL 1 vocabulary, as shown in List 5.37.

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

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

```

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

```

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

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

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

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

```

owl:minQualifiedCardinality
owl:maxQualifiedCardinality
owl:qualifiedCardinality

```

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

List 5.38 Example of using `owl:minQualifiedCardinality` constraint

```

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

```

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

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

5.4.3.4 Property Cardinality Restrictions: Enhanced Reasoning Power 13

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

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

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

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

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

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

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

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

Being reflexive in real life is quite common. For example, any set is a subset of its own. Also, in a given ontology, every class is its own sub-class. If you use a

reasoner on our camera ontology given in List 5.30 (we will see how to do this in later chapters), you can see statement like the following:

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

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

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

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

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

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

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

Again, `owl:IrreflexiveProperty` is a sub-class of `owl:ObjectProperty`. Therefore, `rdfs:range` of an irreflexive property can only be a resource and cannot be a literal or datatype.

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

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

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

List 5.39 Example of using `owl:AsymmetricProperty`

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

Again, `owl:AsymmetricProperty` is a sub-class of `owl:ObjectProperty`. Therefore, `rdfs:range` of an asymmetric property can only be a resource, rather than a literal or datatype.

5.4.3.6 More About Property Characteristics: Enhanced Reasoning Power 14

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

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

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

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

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

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

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

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

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

5.4.3.7 Disjoint Properties

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

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

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

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

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

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

we know the following statement should not exist:

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

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

List 5.40 Example of using `owl:propertyDisjointWith`

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

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

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

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

Finally, note that `owl:AllDisjointProperties` can be used on datatype properties with the same syntax as shown above.

5.4.3.8 Disjoint Properties: Enhanced Reasoning Power 15

The benefits of disjoint properties are again related to the open world assumption, and these properties can help us to eliminate unknowns. For example, given

the definition of `owned_by` property as shown in List 5.40 and the following statement,

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

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

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

5.4.3.9 Property Chains

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

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

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

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

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

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

How can we make our application understand this fact? What we could have done is to define a new property, `example:hasUncle`, as follows:

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

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

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

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

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

List 5.41 Example of using owl:propertyChainAxiom

```

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

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

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

```

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

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

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

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

```

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

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

```

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

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

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

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

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

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

With the knowledge about property chains, we need to think carefully when defining properties. It is always good to consider the choices between defining it as a simple property or using property chain for the property. Using property chain will make our ontology more expressive and powerful when inferences are made.

Finally, note that property chain is only used on object properties, not on datatype properties.

5.4.3.10 Property Chains: Enhanced Reasoning Power 16

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

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

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

it will add the following statement automatically:

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

5.4.3.11 Keys

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

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

List 5.43 Example of using `owl:hasKey`

```
<owl:Class rdf:about="&myCamera;Photographer">
  <owl:intersectionOf rdf:parseType="Collection">
```

```

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

```

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

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

5.4.3.12 Keys: Enhanced Reasoning Power 17

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

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

5.4.4 Extended Support for Datatypes

As we know, OWL 1 depends on XML Schema (represented by `xsd:` prefix) for its built-in datatypes, and it has been working well in general. However, with more experience gained from ontology development in practice, some further requirements about datatypes have been identified. These new requirements can be summarized as follows:

- a wider range of supported datatypes is needed;
- the capability of adding constraints on datatypes should be supported; and
- the capability of creating new user-defined datatypes is also required.

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

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

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

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

5.4.4.2 Restrictions on Datatypes and User-Defined Datatypes

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

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

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

Table 5.1 Datatypes supported by OWL 2

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

Table 5.2 Two new built-in datatypes of OWL 2

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

⁹<http://www.w3.org/TR/2009/REC-owl2-syntax-20091027/>

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

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

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

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

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

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

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

With this definition, `AdultAge`, as a user-defined datatype, can be used as the `rdfs:range` value for some property, like any other built-in datatype. To make this more interesting, we can use another facet to add an upper bound, therefore creating another new datatype called `PersonAge`, as shown in List 5.45.

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

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

```

    <xsd:maxInclusive
      rdf:datatype="&xsd;integer">150</xsd:maxInclusive>
  </rdf:Description>
</owl:withRestrictions>
</rdfs:Datatype>

```

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

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

```

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

```

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

5.4.4.3 Data Range Combinations

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

```

owl:datatypeComplementOf
owl:intersectionOf
owl:unionOf

```

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

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

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

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

Therefore, a person who has a `MinorAge` will be younger than 18 years.

5.4.5 Punning and Annotations

5.4.5.1 Understanding Punning

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

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

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

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

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

```
http://www.liyangyu.com/camera#Nikon_D300
```

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

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

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

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

There are also some restrictions in OWL 2 about punning:

- one IRI cannot denote both a datatype property and an object property, also
- one IRI cannot be used for both a class and a datatype.

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

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

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

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

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

5.4.5.2 OWL Annotations, Axioms About Annotation Properties

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

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

Note that OWL 1 annotation simply associates property–value pairs to ontology entities, or to the entire ontology itself. This information is merely for human eyes and is not part of the semantics of the ontology, and will therefore be ignored by

Table 5.3 OWL 1's annotation properties

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

most reasoning engines. The commonly used annotation constructs offered by OWL 1 is summarized in Table 5.3.

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

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

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

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

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

```
1: <?xml version="1.0"?>
2: <rdf:RDF
2a:   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
3:   xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
4:   xmlns:owl="http://www.w3.org/2002/07/owl#"
5:   xmlns:dc="http://www.purl.org/metadata/dublin-core#"
6:   xmlns:myCamera="http://www.liyangyu.com/camera#"
7:   xml:base="http://www.liyangyu.com/camera#"
8:
9:   <owl:AnnotationProperty rdf:about=
9a:     "http://www.purl.org/metadata/dublin-core#date">
10: </owl:AnnotationProperty>
11:
12: <owl:Class rdf:about="http://www.liyangyu.com/camera#Lens">
13:   <dc:date rdf:datatype=
13a:     "http://www.w3.org/2001/XMLSchema#date">
13b:     2009-09-10</dc:date>
14: </owl:Class>
```


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

With the understanding about annotations in OWL 1, let us take a look at what has been offered by OWL 2. An obvious improvement is that OWL 1 did not allow annotations of axioms, but OWL 2 does. As a summary, OWL 2 allows for annotations on ontologies, entities (classes, properties and individuals), anonymous individuals, axioms and also on annotations themselves.

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

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

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

List 5.49 shows some possible annotation on the following axiom:

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

List 5.49 Annotations on a given axiom

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

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

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

If an annotation property's `rdfs:domain` value has been specified, that annotation property can only be used to add annotations to the entities whose type is

the specified type. Similarly, if the `rdfs:range` property of an annotation property has been specified, the added annotation information can only assume values that have the type specified by its `rdfs:range` property. Since these are all quite straightforward, we will skip the examples.

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

5.4.6 Other OWL 2 Features

5.4.6.1 Entity Declarations

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

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

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

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

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

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

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

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

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

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

5.4.6.2 Top and Bottom Properties

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

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

Table 5.4 Top and bottom object/data properties

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

5.4.6.3 Imports and Versioning

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

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

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

List 5.50 Ontology header of our camera ontology

```

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

```

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

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

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

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

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

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

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

```
http://www.liyangyu.com/camera#mdoel
```

and this is exactly what we want to achieve.

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

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

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

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

First off, understand `owl:imports` is a property with class `owl:Ontology` as both its `rdfs:domain` and `rdfs:range` value. It is used to make reference to another OWL ontology that contains definitions, and those definitions will be considered as part of the definitions of the importing ontology. The `rdf:resource` attribute of `owl:imports` specifies the URI (name) of the ontology being imported.

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

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

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

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

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

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

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

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

List 5.52 Ontology header of our camera ontology using `owl:versionIRI`

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

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

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

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

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

With this said, the importing schema is quite simple:

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

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

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

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

5.4.7 OWL Constructs in Instance Documents

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

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

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

List 5.53 Example of using owl:sameAs

```

1: <?xml version="1.0"?>
2: <rdf:RDF
2a:   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
3:   xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
4:   xmlns:owl="http://www.w3.org/2002/07/owl#"
5:   xmlns:myCamera="http://www.liyangyu.com/camera#">
6:
7:   <rdf:Description
7a:     rdf:about="http://www.liyangyu.com/camera#Nikon_D300">
8:     <rdf:type
8a:       rdf:resource="http://www.liyangyu.com/camera#DSLR"/>
9:     <owl:sameAs
9a:       rdf:resource="http://dbpedia.org/resource/Nikon_D300"/>
10:   </rdf:Description>
11:
12: </rdf:RDF>

```

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

```

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

```

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

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

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

```

1: <?xml version="1.0"?>
2: <rdf:RDF
2a:   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
3:   xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
4:   xmlns:owl="http://www.w3.org/2002/07/owl#"
5:   xmlns:myCamera="http://www.liyangyu.com/camera#">
6:
7:   <owl:Class rdf:about=
7a:     "http://www.liyangyu.com/camera#DigitalSingleLensReflex">
8:     <owl:sameAs

```

```

8a:         rdf:resource="http://www.liyangyu.com/camera#DSLR" />
9:   </owl:Class>
10:
11: </rdf:RDF>

```

Note that the definition in List 5.55 is quite different from the one in List 5.54.

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

```

1: <?xml version="1.0"?>
2: <rdf:RDF
3:   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
4:   xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
5:   xmlns:owl="http://www.w3.org/2002/07/owl#"
6:   xmlns:myCamera="http://www.liyangyu.com/camera#">
7:   <owl:Class rdf:about=
8:     "http://www.liyangyu.com/camera#DigitalSingleLensReflex">
9:     <owl:equivalentClass
10:      rdf:resource="http://www.liyangyu.com/camera#DSLR" />
11:   </owl:Class>

```

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

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

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

List 5.56 Example of using `owl:differentFrom`

```

1: <?xml version="1.0"?>
2: <rdf:RDF
3:   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
4:   xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
5:   xmlns:owl="http://www.w3.org/2002/07/owl#"
6:   xmlns:myCamera="http://www.liyangyu.com/camera#">
7:   <rdf:Description
8:     rdf:about="http://www.liyangyu.com/camera#Nikon_D3X">
9:     <rdf:type
10:      rdf:resource="http://www.liyangyu.com/camera#DSLR" />
11:     <owl:differentFrom
12:      rdf:resource="http://www.liyangyu.com/camera#Nikon_D3S" />

```



```
10: </rdf:Description>
11:
12: </rdf:RDF>
```

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

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

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

However, this is not the assumption made by OWL, which actually follows the non-unique-names assumption: even if two individuals (or classes or properties) have different names, they can still be the same individual. This can be derived by inference, or explicitly asserted by using `owl:sameAs`, as shown in List 5.53. The reason why adapting non-unique-names assumption in the world of the Semantic Web is simple because it is the most plausible one to make in the given environment.

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

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

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

```
1: <?xml version="1.0"?>
2: <rdf:RDF
3:   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
4:   xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
5:   xmlns:owl="http://www.w3.org/2002/07/owl#"
6:   xmlns:myCamera="http://www.liyangyu.com/camera#">
7:   <owl:AllDifferent>
8:     <owl:distinctMembers rdf:parseType="Collection">
9:       <myCamera:DSLR
10:        rdf:about="http://www.liyangyu.com/camera#Nikon_D3" />
11:     </owl:distinctMembers>
```

```

10a:      rdf:about="http://www.liyangyu.com/camera#Nikon_D3X" />
11:      <myCamera:DSLR
11a:      rdf:about="http://www.liyangyu.com/camera#Nikon_D3S" />
12:      <myCamera:DSLR
12a:      rdf:about="http://www.liyangyu.com/camera#Nikon_D300S" />
13:      <myCamera:DSLR
13a:      rdf:about="http://www.liyangyu.com/camera#Nikon_D300" />
14:      <myCamera:DSLR
14a:      rdf:about="http://www.liyangyu.com/camera#Nikon_D700" />
15:      </owl:distinctMembers>
16: </owl:AllDifferent>

```

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

5.4.8 OWL 2 Profiles

5.4.8.1 Why We Need All These?

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

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

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

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

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

The reason behind this dual assignment was largely due to the fact that OWL was originally designed to use a notational variant of Description Logic (DL), which has

been extensively investigated in the literature, and its expressiveness and computational properties are well understood. Meanwhile, it was also very important for OWL to be semantically compatible with existing Semantic Web languages such as RDF and RDFS. The semantic differences between DL and RDF made it difficult to satisfy both requirements, and the solution chosen by W3C was to provide two coexisting semantics for OWL, therefore two ways of assigning semantics to a given OWL ontology.

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

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

5.4.8.3 Three Faces of OWL 1

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

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

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

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

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

- every legal OWL 1 Lite feature is a legal OWL 1 DL feature; therefore, every legal OWL 1 Lite ontology is a legal OWL 1 DL ontology;

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

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

- OWL 1 Full

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

- OWL 1 DL

As we have mentioned, OWL 1 DL has the same language feature as OWL 1 Full, but it has restrictions about the ways in which the constructs from OWL 1 and RDF can be used. More specifically, the following rules must be observed when building ontologies:

- No arbitrary combination is allowed: a resource can be only a class, a datatype, a datatype property, an object property, an instance, a data value, and not more than one of these. In other words, a class cannot be at the same time a member of another class (no punning is allowed).
- Restrictions on functional property and inverse functional property: recall these two properties are sub-classes of `rdf:Property`; therefore they can connect resource to resource or resource to value. However, in OWL 1 DL, they can only be used with object property, not datatype property.
- Restriction on transitive property: `owl:cardinality` cannot be used with transitive property or their sub-properties because these sub-properties are transitive properties by implication.
- Restriction on `owl:imports`: if `owl:imports` is used by an OWL 1 DL ontology to import an OWL 1 Full ontology, the importing ontology will not be qualified as an OWL 1 DL.

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

- Object properties, datatype properties, annotation properties, and ontology properties must be mutually disjoint. For example, a property cannot be at the same time a datatype property and an annotation property.

- Annotation properties must not be used in property axioms. In other words, no sub-properties or domain/range constraints for annotation properties can be defined.
 - Annotation properties must be explicitly declared, as shown in List 5.48.
 - The object of an annotation property must be either a data literal, a URI reference, or an individual; nothing else is permitted.
- OWL 1 Lite

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

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

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

Recall that List 5.30 is an ontology written only by using OWL 1 features. Let us decide what face this ontology has. Clearly, it is not OWL 1 Lite since we did use `owl:hasValue`, and it is also not OWL 1 DL since we also used functional property on `owl:DatatypeProperty`. Therefore, our camera ontology shown in List 5.30 is an OWL 1 Full version ontology.

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

<http://www.mygrid.org.uk/OWL/Validator>

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

5.4.8.4 Understanding OWL 2 Profiles

Recall we have mentioned that for any ontology created by using OWL 1, we had two alternative ways to assign semantics to this ontology. The same situation still holds for OWL 2. In addition, the first method is still called the Direct Model-Theoretic Semantics, and it is specified by the W3C Recommendation OWL 2 Web Ontology Language Direct Semantics.¹⁰ The second one is again called

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

RDF-based Semantics, and it is specified by the W3C Recommendation OWL 2 Web Ontology Language RDF-Based Semantics.¹¹ Also, *OWL 2 DL* refers to those OWL 2 ontologies interpreted by using the Direct Semantics and *OWL 2 Full* refers to those ontologies interpreted by using the RDF-based Semantics. Another way to understand this is to consider the fact that the Direct Model-Theoretic Semantics assigns meaning to OWL 2 ontologies by using Description Logic, therefore the name OWL 2 DL.

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

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

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

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

In addition to OWL 2 DL, OWL 2 further specifies language profiles. An OWL 2 *profile* is a trimmed down version of the OWL 2 language that trades some expressive power for efficiency of reasoning. In computational logic, profiles are usually called *fragments* or *sub-languages*.

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

5.4.8.5 OWL 2 EL, QL, and RL

- OWL 2 EL

OWL 2 EL is designed with very large ontologies in mind. For example, life sciences commonly require applications that depend on large ontologies. These

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

¹²<http://www.w3.org/TR/2009/REC-owl2-profiles-20091027/>

ontologies normally have huge number of classes, complex structural descriptions. Classification is the main goal of the related applications. With the restrictions added by OWL 2 EL, the complexity of reasoning algorithms (including query answering algorithms) is known to be worst-case polynomial, therefore these algorithms are often called *PTime-complete* algorithms.

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

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

And these features are not supported by OWL 2 EL:

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

• OWL 2 QL

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

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

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

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

And these features are not supported by OWL 2 QL:

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

• Owl 2 RL

Owl 2 RL is designed for those applications that require scalable reasoning without sacrificing too much expressive power. Therefore, OWL 2 applications that are willing to trade the full expressiveness of the language for efficiency and RDF(S) applications that need some added expressiveness from OWL 2 are all good candidates for this profile. OWL 2 RL can also guarantee polynomial time performance.

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

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

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

Table 5.5 Syntactic restrictions on class expressions in OWL 2 RL

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

5.4.9 Our Camera Ontology in OWL 2

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

List 5.58 Our camera ontology written by using OWL 2 features

```

1: <?xml version="1.0"?>
2: <!DOCTYPE rdf:RDF [
3:     <!ENTITY owl "http://www.w3.org/2002/07/owl#">
4:     <!ENTITY xsd "http://www.w3.org/2001/XMLSchema#">
5:     <!ENTITY rdfs "http://www.w3.org/2000/01/rdf-schema#">
6:     <!ENTITY myCamera "http://www.liyangyu.com/camera#">
7: ]>
8:
9: <rdf:RDF
10:     xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
11:     xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
12:     xmlns:owl="http://www.w3.org/2002/07/owl#"
13:     xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
14:     xmlns:myCamera="http://www.liyangyu.com/camera#"
15:     xml:base="http://www.liyangyu.com/camera#"
16: <owl:Ontology rdf:about="">
17:     <owl:versionIRI>
18:         http://www.liyangyu.com/camera/v1
19:     </owl:versionIRI>
20:     <rdfs:comment>our camera ontology</rdfs:comment>
21: </owl:Ontology>

```

```

22:
23: <owl:Class rdf:about="&myCamera;Camera" />
24: <owl:Class rdf:about="&myCamera;Lens" />
25: <owl:Class rdf:about="&myCamera;Body" />
26: <owl:Class rdf:about="&myCamera;ValueRange" />
27: <owl:Class rdf:about="&myCamera;Digital" />
28: <owl:Class rdf:about="&myCamera;Film" />
29: <owl:Class rdf:about="&myCamera;DSLR" />
30: <owl:Class rdf:about="&myCamera;PointAndShoot" />
31: <owl:Class rdf:about="&myCamera;Photographer" />
32: <owl:Class rdf:about="&myCamera;Professional" />
33: <owl:Class rdf:about="&myCamera;Amateur" />
34: <owl:Class rdf:about="&myCamera;ExpensiveDSLR" />
35:
36: <owl:AsymmetricProperty rdf:about="&myCamera;owned_by" />
37: <owl:ObjectProperty rdf:about="&myCamera;manufactured_by" />
38: <owl:ObjectProperty rdf:about="&myCamera;body" />
39: <owl:ObjectProperty rdf:about="&myCamera;lens" />
40: <owl:DatatypeProperty rdf:about="&myCamera;model" />
41: <owl:ObjectProperty rdf:about="&myCamera;effectivePixel" />
42: <owl:ObjectProperty rdf:about="&myCamera;shutterSpeed" />
43: <owl:DatatypeProperty rdf:about="&myCamera;focalLength" />
44: <owl:ObjectProperty rdf:about="&myCamera;aperture" />
45: <owl:DatatypeProperty rdf:about="&myCamera;minValue" />
46: <owl:DatatypeProperty rdf:about="&myCamera;maxValue" />
47: <owl:ObjectProperty rdf:about="&myCamera;own" />
48: <owl:DatatypeProperty rdf:about="&myCamera;reviewerID" />
49:
50: <rdfs:Datatype rdf:about="&xsd;string" />
51: <rdfs:Datatype rdf:about="&myCamera;MegaPixel" />
52: <rdfs:Datatype rdf:about="&xsd;float" />
53:
54: <owl:Class rdf:about="&myCamera;Camera">
55:   <rdfs:subClassOf>
56:     <owl:Restriction>
57:       <owl:onProperty rdf:resource="&myCamera;model" />
58:       <owl:minCardinality
59:         rdf:datatype="&xsd;nonNegativeInteger">
60:         1
61:       </owl:minCardinality>
62:     </owl:Restriction>
63:   </rdfs:subClassOf>
64: </owl:Class>
65:
66: <owl:Class rdf:about="&myCamera;Lens">
67: </owl:Class>
68:
69: <owl:Class rdf:about="&myCamera;Body">
70: </owl:Class>
71:
72: <owl:Class rdf:about="&myCamera;ValueRange">

```

```
73:   </owl:Class>
74:
75:   <owl:Class rdf:about="&myCamera;Digital">
76:     <rdfs:subClassOf rdf:resource="&myCamera;Camera" />
77:     <rdfs:subClassOf>
78:       <owl:Restriction>
79:         <owl:onProperty
79a:           rdf:resource="&myCamera;effectivePixel" />
80:         <owl:cardinality
81:           rdf:datatype="&xsd;nonNegativeInteger">
82:             1
83:         </owl:cardinality>
84:       </owl:Restriction>
85:     </rdfs:subClassOf>
86:   </owl:Class>
87:
88:   <owl:Class rdf:about="&myCamera;Film">
89:     <rdfs:subClassOf rdf:resource="&myCamera;Camera" />
90:   </owl:Class>
91:
92:   <owl:Class rdf:about="&myCamera;DSLR">
93:     <rdfs:subClassOf rdf:resource="&myCamera;Digital" />
94:   </owl:Class>
95:
96:   <owl:Class rdf:about="&myCamera;PointAndShoot">
97:     <rdfs:subClassOf rdf:resource="&myCamera;Digital" />
98:   </owl:Class>
99:
100:  <owl:Class rdf:about="&myCamera;Photographer">
101:    <owl:intersectionOf rdf:parseType="Collection">
102:      <owl:Class
102a:        rdf:about="http://xmlns.com/foaf/0.1/Person" />
103:      <owl:Class>
104:        <owl:hasKey rdf:parseType="Collection">
105:          <owl:DatatypeProperty
105a:            rdf:about="&myCamera;reviewerID" />
106:          </owl:hasKey>
107:        </owl:Class>
108:      </owl:intersectionOf>
109:    </owl:Class>
110:
111:
112:  <owl:Class rdf:about="&myCamera;Professional">
113:    <rdfs:subClassOf rdf:resource="&myCamera;Photographer" />
114:    <rdfs:subClassOf>
115:      <owl:Restriction>
116:        <owl:minQualifiedCardinality
116a:          rdf:datatype="&xsd;nonNegativeInteger">
117:            1
118:        </owl:minQualifiedCardinality>
119:      <owl:onProperty rdf:resource="&myCamera;own" />
```

```

120:         <owl:onClass rdf:resource="&myCamera;ExpensiveDSLR" />
121:     </owl:Restriction>
122: </rdfs:subClassOf>
123: </owl:Class>
124:
125:
126: <owl:Class rdf:about="&myCamera;Amateur">
127:     <owl:intersectionOf rdf:parseType="Collection">
128:         <owl:Class
128a:             rdf:about="http://xmlns.com/foaf/0.1/Person"/>
129:         <owl:Class>
130:             <owl:complementOf
130a:                 rdf:resource="&myCamera;Professional" />
131:             </owl:Class>
132:         </owl:intersectionOf>
133:     </owl:Class>
134:
135: <owl:Class rdf:about="&myCamera;ExpensiveDSLR">
136:     <rdfs:subClassOf rdf:resource="&myCamera;DSLR" />
137:     <rdfs:subClassOf>
138:         <owl:Restriction>
139:             <owl:onProperty rdf:resource="&myCamera;owned_by" />
140:             <owl:someValuesFrom
140a:                 rdf:resource="&myCamera;Professional" />
141:             </owl:Restriction>
142:         </rdfs:subClassOf>
143:     </owl:Class>
144:
145: <owl:AsymmetricProperty rdf:about="&myCamera;owned_by">
146:     <owl:propertyDisjointWith rdf:resource="&myCamera;own" />
147:     <rdfs:domain rdf:resource="&myCamera;DSLR" />
148:     <rdfs:range rdf:resource="&myCamera;Photographer" />
149: </owl:AsymmetricProperty>
150:
151: <owl:ObjectProperty rdf:about="&myCamera;manufactured_by">
152:     <rdf:type rdf:resource="&owl;FunctionalProperty" />
153:     <rdfs:domain rdf:resource="&myCamera;Camera" />
154: </owl:ObjectProperty>
155:
156: <owl:ObjectProperty rdf:about="&myCamera;body">
157:     <rdfs:domain rdf:resource="&myCamera;Camera" />
158:     <rdfs:range rdf:resource="&myCamera;Body" />
159: </owl:ObjectProperty>
160:
161: <owl:ObjectProperty rdf:about="&myCamera;lens">
162:     <rdfs:domain rdf:resource="&myCamera;Camera" />
163:     <rdfs:range rdf:resource="&myCamera;Lens" />
164: </owl:ObjectProperty>
165:
166: <owl:DatatypeProperty rdf:about="&myCamera;model">
167:     <rdfs:domain rdf:resource="&myCamera;Camera" />

```

```

168:   <rdfs:range rdf:resource="&xsd:string" />
169: </owl:DatatypeProperty>
170: <rdfs:Datatype rdf:about="&xsd:string" />
171:
172: <owl:ObjectProperty rdf:about="&myCamera;effectivePixel">
173:   <rdfs:domain rdf:resource="&myCamera;Digital" />
174:   <rdfs:range rdf:resource="&myCamera;MegaPixel" />
175: </owl:ObjectProperty>
176:
177: <rdfs:Datatype rdf:about="&myCamera;MegaPixel">
178:   <owl:onDatatype rdf:resource="&xsd;integer" />
179:   <owl:withRestrictions rdf:parseType="Collection">
180:     <rdf:Description>
181:       <xsd:minInclusive rdf:datatype="&xsd;decimal">
182:         1.0
183:       </xsd:minInclusive>
184:     </rdf:Description>
185:     <rdf:Description>
186:       <xsd:maxInclusive rdf:datatype="&xsd;decimal">
187:         24.0
188:       </xsd:maxInclusive>
189:     </rdf:Description>
190:   </owl:withRestrictions>
191: </rdfs:Datatype>
192:
193: <owl:ObjectProperty rdf:about="&myCamera;shutterSpeed">
194:   <rdfs:domain rdf:resource="&myCamera;Body" />
195:   <rdfs:range rdf:resource="&myCamera;ValueRange" />
196: </owl:ObjectProperty>
197:
198: <owl:DatatypeProperty rdf:about="&myCamera;focalLength">
199:   <rdfs:domain rdf:resource="&myCamera;Lens" />
200:   <rdfs:range rdf:resource="&xsd:string" />
201: </owl:DatatypeProperty>
202: <rdfs:Datatype rdf:about="&xsd:string" />
203:
204: <owl:ObjectProperty rdf:about="&myCamera;aperture">
205:   <rdfs:domain rdf:resource="&myCamera;Lens" />
206:   <rdfs:range rdf:resource="&myCamera;ValueRange" />
207: </owl:ObjectProperty>
208:
209: <owl:DatatypeProperty rdf:about="&myCamera;minValue">
210:   <rdfs:domain rdf:resource="&myCamera;ValueRange" />
211:   <rdfs:range rdf:resource="&xsd;float" />
212: </owl:DatatypeProperty>
213: <rdfs:Datatype rdf:about="&xsd;float" />
214:
215: <owl:DatatypeProperty rdf:about="&myCamera;maxValue">
216:   <rdfs:domain rdf:resource="&myCamera;ValueRange" />
217:   <rdfs:range rdf:resource="&xsd;float" />
218: </owl:DatatypeProperty>

```

```

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

```

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

5.5 Summary

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

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

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

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

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

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

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

With the material presented in this chapter and [Chap. 4](#), you should be technically sound when it comes to ontology development. In [Chap. 12](#), we will present a methodology that will help you further with ontology design and development. Together, this will prepare you well for your work on the Semantic Web.

Chapter 6

SPARQL: Querying the Semantic Web

This chapter covers SPARQL, the last core component of the Semantic Web. With SPARQL, you will be able to locate specific information on the machine-readable Web, and the Web can therefore be viewed as a gigantic database, as many of us have been dreaming about.

This chapter will cover all the main aspects of SPARQL, including its concepts, its main language constructs and features, and certainly, real-world examples and related tools you can use when querying the Semantic Web. Once you are done with this chapter, you will have a complete tool collection that you can use to continue exploring the world of the Semantic Web.

6.1 SPARQL Overview

6.1.1 SPARQL in Official Language

SPARQL (pronounced “sparkle”) is an RDF query language and data access protocol for the Semantic Web. Its name is a recursive acronym that stands for *SPARQL Protocol and RDF Query Language*. It was standardized by W3C’s SPARQL Working Group (formerly known as the RDF Data Access Working Group) on 15 January 2008. You can follow its activities from the official Web site:

http://www.w3.org/2009/sparql/wiki/Main_Page

which also lists the specifications contained in the official W3C Recommendation.

The W3C Recommendation of SPARQL consists of three separate specifications. The first one *SPARQL Query Language specification*¹ makes up the core. Together with this language specification is the *SPARQL Query XML Results Format specification*² which describes an XML format for serializing the results of a SPARQL query (including both `SELECT` and `ASK` query). The third specification is

¹<http://www.w3.org/TR/rdf-sparql-query/>

²<http://www.w3.org/TR/rdf-sparql-XMLres/>

the *SPARQL Protocol for RDF specification*³ that uses WSDL 2.0 to define simple HTTP and SOAP protocols for remotely querying RDF databases.

In total, therefore, SPARQL Recommendation consists of a query language, a XML format in which query results will be returned, and a protocol of submitting a query to a query processor service remotely.

The main focus of this chapter is on SPARQL query language itself; all of the key language constructs will be covered with ample examples. It will also cover the new features proposed by SPARQL 1.1, which have not been standardized at the time of this writing. Nevertheless, these new features will likely to remain stable and should be very useful for you to become even more productive with SPARQL.

6.1.2 SPARQL in Plain English

At this point, we have learned RDF, a model and data format that we can use to create structured content for machine to read. We have also learned RDF schema and OWL language, and we can use these languages to create ontologies.

With ontologies, anything we say in our RDF documents, we have a reason to say them. And more importantly, since the RDF documents we create all share these common ontologies, it becomes much easier for machines to make inferences based on these RDF contents, therefore generating even more RDF statements, as we have seen in the last chapter.

As a result, there will be more and more content being expressed in RDF format. And indeed, for last several years, a large amount of RDF documents have been published on the Internet and a machine-readable Web has started to take shape. Following all these is the need to locate specific information on this data Web.

A possible solution is to build a new kind of search engine that will work on this emerging Semantic Web. Since the underlying Web is machine readable, this new search engine will have a much better performance than that delivered by the search engines working under traditional Web environment.

However, this solution will not be able to take full advantage of the Semantic Web. More specifically, a search engine does not directly give us answers; instead, it returns to us a collection of pages that might contain the answer. Since we are working a machine-readable Web, why not directly ask for the answer?

Therefore, to push the solution one step further, we need a query language that we can use on this data Web. By simply submitting a query, we should be able to directly get the answer.

SPARQL query language is what we are looking for. In plain English,

SPARQL is a query language that we can use to query the RDF data content and SPARQL also provides a protocol that we need to follow if we want to query a remote RDF data set.

The benefit of having a query language such as SPARQL is also obvious. To name a few,

³<http://www.w3.org/TR/rdf-sparql-protocol/>

- to query RDF graphs to get specific information;
- similarly, to query a remote RDF server and to get streaming results back;
- to run automated regular queries against RDF dataset to generate reports;
- to enable application development at a higher level, i.e., application can work with SPARQL query results, not directly with RDF statements.

6.1.3 Other Related Concepts: RDF Data Store, RDF Database, and Triple Store

RDF data store, RDF database, and triple store are three concepts you will hear a lot when you start to work with SPARQL. In fact, all these three phrases mean exactly the same thing and are interchangeable. To understand them, we only need to understand any one of them. Let us look at RDF data store in more detail.

To put it simple, an *RDF data store* is a special database system built for the storage and retrieval of RDF statements.

As we know, a relational database management system (DBMS) is built for general purpose. Since no one can predict what data model (tables, schemas, etc.) will be needed for a specific project, all the functionalities, such as adding, deleting, updating, and locating a record, have to be built as general as possible. To gain this generality, performance cannot be the top priority at all time.

If we devote a database system to store RDF statements *only*, we know what kind of data model will be stored already. More specifically, every record is a short statement in the form of subject–predicate–object. As a result, we can modify a DBMS so that it is optimized for the storage and retrieval of RDF statements only.

Therefore, an RDF data store is like a relational database in that we can store RDF statements there and retrieve them later by using a query language. However, an RDF data store is specially made and optimized for storing and retrieving RDF statement only.

An RDF data store can be built as a specialized database engine from scratch, or it can be built on top of existing commercial relational database engines. Table 6.1 shows some RDF data stores created by different parties using different languages.

Any given RDF data store, be it built from scratch or on top of an existing commercial data base system, should have the following features:

Table 6.1 Examples of RDF data store implementations

RDF data store name	implementation language	home page
4store	C	http://www.4store.org
ARC	C	http://arc.semsol.org
Joseki	Java	http://www.joseki.org
Redland	C	http://librdf.org
Sesame	Java	http://www.openrdf.org
Virtuoso	C	http://virtuoso.openlinksw.com

- a common storage medium for any application that manipulates RDF content;
- a set of APIs that allow applications to add triples to the store, query the store, and delete triples from the store.

There can be different add-on features provided by different implementations. For example, some RDF data stores will support loading an RDF document from a given URL, and some will support RDF schema, therefore offering some basic forms of inferencing capability. It is up to you to understand the features of the RDF data store that you have on hand.

In this chapter, we will be using an RDF data store called Joseki, and we will help you to set it up in the next section.

6.2 Set up Joseki SPARQL Endpoint

A *SPARQL endpoint* can be understood as an interface that users (human or application) can access to query an RDF data store by using SPARQL query language. Its function is to accept queries and return result accordingly. For human users, this endpoint could be a stand-alone or a Web-based application. For applications, this endpoint takes the form of a set of APIs that can be used by the calling agent.

A SPARQL endpoint can be configured to return results in a number of different formats. For instance, when used by human users in an interactive way, it presents the result in the form of a HTML table, which is often constructed by applying XSL transforms to the XML result that is returned by the endpoint. When accessed by applications, the results are serialized into machine-processable formats, such as RDF/XML or Turtle format, just to name a few.

SPARQL endpoints can be categorized as *generic* endpoints and *specific* endpoints. A generic endpoint works against any RDF dataset, which could be stored locally or accessible from the Web. A specific endpoint is tied to one particular dataset, and this dataset cannot be switched to another endpoint.

In this chapter, our main goal is to learn SPARQL's language feature; we are going to use a SPARQL endpoint that works in a command line fashion so that we can test our queries right away. In later chapters, we will see how to query RDF statements by using programmable SPARQL endpoint in a soft agent.

There are quite a few SPARQL endpoints available and we have selected *Joseki* as our test bed throughout this chapter. Joseki comes with the Jena Semantic Web framework developed by HP, which is arguably the most popular tool suite for developing Semantic Web applications (more details about Jena in later chapters).

More specifically, Joseki is a Web-based SPARQL endpoint. It contains its own Web server, which hosts a Java servlet engine to support its SPARQL endpoint. It renders a Web query form so that we can enter our query manually, and by submitting the query form, we will be presented with the query result right away. Joseki endpoint also provides a set of SPARQL APIs that we can use to submit queries programmatically, as we will see in later chapters.

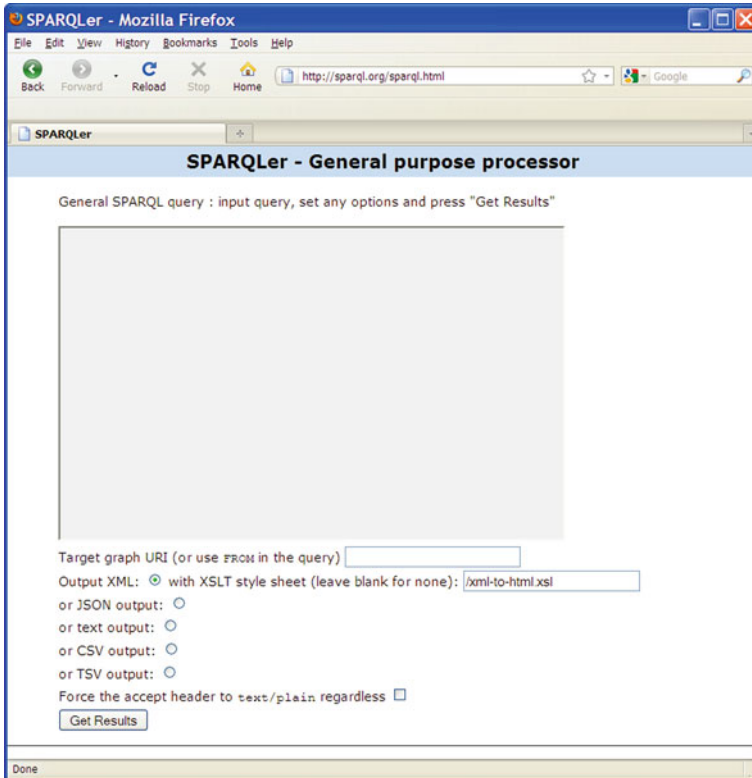


Fig. 6.1 Joseki SPARQL endpoint

There are two ways to access Joseki. The easiest way is to access Joseki endpoint directly online:

`http://sparql.org/sparql.html`

and this endpoint is shown in Fig. 6.1.

You can use the simple text form shown in Fig. 6.1 to enter your query and click Get Results button to see the query result – we will see more details in the coming sections.

Another choice is to download Joseki package and install it on your machine so that you can test SPARQL query anytime you want, even when you are offline. You will be seeing exactly the same interface as shown in Fig. 6.1, except that it is hosted at this location:

`http://localhost:2020/`

We will now discuss how to setup Joseki on your local machine. If you decide to go with online endpoint, you can simply skip the following discussion.

To start, download Joseki from this location

`http://www.joseki.org/download.html`

and remember to get the latest version. Once you have downloaded it, you can install it on your machine, at any location you want. To start using it, you need to finish the following two simple setup steps.

The first step is to set the `JOSEKIROOT` environment variable so that it points to the location of your installation. For example, I have installed Joseki at this location:

```
C:\liyong\DevApp\Joseki-3.2
```

Therefore, I have added the following environment variable:

```
JOSEKIROOT=C:\liyong\DevApp\Joseki-3.2
```

The second step is to update your `CLASSPATH` environment variable and note that every jar file under lib directory (`C:\liyong\DevApp\Joseki-3.2\lib`) has to be added into the `CLASSPATH` variable. Again, use my installation as an example; part of my `CLASSPATH` variable should look like the following:

```
CLASSPATH=.;C:\liyong\DevApp\Joseki-3.2\lib\antlr-2.7.5.jar;
          C:\liyong\DevApp\Joseki-3.2\lib\arq.jar; .....
```

where `.....` represents the other jar files.

Now, you are ready to start Joseki SPARQL endpoint. Go to the following location (using my installation as example):

```
C:\liyong\DevApp\Joseki-3.2
```

type in the following command to start it:

```
bin\rdfserver
```

This should start Joseki server successfully. If everything works fine, you should see a window output that is similar to the one shown in Fig. 6.2.

```
C:\WINDOWS\system32\cmd.exe - bin\rdfserver
C:\liyong\DevApp>cd jo*
C:\liyong\DevApp\Joseki-3.2>bin\rdfserver
17:56:26 INFO Configuration :==== Configuration =====
17:56:26 INFO Configuration : Loading : <joseki-config.ttl>
17:56:27 INFO ServiceInitsImple : Init: Example initializer
17:56:27 INFO Configuration :==== Datasets =====
17:56:27 INFO Configuration : New dataset: Books
17:56:27 INFO Configuration : Default graph : books.n3
17:56:27 INFO Configuration : New dataset: MM
17:56:27 INFO Configuration : Default graph : <<blank nodes>>
17:56:27 INFO Configuration :==== Services =====
17:56:27 INFO Configuration : Service references: "books"
17:56:27 INFO Configuration : Class name: org.joseki.processors.SPARQL
17:56:27 INFO SPARQL : SPARQL processor
17:56:27 INFO SPARQL : Locking policy: multiple reader, single writer
17:56:27 INFO SPARQL : Dataset description: false // Web loading: false
17:56:27 INFO Configuration : Dataset: Books
17:56:27 INFO Configuration : Service reference: "sparql"
17:56:27 INFO Configuration : Class name: org.joseki.processors.SPARQL
17:56:27 INFO SPARQL : SPARQL processor
17:56:27 INFO SPARQL : Locking policy: none
17:56:27 INFO SPARQL : Dataset description: true // Web loading: true
17:56:27 INFO Configuration : Bind services to the server :====
17:56:27 INFO Configuration : Service: <books>
17:56:27 INFO Configuration : Service: <sparql>
17:56:27 INFO Configuration :==== Initialize datasets =====
17:56:28 INFO Configuration :==== End Configuration =====
17:56:28 INFO Dispatcher : Loaded data source configuration: joseki-config.ttl
17:56:28 INFO log : Logging to org.slf4j.impl.Log4jLoggerAdapter@27cd63 via org.mortbay.log.Slf4jLog
17:56:28 INFO log : jetty-6.1.10
17:56:29 INFO log : NO JSP Support for /, did not find org.apache.jasper.servlet.JspServlet
17:56:30 INFO log : Started SelectChannelConnector@0.0.0.0:2020
```

Fig. 6.2 Check if you have installed Joseki correctly

Now open a browser and put the following into the address bar:

```
http://localhost:2020/sparql.html
```

You will see a SPARQL query form that looks exactly like the one shown in Fig. 6.1. For the rest of this chapter, this is what we are going to use for all our SPARQL queries.

6.3 SPARQL Query Language

Now, we are ready to study the query language itself. We need to select an RDF data file first and use it as our example throughout this chapter. This example data file has to be clear enough for us to understand, yet it cannot be too trivial to reflect the power of SPARQL query language.

Let us take Dan Brickley's FOAF document as our example. FOAF represents a project called *Friend Of A Friend*, and Dan Brickley is one of the two founders of this project. We have a whole chapter coming up to discuss FOAF in detail; for now, understanding the following about FOAF is good enough for you to continue:

- The goal of FOAF project is to build a social network using the Semantic Web technology so that we can do experiments with it and build applications that are not easily built under traditional Web.
- The core element of FOAF project is the FOAF ontology, a collection of terms that can be used to describe a person: name, home page, e-mail address, interest, and people he/she knows, etc.
- Anyone can create an RDF document to describe himself/herself by using this FOAF ontology, and he/she can join the friends network as well.

Dan Brickley's FOAF document is therefore a collection of RDF statements that he created to describe himself, and the terms he used to do so come from FOAF ontology. You can find his file at this URL:

```
http://danbri.org/foaf.rdf
```

And since the file is quite long, List 6.1 shows only part of it, so that you can get a feeling about how a FOAF document looks like. Note that Dan Brickley can change his FOAF document at any time, therefore at the time you are reading this book, the exact content of this RDF document could be different. However, the main idea is the same, and all the queries against this file will still be valid.

List 6.1 Part of Dan Brickley's FOAF document

```
1: <?xml version="1.0"?>
2:
3: <rdf:RDF
4:     xml:lang="en"
5:     xmlns:wot="http://xmlns.com/wot/0.1/"
```

```

6:      xmlns:rdf=
6a:          "http://www.w3.org/1999/02/22-rdf-syntax-ns#"
7:      xmlns:dct="http://purl.org/dc/terms/"
8:      xmlns:lang=
8a:          "http://purl.org/net/inkel/rdf/schemas/lang/1.1#"
9:      xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
10:     xmlns:rss="http://purl.org/rss/1.0/"
11:     xmlns="http://xmlns.com/foaf/0.1/"
12:     xmlns:foaf="http://xmlns.com/foaf/0.1/"
13:     xmlns:wn="http://xmlns.com/wordnet/1.6/"
14:     xmlns:air=
14a:         "http://www.megginson.com/exp/ns/airports#"
15:     xmlns:contact=
15a:         "http://www.w3.org/2000/10/swap/pim/contact#"
16:     xmlns:dc="http://purl.org/dc/elements/1.1/">
17:
18: <Person rdf:ID="danbri">
19:
20:   <foaf:name>Dan Brickley</foaf:name>
21:   <foaf:nick>danbri</foaf:nick>
22:
23:   <mbx rdf:resource="mailto:danbri@danbri.org"/>
24:   <mbx rdf:resource="mailto:danbri@porkklips.org"/>
25:
26:   <plan>Save the world and home in time for tea.</plan>
27:
28:   <knows>
29:     <Person>
30:       <mbx rdf:resource=
30a:         "mailto:libby.miller@bristol.ac.uk"/>
31:       <mbx rdf:resource="mailto:libby@asemantics.com"/>
32:     </Person>
33:   </knows>
34:
35:   <knows>
36:     <Person rdf:about=
36a:       "http://www.w3.org/People/Berners-Lee/card#i">
37:       <name>Tim Berners-Lee</name>
38:       <isPrimaryTopicOf rdf:resource=
38a:         "http://en.wikipedia.org/wiki/Tim_Berners-Lee"/>
39:       <homepage rdf:resource=
39a:         "http://www.w3.org/People/Berners-Lee/" />
40:       <mbx rdf:resource="mailto:timbl@w3.org"/>
41:       <rdfs:seeAlso rdf:resource=
41a:         "http://www.w3.org/People/Berners-Lee/card"/>

```

```

42:     </Person>
43:   </knows>
44:
45: </Person>
46:
47: </rdf:RDF>

```

As you can see, he has included his name and nick name (line 20, 21), his e-mail addresses (line 23, 24), and his plan (line 26). He has also used `foaf:knows` to include some of his friends, as shown in line 28–33, line 35–43. Note that a default namespace is declared in line 11, and that default namespace is the FOAF ontology namespace (see next chapter for details). As a result, he can use terms from FOAF ontology without adding any prefix, such as `Person`, `knows`, `mbox`, `plan`.

6.3.1 The Big Picture

SPARQL provides four different forms of query:

- SELECT query
- ASK query
- DESCRIBE query
- CONSTRUCT query

Among these forms, `SELECT` query is the most frequently used query form. In addition, all these query forms are based on two basic SPARQL concepts: triple pattern and graph pattern. Let us understand these two concepts first before we start to look at SPARQL queries.

6.3.1.1 Triple Pattern

As we have learned, RDF model is built on the concept of triple, a three-tuple structure consisting of subject, predicate, and object. Likewise, SPARQL is built upon the concept of *triple pattern*, which is also written as subject, predicate, and object, and has to be terminated with a full stop. The difference between RDF triple and SPARQL triple pattern is that a SPARQL triple pattern can include variables: any or all of the subject, predicate, and object values in a triple pattern can be a variable. Clearly, an RDF triple is also a SPARQL triple pattern.

The second line in the following example is a SPARQL triple pattern (note that the Turtle syntax is used here):

```

@prefix foaf: <http://xmlns.com/foaf/0.1/>.
<http://danbri.org/foaf.rdf#danbri> foaf:name ?name.

```

As you can tell, the subject of this triple pattern is Dan Brickley’s URI, the predicate is `foaf:name`, and the object component of this triple pattern is a variable, identified by the `?` character in front of a string `name`.

Note that a SPARQL variable can be prefixed with either a `?` character or a `$` character, and these two are interchangeable. In this book, we will use the `?` character. In other words, a SPARQL variable is represented by the following format:

```
?variableName
```

where the `?` character is necessary, and `variableName` is given by the user.

The best way to understand a variable in a triple pattern is to view it as a placeholder that can match any value. More specifically, here is what happens when the above triple pattern is used against an RDF graph:

0. Create an empty RDF document, call it `resultSet`.
1. Get the next triple from the given RDF graph; if there is no more triple, return the `resultSet`.
2. Match the current triple with the triple pattern: if both the subject and the predicate of the current triple match the given subject and predicate in the triple pattern, the actual value of the object from the current triple will *bind* to the variable called `name`, therefore creates a new concrete triple that will be collected into the `resultSet`.
3. Go back to step 1.

Obviously, the above triple pattern can be read as follows:

```
find the value of foaf:name property defined for RDF resource identified by http://danbri.org/foaf.rdf#danbri.
```

And based on the above steps, it is clear that all possible bindings are included. Therefore, if we have multiple instances of `foaf:name` property defined for <http://danbri.org/foaf.rdf#danbri>, all these multiple bindings will be returned.

It is certainly fine to have more than one variable in a triple pattern. For example, the following triple pattern has two variables:

```
<http://danbri.org/foaf.rdf#danbri> ?property ?name.
```

And it means to find all the properties and their values that have been defined to the resource identified by <http://danbri.org/foaf.rdf#danbri>.

It is also fine to have all components as variables:

```
?subject ?property ?name.
```

This triple pattern will then match all triples in a given RDF graph.

6.3.1.2 Graph Pattern

Another important concept in SPARQL is called *graph pattern*. Similar to triple pattern, graph pattern is also used to select triples from a given RDF graph, but it can specify a much more complex “selection rule” compared to simple triple pattern.

First off, a collection of triple patterns is called a graph pattern. In SPARQL, { and } are used to specify a collection of triple patterns. For example, the following three triple patterns present one graph pattern:

```
{
  ?who foaf:name ?name.
  ?who foaf:interest ?interest.
  ?who foaf:knows ?others.
}
```

To understand how graph pattern is used to select resources from a given RDF graph, we need to remember one key point about the graph pattern: if a given variable shows up in multiple triple patterns within the graph pattern, its value in all these patterns has to be the same. In other words, each resource returned must be able to substitute into all occurrences of the variable. More specifically,

0. Create an empty set called `resultSet`.
1. Get the next resource from the given RDF graph. If there is no more resource left, return `resultSet` and stop.
2. Process the first triple pattern:
 - If the current resource does not have a property instance called `foaf:name`, go to 6.
 - Otherwise, bind the current resource to variable `?who` and bind the value of property `foaf:name` to variable `?name`.
3. Process the second triple pattern:
 - If the current resource (represented by variable `?who`) does not have a property instance called `foaf:interest`, go to 6.
 - Otherwise, bind the value of property `foaf:interest` to variable `?interest`.
4. Process the third triple pattern:
 - If the current resource (represented by variable `?who`) does not have a property instance called `foaf:knows`, go to 6.
 - Otherwise, bind the value of property `foaf:knows` to variable `?others`.
5. Collect the current resource into `resultSet`.
6. Go to 1.

Based on these steps, it is clear that this graph pattern in fact tries to find any resource that has all three of the desired properties defined. The above process will stop its inspection at any point and move on to a new resource if the current resource does not have any of the required property defined.

You should be able to understand other graph patterns in a similar way just by remembering this basic rule: within a graph pattern, a variable must always be bound to the same value no matter where it shows up.

And now, we are ready to dive into the world of SPARQL query language.

6.3.2 *SELECT* Query

The *SELECT* query form is used to construct standard queries, and it is probably the most popular form among the four. In addition, most of its features are shared by other query forms.

6.3.2.1 Structure of a *SELECT* Query

List 6.2 shows the structure of a SPARQL *SELECT* query:

List 6.2 The structure of a SPARQL *SELECT* query

```
# base directive
BASE <URI>

# list of prefixes
PREFIX pref: <URI>
...

# result description
SELECT...

# graph to search
FROM ...

# query pattern
WHERE {
    ...
}

# query modifiers
ORDER BY...
```

As shown in List 6.2, a *SELECT* query starts with a *BASE* directive and a list of *PREFIX* definitions which may contain an arbitrary number of *PREFIX* statements. These two parts are optional and they are used for URI abbreviations. For example, if you assign a label *pref* to a given URI, then the label can be used anywhere in a query in place of the URI itself. Also note that *pref* is simply a label, we can name it anyway we want. This is all quite similar to Turtle language abbreviation we have discussed in [Chap. 2](#), and we will see more details about these two parts in the upcoming query examples.

The *SELECT* clause comes next. It specifies which variable bindings, or data items, should be returned from the query. As a result, it “picks up” what information to return from the query result.

The *FROM* clause tells the SPARQL endpoint against which graph the search should be conducted. As you will see later, this is also an optional item – in some cases, there is no need to specify the dataset that is being queried against.

The *WHERE* clause contains the graph patterns that specify the desired results; it tells the SPARQL endpoint what to query for in the underlying data graph. Note

that the `WHERE` clause is not optional, although the `WHERE` keyword itself is optional. However, for clarity and readability, it is a good idea not to omit `WHERE`.

The last part is generally called query modifiers. The main purpose is to tell the SPARQL endpoint how to organize the query results. For instance, `ORDER BY` clause and `LIMIT` clause are examples of query modifiers. Obviously, query modifiers are also optional.

6.3.2.2 Writing Basic `SELECT` Query

As we have discussed, our queries will be issued against Dan Brickley’s FOAF document. And our first query will accomplish the following: since FOAF ontology has defined a group of properties that one can use to describe a person, it would be interesting to see which of these properties are actually used by Brickley. List 6.3 shows the query:

List 6.3 What FOAF properties did Dan Brickley use to describe himself?

```
1: base <http://danbri.org/foaf.rdf>
2: prefix foaf: <http://xmlns.com/foaf/0.1/>
3: select *
4: from <http://danbri.org/foaf.rdf>
5: where
6: {
7:   <#danbri> ?property ?value.
8: }
```

Since this is our first query, let us study it in greater detail. First off, note that SPARQL is not case sensitive, so all the keywords can be either in small letters or in capital letters.

Now, lines 1 and 2 are there for abbreviation purpose. Line 1 uses `BASE` keyword to define a base URI against which all relative URIs in the query will be resolved, including the URIs defined with `PREFIX` keyword. In this query, `PREFIX` keyword specifies that `foaf` will be the shortcut for an absolute URI (line 2), so the URI `foaf` stands for does not have to be resolved against the `BASE` URI.

Line 3 specifies which data items should be returned by the query. Note that only variables in the graph pattern (line 6–8) can be chosen as returned data items. In this example, we would like to return both the property names and their values, so we should have written the `SELECT` clause like this:

```
select ?property ?value
```

Since `?property` and `?value` are the only two variables in the graph pattern, we do not have to specify them one by one as shown above, we can simply use a `*` as a wildcard for all the variables, as shown in line 3.

Line 4 specifies the data graph against which we are doing our search. Note that Joseki can be used as either a generic or a specific SPARQL endpoint, and in this chapter, we will always explicitly specify the location of Brickley’s FOAF document.

Line 5 is the `where` keyword, indicating that the search criteria will be the next, and lines 6–8 give the criteria represented by a graph pattern. Since we have discussed the concepts of triple pattern and graph pattern already, we understand how they are used to select the qualified triples. For this particular query, the graph pattern should be quite easy to follow. More specifically, this graph pattern has only one triple pattern, and it tries to match all the property instances and their values that are ever defined for the resource representing Brickley in real life.

Note that the resource representing Brickley has a relative URI as shown in line 7, and it is resolved by concatenating the `BASE` URI with this relative URI. The resolved URI is given as

`http://danbri.org/foaf.rdf#danbri`

which is the one that Brickley has used in his FOAF file.

Now you can put List 6.3 into the query box as shown in Fig. 6.1 and click Get Result button; you should be able to get the results back. Figure 6.3 shows part of the result.

property	value
<http://www.w3.org/2000/01/rdf-schema#seeAlso>	<http://rdfweb.org/people/danbri/rdfweb/webwho.xrd>
<http://xmlns.com/foaf/0.1/mbox>	<mailto:danbri@w3.org>
<http://xmlns.com/wot/0.1/keyid>	"B573B63A" @en
<http://xmlns.com/foaf/0.1/plan>	"Save the world and home in time for tea." @en
<http://xmlns.com/foaf/0.1/knows>	_:b0
<http://xmlns.com/foaf/0.1/knows>	_:b1
<http://xmlns.com/foaf/0.1/knows>	_:b2
<http://www.w3.org/2000/01/rdf-schema#seeAlso>	<http://del.icio.us/rss/danbri>
<http://xmlns.com/foaf/0.1/jabberID>	"danbri@jabber.org" @en
<http://xmlns.com/foaf/0.1/pubkeyAddress>	<http://danbri.org/danbri-pubkey.txt>
<http://xmlns.com/foaf/0.1/holdsAccount>	<http://test.foaf-ssl.org/certs/1240478394168.rdf#acctnt>
<http://xmlns.com/foaf/0.1/knows>	_:b3
<http://xmlns.com/foaf/0.1/knows>	_:b4
<http://xmlns.com/foaf/0.1/knows>	_:b5
<http://xmlns.com/foaf/0.1/mbox>	<mailto:danbri@rdfweb.org>
<http://www.w3.org/2000/10/swap/pim/contact#nearestAirport>	_:b6
<http://xmlns.com/foaf/0.1/knows>	<http://mmt.me.uk/foaf.rdf#mischa>
<http://xmlns.com/foaf/0.1/holdsAccount>	_:b7
<http://www.w3.org/2000/01/rdf-schema#seeAlso>	<http://swordfish.rdfweb.org/discovery/2001/08/codepict/scutterplan.rdf>
<http://purl.org/net/inkel/rdf/schemas/lang/1.1#masters>	"en" @en
<http://kota.s12.xrea.com/vocab/uranaibloodtype>	"A+" @en
<http://xmlns.com/foaf/0.1/img>	<http://www.w3.org/People/DanBri/mugshot1.jpg>
<http://xmlns.com/foaf/0.1/knows>	_:b8
<http://xmlns.com/foaf/0.1/knows>	_:b9

Fig. 6.3 Part of the query result when running the query shown in List 6.3

From the result, we can see which properties have been used. For instance, `foaf:knows` and `foaf:mbox` are the most commonly used ones. Other properties such as `foaf:name`, `foaf:nick`, `foaf:homepage`, `foaf:holdsAccount` are also used.

Note that Brickley's FOAF file could be under constant updation, so at the time you are trying this query, you might not see the same result as shown in Fig. 6.3. Also, we will not continue to show the query results from now on unless it is necessary to do so, so this chapter will not be too long.

Let us try another simple query: find all the people known by Brickley. List 6.4 shows the query:

List 6.4 Find all the people known by Dan Brickley

```
base <http://danbri.org/foaf.rdf>
prefix foaf: <http://xmlns.com/foaf/0.1/>

select *
from <http://danbri.org/foaf.rdf>
where
{
  <#danbri> foaf:knows ?friend.
}
```

Try to run the query, and right away you will see the problem: the query itself is right, but the result does not really tell us anything: there are way too many blank nodes.

In fact, it is quite often that when we include a friend we know in our FOAF documents, instead of using his/her URI, we simply use a blank node to represent this friend. For example, in List 6.1, one friend, Libby Miller, is represented by a blank node. As we will see in Chap. 7, even a blank node is used to represent a friend; as long as `foaf:mbox` property value is also included for this resource, any application will be able to recognize the resource.

Let us change List 6.4 to accomplish the following: find all the people known by Brickley and show their name, e-mail address, and home page information. List 6.5 is the query that can replace the one in List 6.4:

List 6.5 Find all the people known by Brickley, show their names, e-mail addresses, and home page information

```
base <http://danbri.org/foaf.rdf>
prefix foaf: <http://xmlns.com/foaf/0.1/>

select *
from <http://danbri.org/foaf.rdf>
where
{
  <#danbri> foaf:knows ?friend.
  ?friend foaf:name ?name.
```

```
?friend foaf:mbox ?email.
?friend foaf:homepage ?homepage.
}
```

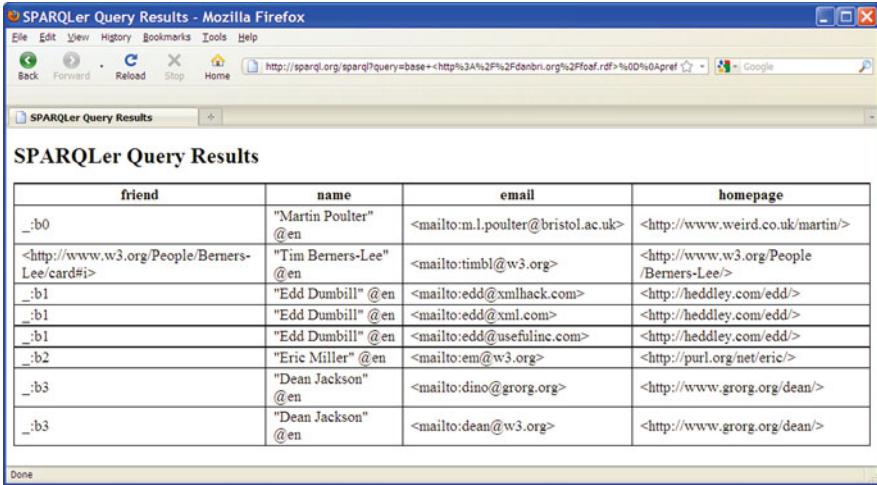


Fig. 6.4 Results from running the query shown in List 6.5

The graph pattern in List 6.5 contains four triple patterns. Also, variable ?friend is used as the object in the first triple pattern, but used as subject of the other three triple patterns. This is the so-called object-to-subject transfer in SPARQL queries. By doing this transfer, we can traverse multiple links in the RDF graph.

If we run this query against Brickley’s FOAF graph, we do see the names, e-mails, and home pages of Brickley’s friends, which make the result much more readable, as shown in Fig. 6.4.

However, the number of friends showing up in this result is much less than the number indicated by the result from running the query in List 6.4 – looks like some friends are missing. What is wrong? We will leave the answer to the next section, and before that, let us try some more SPARQL queries.

Some of Brickley’s friends do have their pictures posted on the Web, and let us say for some reason we are interested in the formats of these pictures. The query shown in List 6.6 tries to find all the picture formats that have been used by Brickley’s friends:

List 6.6 Find all the picture formats used by Brickley’s friends

```
base <http://danbri.org/foaf.rdf>
prefix foaf: <http://xmlns.com/foaf/0.1/>
prefix dc: <http://purl.org/dc/elements/1.1/>

select *
from <http://danbri.org/foaf.rdf>
```

```
where
{
  <#danbri> foaf:knows ?friend.
  ?friend foaf:depiction ?picture.
  ?picture dc:format ?imageFormat.
}
```

We have seen and discussed the object-to-subject transfer in List 6.5. In List 6.6, this transfer happens at a deeper level. The graph pattern in List 6.6 tries to match Brickley's friend, who has a `foaf:depiction` property instance defined, and this instance further has a `dc:format` property instance created, and we will like to return the value of this property instance. This chain of reference is the key to write queries using SPARQL, and is also frequently used.

In order to construct the necessary chain of references when writing SPARQL queries, we need to understand the structure of the ontologies that the given RDF graph file has used. Sometimes, in order to confirm our understanding, we need to read the graph which we are querying against. This should not surprise you at all; if you have to write SQL queries against some database tables, the first thing is to understand the structures of these tables, including the relations between them. The table structures and the relations between these tables can in fact be viewed as the underlying ontologies for these tables.

6.3.2.3 Using **OPTIONAL** Keyword for Matches

Optional keyword is another frequently used SPARQL feature, and the reason why optional keyword is needed is largely due to the fact that RDF data graph is only a semi-structured data model. For example, two instances of the same class type in a given RDF graph may have different set of property instances created for each one of them.

Let us take a look at Brickley's FOAF document, which has defined a number of `foaf:Person` instances. For example, one instance is created to represent Brickley himself, and quite a few others are defined to represent people he knows. Some of these `foaf:Person` instances do not have `foaf:name` property defined, and similarly, not every instance has `foaf:homepage` property instance created either.

This is perfectly legal, since there is no `owl:minCardinality` constraint defined on class `foaf:Person` regarding any of the above properties. For instance, not everyone has a home page; it is therefore not reasonable to require each `foaf:Person` instance to have a `foaf:homepage` property value. Also, recall that `foaf:mbox` is an inverse functional property, which is in fact used to uniquely identify a given person. As a result, having a `foaf:name` value or not for a given `foaf:Person` instance is not vital either.

This has answered the question we had in the previous section from List 6.5: not every Brickley's friend has a name, e-mail, and home page defined. And since the query in List 6.5 works like a logical AND, it only matches a friend whom Brickley knows and has *all* these three properties defined. Obviously, this will

return less number of people compared to the result returned by the query in List 6.4.

Now, we can change this query in List 6.5 a little bit: find all the people known by Brickley and show their name, e-mail, and home page if *any* of that information is available.

To accomplish this, we need `optional` keyword, as shown in List 6.7:

List 6.7 Change List 6.5 to use `optional` keyword

```
base <http://danbri.org/foaf.rdf>
prefix foaf: <http://xmlns.com/foaf/0.1/>

select *
from <http://danbri.org/foaf.rdf>
where
{
  <#danbri> foaf:knows ?friend.
  optional { ?friend foaf:name ?name. }
  optional { ?friend foaf:mbox ?email. }
  optional { ?friend foaf:homepage ?homepage. }
}
```

This query says, find all the people known by Brickley and show their name, e-mail, and home page information if *any* of this information is available. Run this query, you will see the difference between this query and the one shown in List 6.5. And here is the rule about `optional` keyword: the search will try to match all the graph patterns but does not fail the whole query if the graph pattern modified by `optional` keyword fails.

Note that in List 6.7, there are three different graph patterns modified by `optional` keyword, and any number of these graph patterns can fail, yet without causing the solution to be dropped. Clearly, if a query has multiple `optional` blocks, these `optional` blocks act independently of one another, any one of them can be omitted from or present in a solution.

Also note that the graph pattern modified by an `optional` keyword can have any number of triple patterns inside it. In List 6.7, each graph pattern modified by `optional` keyword happens to contain only one triple pattern. If a graph pattern modified by `optional` keyword contains multiple triple patterns, every single triple pattern in this graph pattern has to be matched in order to include a solution in the result set. For example, consider the query in List 6.8:

List 6.8 Use `optional` keyword on the whole graph pattern (compared with List 6.7)

```
base <http://danbri.org/foaf.rdf>
prefix foaf: <http://xmlns.com/foaf/0.1/>

select *
from <http://danbri.org/foaf.rdf>
```

```

where
{
  <#danbri> foaf:knows ?friend.
  optional {
    ?friend foaf:name ?name.
    ?friend foaf:mbox ?email.
    ?friend foaf:homepage ?homepage.
  }
}

```

and compare the result from the one returned by List 6.7, you will see the difference. List 6.8 says, find all the people known by Brickley, show their name, e-mail, and home page information if *all* these information are available.

Let us look at one more example before we move on, which will be used in our later chapters: find all the people known by Brickley, for anyone of them, if she/he has e-mail address, show it, and if she/he has `rdfs:seeAlso` value, also get this value. This query is shown in List 6.9, and I will leave it for you to understand:

List 6.9 Find Dan Brickley’s friends, who could also have e-mail addresses and `rdfs:seeAlso` defined

```

base <http://danbri.org/foaf.rdf>
prefix foaf: <http://xmlns.com/foaf/0.1/>
prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>

select *
from <http://danbri.org/foaf.rdf>
where
{
  <#danbri> foaf:knows ?friend.
  optional { ?friend foaf:mbox ?email. }
  optional { ?friend rdfs:seeAlso ?ref. }
}

```

6.3.2.4 Using Solution Modifier

At this point, we know SPARQL query is about matching patterns. More specifically, SPARQL engine tries to match the triples contained in the graph patterns against the RDF graph, which is a collection of triples. Once a match is successful, it will bind the graph pattern’s variables to the graph’s nodes, and one such variable binding is called a *query solution*. Since the `select` clause has specified a list of variables (or all the variables, as shown in our examples so far), the values of these listed variables will be selected from the current query solution to form a row that will be included in the final query result, and this row is called a *solution*. Obviously, another successful match will add another new solution, so on and so forth, therefore creating a table as the final result, with each solution presented as a row in this table.

Sometimes, it is better or even necessary for the solutions in the result table to be reorganized according to our need. For this reason, SPARQL has provided several *solution modifiers*, which will be the topic of this section.

The first one to look at is the `distinct` modifier, which eliminates duplicate solutions from the result table. Recall the query presented in List 6.3, which tries to find all the properties and their values that Brickley has used to describe himself. Now, let us change the query so that only the property names are returned, as shown in List 6.10:

List 6.10 Change List 6.3 to return only one variable back

```
base <http://danbri.org/foaf.rdf>
prefix foaf: <http://xmlns.com/foaf/0.1/>

select ?property
from <http://danbri.org/foaf.rdf>
where
{
  <#danbri> ?property ?value.
}
```

run the query, and you can see a lot of repeated properties. Modify List 6.10 once more time to make it look as the query in List 6.11:

List 6.11 Use `distinct` keyword to eliminate repeated solutions from List 6.10

```
base <http://danbri.org/foaf.rdf>
prefix foaf: <http://xmlns.com/foaf/0.1/>

select distinct ?property
from <http://danbri.org/foaf.rdf>
where
{
  <#danbri> ?property ?value.
}
```

and you will see the difference: all the duplicated solutions are now gone.

Another frequently used solution modifier is `order by`, which is also quite often used together with `asc()` or `desc()`. It orders the result set based on one of the variables listed in the `where` clause. For example, the query in List 6.12 tries to find all the people that have ever been mentioned in Brickley's FOAF file, and they are listed in a more readable way:

List 6.12 Use `order by` and `asc()` to modify results

```
base <http://danbri.org/foaf.rdf>
prefix foaf: <http://xmlns.com/foaf/0.1/>
```

```

select ?name ?email
from <http://danbri.org/foaf.rdf>
where
{
  ?x a foaf:Person.
  ?x foaf:name ?name.
  ?x foaf:mbox ?email.
}
order by asc(?name)

```

Note that a pair of solution modifiers, namely `offset/limit`, is often used together with `order by` to take a defined slice from the solution set. More specifically, `limit` sets the maximum number of solutions to be returned, and `offset` sets the number of solutions to be skipped. These modifiers can certainly be used separately, for example, using `limit` alone will help us to ensure that not too many solutions are collected. Let us modify the query in List 6.12 to make it look like the one shown in List 6.13:

List 6.13 Use `limit/offset` to modify results

```

base <http://danbri.org/foaf.rdf>
prefix foaf: <http://xmlns.com/foaf/0.1/>

select ?name ?email
from <http://danbri.org/foaf.rdf>
where
{
  ?x a foaf:Person.
  ?x foaf:name ?name.
  ?x foaf:mbox ?email.
}
order by asc(?name)
limit 10 offset 1

```

Run this query and compare with the result from List 6.12, you will see the result from `offset/limit` clearly.

6.3.2.5 Using **FILTER** Keyword to Add Value Constraints

If you have used SQL to query a database, you know it is quite straightforward to add value constraints in SQL. For example, if you are querying against a student database system, you may want to find all the students whose GPA is within a given range. In SPARQL, you can also add value constraints to filter the solutions in the result set, and the keyword to use is called `filter`.

More specifically, value constraints specified by `filter` keyword are logical expressions that evaluate to `boolean` values when applied on values of bound variables. Since these constraints are logical expressions, we can therefore combine them together by using logical `&&` and `||` operators. Only those solutions that are

evaluated to be `true` by the given value constraints will be included in the final result set. Let us take a look at some examples.

List 6.14 will help us to accomplish the following: if Tim Berners-Lee is mentioned in Brickley's FOAF file, then we want to know what has been said about him:

List 6.14 What has been said about Berners-Lee?

```
base <http://danbri.org/foaf.rdf>
prefix foaf: <http://xmlns.com/foaf/0.1/>

select distinct ?property ?propertyValue
from <http://danbri.org/foaf.rdf>
where
{
  ?person foaf:name "Tim Berners-Lee"@en.
  ?person ?property ?propertyValue.
}
```

When you run this query against Brickley's FOAF document, you will indeed see some information about Tim Berners-Lee, such as his home page and his e-mail address. However, note that this query does not use any `filter` keyword at all; instead, it directly adds the constraints to the triple pattern.

This is certainly fine, but with the major drawback that you have to specify the information with exact accuracy. For example, in List 6.14, if you replace the line

```
?person foaf:name "Tim Berners-Lee"@en.
```

with this line

```
?person foaf:name "tim Berners-Lee"@en.
```

the whole query will not work at all.

A better way to put constraints on the solution is to use `filter` keyword. List 6.15 shows how to do this:

List 6.15 Use `filter` keyword to add constraints to the solution

```
1: base <http://danbri.org/foaf.rdf>
2: prefix foaf: <http://xmlns.com/foaf/0.1/>

3: select distinct ?property ?propertyValue
4: from <http://danbri.org/foaf.rdf>
5: where
6: {
7:   ?timB foaf:name ?y.
8:   ?timB ?property ?propertyValue.
9:   filter regex(str(?y), "tim berners-Lee", "i").
10: }
```

Table 6.2 Functions and operators provided by SPARQL

Category	Functions and operators
Logical	!, &&,
Math	+, -, *, /
Comparison	=, !=, >, <
SPARQL testers	isURL(), isBlank(), isLiteral(), bound()
SPARQL accessors	str(), lang(), datatype()
Other	sameTerm(), langMatches(), regex()

Line 9 uses `filter` keyword to put constraints on the solution set. It uses a regular expression function called `regex()` to do the trick: for a given triple, its object component is taken and converted into a string by using the `str()` function, and if this string matches the given string, “tim berners-Lee,” this filter will be evaluated to be `true`, in which case, the subject of the current triple will be bound to a variable called `?timB`. Note that “i” means ignore the case, so the string is matched even if it starts with a small t.

The rest of List 6.15 is easy: line 7 together with line 9 will bind variable `?timB` to the right resource, and line 8 will pick up all the properties and their related values that are ever used on this resource, which accomplishes our goal.

Run this query, and you will see it gives exactly the same result as given by List 6.14. However, it does not require us to know exactly how the desired resource is named in a given RDF file.

Note that `str()` is a function provided by SPARQL for us to use together with `filter` keyword. Table 6.2 summarizes the frequently used functions and operators; we will not go into the detailed description of each one of them, you can easily check them out.

Let us take a look at another example of using `filter` keyword: we want to find those who are known by Brickley and are also related to W3C (note that if someone has an e-mail address such as `someone@w3.org`, we will then assume he/she is related to W3C). List 6.16 is the query we can use:

List 6.16 Find Dan Brickley’s friends who are related to W3C

```
base <http://danbri.org/foaf.rdf>
prefix foaf: <http://xmlns.com/foaf/0.1/>
prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>

select ?name ?email
from <http://danbri.org/foaf.rdf>
where
{
  <#danbri> foaf:knows ?friend.
  ?friend foaf:mbox ?email.
  filter regex(str(?email), "w3.org", "i" ).
  optional { ?friend foaf:name ?name. }
}
```

Note that this query does not put e-mail address as an optional item; in other words, if someone known by Brickley is indeed related to W3C, however without his/her `foaf:mbox` information presented in the FOAF document, this person will not be selected.

List 6.17 gives the last example of using `filter`. It tries to find all those defined in Brickley's FOAF file and whose birthday is after the start of 1970 and before the start of 1980. It shows another flavor of `filter` keyword and also how to do necessary data conversions for the correct comparison we need.

List 6.17 Find all the person whose birthday is within a given time frame

```
base <http://danbri.org/foaf.rdf>
prefix foaf: <http://xmlns.com/foaf/0.1/>
prefix xsd: <http://www.w3.org/2001/XMLSchema#>

select ?name ?dob
from <http://danbri.org/foaf.rdf>
where
{
    ?person a foaf:Person.
    ?person foaf:name ?name.
    ?person foaf:dateOfBirth ?dob.
    filter ( xsd:date(str(?dob)) >= "1970-01-01"^^xsd:date &&
            xsd:date(str(?dob)) < "1980-01-01"^^xsd:date )
}
```

6.3.2.6 Using Union Keyword for Alternative Match

Sometimes, a query needs to be expressed by multiple graph patterns that are mutually exclusive, and any solution will have to match exactly one of these patterns. This situation is defined as an *alternative match* situation, and SPARQL has provided `union` keyword for us to accomplish this.

A good example is from FOAF ontology, which provides two properties for e-mail address: `foaf:mbox` and `foaf:mbox_sha1sum`. The first one takes a readable plain text as its value, and the second one uses hash codes of an e-mail address as its value to further protect the owner's privacy. If we want to collect all the e-mail addresses that are included in Brickley's FOAF file, we have to accept either one of these two alternative forms, as shown in List 6.18:

List 6.18 Using union keyword to collect e-mail information

```
base <http://danbri.org/foaf.rdf>
prefix foaf: <http://xmlns.com/foaf/0.1/>

select ?name ?mbox
from <http://danbri.org/foaf.rdf>
where
{
```

```

?person a foaf:Person.
?person foaf:name ?name.
{
  { ?person foaf:mbox ?mbox. }
  union
  { ?person foaf:mbox_sha1sum ?mbox. }
}

```

Now, any solution has to match one and exactly one of the two graph patterns that are connected by the `union` keyword. If someone has both a plain text e-mail address and a hash-coded address, then both of these addresses will be included in the solution set. This is also the difference between `union` keyword and `optional` keyword; as seen in List 6.19, since `optional` keyword is used, a given solution can be included in the result set without matching any of the two graph patterns at all:

List 6.19 Using `optional` keyword is different from using `union`, as shown in List 6.18

```

base <http://danbri.org/foaf.rdf>
prefix foaf: <http://xmlns.com/foaf/0.1/>

select ?name ?mbox
from <http://danbri.org/foaf.rdf>
where
{
  ?person a foaf:Person.
  ?person foaf:name ?name.
  optional { ?person foaf:mbox ?mbox. }
  optional { ?person foaf:mbox_sha1sum ?mbox. }
}

```

After you run the query in List 6.19, you can find those solutions in the result set which do not have any e-mail address, and these solutions will for sure not be returned when the query in List 6.18 is used.

Another interesting change of List 6.18 is shown in List 6.20:

List 6.20 Another example using `union` keyword, different from List 6.18

```

base <http://danbri.org/foaf.rdf>
prefix foaf: <http://xmlns.com/foaf/0.1/>

select ?name ?mbox ?mbox1
from <http://danbri.org/foaf.rdf>
where
{
  ?person a foaf:Person.
  ?person foaf:name ?name.

```



```

{
  { ?person foaf:mbox ?mbox. }
  union
  { ?person foaf:mbox_sha1sum ?mbox1. }
}

```

I will leave it to you to run the query in List 6.20 and to understand the query result.

Before we move on to the next section, let us take a look at one more example of `union` keyword, and this example will be useful in a later chapter.

As we know, almost any given Web page has links to other pages, and these links are arguably what makes the Web interesting. A FOAF file is also a Web page, with the only difference of being a page that machine can understand. Therefore, it should have links pointing to the outside world as well. The question is, what are these links in a given FOAF page (we will see how to use these links in a later chapter)?

At this point, at least the following three links can be identified:

- `rdfs:seeAlso`
- `owl:sameAs`
- `foaf:isPrimaryTopicOf`

Since all these properties can take us to either another document or another resource on the Web, they can be understood as links to the outside world.

The query shown in List 6.21 can help us to collect all these links from Brickley's FOAF document. If you can think of more properties that can be used as links, you can easily add them into List 6.21:

List 6.21 Using `union` keyword to collection links to the outside world

```

base <http://danbri.org/foaf.rdf>
prefix foaf: <http://xmlns.com/foaf/0.1/>
prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>
prefix owl: <http://www.w3.org/2002/07/owl#>

select ?name ?seeAlso ?sameAs ?topicOf
from <http://danbri.org/foaf.rdf>
where
{
  ?person a foaf:Person.
  ?person foaf:name ?name.
  {
    { ?person rdfs:seeAlso ?seeAlso. }
    union
    { ?person owl:sameAs ?sameAs. }
    union

```

```

    { ?person foaf:isPrimaryTopicOf ?topicOf. }
  }
}

```

6.3.2.7 Working with Multiple Graphs

So far, all the queries we have seen have involved only one RDF graph, and we have specified it by using the `from` clause. This graph, in the world of SPARQL, is called a *background graph*. In fact, in addition to this background graph, SPARQL allows us to query any number of *named graphs*, and this is the topic of this section.

The first thing to know is how to make a named graph available to a query. As with the background graph, named graphs can be specified by using the following format:

```
from named <uri>
```

where `<uri>` specifies the location of the graph. Within the query itself, named graphs are used with the `graph` keyword, together with either a variable name that will bind to a named graph or the same `<uri>` for a named graph, as we will see in the examples. In addition, each named graph will have its own graph patterns to match against.

To show the examples, we need to have another RDF graph besides the FOAF file created by Brickley. For our testing purpose, we will use my own FOAF document as the second graph. You can see my FOAF file here:

```
http://www.liyangyu.com/foaf.rdf
```

Now, let us say that we would like to find those people who are mentioned by both Brickley and myself in our respective FOAF files. List 6.22 is our initial solution, which works with multiple graphs and uses `graph` keyword in conjunction with a variable called `graph_uri`:

List 6.22 Find those who are mentioned by both Brickley's and my own FOAF documents

```

1: prefix foaf: <http://xmlns.com/foaf/0.1/>
2: select distinct ?graph_uri ?name ?email
3: from named <http://www.liyangyu.com/foaf.rdf>
4: from named <http://danbri.org/foaf.rdf>
5: where
6: {
7:   graph ?graph_uri
8:   {
9:     ?person a foaf:Person.
10:    ?person foaf:mbox ?email.
11:    optional { ?person foaf:name ?name. }
12:   }
13: }

```

Table 6.3 Partial result from query List 6.22

Graph_uri	Name	E-mail
...
< http://danbri.org/foaf.rdf >	"Libby Miller"@en	<mailto:libby.miller@bristol.ac.uk>
< http://danbri.org/foaf.rdf >	"Tim Berners-Lee"@en	<mailto:timbl@w3.org>
...
< http://www.liyangyu.com/foaf.rdf >		<mailto:libby.miller@bristol.ac.uk>

First of all, note that lines 3 and 4 specify the two named graphs by using `from` named keyword, and each graph is given by its own `<uri>`. As you can tell, one of the graphs is the FOAF file created by Brickley and the other one is my own FOAF document. Furthermore, lines 8–12 define a graph pattern, which will be applied to each of the named graphs available to this query.

When this query is executed by SPARQL engine, variable `graph_uri` will be bound to the URI of one of the named graphs, and the graph pattern shown from lines 8–12 will be matched against this named graph. Once this is done, variable `graph_uri` will be bound to the URI of the next name graph, and the graph pattern is again matched against this current named graph, so on and so forth, until all the named graphs are finished. If a match is found during this process, the matched person's `foaf:mbox` property value will be bound to `email` variable, and the `foaf:name` property value (if exists) will be bound to `name` variable. Finally, line 2 shows the result: it not only shows all the names and e-mails of the selected people but also shows from which file the information is collected.

To make our discussion easier to follow, Table 6.3 shows part of the result. At the time you are running this query, it is possible that you will see different results, but the discussion here will still apply.

Table 6.3 shows that both FOAF files have mentioned a person whose e-mail address is given by the following:

```
<mailto:libby.miller@bristol.ac.uk>
```

As we will see in Chap. 7, a person can be uniquely identified by her/his `foaf:mbox` property value, no matter whether we have assigned a `foaf:name` property value to this person or not. And to show this point, in my FOAF file, the name of the person identified by the above e-mail address is intentionally not provided.

Note that in order to find those people who are mentioned by both FOAF documents, we have to manually read the query result shown in Table 6.3, i.e., to find common e-mail addresses from both files. This is certainly not the best solution for us, and we need to change our query to *directly* find those who are mentioned in both graphs.

And this new query is shown in List 6.23:

List 6.23 Change List 6.22 to direct get the required result

```

1: prefix foaf: <http://xmlns.com/foaf/0.1/>
2: select distinct ?name ?email
3: from named <http://www.liyangyu.com/foaf.rdf>
4: from named <http://danbri.org/foaf.rdf>
5: where
6: {
7:   graph <http://www.liyangyu.com/foaf.rdf>
8:     {
9:       ?person a foaf:Person.
10:      ?person foaf:mbox ?email.
11:      optional { ?person foaf:name ?name. }
12:    }.
13:   graph <http://danbri.org/foaf.rdf>
14:     {
15:       ?person1 a foaf:Person.
16:       ?person1 foaf:mbox ?email.
17:       optional { ?person1 foaf:name ?name. }
18:     }.
19: }

```

In this query, the `graph` keyword is used with the URI of a named graph (line 7, 13). The graph pattern defined in lines 8–12 will be applied on my FOAF file, and if matches are found in this graph, they become part of a query solution; the value of `foaf:mbox` property is bound to a variable named `email`, and the value of `foaf:name` property (if exists) is bound to a variable called `name`. The second graph pattern (lines 14–18) will be matched against Brickley’s FOAF graph, and the bound variables from the previous query solution will be tested here; the same variable `email` will have to be matched here, and if possible, the same `name` variable should match as well. Recall the key of graph patterns: any given variable, once bound to a value, has to bind to that value during the whole matching process.

Note that the variable representing a person is different in two graph patterns: in the first graph pattern, it is called `person` (line 9) and in the second graph pattern, it is called `person1` (line 15). The reason should be clear now; it does not matter whether this variable holds the same value or not in both patterns, since `email` value is used to uniquely identify a person resource. Also, it is possible that a given person resource is represented by blank nodes in both graphs, and blank nodes only have a scope that is within the graph that contains them. Therefore, even if they represent the same resource in the real world, it is simply impossible to match them at all.

Now run the query in List 6.23, we will be able to find all those people who are mentioned simultaneously in both graphs. Table 6.4 shows the query result.

As we have discussed earlier, part of the power of RDF graphs comes from data aggregation. Since both RDF files have provided some information about Libby

Table 6.4 result from query List 6.23

Name	E-mail
"Libby Miller"@en	<mailto:libby.miller@bristol.ac.uk> <mailto:libby.miller@bristol.ac.uk>

Miller, it will be interesting to aggregate these two pieces of information together and see what have been said about Miller as a `foaf:Person` instance. List 6.24 accomplishes this:

List 6.24 Data aggregation for Libby Miller

```

1: prefix foaf: <http://xmlns.com/foaf/0.1/>
2: select distinct ?graph_uri ?property ?hasValue
3: from named <http://www.liyangyu.com/foaf.rdf>
4: from named <http://danbri.org/foaf.rdf>

5: where
6: {
7:   graph <http://www.liyangyu.com/foaf.rdf>
8:   {
9:     ?person1 a foaf:Person.
10:    ?person1 foaf:mbox ?email.
11:    optional { ?person1 foaf:name ?name. }
12:   }.

13:  graph <http://danbri.org/foaf.rdf>
14:  {
15:    ?person a foaf:Person.
16:    ?person foaf:mbox ?email.
17:    optional { ?person foaf:name ?name. }
18:  }.

19:  graph ?graph_uri
20:  {
21:    ?x a foaf:Person.
22:    ?x foaf:mbox ?email.
23:    ?x ?property ?hasValue.
24:  }

25: }

```

So far, we have seen examples where `graph` keyword is used together with a variable that will bind to a named graph (List 6.22), or it is used with an `<uri>` that represents a named graph (List 6.23). In fact, these two usage patterns can be mixed

Table 6.5 Result from query List 6.24

Graph_uri	Property	hasValue
<danbri:foaf.rdf>	<rdfs:seeAlso>	< http://www.libbymiller.com/webwho.xrdf >
<danbri:foaf.rdf>	<foaf:workplaceHomepage>	< http://ilrt.org/ >
<danbri:foaf.rdf>	<foaf:mbox>	<mailto:libby.miller@bristol.ac.uk>
<danbri:foaf.rdf>	<foaf:name>	"Libby Miller"@en
<danbri:foaf.rdf>	<rdf:type>	< http://xmlns.com/foaf/0.1/Person >
<danbri:foaf.rdf>	<foaf:depiction>	< http://rdfweb.org/people/danbri/rdfweb/libby.gif >
<danbri:foaf.rdf>	<foaf:img>	< http://swordfish.rdfweb.org/~libby/libby.jpg >
<danbri:foaf.rdf>	<foaf:mbox>	<mailto:libby@asemantics.com>
<liyang:foaf.rdf>	<foaf:homepage>	< http://www.ilrt.bris.ac.uk/~ecemm/ >
<liyang:foaf.rdf>	<foaf:mbox>	<mailto:libby@asemantics.com>
<liyang:foaf.rdf>	<rdf:type>	< http://xmlns.com/foaf/0.1/Person >

liyang: <http://www.liyangyu.com/>

danbri: <http://danbri.org/>

together, as shown in List 6.24. After the discussion of Lists 6.22 and 6.23, List 6.24 is quite straightforward; clearly, lines 7–18 find those who have been included in both graphs, and lines 19–24 provide a graph pattern that will collect everything that has been said about those instances from all the named graphs.

Table 6.5 shows the query result generated by List 6.24. Again, at the time you are running the query, the result could be different. Nevertheless, as shown in Table 6.5, statements about Miller from both FOAF files have been aggregated together. This simple example in fact shows the basic flow of how an aggregation agent may work by using the search capabilities provided by SPARQL endpoints.

The last example of this section is given by List 6.25, where a background graph and a named graph are used together. Read this query and try to find what it does before you read on:

List 6.25 What this query does? Think about it before reading on

```

1: prefix foaf: <http://xmlns.com/foaf/0.1/>
2: select distinct ?property ?hasValue
3: from <http://danbri.org/foaf.rdf>
4: from named <http://www.liyangyu.com/foaf.rdf>
5: from named <http://danbri.org/foaf.rdf>
6: where
7: {
8:   graph <http://www.liyangyu.com/foaf.rdf>
9:   {
10:     ?person1 a foaf:Person.

```

```

11:      ?person1 foaf:mbox ?email.
12:      optional { ?person1 foaf:name ?name. }
13:    }.

14:    graph <http://danbri.org/foaf.rdf>
15:    {
16:      ?person a foaf:Person.
17:      ?person foaf:mbox ?email.
18:      optional { ?person foaf:name ?name. }
19:    }.
20:    ?x a foaf:Person.
21:    ?x foaf:mbox ?email.
22:    ?x ?property ?hasValue.
23:  }

```

The interesting part of this query is at lines 3–5, where a background graph is specified in line 3, and two named graphs are introduced in lines 4 and 5. Note that lines 3 and 5 are in fact the same graph.

Now, lines 8–19 are the same as the query in List 6.24 (trying to find those who are mentioned in both graphs), and lines 20–22 is a graph pattern that does not specify any graph, so this pattern is matched against the background graph. Therefore, this query, after finding those who are mentioned in both Brickley’s file and my file, tries to collect everything that has been said about them from Brickley’s file *only*. This is not a data aggregation case as shown by List 6.24, but it shows the combination usage of a background graph and a named graph.

At this point, we have covered quite some features about SPARQL’s `select` query, and the examples presented here should have given you enough to explore their other features on your own. Let us move on to SPARQL’s other query styles, and we will briefly discuss them in the next several sections.

6.3.3 CONSTRUCT Query

`construct` query is another query form provided by SPARQL which, instead of returning a collection of query solutions, returns a new RDF graph. Let us take a look at some examples.

List 6.26 creates a new FOAF graph, which has a collection of all the names and e-mails of those who are mentioned in Brickley’s FOAF document. List 6.27 shows part of this new graph:

List 6.26 Example of a `construct` query

```

prefix foaf: <http://xmlns.com/foaf/0.1/>

construct {
  ?person a foaf:Person.

```

```

    ?person foaf:name ?name.
    ?person foaf:mbox ?email.
}
from <http://danbri.org/foaf.rdf>
where
{
    ?person a foaf:Person.
    ?person foaf:name ?name.
    ?person foaf:mbox ?email.
}

```

List 6.27 Part of the generated RDF graph

```

<?xml version="1.0"?>
<rdf:RDF
  xmlns:foaf="http://xmlns.com/foaf/0.1/"
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
  <foaf:Person>
    <foaf:mbox rdf:resource="mailto:craig@coolstuffhere.co.uk"/>
    <foaf:name xml:lang="en">Craig Dibble</foaf:name>
  </foaf:Person>
  <foaf:Person>
    <foaf:name xml:lang="en">Joe Brickley</foaf:name>
    <foaf:mbox rdf:resource=
      "mailto:joe.brickley@btopenworld.com"/>
  </foaf:Person>
  <foaf:Person>
    <foaf:mbox rdf:resource="mailto:libby.miller@bristol.ac.uk"/>
    <foaf:name xml:lang="en">Libby Miller</foaf:name>
  </foaf:Person>
  ... more ...

```

This generated new graph is indeed clean and nice, but it is not that much interesting. In fact, a common use of `construct` query form is to transform a given graph to a new graph that uses a different ontology.

For example, List 6.28 will transfer FOAF data to vCard data, and List 6.29 shows part of the resulting graph:

List 6.28 Another `construct` query which changes FOAF document into vCard document

```

prefix foaf: <http://xmlns.com/foaf/0.1/>
prefix vCard: <http://www.w3.org/2001/vcard-rdf/3.0#>

construct {
    ?person vCard:FN ?name.
    ?person vCard:URL ?homepage.
}

```



```

from <http://danbri.org/foaf.rdf>

where
{
  optional {
    ?person foaf:name ?name.
    filter isLiteral(?name).
  }
  optional {
    ?person foaf:homepage ?homepage.
    filter isURI(?homepage).
  }
}

```

List 6.29 Part of the new graph expressed as vCard data

```

<?xml version="1.0"?>
<rdf:RDF
  xmlns:foaf="http://xmlns.com/foaf/0.1/"
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:vCard="http://www.w3.org/2001/vcard-rdf/3.0#">
<rdf:Description>
  <vCard:FN xml:lang="en">Dan Connolly</vCard:FN>
</rdf:Description>
<rdf:Description>
  <vCard:FN xml:lang="en">Dan Brickley</vCard:FN>
</rdf:Description>
<rdf:Description>
  <vCard:FN xml:lang="en">Jim Ley</vCard:FN>
</rdf:Description>
<rdf:Description>
  <vCard:FN xml:lang="en">Eric Miller</vCard:FN>
  <vCard:URL rdf:resource="http://purl.org/net/eric/" />
</rdf:Description>
... more ...

```

6.3.4 DESCRIBE Query

At this point, every query we have constructed requires us to know something about the data graph. For example, when we are querying against a FOAF document, at least we know some frequently used FOAF terms, such as `foaf:mbox` and `foaf:name`, so we can provide a search criteria to SPARQL query processor. This is similar to writing SQL queries against a database system; we will have to be familiar with the structures of the tables in order to come up with queries.

However, sometimes, we just don't know much about the data graph, and we don't even know what to ask. If this is the case, we can ask a SPARQL query processor to describe the resource we want to know, and it is up to the processor to provide some useful information about the resource we have asked.

And this is the reason behind the `describe` query. After receiving the query, a SPARQL processor will create and return an RDF graph; the content of the graph is decided by the query processor, not the query itself.

For example, List 6.30 is one such query:

List 6.30 Example of `describe` query

```
prefix foaf: <http://xmlns.com/foaf/0.1/>
describe ?x
from <http://danbri.org/foaf.rdf>
where
{
  ?x foaf:mbox <mailto:timbl@w3.org>.
}
```

In this case, the only thing we know is the e-mail address, so we provide this information and ask SPARQL processor to tell us more about the resource whose e-mail address is given by `<mailto:timbl@w3.org>`. The query result is another RDF graph whose statements are determined by the query processor.

At the time of this writing, SPARQL Working Group has adopted `describe` keyword without reaching consensus. A description will be determined by the particular SPARQL implementation, and the statements included in the description are left to the nature of the information in the data source. For example, if you are looking for a description about a book resource, the author information could be included in the result RDF graph.

For this reason, we are not going to cover any more details. However, understanding the reason why this keyword is proposed will certainly help you in a later time when hopefully some agreement can be reached regarding the semantics of this keyword.

6.3.5 *ASK Query*

SPARQL's `ask` query is identified by `ask` keyword, and the query processor simply returns a `true` or `false` value, depending on whether the given graph pattern has any matches in the dataset or not.

List 6.31 is a simple example of `ask` query:

List 6.31 Example of using `ask` query

```
prefix foaf: <http://xmlns.com/foaf/0.1/>
ask
from <http://danbri.org/foaf.rdf>
```

```

where
{
  ?x foaf:mbox <mailto:danbri@danbri.org>.
}

```

This query has a graph pattern that is equivalent to the following question: is there a resource whose `foaf:mbox` property uses `<mailto:danbri@danbri.org>` as its value? To answer this query, the processor tries to match the graph pattern against the FOAF data graph, and apparently, a successful match is found, therefore, `true` is returned as the answer.

It is fun to work with `ask` query. For example, List 6.32 tries to decide whether it is true that Brickley was either born before 1 January 1970 or after 1 January 1980:

List 6.32 Ask the birthday of Dan Brickley

```

base <http://danbri.org/foaf.rdf>
prefix foaf: <http://xmlns.com/foaf/0.1/>
prefix xsd: <http://www.w3.org/2001/XMLSchema#>

ask
from <http://danbri.org/foaf.rdf>
where
{
  <#danbri> foaf:dateOfBirth ?dob.
  filter ( xsd:date(str(?dob)) <= "1970-01-01"^^xsd:date ||
          xsd:date(str(?dob)) >= "1980-01-01"^^xsd:date )
}

```

And certainly, this will give `false` as the answer. Note that we should understand this answer in the following way: the processor cannot find any binding to compute a solution to the graph pattern specified in this query.

Obviously, a `true` or `false` answer depends on the given graph pattern. In fact, you can use any graph pattern together with `ask` query. As the last example, you can ask whether both Brickley's FOAF file and my own FOAF file have described anyone in common, and this query should look very familiar, as shown in List 6.33:

List 6.33 Ask if the two FOAF documents have described anyone in common

```

prefix foaf: <http://xmlns.com/foaf/0.1/>

ask
from named <http://www.liyangyu.com/foaf.rdf>
from named <http://danbri.org/foaf.rdf>
where
{
  graph <http://www.liyangyu.com/foaf.rdf>
  {
    ?person a foaf:Person.

```

```

    ?person foaf:mbox ?email.
  optional { ?person foaf:name ?name. }
}.
graph <http://danbri.org/foaf.rdf>
{
  ?person1 a foaf:Person.
  ?person1 foaf:mbox ?email.
  optional { ?person1 foaf:name ?name. }
}.
}

```

And if you run the query, you can get `true` as the answer, as you have expected.

6.4 What Is Missing from SPARQL?

At this point, we have covered the core components of the current SPARQL standard. If you are experienced with SQL queries, you have probably realized the fact that there is something missing in the current SPARQL language constructs. Let us briefly discuss these issues in this section, and in the next section, we will take a closer look at SPARQL 1.1, SPARQL Working Group's latest progress.

The most obvious fact about SPARQL is that it is read-only. In its current stage, SPARQL is only a retrieval query language; there are no equivalents of the SQL `insert`, `update` and `delete` statements.

The second noticeable missing piece is that SPARQL does not support any grouping capabilities or aggregate functions, such as `min`, `max`, `avg`, `sum`, just to name a few. There are some implementations of SPARQL that provide these functions; yet, standardization is needed, so a uniform interface can be reached.

Another important missing component is the service description. More specifically, there is no standard way for a SPARQL endpoint to advertise its capabilities and its dataset.

There are other functionalities that are missing, and we are not going to list them all there. The good news is, some of these missing features have long been on the task list of W3C SPARQL Working Group, and a potentially updated standard called SPARQL 1.1 will be ready soon.

6.5 SPARQL 1.1

6.5.1 Introduction: What Is New?

SPARQL 1.1 is the collective name of the work produced by the current SPARQL Working Group. The actual components being worked on include the following major pieces:

- SPARQL 1.1 Query
- SPARQL 1.1 Update
- SPARQL 1.1 Protocol
- SPARQL 1.1 Service Description
- SPARQL 1.1 Uniform HTTP Protocol for Managing RDF Graphs
- SPARQL 1.1 Entailment Regimes
- SPARQL 1.1 Property Paths

and you can also find more details here:

http://www.w3.org/2009/sparql/wiki/Main_Page

At the time of this writing, these standards are still under active discussion and revision. To give you some basic idea of these new standards, we will concentrate on SPARQL 1.1 Query and Update. Not only because these two pieces are relatively stable but also because they are the ones that are most relevant to our day-to-day development work on the Semantic Web.

6.5.2 SPARQL 1.1 Query

In this section, the following new features added by SPARQL 1.1 Query will be briefly discussed:

- aggregate functions
- subqueries
- negation
- expressions with `SELECT`
- property paths

Again understand that at the time of this writing, SPARQL 1.1 Query is not finalized yet. The material presented here is based on the latest working drafts from SPARQL Working Group; it is therefore possible that the final standard will be more or less different. However, the basic ideas that will be discussed here should remain the same.

6.5.2.1 Aggregate Functions

If you are experienced with SQL queries, chance is that you are familiar with aggregate functions. These functions operate over the columns of a result table and can conduct operations such as counting, numerical averaging, or selecting the maximal/minimal data element from the given column. The current SPARQL query standard does not provide these operations, and if an application needs these functions, the application has to take a SPARQL query result set and calculate the aggregate values by itself.

Obviously, enabling a SPARQL engine to calculate aggregates for the users will make the application more light weighted, since the work will be done on the

SPARQL engine side. In addition, this will normally result in significantly smaller result set being returned to the application. If the SPARQL endpoint is accessed over HTTP, this will also help to lessen the traffic on the network.

With these considerations, SPARQL 1.1 will support aggregate functions. As usual, a query pattern yields a solution set, and from this solution set, a collection of columns will be returned to the user as the query result. An aggregation function will then operate on this set to create a new solution set which normally contains a single value representing the result from the aggregate function.

The following aggregate functions will be supported by SPARQL 1.1:

- COUNT
- SUM
- MIN/MAX
- AVG
- GROUP_CONCAT
- SAMPLE

Let us study some examples to understand more.

The first example queries about how many people have their e-mail addresses provided in a given FOAF document. List 6.34 shows the query itself:

List 6.34 Example of using `count ()` aggregate function

```
prefix foaf: <http://xmlns.com/foaf/0.1/>
SELECT count(*)
from <http://danbri.org/foaf.rdf>
WHERE {
    ?x a foaf:Person;
        foaf:mbox ?mbox.
}
```

This does not require much of an explanation at all. Note that at the time of this writing, this query is supported by the online Joseki endpoint which can be accessed at this URL:

<http://sparql.org/sparql.html>

If you are using other SPARQL endpoints, this query might not work well. In addition, even the SPARQL endpoint you are using does support aggregate functions, its implementation might require slightly different syntax.

The following example scans a given FOAF document and tries to sum up all the ages of the people included in this document, as shown in List 6.35:

List 6.35 Example of using `sum ()` aggregate function

```
prefix foaf: <http://xmlns.com/foaf/0.1/>
SELECT ( sum(?age) AS ?ages )
from <http://danbri.org/foaf.rdf>
```

```
WHERE {
    ?x a foaf:Person;
        foaf:age ?age.
}
```

the final result will be stored in the `ages` variable.

Again, this query has been tested by using online Joseki endpoint, and it works well. For the rest of this chapter, without explicitly mentioning, all the new SPARQL 1.1 features will be tested using the same Joseki endpoint. At the time when you are reading this book, you can either use Joseki endpoint or use your favorite one, which should be working as well.

The last example we would like to discuss here is the `SAMPLE` function. To put it simple, the newly added `SAMPLE` aggregate tries to solve the issue where it is not possible to project a particular variable or apply functions over that variable out of a `GROUP` since we are not grouping on that particular variable. Now, with `SAMPLE` aggregate function, this is very easy, as shown in List 6.36:

List 6.36 Example of using `SAMPLE ()` aggregation function

```
SELECT ?subj SAMPLE(?obj)
from <http://danbri.org/foaf.rdf>
WHERE {
    ?subj ?property ?obj.
} GROUP BY ?subj
```

6.5.2.2 Subqueries

Subquery is not something new either. More specifically, it is sometimes necessary to use the result from one query to continue the next query.

For example, let us consider a simple request. Assume we would like to find all the friends I have, and for each one of them, I would like to know their e-mail addresses. As you know, a single query can be constructed by using the current SPARQL constructs to finish the task, there is no need to write two queries at all.

However, let us slightly change the request: find all the friends I have, and for each one of them, I would like to know their e-mail address and I only want one e-mail address for each friend I have.

Now to construct this query using the current SPARQL constructs, you will have to write two queries. The pseudo-code in List 6.37 shows the solution you will have to use:

List 6.37 Pseudo-code that finds friends and only one e-mail address of each friend

```
queryString = "
    SELECT ?person WHERE {
<http://www.liyangyu.com/foaf.rdf#liyang> foaf:knows ?friend.
    }";
```

```
resultSet = do_query(queryString);

foreach (result in resultSet) {
    person = result.get("friend");
    queryString = "SELECT ?mbox WHERE {
                    ?person foaf:mbox ?mbox.
                    } LIMIT 1";
    // do the query and get the e-mail address
}
```

Now, using the subquery feature provided by SPARQL 1.1, only one query is needed to finish the above task, as shown in List 6.38:

List 6.38 Using subquery feature to accomplish the same as by List 6.37

```
prefix foaf: <http://xmlns.com/foaf/0.1/>

SELECT ?friend ?mbox
from <http://www.liyangyu.com/foaf.rdf>
WHERE {
<http://www.liyangyu.com/foaf.rdf#liyang> foaf:knows ?friend.
{
    SELECT ?mbox WHERE {
        ?friend foaf:mbox ?mbox
    } LIMIT 1
}
}
```

6.5.2.3 Negation

Negation is something that can be implemented by simply using the current version of SPARQL. Let us consider the request of finding all the people in a given FOAF document who do not have any e-mail address specified. The query in List 6.39 will accomplish this (and note that it only uses the language features from the current standard of SPARQL):

List 6.39 Find all the people from a given FOAF document who do not have any e-mail address specified

```
prefix foaf: <http://xmlns.com/foaf/0.1/>

SELECT ?name
from <http://danbri.org/foaf.rdf>
WHERE {
    ?x foaf:givenName ?name.
    OPTIONAL { ?x foaf:mbox ?mbox }.
    FILTER (!BOUND(?mbox))
}
```


Let us understand `BOUND()` operator first. `BOUND()` operator is used as follows:

```
xsd:boolean BOUND(variable var)
```

It returns `true` if `var` is bounded to a value, it returns `false` otherwise. `BOUND()` operator is quite often used to test that a graph pattern is *not* expressed by specifying an `OPTIONAL` graph pattern which uses a variable and then to test to see that the variable is not bound. This testing method is called *negation as failure* in logic programming.

With this said, the query in List 6.39 is easy to understand; it matches the people with a name but no expressed e-mail address. Therefore, it accomplishes what we have requested.

However, this negation as failure method is not quite intuitive and has a somewhat convoluted syntax. To fix this, SPARQL 1.1 has adopted at least one new operator called `NOT EXISTS`, and this will make the same query much intuitive and easier, as shown in List 6.40:

List 6.40 Example of negation using `NOT EXISTS` operator

```
prefix foaf: <http://xmlns.com/foaf/0.1/>

SELECT ?name
from <http://danbri.org/foaf.rdf>
WHERE {
    ?x foaf:givenName ?name.
    NOT EXISTS { ?x foaf:mbox ?mbox }.
}
```

The SPARQL Working Group has also considered another different design for negation, namely the `MINUS` operator. List 6.41 shows a possible query which uses `MINUS` operator. Again, this accomplishes the same goal with a cleaner syntax:

List 6.41 Example of negation using `MINUS` operator

```
prefix foaf: <http://xmlns.com/foaf/0.1/>

SELECT ?name
from <http://danbri.org/foaf.rdf>
WHERE {
    ?x foaf:givenName ?name.
    MINUS {
        ?x foaf:mbox ?mbox.
    }
}
```

6.5.2.4 Expressions with **SELECT**

The expressions with `SELECT` feature added by SPARQL 1.1 is closely related to another feature called *projected expressions*. We will present these two closely related features together in this section.

In the current standard SPARQL Query language, a `SELECT` query (also called a *projection query*) may only project out variables bound in the query. More specifically, since variables can be bound only via triple pattern matching, it is impossible to project out values that are not matched in the underlying RDF dataset.

Expressions with `SELECT` query or projected expressions introduced by SPARQL 1.1 offers the ability for `SELECT` queries to project *any* SPARQL expression, rather than just bounded variables. A projected expression can be a variable, a constant URI/literal, or an arbitrary expression which may include functions on variables and constants. Functions could include both SPARQL built-in functions and extension functions supported by an implementation. Also, the variable used in the expression can be one of the following cases:

- a new variable introduced by `SELECT` clause (using the keyword `AS`);
- a variable binding already in the query solution; or
- a variable defined earlier in the `SELECT` clause.

List 6.42 shows one simple example:

List 6.42 Example of using expressions with **SELECT** query

```
prefix foaf:<http://xmlns.com/foaf/0.1/>
prefix fn:<http://www.w3.org/2005/xpath-functions>

SELECT fn:concat(?first, " ", ?last) AS ?name
from <http://danbri.org/foaf.rdf>
WHERE {
    ?person foaf:firstName ?first;
           foaf:lastName ?last.
}
```

This query tries to find the first and last names of all the people included in a given FOAF document. Instead of simply showing the binding variables (`?first` and `?last`), the `SELECT` clause uses projected expression to first concatenate the first name and last name by using the binding variables `?first` and `?last`; the result is saved in a new variable called `?name` by using keyword `AS`. The query result includes only the `?name` variable and could be something like “Dan Brickley.”

The next example (List 6.43) is taken from W3C’s SPARQL Query Language 1.1 Working Draft,⁴ since it is very helpful to show the usage of expressions in `SELECT` clause:

⁴<http://www.w3.org/TR/2010/WD-sparql11-query-20100126/>

List 4.43 Another example of using expressions in SELECT query

Data:

```
@prefix dc: <http://purl.org/dc/elements/1.1/>.
@prefix : <http://example.org/book/>.
@prefix ns: <http://example.org/ns#>.
```

```
:book1 dc:title "SPARQL Tutorial".
:book1 ns:price 42.
:book1 ns:discount 0.1.
```

```
:book2 dc:title "The Semantic Web".
:book2 ns:price 23.
:book2 ns:discount 0.
```

Query:

```
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX ns: <http://example.org/ns#>
SELECT ?title (?p*(1-?discount) AS ?price)
  { ?x ns:price ?p.
    ?x dc:title ?title.
    ?x ns:discount ?discount
  }
```

In this case, the expression makes use of binding variables such as `?title` and `?discount`, and the final price (after the discount) is stored in the new variable `?price`. Table 6.6 shows the result of this query.

Table 6.6 Result of query in List 6.43

Title	Price
"The Semantic Web"	23
"SPARQL Tutorial"	37.8

6.5.2.5 Property Paths

We have covered several SPARQL 1.1 Query features so far at this point, including aggregate functions, subqueries, negation, and projected expressions. These features are currently marked by W3C's SPARQL Query Language 1.1 Working Group as *required*. In addition to these required features, there are several other features being considered by the working group as *time permitting*. For example, property paths, basic federated query, and some commonly used SPARQL functions are all considered as time-permitting features.

For obvious reason, we will not cover these time-permitting features in details. Before we move on to SPARQL 1.1 Update, however, we will take a brief look at property paths, just to give you an idea of these time-permitting features.

If you have been writing quite a lot queries by using the current SPARQL standard, you have probably seen the cases where you need to follow paths to finish your query. More specifically, in order to find what you want, you need to construct a query that covers fixed-length paths to traverse along the hierarchical structure expressed in the given data store. List 6.44 shows one example. This query tries to find the name of my friend’s friend:

List 6.44 Find the name of my friend’s friend

```
prefix foaf:<http://xmlns.com/foaf/0.1/>

SELECT ?name
from <http://www.liyangyu.com/foaf.rdf>
where {
  ?myself foaf:mbox <mailto:liyang910@yahoo.com>.
  ?myself foaf:knows ?friend.
  ?friend foaf:knows ?friendOfFriend.
  ?friendOfFriend foaf:name ?name.
}
```

As shown in List 6.44, the paths we have traversed include the following: myself, friend, friendOfFriend, name of friendOfFriend. This is quite long and cumbersome. The property paths featured by SPARQL Query 1.1 will make this a lot simpler for us. List 6.45 shows the tentative syntax of this feature:

List 6.45 Example of property path

```
prefix foaf:<http://xmlns.com/foaf/0.1/>

SELECT ?name
from <http://www.liyangyu.com/foaf.rdf>
where {
  ?myself foaf:mbox <mailto:liyang910@yahoo.com>.
  ?myself foaf:knows/foaf:knows/foaf:name ?name.
}
```

This accomplishes exactly the same goal as the query in List 6.44, but with a much cleaned syntax.

6.5.3 SPARQL 1.1 Update

If you are experienced with SQL queries, you know how easy it is to change the data in the database. There are SQL statements provided for you to do that, you don’t have to know the mechanisms behind these statements.

We all know that SPARQL to RDF data stores is as SQL to databases. However, changing an RDF graph is not as easy as updating a table in a database. To add,

update, or remove statements from a given RDF graph, there is no SPARQL language constructs we can use; instead, we have to use a programming language and a set of third-party APIs to accomplish this.

To allow an RDF graph or an RDF store to be manipulated the same way as SQL to database, a language extension to the current standard of SPARQL is proposed. Currently, this language extension is called SPARQL Update 1.1, and it includes the following features:

- Insert new triples into an RDF graph.
- Delete triples from an RDF graph.
- Perform a group of update operations as a single action.
- Create a new RDF graph in a graph store.
- Delete an RDF graph from a graph store.

The first three operations are called *graph update*, since they are responsible for addition and removal of triples from one specific graph. The next two operations are called *graph management*, since they are responsible for creating and deleting graphs within a given graph store.

In this section, we will discuss these operations briefly. Again, understand this standard is not finalized yet, and by the time you are reading this book, it could be changed or updated. However, the basic idea should remain the same.

6.5.3.1 Graph Update: Adding RDF Statements

One way to add one or more RDF statements into a given graph is to use the `INSERT DATA` operation. This operation creates the graph if it does not exist. List 6.46 shows one example:

List 6.46 Example of using `INSERT DATA` to update a given graph

```
prefix foaf:<http://xmlns.com/foaf/0.1/>
prefix liyang: <http://www.liyangyu.com/foaf.rdf#>
INSERT DATA
{
  GRAPH <http://www.liyangyu.com/foaf.rdf>
  {
    liyang:liyang foaf:workplaceHomepage <http://www.delta.com> ;
                  foaf:schoolHomepage <http://www.osu.edu> .
  }
}
```

This will insert two new statements into my personal FOAF document, namely <http://www.liyangyu.com/foaf.rdf>. And these two statements show the home page of the company I work for and the home page of the school I graduated from.

Note that you can insert any number of RDF statements within one `INSERT DATA` request. In addition, the `GRAPH` clause is optional; an `INSERT DATA` request without the `GRAPH` clause will simply operate on the default graph in the RDF store.

Another way to add one or more RDF statements into a given graph is to use the `INSERT` operation. List 6.47 shows one example to use the `INSERT` operation. This example copies RDF statement(s) from my old FOAF document into my current FOAF document. More specifically, all the information about my interests are moved into the new FOAF document so I don't have to add each one of them manually.

List 6.47 Example of using `INSERT` operation to update a given graph

```
prefix foaf:<http://xmlns.com/foaf/0.1/>
prefix liyang: <http://www.liyangyu.com/foaf.rdf#>

INSERT
{
  GRAPH <http://www.liyangyu.com/foaf.rdf>
  { liyang:liyang foaf:topic_interest ?interest. }
}
WHERE
{
  GRAPH <http://www.liyangyu.com/foafOld.rdf>
  { liyang:liyang foaf:topic_interest ?interest. }
}
```

This is a very useful operation, and with this operation, we can move statements from one graph to another based on any graph pattern we have specified by using the `WHERE` clause.

6.5.3.2 Graph Update: Deleting RDF Statements

Similar to adding statements, deleting statements from a given RDF graph can be done in two ways. One way is to use the `DELETE DATA` operation, which removes triples from a graph. List 6.48 shows one example:

List 6.48 Example of using `DELETE DATA` operation to update a given graph

```
prefix foaf:<http://xmlns.com/foaf/0.1/>
prefix liyang: <http://www.liyangyu.com/foaf.rdf#>
DELETE DATA
{
  GRAPH <http://www.liyangyu.com/foaf.rdf>
  {
    liyang:liyang foaf:workplaceHomepage
      <http://www.delta.com> ;
      foaf:schoolHomepage <http://www.osu.edu>.
  }
}
```

This will delete the two statements we have just added into my FOAF document. Similarly, you can delete any number of statements in one `DELETE DATA` request, and the `GRAPH` clause is optional; an `DELETE DATA` statement without the `GRAPH` clause will simply operate on the default graph in the RDF store.

Another way to delete one or more RDF statements from a given graph is to use the `DELETE` operation. List 6.49 shows one example. And as you can tell, we will be able to specify a graph pattern using `WHERE` clause so as to delete statements more effectively. In this particular example, we would like to delete all the e-mail information that have been included in my FOAF document:

List 6.49 Example of using **DELETE** operation to update a given graph

```
prefix foaf:<http://xmlns.com/foaf/0.1/>
DELETE
{
  GRAPH <http://www.liyangyu.com/foaf.rdf>
  { ?person foaf:mbox ?mbox. }
}
```

Similar to `INSERT` operation, `DELETE` operation can have a `WHERE` clause to specify more interesting graph pattern one can match, as shown in List 6.50:

List 6.50 Example of using **DELETE** operation together with **WHERE** clause

```
prefix foaf:<http://xmlns.com/foaf/0.1/>
prefix dc: <http://purl.org/dc/elements/1.1/>
prefix liyang: <http://www.liyangyu.com/foaf.rdf#>

DELETE
{
  GRAPH <http://www.liyangyu.com/foaf.rdf>
  { ?logItem ?pred ?obj. }
}
WHERE
{
  GRAPH <http://www.liyangyu.com/foaf.rdf>
  {
    liyang:liyang foaf:weblog ?logItem;
    ?logItem dc:date ?date.
    FILTER ( ?date < "2005-01-01T00:00:00-2:00"^^xsd:
      dateTime )
    ?logItem ?pred ?obj
  }
}
```

As you can tell, this will delete all the blog items I have written before January 1st of 2005.

6.5.3.3 Graph Update: **LOAD** and **CLEAR**

Two more operations that can be useful are the `LOAD` and `CLEAR` operations. With what we have learned so far, these two operations are not necessarily needed since their functionalities can be implemented by simply using `INSERT` and `DELETE` operations. However, they do provide a more convenient and effective choice when needed.

The `LOAD` operation copies all the triples of a remote graph into the specified target graph. If no target graph is specified, the default graph will be used. For example, List 6.51 will load all the statements from my old FOAF document to my current FOAF document:

List 6.51 Example of using `LOAD` operation to update a given graph

```
LOAD <http://www.liyangyu.com/foafOld.rdf>  
INTO <http://www.liyangyu.com/foaf.rdf>
```

The `CLEAR` operation deletes all the statements from the specified graph. If no graph is specified, it will operate on the default graph. Note that this operation does not remove the graph from the RDF graph store. For example, List 6.52 will delete all the statements from my old FOAF document:

List 6.52 Example of using `CLEAR` operator to update a given graph

```
CLEAR GRAPH <http://www.liyangyu.com/foafOld.rdf>
```

6.5.3.4 Graph Management: Graph Creation

As we have mentioned earlier, graph management operations create and destroy named graphs in the graph store. Note that, however, these operations are optional since based on the current standard, graph stores are not required to support named graphs.

The following is used to create a new named graph:

```
CREATE [SILENT] GRAPH <uri>
```

This creates a new empty graph whose name is specified by *uri*. After the graph is created, we can manipulate its content by adding new statements into it, as we have discussed in previous sections.

Note that the optional `SILENT` keyword is for error handling. If the graph named *uri* already exists in the store, unless this keyword is present, the SPARQL 1.1 Update service will flag an error message back to the user.

6.5.3.5 Graph Management: Graph Removal

The following operation will remove the specified named graph from the graph store:

```
DROP [SILENT] GRAPH <uri>
```


Once this operation is successfully completed, the named graph cannot be accessed with further operations. Similarly, SPARQL 1.1 Update service will report an error message if the named graph does not exist. If the optional `SILENT` keyword is present, no error message will be generated.

6.6 Summary

We have covered SPARQL in this chapter, the last core technical component of the Semantic Web.

First off, understand how SPARQL fits into the technical structure of the Semantic Web and how to set up and use a SPARQL endpoint to submit queries against RDF models.

Second, understand the following main points about SPARQL query language:

- basic SPARQL query language concepts such as triple pattern and graph pattern;
- basic SPARQL query forms such as `SELECT` query, `ASK` query, `DESCRIBE` query and `CONSTRUCT` query;
- key SPARQL language features and constructs, and use them effectively to build queries, including working with multiple graphs.

Finally, this chapter has also covered SPARQL 1.1, a collection of new features added to the current SPARQL standard. Make sure you understand the following about SPARQL 1.1:

- the SPARQL 1.1 technical components;
- the language features and constructs of SPARQL 1.1 Query, including aggregate functions, subqueries, negation, expressions with `SELECT` query, and property paths;
- the language features and constructs of SPARQL 1.1 Update, including inserting and deleting operations on a single graph, creating and deleting graphs from a graph store.

At this point in the book, we have covered all the core technical components of the Semantic Web. The next five chapters will give you some concrete examples of the Semantic Web at work, which will further enhance your understanding about the materials presented so far in the book.

Chapter 7

FOAF: Friend of a Friend

At this point, we have learned the major technical components of the Semantic Web, and it is time for us to take a look at some real-world examples. Starting from FOAF is a good choice since it is simple and easy to understand, yet it does tell us a lot about how the Semantic Web looks like, especially in the area of social networking.

Studying FOAF will also give us a chance to practice what we have learned about RDF, RDFS and OWL. Another good reason is that FOAF namespace shows up in many ontology documents and in many literatures; understanding FOAF seems to be necessary.

As usual, we will first examine what exactly is FOAF and what it accomplishes for us. Then we will dive inside FOAF to see how it works. Finally we will take a look at some real examples and also come up with our own FOAF document.

Another interesting topic we will cover in this chapter is semantic markup. As you will see, semantic markup is the actual implementation of the idea of adding semantics to the current Web so as to make it machine readable. However, once you understand semantic markup, you will see the issues associated with it. The possible solutions to these issues will be covered in later chapters, and they will give you even more chances to further understand the idea of the Semantic Web.

7.1 What Is FOAF and What It Does

7.1.1 FOAF in Plain English

In the early days of the Semantic Web, developers and researchers were eager to build some running examples of the Semantic Web for the purpose of experimenting with the idea and hopefully showing the benefits of the Semantic Web. Yet, as we have seen in the previous chapters, to build applications on the Semantic Web, we need to have some ontologies, and we will have to mark up Web documents by using these ontologies so that we can turn them into the documents that are machine readable.

Obviously, in order to promptly create such an application example, it would be easier to focus on some specific domain, so the creation of the ontologies would be constrained in scope and would not be too formidably hard. In addition, to rapidly

yield a large number of Web documents that would be created by using this specific ontology, it would have to involve a lot of people who were willing to participate in the effort. Therefore, a fairly straightforward project to start with would be some people-centric Semantic Web application.

There are tons of millions of personal Web pages on the Web. On each such Web site, the author often provides some personal information, such as e-mails, pictures, interests. The author may also include some links to his/her friends' Web sites, therefore creating a social network. And with this network, we can answer questions such as "who has the same interest as I do?", and maybe that means we can sell our old camera to him. And also, we can find someone who lives close to us and also works at roughly the same location, so we can start to contact him and discuss the possibility of carpooling.

All these sound great. However, since all the personal Web sites are built for human eyes, we will have to do all the above manually, and it is very hard to create any application to do all that for us.

To make these documents understandable to an application, two major steps have to be accomplished: (1) a machine-readable ontology about person has to be created and (2) each personal home page has to be marked up, i.e., it has to be connected to some RDF statement document written by using this ontology.

This was the motivation behind FOAF project. Founded by Dan Brickley and Libby Miller in the mid 2000, FOAF is an open community-lead initiative with the goal of creating a machine-readable Web of data in the area of personal home pages and social networking.

It is important to understand the concept of "machine-readable Web of data." Just like the HTML version of your home page, FOAF documents can be linked together to form a Web of data. The difference is that this web of data is formed with well-defined semantics, expressed in the person ontology. We will definitely come back to this point later in this chapter and in the coming chapters as well.

In plain English, FOAF is simply a vocabulary (or, ontology) which includes the basic terms to describe personal information, such as who you are, what you do, and who your friends are. It serves as a standard for everyone who wants to mark up their home pages and turn them into the documents that can be processed by machines.

7.1.2 FOAF in Official Language

First off, FOAF stands for *Friend of a Friend*, and its official Web site can be found at

<http://www.foaf-project.org/>

which has an official definition of FOAF:

The Friend of a Friend (FOAF) project is creating a Web of machine-readable pages describing people, the links between them and the things they create and do.

This definition should be clear enough based on our discussion so far. Again, you can simply understand FOAF as a machine-readable ontology describing persons,

their activities, and their relations to other people. Therefore, FOAF and FOAF ontology are interchangeable concepts.

Note that FOAF ontology is not a standard from W3C; it is managed by following the style of an Open Source¹ or Free Software² project standards and maintained by a community of developers. However, FOAF does depend on W3C standards, such as RDF and OWL. More specifically,

- FOAF ontology is written in OWL.
- FOAF documents must be well-formed RDF documents.

FOAF ontology's official specification can be found at the location

`http://xmlns.com/foaf/spec/`

New updates and related new releases can be found at this page as well. In addition, the FOAF ontology itself can be found (and downloaded) from the following URL:

`http://xmlns.com/foaf/spec/index.rdf`

As usual, FOAF ontology is a collection of terms and all these terms are identified by pre-defined URIs, which all share the following leading string:

`http://xmlns.com/foaf/0.1/`

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

Finally, there is also a wiki site for FOAF project, and here is the URL for this site:

`http://wiki.foaf-project.org/w/Main_Page`

and you can use this wiki to learn more about FOAF project as well.

7.2 Core FOAF Vocabulary and Examples

With what we have learned so far, and given the fact that FOAF ontology is written in OWL, understanding FOAF ontology should not be difficult. In this section, we will cover the core terms in this ontology and also present examples to show how the FOAF ontology is used.

7.2.1 *The Big Picture: FOAF Vocabulary*

FOAF terms are grouped in categories. Table 7.1 summarizes these categories and the terms in each category. Note that FOAF is also under constant change and

¹<http://www.opensource.org/>

²<http://www.gnu.org/philosophy/free-sw.html>

Table 7.1 FOAF vocabulary

Category	Terms
Basic FOAF classes and properties	foaf:Agent, foaf:Person, foaf:name, foaf:nick, foaf:title, foaf:homepage, foaf:mbox, foaf:mbox_shalsum, foaf:img, foaf:depiction, foaf:depict, foaf:surname, foaf:familyName, foaf:givenName, foaf:firstName, foaf:lastName.
Properties about personal information	foaf:weblog, foaf:knows, foaf:interest, foaf:currentProject, foaf:pastProject, foaf:plan, foaf:based_near, foaf:age, foaf:workplaceHomepage, foaf:workInfoHomepage, foaf:schoolHomepage, foaf:topic_interest, foaf:publications, foaf:geekcode, foaf:myersBriggs, foaf:dnaChecksum
Classes and properties about online accounts and instance messaging	foaf:OnlineAccount, foaf:OnlineChatAccount, foaf:OnlineEcommerceAccount, foaf:OnlineGamingAccount, foaf:account, foaf:accountServiceHomepage, foaf:accountName, foaf:icqChatID, foaf:msnChatID, foaf:jabberID, foaf:yahooChatID, foaf:skypeID
Classes and properties about projects and groups	foaf:Project, foaf:Organization, foaf:Group, foaf:member, foaf:membershipClass
Classes and properties about documents and images	foaf:Document, foaf:Image, foaf:PersonalProfileDocument, foaf:topic, foaf:page, foaf:primaryTopic, foaf:primaryTopicOf, foaf:tipjar, foaf:shal, foaf:made, foaf:maker, foaf:thumbnail, foaf:logo

update; it will not be surprising at the time you read this book that you may find more terms in some categories.

As you can see, FOAF ontology is not a big ontology at all, and most of the terms are quite intuitive. Note that a term starting with capital letter identifies a class; otherwise, it identifies a property.

7.2.2 Core Terms and Examples

It is not possible to cover all the FOAF terms in detail. In this section, some most frequently used terms will be discussed, with the rest of them left for you to study.

`foaf:Person` class is one of the core classes defined in FOAF vocabulary, and it represents people in the real world. List 7.1 is the definition of `Person` class, taken directly from the FOAF ontology.

List 7.1 Definition of `Person` class

```
<rdfs:Class rdf:about="http://xmlns.com/foaf/0.1/Person"
            rdfs:label="Person"
            rdfs:comment="A person."
            vs:term_status="stable">
  <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#Class"/>
  <rdfs:subClassOf>
    <owl:Class rdf:about="http://xmlns.com/wordnet/1.6/Person"/>
  </rdfs:subClassOf>
  <rdfs:subClassOf>
    <owl:Class rdf:about="http://xmlns.com/foaf/0.1/Agent"/>
  </rdfs:subClassOf>
  <rdfs:subClassOf>
    <owl:Class rdf:about="http://xmlns.com/wordnet/1.6/Agent"/>
  </rdfs:subClassOf>
  <rdfs:subClassOf>
    <owl:Class rdf:about=
      "http://www.w3.org/2000/10/swap/pim/contact#Person"/>
  </rdfs:subClassOf>
  <rdfs:subClassOf>
    <owl:Class rdf:about=
      "http://www.w3.org/2003/01/geo/wgs84_pos#SpatialThing"/>
  </rdfs:subClassOf>
  <rdfs:isDefinedBy rdf:resource="http://xmlns.com/foaf/0.1/" />
  <owl:disjointWith
    rdf:resource="http://xmlns.com/foaf/0.1/Document"/>
  <owl:disjointWith
    rdf:resource="http://xmlns.com/foaf/0.1/Organization"/>
  <owl:disjointWith
    rdf:resource="http://xmlns.com/foaf/0.1/Project"/>
</rdfs:Class>
```

As you can see, `foaf:Person` is defined as a sub-class of `Person` class defined in WordNet. WordNet is a semantic lexicon for the English language. It groups English words into sets of synonyms called synsets and provides short and general definitions, including various semantic relations between these synonym sets. Developed by Cognitive Science Laboratory of Princeton University, WordNet has two goals: first, to produce a combination of dictionary and thesaurus that is more intuitively usable and second, to support automatic text analysis and artificial intelligence applications.

During the past several years, WordNet finds more and more usage in the area of the Semantic Web, and FOAF class `foaf:Person` is a good example. By being a sub-class of `wordnet:Person`, FOAF vocabulary can fit into a much broader picture. For example, an application which only knows WordNet can also understand `foaf:Person` even if it has never seen FOAF vocabulary before.

By the same token, `foaf:Person` is also defined to be a sub-class of several outside classes defined by other ontologies, such as the following two classes:

```
http://www.w3.org/2000/10/swap/pim/contact#Person
http://www.w3.org/2003/01/geo/wgs84_pos#SpatialThing
```

Note that `foaf:Person` is a sub-class of `foaf:Agent`, which can represent a person, a group, a software, or some physical artifacts, and similar agent concept is also defined in WordNet. Furthermore, `foaf:Person` cannot be anything such as a `foaf:Document`, a `foaf:Organization`, or a `foaf:Project`.

Besides `foaf:Person` class, FOAF ontology has defined quite a few other classes; the goal is to include the main concepts that can be used to describe a person as a resource. You can read these definitions just as the way we have understood `foaf:Person`'s definition. For example, `foaf:Document` represents the things which are considered to be documents used by a person, such as `foaf:Image`, a sub-class of `foaf:Document`, since all images are indeed documents.

Properties defined by FOAF can be used to describe a person on a quite detailed level. For example, `foaf:firstName` is a property that describes the first name of a person. This property has `foaf:Person` as its domain, and <http://www.w3.org/2000/01/rdf-schema#Literal> as its value range. Similarly, `foaf:givenname` is the property describing the given name of a person, and it has the same domain and value range. Note that a simpler version of these two properties is the `foaf:name` property.

`foaf:homepage` property relates a given resource to its home page. Its domain is <http://www.w3.org/2002/07/owl#Thing>, and range is `foaf:Document`. It is important to realize that this property is an inverse functional property. Therefore, a given `Thing` can have multiple home pages; however, if two `Things` have the same home page, then these two `Things` are in fact the same `Thing`.

A similar property is the `foaf:mbox` property, which describes a relationship between the owner of a mailbox and a mailbox. This is also an inverse functional property; if two `foaf:Person` resources have the same `foaf:mbox` value, these two `foaf:Person` instances have to be exactly the same person. On the other hand, a `foaf:Person` can indeed own multiple `foaf:mbox` instances. We will come back to this important property soon.

Let us take a look at some examples, and we will also cover some other important properties in these examples.

First off, List 7.2 shows a typical description of a person.

List 7.2 Example of using `foaf:Person`

```
1: <rdf:RDF
1a:   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
2:   xmlns:foaf="http://xmlns.com/foaf/0.1/">
3:
4:   <foaf:Person>
5:     <foaf:name>Liyang Yu</foaf:name>
6:     <foaf:mbox rdf:resource="mailto:liyang910@yahoo.com"/>
```

```
7: </foaf:Person>
8:
9: </rdf:RDF>
```

List 7.2 simply says that there is a person, this person's name is Liyang Yu, and e-mail address is liyang910@yahoo.com.

The first thing to note is the fact that there is no URI to identify this person at all. More specifically, you don't see the following pattern where `rdf:about` attribute is used on `foaf:Person` resource:

```
<foaf:Person rdf:about="some_URI" />
```

This seems to have broken one of the most important rules we have for the world of the Semantic Web. This rule says, whenever you decide to publish some RDF document to talk about some resource on the Web (in this case, Liyang Yu as a `foaf:Person` instance), you need to use a URI to represent this resource, and you should always use the existing URI for this resource if it already has one.

In fact, List 7.2 is correct and this is done on purpose. This is also one of the important features a FOAF document has. Let us understand the reason here.

It is certainly not difficult to come up with a URI to uniquely identify a person. For example, I can use the following URI to identify myself:

```
<foaf:Person
  rdf:about="http://www.liyangyu.com/people#LiyangYu" />
```

The difficult part is how to make sure other people know this URI and when they want to add additional information about me, they can reuse this exact URI.

One solution comes from `foaf:mbox` property. Clearly, an e-mail address is closely related to a given person, and it is also safe to assume that this person's friends should all know this e-mail address. Therefore, it is possible to use an e-mail address to uniquely identify a given person, and all we need to do is to make sure if two people have the same e-mail address and these two people are in fact the same person.

As we have discussed earlier, FOAF ontology has defined `foaf:mbox` property as an inverse functional property, as shown in List 7.3.

List 7.3 Definition of `foaf:mbox` property

```
<rdf:Property rdf:about="http://xmlns.com/foaf/0.1/mbox"
  vs:term_status="stable"
  rdfs:label="personal mailbox"
  rdfs:comment="...">
  <rdf:type rdf:resource=
    "http://www.w3.org/2002/07/owl#InverseFunctionalProperty" />
  <rdf:type rdf:resource=
    "http://www.w3.org/2002/07/owl#ObjectProperty" />
  <rdfs:domain rdf:resource="http://xmlns.com/foaf/0.1/Agent" />
  <rdfs:range
    rdf:resource="http://www.w3.org/2002/07/owl#Thing" />
  <rdfs:isDefinedBy rdf:resource="http://xmlns.com/foaf/0.1/" />
</rdf:Property>
```


Now if one of my friends has the following descriptions in her FOAF document:

```
<foaf:Person>
  <foaf:nick>Lao Yu</foaf:nick>
  <foaf:title>Dr</foaf:title>
  <foaf:mbox rdf:resource="mailto:liyang910@yahoo.com" />
</foaf:Person>
```

An application that understands FOAF ontology will be able to recognize `foaf:mbox` property and conclude that this is exactly the same person as described in List 7.2. And apparently, among other extra information, at least we now know this person has a nick name called Lao Yu.

Clearly, property `foaf:mbox` has solved the problem of identifying a person as a resource: when describing a person, you don't have to find the URI that identifies this person and you certainly don't have to invent your own URI either, all you need to do is to make sure you include his/her e-mail address in your description, as shown here.

`foaf:mbox_sha1sum` is another property defined by FOAF vocabulary which functions just like `foaf:mbox` property. You will see this property quite often in related documents and literatures, so let us talk about it here as well.

As you can tell, the value of `foaf:mbox` property is a simple textual representation of your e-mail address. In other words, after you have published your FOAF document, your e-mail address is open to the public. This may not be what you wanted. For one thing, spam can influx your mailbox within a few hours. For this reason, FOAF provides another property `foaf:mbox_sha1sum`, which offers a different representation of your e-mail address. You can get this representation by taking your e-mail address and applying the SHA1 algorithm to it. The resulting representation is indeed long and ugly, but your privacy is well protected.

There are several different ways to generate the sha1 sum of your e-mail address, we will not cover the details here. Remember to use `foaf:mbox_sha1sum` as much as you can, and it is also defined as an inverse functional property, so it can be used to uniquely identify a given person.

Now let us move on to another important FOAF property `foaf:knows`. We use it to describe our relationships with other people, and it is very useful when it comes to building the social network using FOAF documents. Let us take a look at one example. Suppose part of my friend's FOAF document looks like the following:

```
<foaf:Person>
  <foaf:name>Connie</foaf:name>
  <foaf:mbox rdf:resource="mailto:connie@liyangyu.com" />
</foaf:Person>
```

If I want to indicate in my FOAF document that I know her, I can include the code in List 7.4 into my FOAF document.

List 7.4 Example of using foaf:knows property

```

1: <foaf:Person>
2:   <foaf:name>Liyang Yu</foaf:name>
3:   <foaf:mbox rdf:resource="mailto:liyang910@yahoo.com"/>
4:   <foaf:knows>
5:     <foaf:Person>
6:       <foaf:mbox rdf:resource="mailto:connie@liyangyu.com"/>
7:     </foaf:Person>
8:   </foaf:knows>
9: </foaf:Person>

```

This shows that I know a person who has an e-mail address given by `connie@liyangyu.com`. Again, since property `foaf:mbox` is used, a given application will be able to understand that the person I know has a name called `Connie`; note that no URI has been used to identify her at all.

Also note that you cannot assume `foaf:knows` property is a symmetric property; in other words, I know `Connie` does not imply that `Connie` knows me. If you check the FOAF vocabulary definition, you can see `foaf:knows` is indeed not defined as symmetric.

Perhaps the most important use of `foaf:knows` property is to connect FOAF files together. Often by mentioning other people (`foaf:knows`), and by providing a `rdfs:seeAlso` property at the same time, we can link different RDF documents together. Let us discuss this a little further at this point, and in the later chapters, we will see application built upon this relationships.

We have seen property `rdfs:seeAlso` already in previous chapters. It is defined in RDF schema namespace, and it indicates the fact that there is some additional information about the resource this property is describing. For instance, I can add one more line into List 7.1, as shown in List 7.5.

List 7.5 Example of using rdfs:seeAlso property

```

1: <rdf:RDF
1a:   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
2:   xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
3:   xmlns:foaf="http://xmlns.com/foaf/0.1/">
4:
5: <foaf:Person>
6:   <foaf:name>Liyang Yu</foaf:name>
7:   <foaf:mbox rdf:resource="mailto:liyang910@yahoo.com"/>
8:   <rdfs:seeAlso
8a:     rdf:resource="http://www.yuchen.net/liyang.rdf"/>
9: </foaf:Person>
10:
11: </rdf:RDF>

```

Line 8 says, if you want to know more about this `Person` instance, you can find it in the resource pointed by <http://www.yuchen.net/liyang.rdf>.

Here, the resource pointed to by <http://www.yuchen.net/liyang.rdf> is an old FOAF document that describes myself, but I can in fact point to a friend's FOAF document using `rdfs:seeAlso`, together with property `foaf:knows`, as shown in List 7.6.

List 7.6 Use `foaf:knows` and `rdfs:seeAlso` to link RDF documents together

```

1: <rdf:RDF
1a:   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
2:   xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
3:   xmlns:foaf="http://xmlns.com/foaf/0.1/">
4:
5:   <foaf:Person>
6:     <foaf:name>Liyang Yu</foaf:name>
7:     <foaf:mbox rdf:resource="mailto:liyang910@yahoo.com"/>
8:     <rdfs:seeAlso
9:       <foaf:knows>
10:        <foaf:Person>
11:          <foaf:mbox rdf:resource="mailto:connie@liyangyu.com"/>
12:          <rdfs:seeAlso
12a:            rdf:resource="http://www.liyangyu.com/connie.rdf"/>
13:        </foaf:Person>
14:      </foaf:knows>
15:    </foaf:Person>
16:
17:</rdf:RDF>

```

Now, an application sees the document shown in List 7.6 will move on to access the document identified by the following URI (line 12):

`http://www.liyangyu.com/connie.rdf`

and by doing so, FOAF aggregators can be built without the need for a centrally managed directory of FOAF files.

As a matter of fact, property `rdfs:seeAlso` is treated by the FOAF community as the hyperlink of the FOAF documents. More specifically, one FOAF document is considered to contain a hyperlink to another document if it has included `rdfs:seeAlso` property, and the value of this property is where this hyperlink is pointing to. Here, this FOAF document can be considered as a root HTML page, and `rdfs:seeAlso` property is just like a `<href>` tag contained in the page. It is through the `rdfs:seeAlso` property that a whole web of machine-readable metadata can be built. We will see more about this property and its important role in the chapters yet to come.

The last two FOAF terms we would like to discuss here are `foaf:depiction` and `foaf:depicts`. It is quite common that people will put their pictures on their Web sites. To help us add statements about the pictures into the related FOAF document, FOAF vocabulary provides two properties to accomplish this. The first property is the `foaf:depiction` property and second one is `foaf:depicts` property, make sure you know the difference between these two.

`foaf:depiction` property is a relationship between a thing and an image that depicts the thing. In other words, it makes the statement such as “this person (Thing) is shown in this image.” On the other hand, `foaf:depicts` is the inverse property; it is a relationship between an image and something that image depicts. Therefore, to indicate the fact that I have a picture, I should use line 9 as shown in List 7.7.

List 7.7 Example of using `foaf:depiction` property

```

1: <rdf:RDF
1a:   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
2:   xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
3:   xmlns:foaf="http://xmlns.com/foaf/0.1/">
4:
5:   <foaf:Person>
6:     <foaf:name>Liyang Yu</foaf:name>
7:     <foaf:mbox rdf:resource="mailto:liyang910@yahoo.com"/>
8:     <rdfs:seeAlso
8a:       rdf:resource="http://www.yuchen.net/liyang.rdf"/>
9:     <foaf:depiction rdf:resource=
9a:       "http://www.liyangyu.com/pictures/yu.jpg"/>
10:   </foaf:Person>
11:
12: </rdf:RDF>

```

I will leave it to you to understand the usage of `foaf:depicts` property.

Up to this point, we have talked about several classes and properties defined in the FOAF vocabulary. Again, you should have no problem reading and understanding the whole FOAF ontology. Let us move on to the topic of how to create your own FOAF document and also make sure that you know how to get into the “friend circle.”

7.3 Create Your FOAF Document and Get into the Friend Circle

In this section, we will talk about several issues related to creating your own FOAF document and joining the circle of friends. Before we can do all these, we need to know how FOAF project has designed the flow, as we will see in the next section.

7.3.1 How Does the Circle Work?

The circle of FOAF documents is created and maintained by the following steps.

Step 1. A user creates the FOAF document.

As a user, you create a FOAF document by using the FOAF vocabulary as we discussed in the previous section. The only thing you need to remember is that you should use `foaf:knows` property together with `rdfs:seeAlso` property to connect your document with the documents of other friends.

Step 2. Link your home page to your FOAF document.

Once you have created your FOAF document, you should link it from your home page. And once you have finished this step, you are done, it is now up to the FOAF project to find you.

Step 3. FOAF uses its crawler to visit the Web and collect all the FOAF documents.

In the context of FOAF project, a crawler is called a *scutter*. Its basic task is not much different from a crawler: it visits the Web and tries to find RDF files. In this case, it has to find a special kind of RDF file: a FOAF document. Once it finds one, the least it will do is to parse the document and store the triples into its data system for later use.

An important feature about *scutter* is that it has to know how to handle `rdfs:seeAlso` property. Whenever the *scutter* sees this, it will follow the link to reach the document pointed by `rdfs:seeAlso` property. This is the way FOAF uses to construct a network of FOAF documents.

Another important fact about *scutter* is that it has to take care of the data merging issue. To do so, the *scutter* has to know which FOAF properties can uniquely identify resources. More specifically, `foaf:mbox`, `foaf:mbox_sha1sum` and `foaf:homepage` are all defined as inverse functional properties; therefore, they can all uniquely identify individuals that have one of these properties. In the real operation, one solution the *scutter* can use is to keep a list of RDF statements which involve any of these properties, and when it is necessary, it can consult this list to merge together different triples that are in fact describing the same individuals.

Step 4. FOAF maintains a central repository and is also responsible for keeping the information up to date.

FOAF also has to maintain a centralized database to store all the triples it has collected and other relevant information. To keep this database up to date, it has to run the *scutter* periodically to visit the Web.

Step 5. FOFA provides a user interface so that we can find our friends and conduct other interesting activities.

FOAF offers some tools one can use to view the friends in the circle, which further defines the look and feel of the FOAF project. Among these tools, FOAF explorer is quite popular, and you can find this tool as the following location:

```
http://xml.mfd-consult.dk/foaf/explorer/
```

Figure 7.1 is an example of viewing FOAF document using FOAF explorer. The FOAF document being viewed is created by Dan Brickley, one of the founders of the FOAF project.

Up to this point, we have gained understanding about how FOAF project works to build a network of FOAF documents. It is time to create our own FOAF document and join the circle.

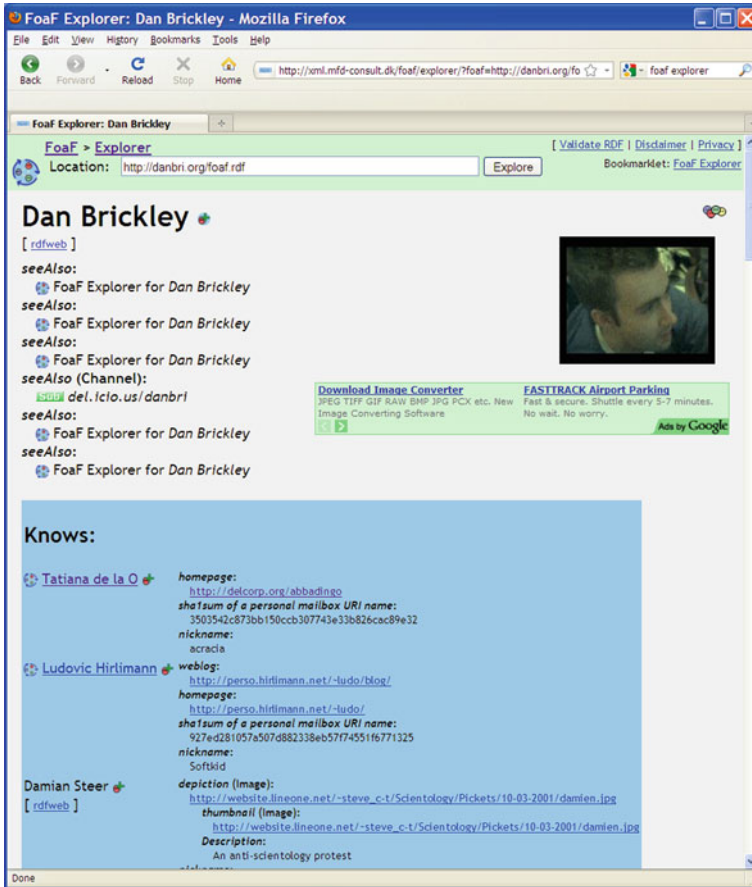


Fig. 7.1 FOAF explorer shows Dan Brickley’s FOAF document

7.3.2 Create Your FOAF Document

The most straightforward way to create a FOAF document is to use a simple text editor. This requires you to directly use the FOAF vocabulary. Given the self-explanatory nature of the FOAF ontology, this is not difficult to do. Also you need to validate the final document, just to make sure its syntax is legal.

The other choice is to use tools to create FOAF document. The most popular one is called “FOAF-a-matic”; you can find the link to this tool from the FOAF official Web site, and at the current time, its URL is

`http://www.ldodds.com/foaf/Foaf-a-matic.html`

Figure 7.2 shows the main interface of this authoring tool.

To use this form, you don’t have to know any FOAF terms, you just need to follow the instructions to create your FOAF document. More specifically, this form allows you to specify your name, e-mail address, home page, your picture and phone

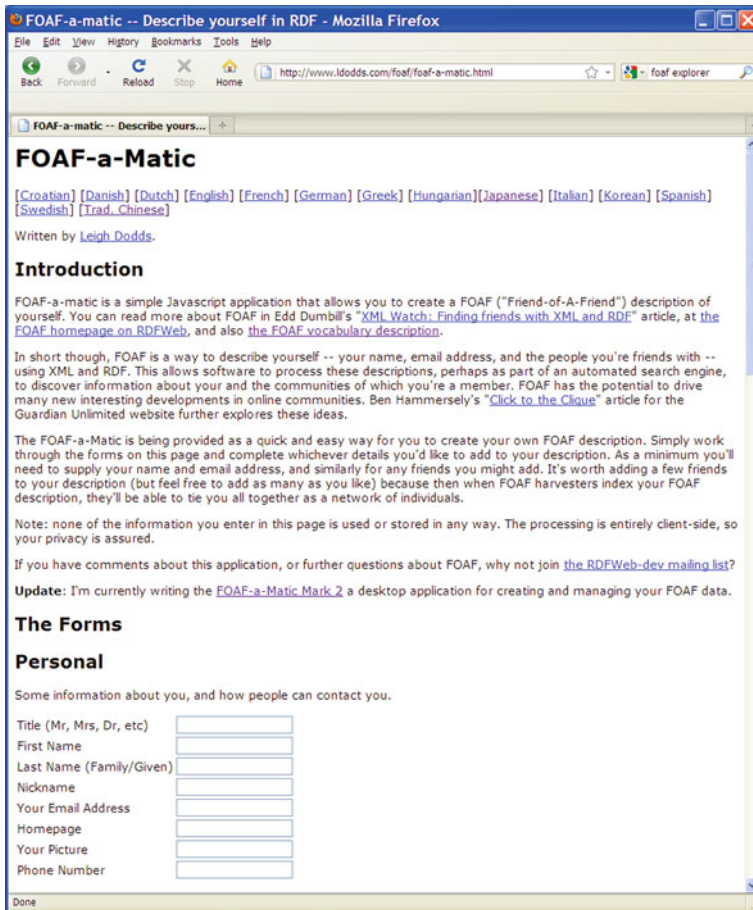


Fig. 7.2 Use FOAF-a-matic to create your own FOAF document

number, and other personal information. It also allows you to enter information about your work, such as work home page and a small page describing what you do at your work. More importantly, you will have a chance to specify your friends and provide their FOAF documents as well. Based on what we have learned so far, this will bring both you and your friends into the FOAF network.

Note that you can leave a lot of fields on the form empty. The only required fields are “First Name,” “Last Name,” and “Your Email Address.” By now, you should understand the reason why you have to provide an e-mail address – FOAF does not assign an URI to you at all, and later on in life, it will use this e-mail address to uniquely identify you.

Once you have finished filling the form, by clicking the “FOAF me!” button, you will get an RDF document which uses FOAF vocabulary to present a description about yourself. At this point, you need to exercise your normal “copy-and-paste” trick in the output window, copy the generated statements into your favorite editor,

and save it to a file so that you can later on join the circle of friends, as will be discussed next.

7.3.3 *Get into the Circle: Publish Your FOAF Document*

Once you have created your FOAF document, the next step is to publish it in a way that it can be easily harvested by the scutter (FOAF's crawler) or other applications that can understand FOAF documents. This is what we mean when we say "get into the circle." There are three different ways to get into the circle, and we will discuss these different methods in this section.

- Add a link from you home page to your FOAF document

The easiest solution is to link your home page to your FOAF document. This can be done using the `<link>` element as shown in List 7.8.

List 7.8 Add a link from your home page to your FOAF document

```

<!-- this is your homepage -->
<html>
<head>
... ..
<link rel="meta" type="application/rdf+xml" title="FOAF"
      href="http://www.liyangyu.com/foaf.rdf"/>
... ..
</head>
<body>
... ..
</body>
</html>

```

Remember to substitute `href` to point to your own FOAF document. Also note that your FOAF file can be any name you like, but `foaf.rdf` is a common choice.

This is quite easy to implement; however, the downside is the fact that you have to wait for the crawler to visit your home page to discover your FOAF document. Without this discovery, you will never be able to get into the circle. Given the fact that there are millions of personal Web sites out there on the Web, the FOAF scutter will have to traverse the Web for long time to find you, if it can find you at all.

To solve this problem, you can use the second solution, which will make the discovery process much more efficient.

- Ask your friend to add a `rdfs:seeAlso` link that points to your document

This is a recommended way to get your FOAF document indexed. Once your friend has added a link to your document by using `rdfs:seeAlso` in his/her document, you can rest assured that your data will appear in the network.

To implement this, your friend needs to remember that he/she has to use `foaf:knows` and `rdfs:seeAlso` together by inserting the following lines into his/her FOAF document:

```
<foaf:knows>
  <foaf:Person>
    <foaf:mbox rdf:resource="mailto:you@yourEmail.com"/>
    <rdfs:seeAlso rdf:resource="http://
      path_to_your_foaf.rdf"/>
  </foaf:Person>
</foaf:knows>
```

Now, the fact that your friend is already in the circle means that FOAF scutter has visited his/her document already. Since the scutter will periodically revisit the same files to pick up any updates, it will see the `rdfs:seeAlso` link and will then pick up yours; this is the reason why your FOAF document will be guaranteed to be indexed.

Obviously, this solution is feasible only when you have a friend who is already in the circle. What if you do not have anyone in the circle at all? We will then need the third solution discussed next.

- Use the “FOAF Bulletin Board”

Obviously, instead of waiting for FOAF network to find you, you can report to it voluntarily. FOAF project does provide a service for you to do this, and it is the so-called FOAF Bulletin Board. To access this service, visit the FOAF Wiki site, and find the FOAF Bulletin Board page. You can also use the following URL to directly access the page:

<http://wiki.foaf-project.org/w/FOAFBulletinBoard>

Once you are on the page, you will see a registry of people whose FOAF document has been collected by FOAF. To add your own FOAF document, you need to log in first. Once you log in, you will see an `Edit` tab. Click this tab, you will then be able to edit a document in the editing window. Add your name, and a link to your FOAF document, click “save page” when you are done, and you are in the FOAF network already.

There are other ways you can use to join the circle, and we are not going to discuss them here. A more interesting question at this point is, what does the FOAF world look like, especially after more and more people have joined the circle? In other words, how does FOAF project change the world of personal Web pages for human eyes into a world of personal Web pages that are suitable for machine processing? Let us take a look at this interesting topic in the next section.

7.3.4 From Web Pages for Human Eyes to Web Pages for Machines

Let us take a look at the world of personal Web pages first. Assuming in my Web page, www.liyangyu.com, I have included links pointing to my friends' Web sites. One of my friends, on his Web site, has also included links that point to his friends, so on and so forth. This has created a linked documents on the Web, just as what we have today.

Now using FOAF vocabulary, I have created a FOAF document that describes myself. Quite similar to my personal Web site, in this FOAF document, I talk about myself, such as my e-mail, my name, my interest. Yet there is a fundamental difference: when I talk about myself in this FOAF document, I have used a language that machine can understand. For the machine, this FOAF document has become my new personal home page; it might look ugly to human eyes, but it looks perfectly understandable to machines.

Now, assuming that all my friends have created their machine-readable home pages, and just like what I have done in my human-readable home page, I can now put links that point to my friends' FOAF documents in my machine-readable home page. This is done by using `foaf:knows` together with `rdfs:seeAlso` property. Furthermore, this is also true for all my friends: in their machine-readable home pages, they can add links to their friends' machine-readable home pages, so on and so forth.

This will then create a brand new social network on the Web, co-existing with the traditional linked documents on the current Web. This whole new network is now part of the Semantic Web in the domain of human network.

The above two different Web networks are shown in Fig. 7.3. By now, probably two things have become much more clear. First, the reason why FOAF is called

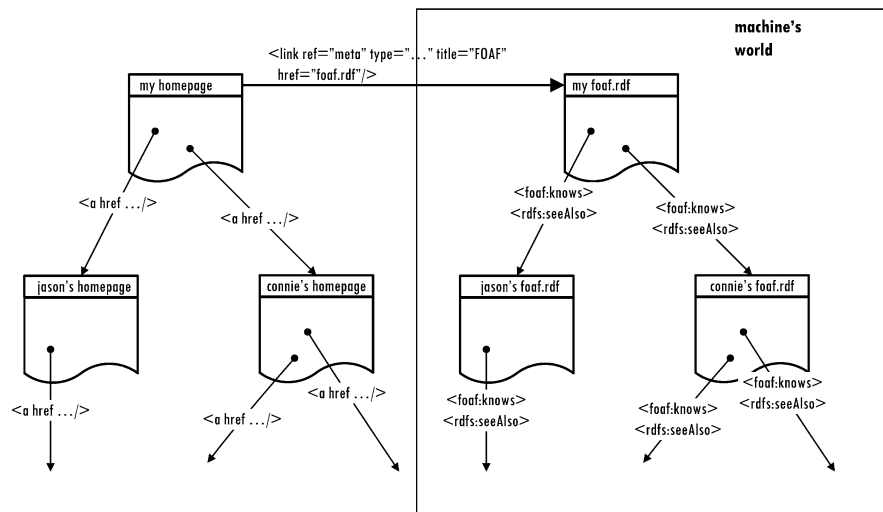


Fig. 7.3 Home pages for human eyes vs. home pages for machines

“Friend of a Friend” has become clearer; and second, the reason why `foaf:knows` together with `rdfs:seeAlso` is considered by the FOAF community the hyperlink of the FOAF documents has become clear as well.

7.4 Semantic Markup: a Connection Between the Two Worlds

Before we move on to continue exploring the Semantic Web world, we need to talk about one important issue: semantic markup.

7.4.1 What Is Semantic Markup

So far in this book, we have used the phrase semantic markup quite a few times already. So, what exactly is semantic markup? How does it fit into the whole picture of the Semantic Web?

First of all, after all these chapters, we have gained a much better understanding about the Semantic Web. In fact, at this moment if we had to use one simple sentence to describe what exactly the Semantic Web is, it would be really simple: it is all about extending the current Web to make it more machine understandable.

To accomplish this goal, we first need some language(s) to express meanings that machine can understand. This is one of the things we have learned the most at this point: we have covered RDF, RDFS and OWL. These languages can be used to develop a formal ontology and create RDF documents that machine can process. And as we have learned in this chapter, we used FOAF ontology to create RDF documents that describe myself. Obviously, we can use other ontologies in other domains to create more and more RDF documents that describe resources in the world.

However, when we look at our goal and what we have accomplished so far, we realize the fact that there is something missing: the current Web is one world, the machine-readable semantics expressed by ontologies is another world, and where is the connection between these two? If these two worlds always stand independent of each other, there will be no way we can extend the current Web to make it more machine readable.

Therefore, we need to build a connection between the current Web and the semantic world. This is what we call “adding semantics to the current Web.”

As you might have guessed, adding semantics to the current Web is called *semantic markup*; sometimes, it is also called *semantic annotation*.

7.4.2 Semantic Markup: Procedure and Example

In general, a semantic markup file is an RDF document containing RDF statements which describe the content of a Web page by using the terms defined in one or several ontologies. For instance, suppose a Web page describes some entities in the real world, the markup document for this Web page may specify that these entities

are instances of some classes defined in some ontology, and these instances have some properties and some relationships among them.

When an application reaches a Web page and somehow finds this page has a markup document (more details on this later), it will read this markup file and will also load the related ontologies into its memory. At this point, the application can act as if it understands the content of the current Web page, and it can also discover some implicit facts about this page. The final result is that the same Web page not only continues to look great to human eyes but also makes perfect sense to machines.

More specifically, there are several steps you need to follow when semantically marking up a Web page:

Step 1. Decide which ontology or ontologies to use for semantic markup.

The first thing is to decide which ontology to use. Sometimes, you might need more than one ontologies. This involves reading and understanding the ontology to decide whether the given ontology fits your need, or, whether you agree with the semantics expressed by the ontology. It is possible that you have to come up with your own ontology; in that case, you need to remember the rule of always trying to reuse existing ontologies, or simply constructing your new ontology by extending some given ontology.

Step 2. Mark up the Web page.

Once you have decided the ontology you are going to use, you can start to mark up the page. At this point, you need to decide exactly what content on your page you want to mark up. Clearly, it is neither possible nor necessary to mark up everything on your page. Having some sort of application in your mind would help you to make the decision. The question you want to ask yourself is, for instance, if there were an application visiting this page, what information on this page I want the agent to understand? Remember your decision is also constrained by the ontology you have selected; the markup statements have to be constructed based upon the ontology, therefore, you can only mark up the contents that are supported by the selected ontology.

You can elect to create your markup document by using a simple editor or by using some tools. Currently there are tools available to help you to mark up your pages, as we will see in our markup examples later in this chapter. If you decide to use a simple editor to manually mark up a Web page, remember to use a validator to make sure your markup document at least does not contain any syntax errors. The reason is simple: the application that reads this markup document may not be as forgiving as you are hoping; if you make some syntax mistakes, a lot of your markup statements can be totally skipped and ignored.

After you have finished creating the markup document, you need to put it somewhere on your Web server. You also need to remember to grant enough rights to it so that the outside world can access it. This is also related to the last step discussed below.

Step 3. Let the world know your page has a markup document.

The last thing you need to do is to inform the world that your page has a markup document. At the time of this writing, there is no standard way of accomplishing this. A popular method is to add a link in the HTML header of the Web page, as we have seen in this chapter when we discuss the methods we can use to publish FOAF documents (see List 7.8).

With all these said, let us take a look at one example of semantic mark up. My goal is to mark up my own personal home page, www.liyangyu.com, and to do so, we will follow the steps discussed earlier.

The first step is to choose an ontology for markup. Clearly, my home page is all about a person, so quite obviously we are going to use FOAF ontology. We might need other vocabularies down the road, but for now, we will settle down with FOAF ontology only.

The second step is to create the markup document. With all the examples we have seen in this chapter, creating this document should not be difficult at all; it is simply an RDF document that describes me as a resource by using the terms defined in FOAF ontology.

Again, it is up to us to decide what content in the page should be semantically marked up. As a starter, List 7.9 shows a possible markup document:

List 7.9 A markup document for my home page

```

1: <?xml version="1.0" encoding="UTF-8"?>
2: <rdf:RDF
3a:     xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
4:     xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
5:     xmlns:dc="http://www.purl.org/metadata/dublin-core#"
6:     xmlns:foaf="http://xmlns.com/foaf/0.1/">
7: <rdf:Description rdf:about="http://www.liyangyu.com">
8:   <rdf:type
9a:     rdf:resource="http://xmlns.com/foaf/0.1/Document"/>
9:   <dc:title>liyang yu's home page</dc:title>
10:  <dc:creator
10a:    rdf:resource="http://www.liyangyu.com/foaf.rdf#liyang"/>
11: </rdf:Description>
12:
13: <rdf:Description
13a:   rdf:about="http://www.liyangyu.com/foaf.rdf#liyang">
14:
15:   <rdf:type rdf:resource="http://xmlns.com/foaf/0.1/Person"/>
16:   <foaf:name>liyang yu</foaf:name>
17:   <foaf:title>Dr</foaf:title>
18:   <foaf:givenname>liyang</foaf:givenname>
19:   <foaf:family_name>yu</foaf:family_name>
20:   <foaf:mbox_sha1sum>
20a:     1613a9c3ec8b18271a8fe1f79537a7b08803d896

```

```

20b: </foaf:mbox_shalsum>
21: <foaf:homepage rdf:resource="http://www.liyangyu.com"/>
22:
23: <foaf:workplaceHomepage
23a:     rdf:resource="http://www.delta.com"/>
24: <foaf:topic_interest
24a:     rdf:resource="http://dbpedia.org/resource/Semantic_Web"/>
25: <foaf:knows>
26:     <foaf:Person>
27:         <foaf:mbox rdf:resource="mailto:connie@liyangyu.com"/>
28:         <rdfs:seeAlso rdf:resource=
28a:             "http://www.liyangyu.com/connie.rdf#connie"/>
29:     </foaf:Person>
30: </foaf:knows>
31:
32: </rdf:Description>
33:
34: </rdf:RDF>

```

As you can tell, this document contains some basic information about myself, and it also includes a link to a friend I know. With what we have learned in this chapter, this document should be fairly easy to follow, and not much explanation is needed.

Now imagine an application that comes across my home page. By reading List 7.9, it will be possible to understand the following facts (not a complete list):

- Resource identified by <http://www.liyangyu.com> is a foaf: Document instance, and it has a dc:title whose value is liyang yu's home page, and a resourced named <http://www.liyangyu.com/foaf.rdf#liyang> has created this document.
- <http://www.liyangyu.com/foaf.rdf#liyang> is a foaf:Person-type resource; its foaf:homepage is identified by <http://www.liyangyu.com>
- <http://www.liyangyu.com/foaf.rdf#liyang> has these properties defined: foaf:name, foaf:title, foaf:mbox_shalsum, etc.
- <http://www.liyangyu.com/foaf.rdf#liyang> also foaf:knows another foaf:Person instance, whose foaf:mbox property is given by the value of connie@liyangyu.com, etc.

Note that Dublin Core vocabulary is used to identify the page title and page author (lines 9–10), and the URI that represents the concept of the Semantic Web is also reused (line 24). Again, this URI is coined by the DBpedia project, which will be discussed in [Chap. 10](#).

Now we are ready for the last step: explicitly indicate the fact that my personal Web page has been marked up by an RDF file. To do so, we can use the first solution presented in [Sect. 7.3.3](#). In fact, by now, you should realize the fact that a FOAF document can be considered as a special markup to a person's home page, and all I have done here was simply creating a FOAF document for myself.

There are also tools available to help us mark up a given Web page. For example, SMORE is one of the projects developed by the researchers and developers in the University of Maryland at College Park, and you can take a look at their work from their official Web page:

<http://www.mindswap.org/>

SMORE allows the user to mark up Web documents without requiring a deep knowledge about OWL terms and syntax. You can create different instances easily by using the provided GUI; it is quite intuitive and straightforward. Also, SMORE lets you visualize your ontology, therefore it can be used as an OWL ontology validator as well.

We are not going to cover the details about how to use it; you can download it from the following Web site and experiment with it on your own:

<http://www.mindswap.org/2005/SMORE/>

If you use it for markup, your final result would be a generated RDF document that you can directly use as your markup document. You might want to make modification if necessary; but generally speaking, it is always a good idea to use a tool to create your markup file whenever it is possible.

7.4.3 Semantic Markup: Feasibility and Different Approaches

As we have mentioned earlier, the process of marking up a document is the process of building the critical link between the current Web and the machine-readable Web. It is the actual implementation of the so-called adding semantics to the current Web. However, as you might have realized already, there are lots of unsolved issues associated with Web page markup.

The first thing you might have noted is that no matter whether we have decided to mark up a page manually or by using some tools, it seems to be quite a lot of work just to mark up a simple Web page such as my personal home page. The question then is, how do we finish all the Web pages on the Web? Given the huge number of pages on the Web, it is just not practical. Also, it is not trivial at all to implement the markup process; a page owner has to learn at least something about ontology, OWL, and RDF among other things. Even all single-page owners agree to mark up their pages, how do we make sure everyone is sharing ontologies to the maximum extent without un-necessarily inventing new ones?

These thoughts have triggered the search for the so-called killer application in the world of the Semantic Web. The idea is that if we could build a killer Semantic Web application to demonstrate some significant benefit to the world, there will then be enough motivation for the page owners to mark up their pages. However, without the link between the current Web and the machine-readable semantics built, the killer application (whatever it is) simply cannot be created.

At the time of this writing, there is still no final call about this killer application yet. However, the good news is, there are at least some solutions to the above

dilemma, and in the upcoming chapters, we will be able to see examples of these solutions. For now, let us briefly introduce some of these solutions:

- Manually mark up in a much more limited domain and scope

Semantic markup by the general public on the whole Web seems to be too challenging to implement, but for a much smaller domain and scope, manual markup is feasible. For example, for a specific community or organization, their knowledge domain is much smaller; publishing and sharing a collection of core ontologies within the community or the organization is quite possible. If a relatively easier way of manually semantic markup is provided, it is then possible to build Semantic Web application for this specific community or organization.

One successful example along this line is semantic wiki. We will present one such example in [Chap. 9](#), where we will have a chance to see that manual markup, within a limited domain and scope, can indeed produce quite impressive results.

- Machine-generated semantic markup

There has been some research in this area, and some automatic markup solutions have been proposed. However, most of these techniques are applied to technical texts. For the Web that contains highly heterogeneous text types which are mainly made up by natural languages, there seems to be no efficient solution yet.

However, some Web content does already provide structured information as part of the content. Therefore, instead of parsing natural languages to produce markup files, machine can take advantage of this existing structured information and generate markup documents based on these structured data.

We will also see one example along this line as well, and it is the popular DBpedia project. We will learn more details later on, but for now, DBpedia is completely generated by machine, and the source for these machine-readable documents all come from the structured information contained in Wikipedia.

- Create a machine-readable Web all on its own

There seems to be one key assumption behind the previous two solutions: there has to be two formats for one piece of Web content, one for human viewing and one for machines.

In fact, do we really have to do this at all? If we start to publish machine-readable data, such as RDF documents, and somehow make all these documents connect to each other, just like what we have done when creating Web pages, then we will be creating a Linked Data Web! And since everything on this Linked Data Web is machine readable, we should be able to develop a lot of interesting applications as well, without any need to do semantic markup.

This is the idea behind the Linked Data Project, and we will study it as well in another future chapter.

At this point, we are ready to move on and to study the above three solutions in detail. The goal is twofold: one, to build more understanding about the Semantic Web and second, these solutions will be used as hints to you, and hopefully you will be able to come up and design even better solutions for the idea of the Semantic Web.

7.5 Summary

In this chapter, we have learned FOAF, an example of the Semantic Web in the domain of social networking.

The first thing we should understand from this chapter is the FOAF ontology itself. This includes its core terms and how to use these terms to describe people, their basic information, the things they do, and their relationships to other people.

It is useful to have your own FOAF document created. You should understand how to create it and how to publish it on the Web and further get into the “circle of trust.”

Semantic markup is an important concept. Not only you should be able to manually mark up a Web document but also you should understand the following about it:

- It provides a connection between the current Web and the collection of knowledge that is machine understandable.
- It is the concrete implementation of so-called “adding semantics to the current Web”.
- It has several issues regarding whether it is feasible in the real world. However, different solutions do exist, and these solutions have already given rise to different applications on the Semantic Web.

Chapter 8

Semantic Markup at Work: Rich Snippets and SearchMonkey

In last chapter we have studied FOAF project and the concept of semantic markup. Personal Web sites have made up a large portion of the current Web, and as we have discussed, FOAF ontology together with semantic markup has changed this large portion from linked Web documents into the Semantic Web.

The obvious and important difference between these two is the added semantic markup. However, exactly how the added semantic markup is going to be used by machine? This key question has not been answered in the last chapter.

In this chapter, we will answer this question by examples. These examples are real-world applications developed by major players such as Google and Yahoo!. More specifically, we will cover Rich Snippets by Google and SearchMonkey by Yahoo!. Once you are done with this chapter, you will gain more understanding about how semantic markup is done, and how semantic markup can be used by machines to change our experience on the Web.

8.1 Introduction

8.1.1 Prerequisite: How Does a Search Engine Work?

To understand the rest of this chapter, some knowledge about how search engine works is necessary. In this section, we will discuss the basic flow of a search engine. If you are familiar with this already, you can skip this section and move on to the next.

8.1.1.1 Basic Search Engine Tasks

It is safe to say that the majority of us have the experience of using a search engine, and probably most of us were also amazed by how fast a search engine can react. For instance, right after a user enters the keywords and hits the search button, the results will be returned, and it even tells the user how long it takes to finish the search (some very tiny fraction of a second). Clearly, it is impossible to search the Web each time a query is submitted and return the results within such a short period of time. So what has happened behind the scene?

The truth is, instead of searching the Web on the fly, every time when a user query is submitted, a search engine queries a highly optimized database of Web pages. In addition, this database is created ahead of the time, by something called *crawler* (or *spider*), i.e., a piece of software that is capable of traversing the Web by downloading Web pages and following links from page to page. The final result is a database that holds the Web is created.

There are many search engines available on the market, and you are probably familiar with at least one of these engines: Google, Yahoo!, Bing, MSN Search, etc. There are differences in the ways these search engines work, but the following three basic tasks are all the same:

- They search the Web or selected pieces of the Web based on important words.
- They maintain an index of the words they find and where they find them.
- They allow users to look for words or combinations of words found in the index databases.

The most important measure for a search engine is its search performance and the quality of the search results. The ability to crawl and index the Web efficiently and effectively is also important for a search engine. The main goal is to provide quality results, given the rapid grow of the current Web.

8.1.1.2 Basic Search Engine Workflow

A search engine's workflow can be summarized by three words: crawling, indexing, and searching. To understand how a search engine works, we need to understand the action behind these three words.

- **Crawling**

A search engine's life starts from crawling. Clearly, before a search engine can tell us anything at all, it must know where everything is in advance. A crawler is used for this purpose. More specifically, a *URL server* sends a list of URLs to the crawler for it to visit. This list of URLs is viewed as the *seed URLs* – the URLs that we want the crawler to start with. For each URL, the crawler downloads the Web document on this URL and finds all the hypertext links on that page that point to other Web pages. It then picks one of these new links and follows that link to download a new page, and finds more links on the new page, so on and so forth, until it decides to stop or there is no more links to follow. As a summary, the following are the main tasks of a given crawler:

1. download the Web page;
2. parse through the downloaded page and retrieve all its links;
3. for each new link retrieved, repeat steps 1 and 2.

Note that in real life, a search engine normally has a number of crawlers that work simultaneously to make a more efficient visit on the Web. And certainly, the URL server is responsible for providing URL lists to all these crawlers.

- Indexing

What about the Web documents downloaded by the crawlers? The crawlers only extract the URL links on the pages and further download the documents; they do not conduct any processing work on these documents. Instead, the downloaded Web pages are sent to the *Store server*, which compresses and stores the pages into a *repository*. To use the information in the repository, the second part of a search engine's life starts: indexing process uses everything there is in the repository and prepares the quick responses a user sees when using the search engine.

More specifically, *indexer*, together with *sorter*, creates the database files we have mentioned earlier. The indexer performs several tasks. It fetches a document from the repository, decompresses it, and breaks it into a set of words. For each word, it creates a record that describes it. This record has quite a lot information, including the word itself, the document in which the word is found, the position of the word in the document, etc. The indexer then distributes these records into a set of *barrels*, creating a partially sorted forward index system.

The indexer has yet another important task: for each Web page, it again extracts all the links in this page and stores important information about these links. For example, for each link, this includes where this link points to, and the related text of the link. This information is aggregated into a file called *anchors*.

One of the main purposes of the anchors file is to provide information to compute the *rank* of each page. Clearly, if the word or words being queried has or have occurred in multiple pages, then in which order should these pages be returned to the user? Obviously, this has a vital influence on the quality of the search result. Intuitively, the page on the top of the returned list should be the most relevant page to the user.

It is certainly true that each search engine applies its own ranking algorithm to solve this problem (for example, the famous *PageRanking* method from Google), and the details of these algorithms are not known to the general public. Again, using Google as the example, a *URL Resolver* reads the anchors file and generates a *links* file, which is used solely to compute page ranks for all the documents.

- Searching

At this point, a search engine is ready for user's query. This is the most exciting part of a search engine's life and it is also the part that we, as users, can directly interact with. The basic workflow is quite intuitive: a *searcher* accepts the user's query, analyzes the query, and uses the barrels and page ranking algorithms to return the search results to the user.

This basic workflow of a search engine can be summarized as shown in Fig. 8.1. Note that this workflow description is based on Google's search engine model, and to make it more understandable, only the major components are mentioned. If you want to know more details, the best source will be the classic paper by Brin and Page (1998).

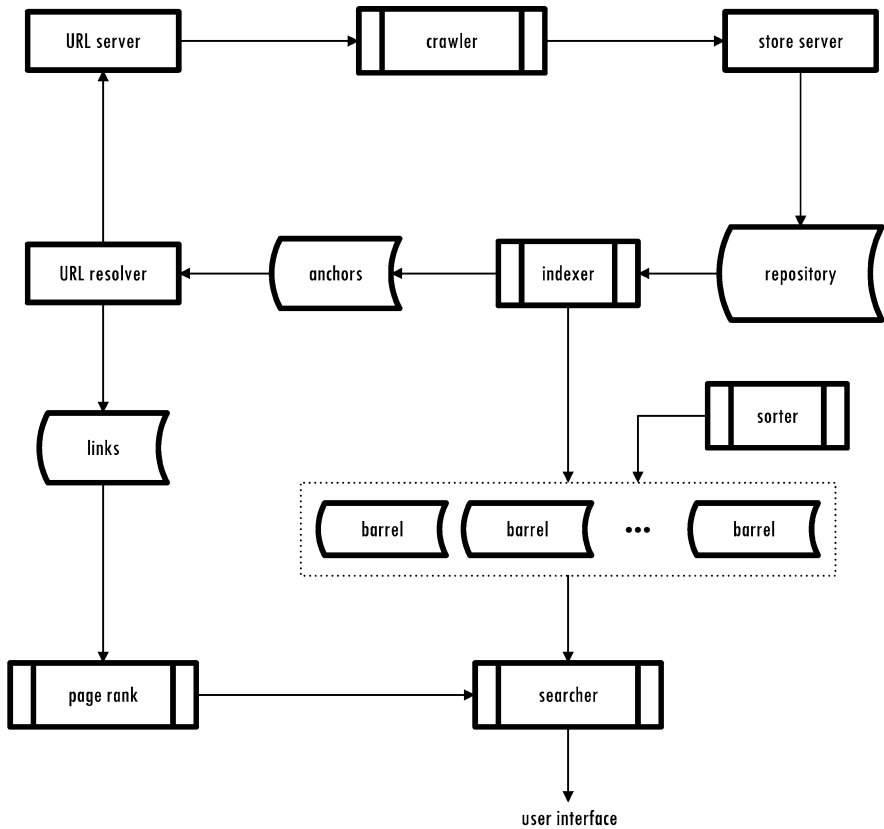


Fig. 8.1 A simplified search engine system architecture (based on Brin and Page 1998)

8.1.2 Rich Snippets and SearchMonkey

A search engine user's direct experience with any search engine comes from the search result page. This not only includes how relevant the results are to the original query but also includes the search result presentation. As far as the search result presentation is concerned, the goal is to summarize and describe each individual result page in such a way that it can give the user a clear indication of how relevant this current page is to the original search.

Based on our understanding about how a search engine works, improving the quality of search result presentation is not an easy task. In general, automatically generated abstract for a given page often provides a poor overview of the page. More specifically, when search engine indexes the words contained in a page, it normally picks up the text contained in the immediate area where the query keywords are found and returns this text as the page summary. This obviously is not a reliable way to describe an individual page. On top of this, any tables or images have to be removed from the page summary, since there are no reliable algorithms to automatically recognize and select the appropriate images and tables. The final result is that the page summaries on the search result page are not much a help to the end users.

Rich Snippets developed by Google and SearchMonkey developed by Yahoo! are improvements along this line. They both use the added semantic markup information contained in each individual page to improve search result display, with benefits to both search users and publishers of Web documents. As you will see when you finish this chapter, these are quite simple and elegant ideas. They do not require much from the publishers of the Web contents, yet they do produce quite impressive results.

To better understand this chapter, you do need some basic understanding about microformats and RDFa. You can find related materials in [Chap. 3](#).

8.2 Rich Snippets by Google

8.2.1 What Is Rich Snippets: An Example

For a submitted user query, Google returns the search result in the form of a collection of pages. In order to help the user to locate the result quickly and easily, for each page in this collection, Google shows a small sample of the page content. This small sample is called a page *snippet*.

Google introduced the so-called *Rich Snippets* to the world on 12 May 2009. This is a new presentation of the snippets that applies the related Semantic Web technology as we will see in the next section. For now, let us take a look at one such example: if you search for “Drooling Dog Bar” using Google, you will get the result as shown in [Fig. 8.2](#).

As you can tell, Rich Snippets in [Fig. 8.2](#) gives a user more convenient summary information about the search results at a glance. More specifically in this case, the review rating and the price range are clearly shown, which are the most important information a user needs when deciding which restaurant to go. You can try the same search using MSN, for example. At the time of this writing, the same restaurant does show up, but without the rating and price range information.

At this point, Google uses Rich Snippets to support the search for reviews and people. More specifically, when a user searches for a product or a service, review and rating information will be shown in the Rich Snippets. Similarly, when a user searches for people, Rich Snippets will be used to help the user to distinguish between people with the same name.

How are Rich Snippets related to the Semantic Web? And how does Google gather the information to create Rich Snippets? Let us answer these questions in the next section.

8.2.2 How Does It Work: Semantic Markup Using Microformats/RDFa

8.2.2.1 Rich Snippets Powered by Semantic Markup

The idea behind Rich Snippets is quite straightforward: it is created by using the structured data embedded in Web pages, and the structured data are added by Web page authors like you and me.

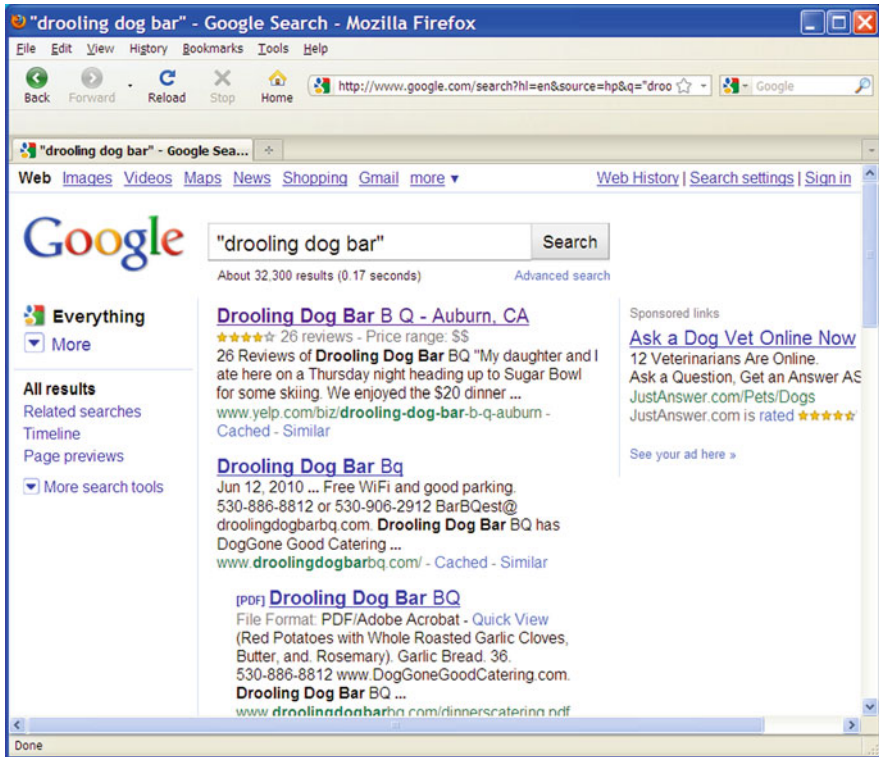


Fig. 8.2 Rich Snippets example when searching for “Drooling Dog Bar”

More specifically, the crawler still works as usual, i.e., traveling from page to page and downloading the page content along its way. However, the indexer’s work has changed quite a bit: when indexing a given page, it also looks for the markup formats Google supports. Once some embedded markups are found, they will be collected and will be used to generate the Rich Snippets.

To generate the Rich Snippets, Google also has to decide a presentation format that will be helpful to its users. For example, what information should be shown in the snippets? and in what order should this information be shown?

What Google has is the collected markups. Google does not tell the outside world the final format it uses to store these markups. For example, these collected markups can be expressed in RDF statements or some other format that Google has invented and used internally. However, no matter what the format is, these structured data can always be expressed in the form of name–value pairs.

Obviously, these name–value pairs are ultimately created by millions of Web page authors all over the world. Therefore, there could be very little consistency exhibited by these pairs. For example, some of these pairs are reviews of products, some are about people, some are about a trip to Shanghai, and some are about a

tennis match, just to name a few. Therefore, if all these pairs were to be considered, it would be very difficult to find a uniform way to present the information, i.e., generate the Rich Snippets.

In general, two solutions can be considered to solve this problem:

1. The first solution is to limit the types of markups a user can add, therefore the variety of the name–value pairs are limited. The final result is a uniform and consistent snippet can be created automatically by the search engine.
2. The second solution is to allow a user to markup anything he/she would like to. However, he/she also has to tell the search engine how to use the embedded structure data to generate a presentation.

These two are obviously quite different solutions. Google has used the first solution, and Yahoo!, as we will see, has adopted the second solution.

To use the first solution, Google has added the following limitations to the semantic markup one can add on a given content page:

- Google accepts and uses only markup data for review Web sites and people/social networking Web sites.
- When providing review or people information, certain data are required in order to automatically generate a Rich Snippet. For example, a review markup without a reviewer or a rating count will not be enough to generate a presentation. Finally,
- At this point, Google supports two markup formats: microformats and RDFa.

In fact, even for microformats and RDFa, there are still limitations: as we will see in the next two sections, only a certain microformats are supported and when using RDFa, the supported ontologies are also limited.

The benefit of this limitation, as we have discussed, is that a user only has to add the markup information and does not have to do anything regarding the presentation at all.

8.2.2.2 Microformats Supported by Rich Snippets

Google currently supports markup information on review and people/social networking sites. To mark up a review site, a user can choose *individual* review markup or *aggregate* review markup. An individual review markup is a single review about a product or a local business, and an aggregate review is an average rating or the total number of user reviews submitted.

To facilitate the use of microformats for review sites, Google accepts a collection of properties derived from the `hReview` microformat for individual reviews and the `hReview-aggregate` microformat for aggregate review properties.

For markup on people/social networking sites, Google recognizes a group of properties derived from the `hCard` microformat and some properties taken from the `XFN` microformat.

To see all these supported properties for both review and people sites, you can always check Google's documentation for Rich Snippets.¹ Note that by the time you are reading this book, it is likely that more and more microformats will be supported by Google already.

8.2.2.3 Ontologies Supported by Rich Snippets

Besides the choice of using microformats for marking up the pages, a user can choose to use RDFa to do the same markup. The same properties derived from the hReview microformat, hReview-aggregate microformat, hCard microformat, and the xFN microformat can be used in RDFa mark up.

In addition, Google supports the use of FOAF ontology and vCard² ontology when it comes to RDFa markup. For more details and updates, always check back to the Rich Snippets documentation Google provides.

8.2.3 Test It Out Yourself

When it comes to actually using microformats or RDFa to mark up the page and hoping Google will show the enhanced presentation, it is the page author's responsibility to make sure Google can understand the added markup. To help this process, Google provides a Rich Snippets Testing Tool³ for us to use. Fig. 8.3 shows the screen of this testing tool.

To use this tool, you need to first mark up your page, then enter the URL of the page. Clicking the `Preview` button will show a section called "Extracted Rich Snippet data from the page." In this section, you can see if Google's parser can extract the markup information you have entered. Once you have confirmed the marked up content can be extracted successfully by Google, you can sign up by filling the *Interested in Rich Snippets* form. Rich Snippets from new sites will be enabled by Google automatically from this list over time.

Note that it is possible even after all the above steps that the Rich Snippet for your site may still not show up in a search result page. We will not discuss the details here, since our goal is not to provide a tutorial on using Google's Rich Snippets but to understand it as an example of the Semantic Web. If you are interested in using Rich Snippets, you can always find more details from Google's Rich Snippets documentation page.

8.3 SearchMonkey from Yahoo!

SearchMonkey is Yahoo!'s version of Rich Snippets. In other words, similar to Rich Snippets, SearchMonkey also uses the semantic markups added by the page authors to enhance the search engine results page.

¹<http://www.google.com/support/webmasters/bin/topic.py?hl=en&topic=21997>

²<http://www.w3.org/2006/vcard/ns#>

³<http://www.google.com/webmasters/tools/richsnippets>

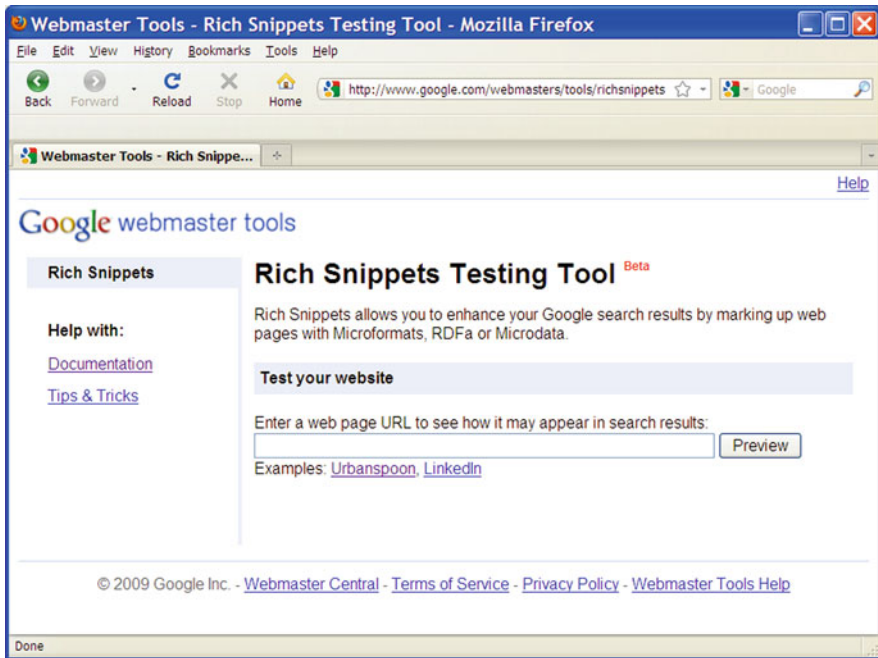


Fig. 8.3 Rich Snippets testing tool provided by Google

The difference between SearchMonkey and Rich Snippets is how to generate the enhanced presentation based on the embedded markup. As discussed in Sect. 8.2.2.1, there are two solutions. Google uses the first solution and SearchMonkey has adopted the second solution.

Therefore, SearchMonkey provides the flexibility of allowing any microformats and any ontologies in the markups; however it has the complexity of creating the presentation by the users themselves. SearchMonkey can actually be viewed as a framework that leverages the added semantic markup (in the form of microformats, RDFa, and eRDF) to enhance the search engine results page.

8.3.1 What Is SearchMonkey: An Example

Yahoo! announced SearchMonkey in May 2008 (about a year before Google introduced its Rich Snippets). Let us take a look at one example of SearchMonkey at work.

Open up Yahoo! search page, and type “Thai Tom University District Seattle, WA” as the query string, and you will see the enhanced search result as shown in Fig. 8.4.

As you can tell, for this particular search for a Thai restaurant, the enhanced search result is not much different from what we have seen in Google’s Rich

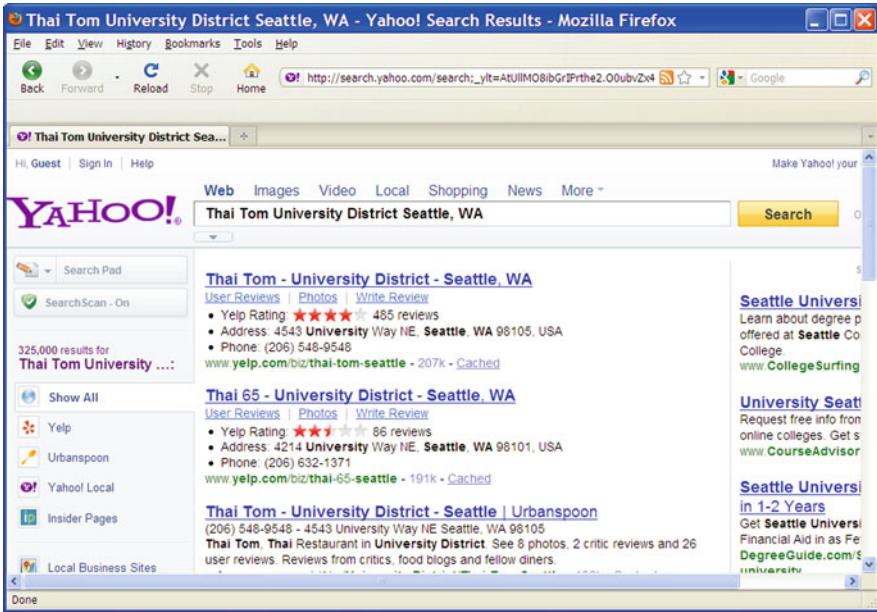


Fig. 8.4 Example of Yahoo!'s SearchMonkey application

Snippets (Fig. 8.2). However, what has happened inside Yahoo! search engine is more complex. Let us study this in the next sections.

8.3.2 How Does It Work: Semantic Markup Using Microformats/RdFa

First off, understand that SearchMonkey works in three ways to change its search engine results page:

- Site owners/publishers use structured data to mark up their content pages.
- Site owners/developers or third-party developers build SearchMonkey applications.
- Yahoo! search engine users add SearchMonkey applications to their search profiles on an opt-in basis, and once they start to use SearchMonkey, they can customize their search experience with *Enhanced Results* or *Infobars*.

All these will become clear when you finish this section. In addition, you can also check out Yahoo! SearchMonkey's official Web site⁴ along with your reading to get more details that are not covered here.

⁴<http://developer.yahoo.com/searchmonkey/>

8.3.2.1 SearchMonkey Architecture

This section presents the high-level architecture of SearchMonkey, and understanding this architecture will help us understand its basic flow, and eventually, how it works. Fig. 8.5 gives the overall structure of a SearchMonkey application.

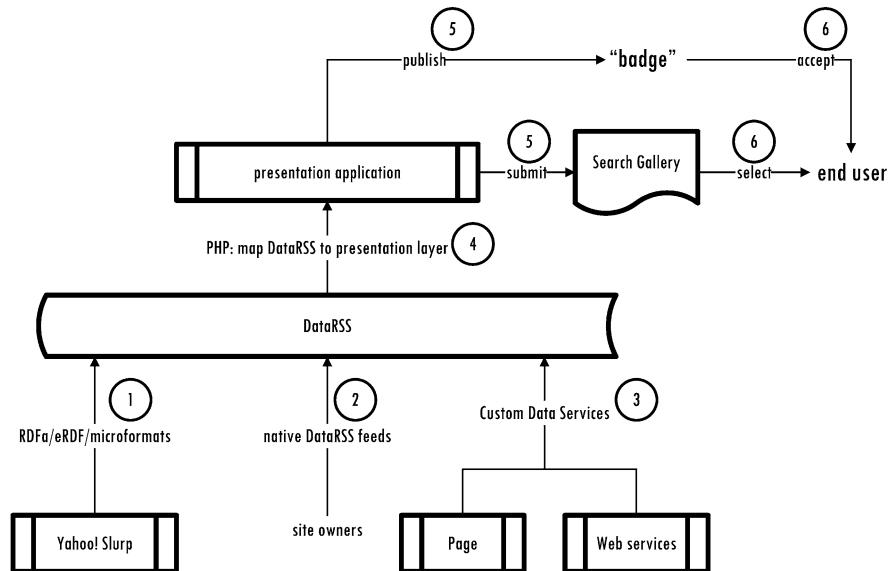


Fig. 8.5 Structure of a SearchMonkey application

Everything starts from the moment when a site owner decides to add some semantic markup to the content pages contained in his/her site. For Yahoo!’s SearchMonkey, these semantic data can be expressed by microformats and RDFa, and most of the markup data will be added manually by the site owners.

Once this is done, it is up to Yahoo! Search Crawler to pick up the added structured data. Yahoo! Search Crawler, also known as *Yahoo! Slurp*, is capable of recognizing the embedded semantic data. At the time of this writing, Yahoo! Slurp can process embedded RDFa and microformats data.

For example, when it hits a page and sees the RDFa markup, it will automatically extract any valid RDFa data it finds. It will then transform this extracted RDFa statements into a chunk of data expressed in *DataRSS* format. Note that *DataRSS* is a special XML language used by Yahoo! to normalize between all the different types of structured data Yahoo! Slurp might have found from a page. At this point, the structured data can come from RDFa, eRDF, and a verity of microformats. Later on, this layer could be expanded to include more possibilities.

Note that any chunk of *DataRSS* is related to a given URL, identifying the page where the original semantic data is found. Also, Yahoo! caches this information on the server side, so retrieving this information is relatively fast. As with the conventional Yahoo! index, extracted semantic data refreshes whenever the page gets

crawled; therefore it is always possible to see a gap between what you have added on the page and what Yahoo! has gathered.

What we have just discussed corresponds to the box marked as “1” in Fig. 8.5. In some cases, however, instead of adding markups and waiting for Yahoo! Slurp to pick up the added semantic data, a site owner can directly submit a feed of native DataRSS to Yahoo!. Similar to the data gathered by Yahoo! Slurp, this submitted chunk of DataRSS is also indexed and cached by Yahoo! on the server side to improve performance.

Obviously, direct data feeds offer another excellent choice to provide rich information about a site, particularly if the site owner cannot currently afford to redesign the whole site to include embedded microformats or RDFa. This component maps to the box marked as “2” in Fig. 8.5.

It is important to understand that in SearchMonkey, any semantic data extracted from an (X)HTML page are finally presented within SearchMonkey as DataRSS data chunks. Besides the above two methods to get the structured data into DataRSS, it is also possible to build the so-called *XSLT Custom Data Service* to accomplish the same (marked as “3” in Fig. 8.5). More specifically, there are two types of custom data services: *Page custom data service* and *Web Service custom data service*. Page custom data service refers to getting structured data from the pages within a given Web site, and Web Service custom data service refers to extracting data from Web services which you may or may not provide.

One of the reasons why Page custom data service is useful can be the fact that the semantic markup used on the particular page is not currently supported by Yahoo! SearchMonkey; therefore it will not be understood and collected by SearchMonkey either.

For example, SearchMonkey understands microformats, RDFa, and eRDF at this point. If a new format is used or if a site owner simply invents his/her own standard, the site owner has to write a Page custom data service to extract the structured data and change it into DataRSS format.

Another less obvious case where Page custom data service can be useful happens when someone is not the owner of a given site. If this is the case, to make sure the structured data contained in that site participate in SearchMonkey, Page custom data service has to be used to extract data from that site. Certainly, if the site already exposes semantic markup that can be understood by SearchMonkey or if it provides a native DataRSS feed already, there will be no need for Page custom data service.

Compared to Page custom data service, the reason why Web Service custom data service is needed is quite obvious: it is useful when the structured data are not contained in a page but returned by a Web service call. In this case, a custom data service is created to call the Web service and the returned data are transformed into DataRSS data format and submitted to SearchMonkey.

Note that custom data services (including both Page and Web services) are defined and created within SearchMonkey framework itself. This is in fact a central piece of the SearchMonkey experience: a developer or a site owner can use an

online tool⁵ provided by SearchMonkey to define and create the data services they need. Since the goal of this chapter is not to provide a tutorial about how to define and create custom data service, we will leave this for you to explore.

At this point, we have described how distributed semantic markup information gets into SearchMonkey's DataRSS. The next thing is to make use of the available data to make the search result page look more appealing to the users. To accomplish this, developer's work is needed.

Recall in Google's Rich Snippets solution, developers do not have to do anything at this step. Since it only supports limited content markup (both the content and the markup languages), Google is able to automatically generate the final presentation.

In SearchMonkey, this step is called *Creating Presentation Applications*. A presentation application is a small PHP application that defines how Yahoo! search engine should display the search results. More specifically, any presentation application is based on a given presentation template, and there are two basic presentation templates: *Enhanced Result* and *Infobar*. The example shown in Fig. 8.4 is the look and feel of an Enhanced Result template, and obviously, it is similar to Google's Rich Snippets. An Infobar, by contrast, is quite different from Enhanced Result. It is an expandable pane beneath a search result that provides additional information about the result.

The difference between Enhanced Result and Infobar is shown by how search users react to them. An Enhanced Result will give a user more information about the search result page, and he/she can use the additional information to determine the relevance of the page. Note that no user action is needed here when this decision is made. When a user views an Infobar, the user has already decided the relevance of the page, and he/she is using the Infobar to look for more information. And obviously, action is needed here: a user has to expand the Infobar to access the additional information.

Therefore, Enhance Result presentation format is limited to a specific set of presentation elements, and these elements are also arranged in a particular format. More specifically,

- Title (a string specifying the search page's title);
- Summary (a string specifying the search page's summary);
- Image (a thumbnail image that represents the search result);
- Link (an HTML link that either provides more information about the search result or indicates some action that a user can take);
- Dict (a key-value pair that displays structured information about the item in the search result, such as review ratings, technical specifications, and hours of operations).

⁵<http://developer.search.yahoo.com/wizard/data/basic>

In real development work, which template to use is the choice of the developer together with the site owner. Again, Enhanced Result is more uniform, but it is more limited as well. Infobar is quite flexible but does require more action from the users.

Once a developer has decided which template to use for building the presentation, he/she can start to code the application itself, which will be later on used by Yahoo! search engine to display the structured information contained in DataRSS.

To make this easier, SearchMonkey provides another online tool⁶ to guide a developer to finish this step by step. Yahoo!'s SearchMonkey online document⁷ has detailed tutorial about how to accomplish this; we will not cover them here. However, to make the picture complete, here are some key points about creating a presentation application:

- The developer will specify which template to use, i.e. Enhanced Result or Infobar.
- The developer will have to use PHP code to build the application.
- The developer will connect the presentation layer to the correct data feed. For example, if he/she has created custom data services in previous steps, this will be the point where the connection is made.
- The developer will specify a *trigger URL*, i.e., a URL pattern to match against Yahoo! Search results and once a match is found, this application will be triggered.

At this point, the basics of building the presentation application are covered. In Fig. 8.5, this process is marked as “4.”

Once the presentation application is finished, the last step is to publish the application so that the search users can benefit from it. This is a relatively simple step as shown in Fig. 8.5 (marked as “5”). More specifically, the presentation application confirmation screen contained in SearchMonkey's online development tool will offer the developer a choice to make the application sharable. To make the application sharable, the developer will require a button (also called a “badge”) that he/she can use to promote the application; this badge can also be used to help spread the application virally.

Another choice available to the developer is to submit the application to Search Gallery, where a collection of applications are featured and search users can visit at any time to select from them.

The last component in SearchMonkey's architecture is the end user. For a search user to enjoy the enhanced search result, he/she has two choices. First, by clicking a badge that represents an application already built, a search user will be taken to his/her search preference screen, where he/she may choose to add that application. The second choice is that the search user can also visit the Search Gallery to select application from the collection. Either way, the user's search profile will be updated,

⁶<http://developer.search.yahoo.com/wizard/pres/basic>

⁷<http://developer.yahoo.com/searchmonkey/smguide/index.html>

as shown in Fig. 8.5 (marked as “6”). Next time, when the user searches the Web, if one of the returned URL matches the triggering pattern, SearchMonkey will fire up the presentation application, and the user will see a much better result screen.

As a summary, Yahoo! search engine users add SearchMonkey applications to their search profiles on an opt-in basis, and once they start to use SearchMonkey, they can customize their search experience with Enhanced Results or Infobars.

At this point, we have finished description of SearchMonkey’s overall architecture, and meanwhile, we have obtained a good understanding about how it works. The final takeaway can be summarized as this: SearchMonkey is a framework for creating applications that enhance Yahoo! search results, and it is made possible by the added semantic markup.

8.3.2.2 Microformats Supported by SearchMonkey

Based on our understanding about SearchMonkey, it is obvious that theoretically, SearchMonkey can support any microformat. However, for the purpose of creating the presentation, only the following microformats are supported:

- hCard
- hCalendar
- hReview
- hFeed
- XFN

If you decide to use other microformats, you will likely have to create custom data service(s) as we have discussed earlier. In addition, you will have to create your own presentation as well, instead of using the default ones provided by Yahoo! Search.

8.3.2.3 Ontologies Supported by SearchMonkey

Any ontology created by using RDFS and OWL can be used in SearchMonkey. But when you choose to use your own custom-made ontology, note that you will likely have to write custom data service(s) so that the markup information can be collected. Also, you will have to create presentation application instead of using default ones.

Finally, note that even when you use RDFa with ontologies, no validation will be performed and no reasoning is conducted either.

8.3.3 *Test It Out Yourself*

Obviously, SearchMonkey is more complex than Google’s Rich Snippets. Fortunately however, there is a way for you to quickly try it out with minimal explanation and learning curve to cover. Yahoo! SearchMonkey has provided a

Developer Quick Start tutorial for you to construct a Hello World SearchMonkey application, and you can access the whole tutorial from the following URL:

<http://developer.yahoo.com/searchmonkey/smguide/quickstart.html>

Follow the instructions in this tutorial, you should be able to build a quick understanding about SearchMonkey.

8.4 Summary

We have discussed Google's Rich Snippets and Yahoo!'s SearchMonkey in this chapter as two Semantic Web application examples. These are important applications not only because they give you a chance to see how to put what you have learned together to make real Semantic Web applications but also because of the fact that they could be a turning point for the Semantic Web: for the first time, there are example applications from major players such as Google and Yahoo!, and more importantly, they could create some economic motivations for more semantic markup to come.

Finally, here are the main points you should have learned from this chapter:

- Microformats and RDFa can be used to add semantic markups to Web documents, and the semantic markups are added manually, by site owners or developers.
- Yahoo!'s SearchMonkey and Google's Rich Snippets are applications which take advantage of the added markup information.
- It is therefore indeed possible to create large-scale applications based on manually created markup information.

Reference

Brin S, Page L (1998) The anatomy of a large-scale hypertextual Web Search Engine. *Computer Networks and ISDN Systems*, 30(1-7):107-117

Chapter 9

Semantic Wiki

In [Chap. 7](#), we have discussed the topic about semantic markup. As we have concluded, semantic markup by the general public on the whole Web seems to be too challenging to implement. However, for a much smaller domain and scope, semantic markup is feasible, largely due to the fact that publishing and sharing a collection of core ontologies within this smaller domain is much more easier.

This chapter will provide one such example so that you can get a detailed understanding about when and where manually semantic markup can be a useful solution.

More specially, we will concentrate on semantic wiki, one successful application based on manual semantic markup. In this chapter, we will first study the limitations of wiki sites, we will then see how the Semantic Web technology can be used to change the way we use wiki sites.

To accomplish this, we will present semantic wiki engines in detail, including an enhanced markup language called wikitext, which includes constructs to implement semantic markup that ordinary user can understand and use. We will also discuss a new query language for using the semantic wiki. Not only you will see the power of manually semantic markup but also you will find the material in this chapter valuable if you want to set up and use a semantic wiki site from scratch.

9.1 Introduction: From Wiki to Semantic Wiki

9.1.1 What Is a Wiki?

In fact, since there are so many writings about wiki out there already, let us not try to describe wiki again. If you are interested in the first wiki developed by Ward Cunningham, or you are interested in the history of wiki, you can easily find your answers on the Web – perhaps at one of those wiki sites – such as Wikipedia, for example.

And once you have used Wikipedia, you know what a wiki is already. In general, the following is what you need to know about wiki:

- A wiki is a Web application that manages a collection of Web pages, where *anyone* can create new pages and edit existing pages by using a simplified markup language.
- A wiki engine is a software system that powers the wiki site and makes it work.

In recent years, wikis have become popular environments for collaboration on the Web. For some organizations, wiki sites are acting as the primary tools for content management and knowledge sharing. For example, in some areas, business executives, project managers, and frontline employees have already started to look at how internal wikis can help them to be more efficient, and at same time promote collaboration, making it easy for users to find and use information about business processes, company policies, and more.

What makes wiki sites so attractive? The following is a list about why wiki is so good to use:

- Creating a new page is simple

To create new page, you can use the browser address bar to enter an URL to a new page (better yet, you can edit the page name part of an URL for an existing page). This newly created URL will take you to the default `no article` message page, where you will find the usual `Edit this page` link, which can then be used to create new page.

It is also possible to start a new page by following a link to the new page, and you can then start your editing right away. Links to non-existing pages are quite common in wiki sites, and they are typically created as a placeholder for new content. In fact, this is also one effective way to invite everyone to participate in the wiki as well.

- Editing a page is easy

To edit a wiki page, we don't need to know or remember HTML at all, we only need to learn a few simple markup rules, i.e., *wikitext*. Wiki engine will produce HTML code for us. This is one and the vital piece that makes wiki a success, since it allows non-technical people to publish their ideas with much ease.

- Changing a page is easy

Each wiki page has an `Edit` link. Anyone can click the link and change the content by editing it. The change will show up once the user submits it by one button click. This very idea of involving every mind in the community is also critical to wiki's success.

- Linking to others is easy

The fact that wiki pages are all inter-linked has provided tremendous benefit to its users. And creating the link is also quite simple. In fact, a wiki engine stores all the pages in an internal hypertext database; it therefore knows about every page you have and every link you make. You don't have to worry about the real location of files, simply name the page, and wiki engine will automatically create a link for you.

Besides the above, there are many other attractive features offered by wiki. For example, wiki keeps a document history, so we can always recover from a mistake, or roll back to any previous version of a given page. Also, wiki allows us to identify the exact changes that have been made to a page over time. In general, wiki's success comes from the fact that it is an open format where anyone can post new articles and change existing articles.

However, as everything else on earth, pros and cons always go hand-in-hand. Despite their success stories, wikis do have some limitations, and it is the limitations that lead to the idea of semantic wiki, which we will introduce in the next section.

9.1.2 From Wiki to Semantic Wiki

Here is a list of wiki's limitations.

- Knowledge discovery

This is perhaps the most visible problem for wiki sites. Today, using a given wiki site primarily means *reading* articles in the site. For example, in Wikipedia, there is no way to request a list of cities with each city having at least 100 years of history, a female mayor, and also being the place for Olympic games, even though the information is indeed contained in a number of pages in the wiki. This information has to be obtained through human reading. For a wiki that has thousands of pages, reading through pages is time consuming at least. Yet ironically, this problem could potentially defeat the main goal of building a wiki site: to share knowledge and to make knowledge discovery easier.

Wiki engine does provide a full text search for its users. However, full text search inevitably suffers from the ambiguity of natural language. As a result, it normally produces a list where most of the pages in the list are irrelevant; users still have to sift through the returned pages to look for the information they need, if they are patient enough to do so.

Based on what we have learned so far, we understand the root of this problem. The required information is there in the wiki pages, yet its meaning remains unclear to the wiki engine. The reason is because it is not presented and stored in a machine-readable way, but only accessible to human eyes.

To solve this problem once and for all, we need to follow the idea that is promoted by the Semantic Web technology: add semantics into the wiki pages so that the information on these pages will be structured enough for machine to process. Once this is done, knowledge discovery will be easier for the wiki users.

- Knowledge reuse

This is closely related to the above problem. Even though many wiki sites are designed and created with the idea of knowledge reuse in mind, it is almost always true that the reuse goal is not accomplished. Again, given the fact that wiki page can be read only by human beings, automatic knowledge reuse is simply not possible.

- Knowledge consistency

One of wiki's main attractive features is its openness: everyone can publish any content and change any content in a given wiki site, in a distributed fashion. However, this also means the fact that the same concept can appear in multiple pages and can be expressed by using different terms. This not only gives confusion to the wiki users but also poses another difficulty to the full text search functionality. If a user searches the wiki by using a keyword term, a highly relevant page will not show up in the retrieved list simply because it uses the "wrong" term.

With all these said, and with what we have learned so far from this book, the solution should be clear: adding formal semantics into wiki pages so that machine can understand these pages. This not only solves the knowledge discovery and reuse issues but also promotes information consistency among the pages.

Adding semantics to the wiki pages turns out to be only the tip of an iceberg; the wiki engine itself has to be enhanced so that it knows how to take advantage of the structured information contained in the page. This further leads to an inside-and-out change of a wiki engine, therefore giving rise to a new breed of wiki site: *semantic wiki*.

Formally speaking, semantic wiki can be defined as a wiki site powered by a semantic wiki engine; it enables users to add semantic markup to wiki pages, and the added structured information can then be used for better searching, browsing, and exchanging of information.

In this chapter, we will cover the following topics in detail:

- How to add semantic information to a wiki page?
- How the added semantic information helps us when it comes to using the wiki?
- What has happened behind the scene? Why the added semantics is so powerful?

To make our discussion easier, we need to have a running example of semantic wiki engine. As you might have guessed, instead of creating a semantic wiki engine from the scratch, most of today's semantic wiki engines are built by making some extensions to some traditional wiki engines.

Semantic MediaWiki is a semantic wiki engine built on top of *MediaWiki* engine. It was originally developed by AIFB,¹ a research institute at the University of Karlsruhe, Germany. Over the years, it has been under constant improvement by developers around the world. Since it is a free extension of *MediaWiki*, it is widely available, and by using it, you can easily build your own semantic wiki sites.

The reason why *MediaWiki* is taken as the source for *Semantic MediaWiki* is largely due to its popularity. In particular, it is the wiki engine that powers many of the most popular wikis in the world, including Wikipedia, the largest wiki site online.

¹AIFB, in English, is *Applied Informatics and Formal Description Methods*.

Above being said, throughout this chapter, we are going to use Semantic MediaWiki as the example semantic wiki engine. The semantic features we are going to discuss are therefore all provided by Semantic MediaWiki. It is certainly true that other implementation of semantic wiki engine may offer different features. However, most of these features do share great similarity, therefore the discussion in this chapter should be general enough for you to understand other semantic wiki engines.

To know more about Semantic MediaWiki, you can visit pages:

http://semanticweb.org/wiki/Semantic_MediaWiki

and

http://semantic-mediawiki.org/wiki/Semantic_MediaWiki

Both these pages will provide abundant information for you to understand more about Semantic MediaWiki itself. Bear in mind, the underlying wiki engine is the software that makes use of the added semantic markup data, and it has been built already for us to use. The real interesting pieces, for example, how the pages are marked up and how the added semantic data can help us to use the wiki better, are the main topics of this chapter.

Note that you do have to have some understanding about non-semantic wiki. If you know nothing about wiki at this point, take some time to understand it; that will help you to understand the material presented in this chapter.

9.2 Adding Semantics to Wiki Site

This section will discuss the first key issue: semantic annotations on a given wiki page.

First off, understand that in the context of wiki, any given wiki document has two elements, namely hyperlink and text. Therefore, semantic annotation in wiki means to annotate any link or text on the page so as to describe the meaning of the hyperlink or the text. The result is that the added annotation turns links and text into explicit property–value pairs with the page being the subject.

Second, besides the properties on links and text, *category system* can be viewed as an existing semantic component. The reason being that it does add structural information to the wiki documents, and it is also used in semantic wiki’s RDF/OWL exported files. Therefore, to have a complete picture about semantic components in a semantic wiki engine, we will start with category system.

Third, keep the following questions in mind when you read this section:

- What is the syntax of semantic annotation in wiki? Is it quite different from the original wikitext?
- What are the terms (property names) we can use when marking up a page? Who is responsible for creating these terms?
- Are there any ontologies we can use when adding semantics to the pages?

These will help you to understand the idea of adding semantics much quicker and easier.

9.2.1 Namespace and Category System

Category system is an existing semantic component in MediaWiki and will be the main topic of this section. To understand the category system, we need to understand the concept of namespace first.

In MediaWiki, a page title always looks like

```
namespace:title
```

For example,

```
Help:Starting_a_new_page
```

is the title of a help page which provides information on how to start a new page. This title string tells us that this page is in `Help` namespace, and its title is `Starting_a_new_page`.

Note that the `namespace` portion of a page title is optional, and if the title string of a given page does not have the `namespace` prefix, it will be simply a title without the colon. For example, the `Goings-on` page, which reports recent goings-on of the wiki community, has a page title that looks like this:

```
Goings-on
```

In this case, this page is by default in `main` namespace.

By default, a wiki powered by MediaWiki will have 18 namespaces, and they are as follows:

- `main` namespace, which is also the default namespace if a given page title does not have any prefix;
- 15 additional namespaces, each having a specific prefix; and
- 2 pseudo-namespaces.

The main reason of having the namespace system is to better structure the content for a given wiki project. The following is to name a few benefits:

- Namespace system can be used to separate the main content from the rest.

This separation will make the wiki management become easier. For example, `main` namespace (and a few others) will form a core set that is open for public, allowing users to view and edit them frequently. Meanwhile, this core set is actively policed by the wiki community; any inappropriate content will be quickly removed. The policing rules for other namespaces are generally more relaxed.

Another example would be the search function. For instance, for most wiki engines, searching can be limited to any subset of namespaces, which will be helpful to users.

- Namespace system can be used to group the content about the same subject and to form a set of relatively unrelated content items.

For example, namespaces are stored as folders on the host file system, which gathers the content files of the same subject inside one single directory. This makes the administrator's work much easier for obvious reason. Also, if namespace is given by the prefix `<ns:>` before each page name, for a given page, its raw text file will then be stored in the folder `<ns>`. When linking to other document within the same namespace, one can simply refer to the document without having to prefix the document with `<ns:>`. The prefix is necessary only when linking pages outside the namespace `<ns:>`.

Note that for a specific wiki project, the namespace structure will most likely be customized to fit the need for that project. For example, not only each project will have its own naming method for these namespaces, but also the number of namespaces can be different.

Throughout this chapter, we are going to use *wikicompany* as our example. A semantic wiki powered by MediaWiki, *wikicompany* is a worldwide business directory. Everyone can add a new company profile into this directory, and we can search for a company from this directory and learn interesting things about the company. It is selected as our example mainly because it is quite intuitive to follow, in addition, there is no special domain knowledge needed. You can find this semantic wiki at the location

http://wikicompany.org/wiki/Main_Page

The following is a list of the namespaces used in *wikicompany* at the time of my writing:

Main, Talk, User, Wikicompany, Image, MediaWiki, Template, Help, Category, Property, Type, Concept, Job, Product, Event, Presence, Planet, Reference, pid, 911, Library, Blog, Filter.

Note that *Category*, as one of the namespaces, logically holds a categorization system that is used in MediaWiki. The goal of this categorization system is to classify articles based on certain criteria, i.e., adding a tag to each page, which will then be useful when searching the entire wiki. For example, click any company name in *wikicompany*, you can see that company's page, and this page will have a category named *Companies*, identified by `Category:Companies` at the bottom of the page.

Understand that a given category, `Category:Companies`, for example, is represented by a page in the *Category* namespace. This page normally contains at least the following information:

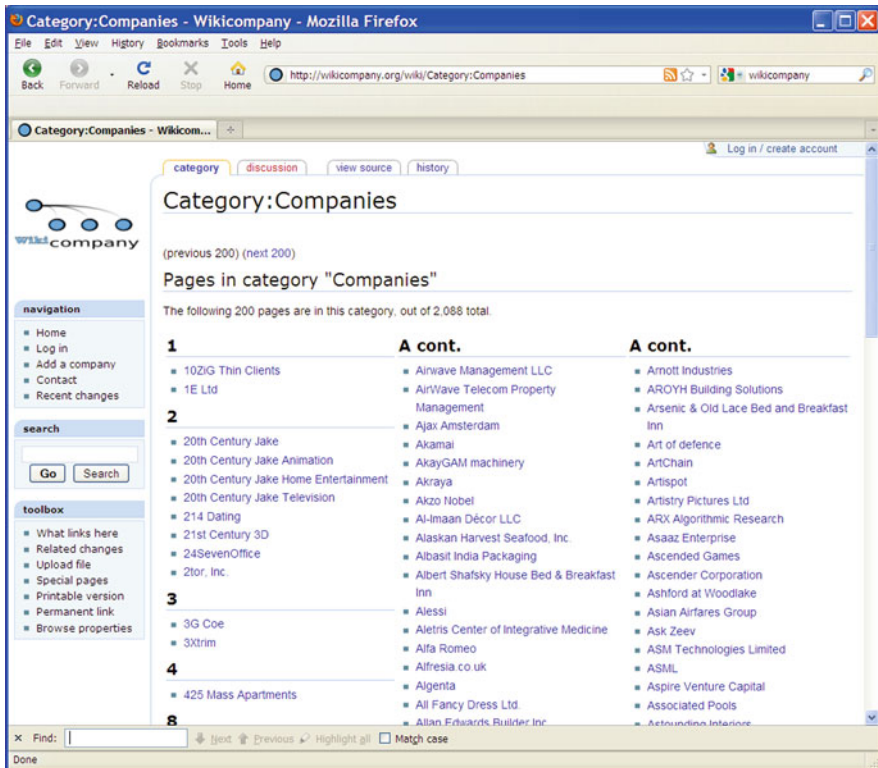


Fig. 9.1 Category Companies is represented as a page in the Category namespace

- editable text that offers a description about this category, which should explain which article should go into this category;
- a list of sub-categories, if applicable; and
- a list of pages that have been tagged with this given category.

For example, Fig. 9.1 shows the page about Category:Companies, and each page in the list shown in Fig. 9.1 is a page that has been tagged as Companies.

As a wiki user, you can create new user category and add it into Category namespace. Nevertheless, you should always try to reuse categories that already exist and should make good effort not to invent categories that represent same or similar concept. As one example, the following is a list of all the categories that are currently used in wikicompany site (and it is clear that some categories in this list are not the best way to use category system):

```

Attribute
Companies
Companies based in Seattle, Washington

```

[Companies bases in the United States](#)
[Internet companies](#)
[Market research](#)
[marketing](#)
[Networking](#)
[Planet](#)
[Planets](#)
[Services](#)
[Templates](#)
[Web application companies](#)
[Wikicompany](#)
[Windows software](#)

As a summary, it is clear that category system does provide some structure to the entire wiki system. However, its most obvious drawback is the lack of precise semantic definition. For instance, if a page describes IBM as a company, we can tag this page as `Category:Companies`. Now, for a page that discusses how to run a company, should we also tag it as `Category:Companies`? Given the openness of wiki engine, lack of formal semantics will finally become the source of ambiguity.

As we will see in the next section, Semantic MediaWiki provides a set of formal semantic definitions that are independent of the existing categorization system. Therefore, users can still use Semantic MediaWiki engine to create traditional wiki sites without worrying about the added semantic layer at all.

9.2.2 *Semantic Annotation in Semantic MediaWiki*

As we have mentioned earlier, semantic annotation introduced by Semantic MediaWiki concentrates only on the links and text contained in a wiki page. The annotation process does not alter anything about the category. In this section, we will focus on this annotation process.

9.2.2.1 **Semantic Annotation: Links**

The basic fact about a link on any given wiki page is that a link identifies some binary relationship between the linking page and the linked page. If we use a traditional MediaWiki engine to construct a page and its links, all these links will be created equally. In other words, there will be no way for a machine to capture the meaning of each specific binary relationship.

Let us go back to wikicompany and use Apple Computer as an example. Here is how the page for Apple Computer looks like at the time of this writing, as shown in Fig. 9.2.

This page has many links to other articles. One of these links is pointing to the page for Microsoft. As every other link, it does have a special meaning: it was there since Microsoft is a *competitor* of Apple Computer.



Fig. 9.2 Wiki page for Apple Computer

Note that Fig. 9.2 does not include this competitor content. To see this content, you can visit the following URL which will bring up the page shown in Fig. 9.2

<http://wikicompany.org/wiki/Apple>

and move to the bottom of the page to see the competitor content.

Now, if this page were created in MediaWiki without the semantic markup, the link to Microsoft would have looked like the following (this requires some understanding of wikitext):

`[[Microsoft]]`

and the *competitor* relationship is not expressed at all.

To make this knowledge available to a machine, Semantic MediaWiki allows us to do the following. First, express this binary relationship by defining a property called *competitor*. Second, in the article text for Apple Computer, instead of

simply using `[[Microsoft]]`, we will annotate this link by putting this property name and `::` in front of the link:

```
[[competitor::Microsoft]]
```

To human eyes, the article text is still displayed as a simple hyperlink to Microsoft, but the semantic markup, `competitor::`, is now available to be used by the machine as we will discuss later. For now, here is the general format for marking up a link:

```
[[propertyname::value]]
```

This statement, in a wiki page, defines a `value` for the property whose name is identified by `propertyname`. The page containing this markup will just show the text for `value` as a link that one can click, but the `propertyname` will not be shown.

Before we discuss further about property, let us clean up some formality issues regarding this simple markup. First off, besides the above, the following will show alternate text as the link text, instead of `link` on the wiki page:

```
[[propertyname::link|alternate text]]
```

And if we use a space to replace `alternate text` like this

```
[[propertyname::link|  ]]
```

then no link will appear in the text at all. Furthermore, if you want to create an ordinary link which contains `“::”` but does not assign any property, you can put `“:”` in the very front, for example:

```
[[[:propertyname::link]]]
```

This will show `propertyname::link` as the link text.

You can also mark up a link with more than one properties, in which case, you use `“::”` to separate these properties:

```
[[propertyname1::propertyname2::value]]
```

This will again only show `value` as the link text, but the wiki engine knows the fact that there are in fact two properties that have been assigned to this link.

Now, after solving the formality issues, let us continue to study the annotation of the links. First question comes to mind is the naming of the property. For example, in `wikicompany`, the example property is named `competitor`. Where does this name come from? Can we use other name at all?

In fact, the name of the property is arbitrary, and it is up to the user who adds the annotation to the link to name the property. Again, each user should try to reuse properties that are already defined and used elsewhere in the wiki site.

The second question then is, how to promote this reuse, i.e., how to find a list of all the properties that are already defined in this wiki site?

Semantic MediaWiki comes up with a simple yet effective solution: just like any given category, each property has its own article in the wiki, and all the articles that are describing properties are collected in a namespace called *Property*. For example, in wikicompany, property *competitor* has its own page named *Property:competitor*, with prefix *Property:* indicating its namespace. Figure 9.3 shows this page.

As you can see, this property page contains a very brief textual description of the property, with the goal of helping the users to use it consistently throughout the wiki. Also, the page contains a list of the articles in the entire wiki site which have used this property as their markup information.

Now, since each property has a page in *Property* namespace, coming up with a list of all the existing properties is quite easy (collecting all the pages from

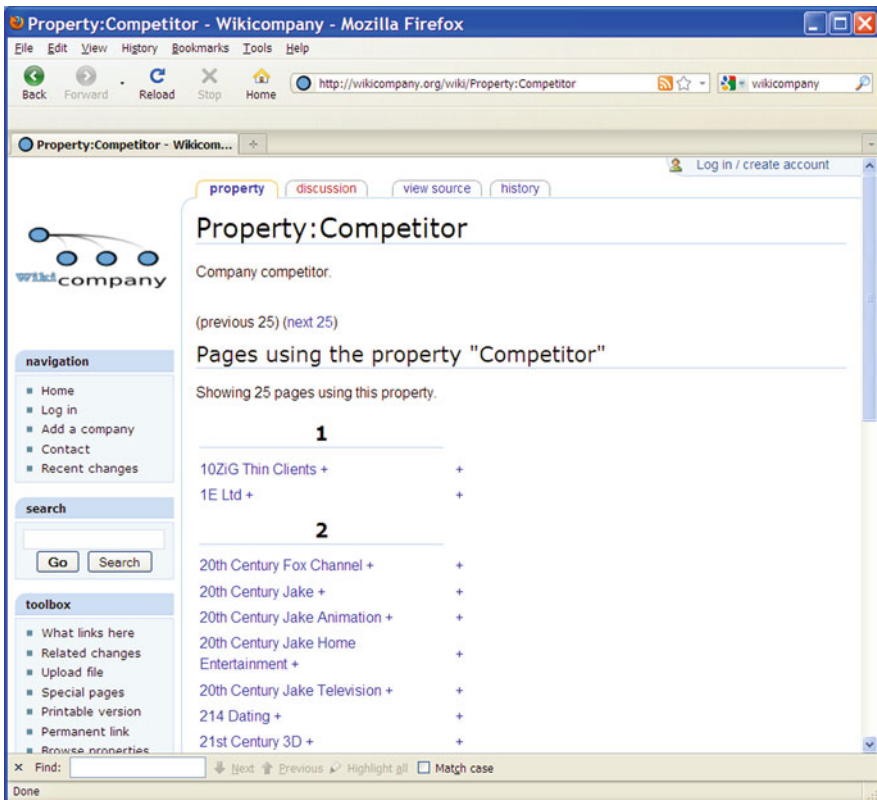


Fig. 9.3 Competitor property page

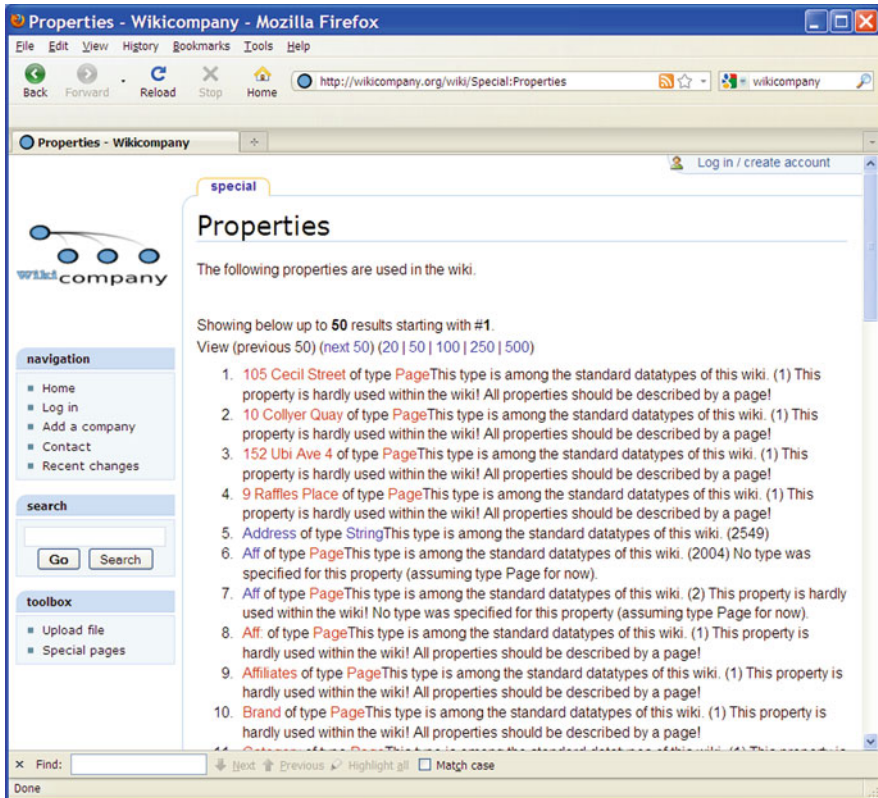


Fig. 9.4 A list of currently available properties

Property namespace will do). When conducting semantic markup, one should always consult this list to decide which property to use, and if necessary, one can invent his/her own. Figure 9.4 shows a list of existing properties at the time of this writing.

At this point, we have a good understanding about marking up the links on the page. The next section discusses the annotation of the second major element on a wiki page: the text.

9.2.2.2 Semantic Annotation: Text

Besides the links, lots of information are buried in the text on a given page. To mark up the text, Semantic MediaWiki decides to continue using properties. Therefore, properties are used to annotate both links and text. However, for text markup, we define properties that don't take other pages as their values. Let us take a look at one example from wikicompany.

Let us go back to the page of Apple Computer. One piece of information on this page is that Apple Computer has about 13,500 employees. To mark up this information, we can first create an `Employees` property by adding a `Property:Employees` page in `Property` namespace and then somehow indicate that Apple Computer has an `Employees` property, whose value is 13,500. Semantic MediaWiki decides to use the same general form as the one used when annotating the links. Therefore, the annotation will look like this:

```
[[Employees::13500]]
```

This seems to be fine. However, at this point, for the wiki engine, every property value is understood as a link to another article. In other words, it understands the above markup as this: Apple Computer has a `Employees` property, whose value is a link to a page that has 13,500 as its title. The wiki engine will therefore display 13,500 as a clickable link, just as it does for marking up links in the previous section.

This is certainly not what we wanted. The solution adopted by Semantic MediaWiki development team requires a user to do the following:

1. To mark up this information, a wiki users has to declare a property named `Employees` in `Property` namespace.
2. On this page, the user has to specify a datatype for this `Employees` property, and for this case, this type has to be `Number`.

On the Semantic MediaWiki engine side, the following two steps have been done to help the user:

Step 1. Semantic MediaWiki has declared several built-in datatypes that one can choose for properties.

At the time of this writing, these built-in datatypes include the following:

```
Annotation URI
Boolean
Code
Date
Email
Geographic coordinate
Number
Page
String
Temperature
Text
URL
```

And each one of these built-in types has a page to describe it. To collect all these datatype pages, Semantic MediaWiki has created a special namespace called `Type`. Therefore, the page for the datatype `Number` will be called `Type:Number`.

When it comes to choosing from these built-in types, one normally has to visit the pages of the candidate types, unless one is already familiar with all these types. Since you can always go to Semantic MediaWiki's document, we will not discuss each one of these types in detail, but just a quick look at some of the frequently used ones.

First off, `Page` type is the default type for any property. If you create a new property without specifying its type, `Page` will be its type. Semantic MediaWiki engine understands `Page` type as a link to another page, as we have seen in the previous section (property `Competitor` has `Page` as its type).

Other frequently used types include the following:

- `String` type is used on character sequences;
- `Date` for calendar dates;
- `Number` represents integer and decimal numbers with optional exponent, and
- `Geographic coordinate` describes geographic locations.

Now, with all these datatypes available for a user to choose from, how does a user indicate the final take for a given property? This lead to the second step that Semantic MediaWiki has done for us.

Step 2. Semantic MediaWiki has created special properties one can use when declaring the datatype.

For example, one of these special properties is called `Property:Has type`, which assigns a type to a property that user declares. Each special property may or may not have its own page in the wiki; however, it does have a special built-in meaning and is not evaluated like other properties.

Here is a list of the special properties created by the wiki engine at the time of this writing. Again, you can go to the official documents for detailed discussion about these properties:

```
Property:Allows value
Property:Corresponds to
Property:Display units
Property:Equivalent URI
Property:Has type
Property:Imported from
Property:Modification date
Property:Provides service
Property:Subproperty of
```

Now, let us go back to `Employees` property in `wikicompany`. When the page for this property is created, one has to add the annotation `[[Has type::Number]]` to indicate that this property expects numerical values. Note that special property `Has type` assumes its value is from `Type` namespace, so we can omit the prefix `Type` for `Number`.

At this point, using `Employees` property to mark up the text 13,500 is feasible, since `Employees` property has its own page, where its datatype has been specified

by using `Has type` property. The wiki engine will be able to interpret it correctly this time, and we don't have to worry that `Employees` property creates a link to a page called 13,500. In general, each type provides its own methods to process user input, and it tells the wiki engine how to display its value as well.

Note that unlike marking up links, where we normally finish all of them on a given page, marking up text is highly flexible: you can choose to mark up any text you think necessary, and there is no standard about which text you should mark up at all. In this section, we marked up only one single piece of text, and you can follow the same pattern to mark up other text if you want to. For example, a user might decide to mark up the phone number and home page information of Apple Computer as well.

As a side note, if you need to create a new property whose type is `Page`, you don't have to add a new page in `Property` namespace for this property, since `Page` is the default type. In other words, you can directly use any property whose type is `Page` without declaring it. However, this is not a good practice: if another user needs a similar property, since you have never added it into `Property` namespace, that user will not be able to reuse your property, and this is certainly what we want to avoid.

Before we move on to the next section, let us use `wikicompany` again as an example to take a look at the properties and their types as defined by its users, as shown in Table 9.1.

Table 9.1 Properties defined in `wikicompany`

Property name	Datatype
Address	String
Brand	Page
Company	Page
Competitor	Page
Coordinates	Geographic coordinate
Customer	Page
E-mail	Page
Employees	Number
Exchange	String
Fax	String
Founding	Page
Fullname	String
Home page	URL
Parent	Page
Partner	Page
Phone	String
Product	Page
Region	String
Sector	String
Sub	Page
Ticker	String
Url	URL

At this point, we have presented the basic process of adding semantic markups to wiki articles. This chapter is not intended as a complete tutorial, so for a complete description about annotating the pages, you need to go to the official documents on Semantic MediaWiki's Web site. For us, the more interesting questions center around how the added semantics information is used and what exactly is the benefit added when we change a traditional wiki site into a semantic wiki site. Let us answer these questions in the reminder of this chapter.

9.3 Using the Added Semantics

In the previous section, we have discussed the process of semantic annotation. Although we have not covered all the details, this process is fairly straightforward and simple.

On the other hand, however simple the process may be, the majority of users will still choose to neglect it if it does not bear some immediate benefits. In this section, we will discuss the powerful features provided by this simple markup, and hopefully, there would then be enough motivation for the users to go through the markup process.

9.3.1 Browsing

9.3.1.1 FactBox

For a user who has gone through the process of marking up the page, even before seeing the benefit of the markup, the first question would be where is the semantic data that has been added? Does the wiki engine actually understand this information?

To address this, the page rendered by the wiki engine has included a so-called *Factbox* that is placed at the bottom of the page to avoid disturbing normal reading. This Factbox summarizes all the semantic information that have been entered on this page.

Let us take Apple Computer as an example. Once you land on Apple Computer page in wikicompany, move to the bottom of the page, and you should see the Factbox as shown in Fig. 9.5. Again note that at the time you are reading this book, what you see could be different.

As you have seen in Fig. 9.5, Factbox shows information in two columns: the left column lists the properties that have been used to annotate the page and the right column shows their corresponding values. Note that each property on the left column is clickable: you will be landing on that property's article in *Property* namespace if you click the property name. Similarly, the values in the right column can also be links to other pages in the wiki. For instance, as shown in Fig. 9.5 property *Competitor* has *Microsoft* as its value, and *Microsoft* is rendered as a link which leads you to *Microsoft* page in the wiki.

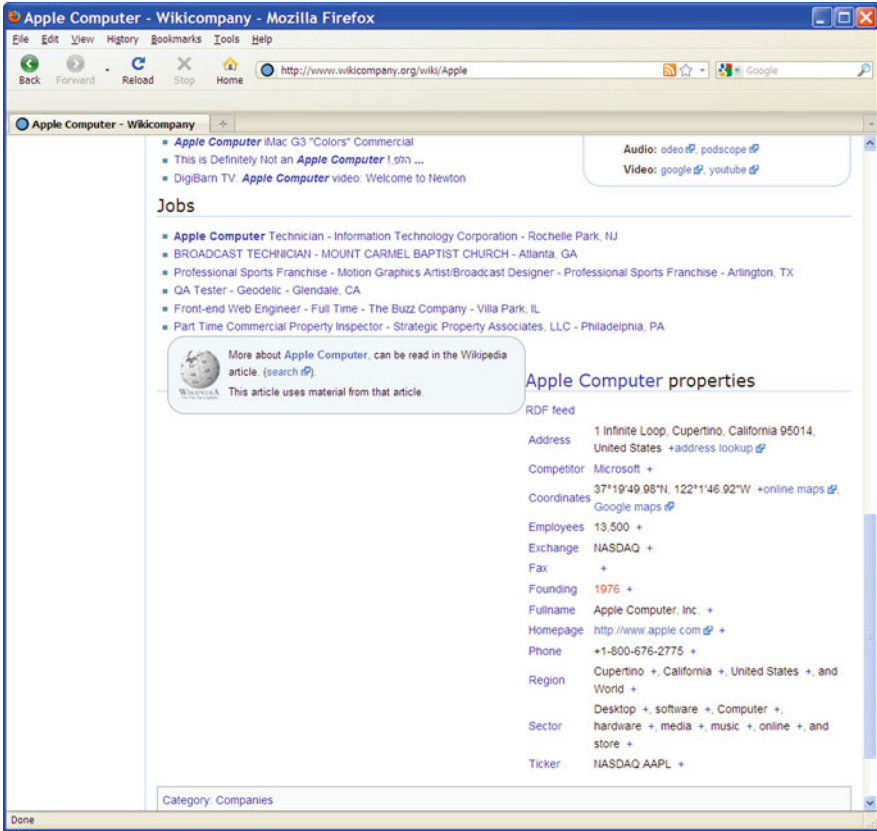


Fig. 9.5 Factbox on Apple Computer page

An interesting feature is the + icon next to Microsoft link (note that it could be other icons in different implementations). This in fact takes you to a simple search result which lists all pages with the same annotation, i.e., a list of all pages that have property Competitor with value Microsoft. In fact, Semantic MediaWiki engine provides a special search page, called Special:SearchByProperty, which offers exactly the same search: you can search for all pages that have a given property and value.

9.3.1.2 Semantic Browsing Interface

Perhaps the real semantic flavor comes into play from two links on the Factbox. The first one is RDF feed link, and we will be discussing this link in detail in later sections, so let us skip it for now and move on to the second link.

The second link is the page title, Apple Computer, shown as the heading of the Factbox. Clicking this link will lead to a so-called semantic browsing interface as shown in Fig. 9.6.

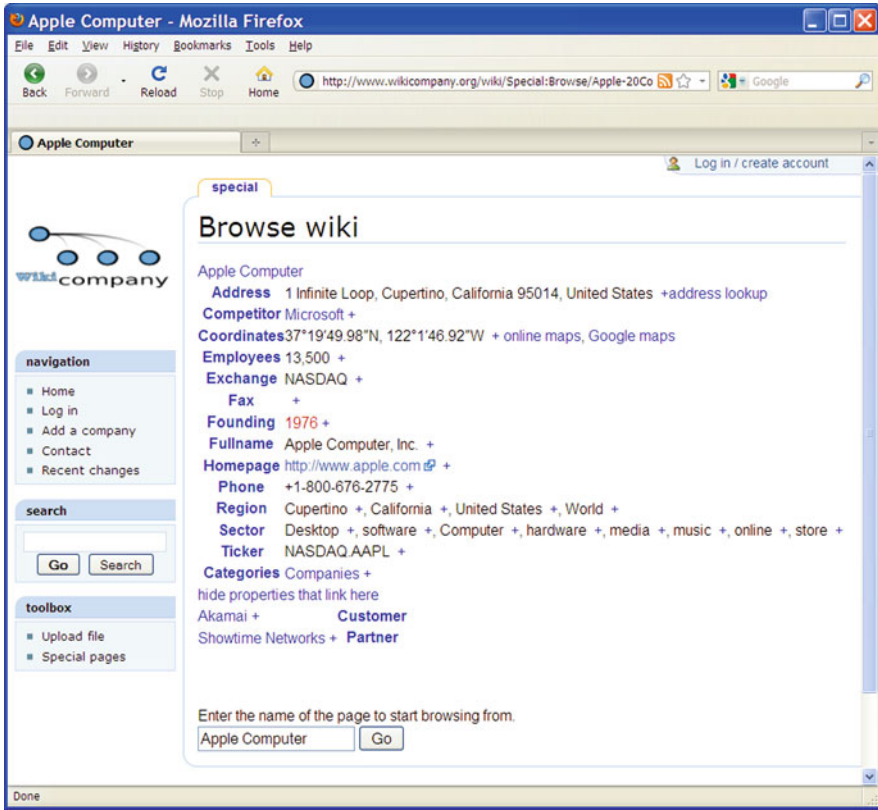


Fig. 9.6 Semantic browsing interface for Apple Computer

As you can see, this page not only shows all the annotations on the page for Apple Computer, but also includes all annotations from other pages which use the page of Apple Computer as their value. For example, as shown in Fig. 9.6, company *Akamai* is a Customer of Apple Computer, and company *Showtime Networks* is a Partner of Apple Computer.

This has in fact answered the question of how other companies are related to Apple Computer. Clearly, in a wiki that collects information about companies, this question is perhaps one of the most important questions you want to have an answer to. Yet, a traditional wiki engine will not be able to answer this, and it is up to you to actually read perhaps all the articles in the wiki and then come up with the answer. Obviously, some simple annotation process has indeed provided some very impressive feature already.

Note that the title link in the Factbox on a company’s page is not the only entrance to this feature; you can bring up this semantic browsing interface by visiting the following special page:

`http://wikicompany.org/wiki/Special:Browse`

Enter the company page name in textbox and hit Go button, the same browsing page will show up.

9.3.2 Wiki Site Semantic Search

To further exploit the semantics added by the annotations on the pages, Semantic MediaWiki has included a query language that allows access to the wiki's knowledge. In this section, we will take a look at this query language and understand it by examples. The good news is that the syntax of this query language is quite similar to the syntax of annotations we have discussed; it is therefore not a completely new language you have to learn.

9.3.2.1 Direct Wiki Query: Basics

One way to conduct a search is to use the query language on the special page called `Special:Ask`. In our wikicompany example, you can access this page at

<http://wikicompany.org/wiki/Special:Ask>

And once you open up this page, you will see the `Query` box on the left side, and the `Additional printouts` box on the right. As their names have suggested, the query is entered in the `Query` box and formatting issues are addressed by using the `Additional printouts` box.

Before we start to learn this query language, let us understand its basic idea. Recall that in semantic wiki, there are two constructs we have used to add structured data to the wiki content: category and property. More specifically, category represents a classification system that has been there for all the wiki sites, and property represents more detailed semantic information that can be added to the links and text within a given page.

Now, here is the basic idea: a query can be constructed by having constraints on either one or both of these constructs. With this in mind, let us start with the following simple query:

```
[[Category:Companies]]
```

This is a query that puts a single constraint on category, and it tries to find every page that has `Companies` as its category value. If you enter the above query into the `Query` box and click `Find results`, you will get a long list of all the pages which have `Companies` as its category value, and you will see `Apple Computer` is among the returned list.

To make this query more interesting, let us add one constraint on the property. For example,

```
[[Category:Companies]][[Competitor::microsoft]]
```

This query has put one constraint on the category and one constraint on the property, and these two constraints have a logical AND relationship (note that the constraint on Category uses :, while the constraint on property value uses ::). Therefore, this query tries to find pages about Companies, and the annotation property Competitor on these pages has to take Microsoft as its value. Therefore, this query aims to find all the companies that treat Microsoft as their competitor. If you enter this query into the Query box and click Find results, you will get the following companies (pages) back:

```
Apple Computer
Opera Software
TIBCO
```

Again, this is the result returned at the time of this writing, and if you execute this query at the moment of reading this book, it is possible that you will get different results back. This is also true for all the coming up examples, so we will not make special notes anymore.

Go back to our query. In fact, we can put more property constraints in a single query. The following query

```
[[Category:Companies]][[Competitor::microsoft]][[Sector::music]]
```

will try to find those companies who treat Microsoft as their competitor, and each one of these companies is also active in the music business. After executing the query, we only see Apple Computer in the result set since both Opera Software and TIBCO are not particularly active in the music sector.

Recall the fact that property has datatypes. This can give us lots of flexibility when asking for pages that have some specific property values. The following is a quick rundown of the examples, just to show you some of the main features:

- Wildcards

Wildcards are written as “+” and can be used to allow any value for a given property. For example,

```
[[Category:Companies]][[Phone::+]][[Homepage::+]]
```

will find all the companies that have provided their phones and their online Web sites.

- Comparators

For properties that have numeric values, we can select pages with the property values within a certain range. For example,

```
[[Category:Companies]][[Employees::>1000]][[Employees::<2000]]
```

will find all the companies that have employee number between 1000 and 2000. Note that > means “greater than or equal,” and < means “less than or equal”; the equality symbol (=) is not needed.

There are indeed several points we have to pay attention to for comparator operations. First off, we can use these range of comparators on those properties whose values can be naturally ordered. For example, `Employees` property here is defined to have a `Type: Number`, and it indeed can be ordered in a natural way. However, it will not make any sense to do the following:

```
[[Homepage::>http://www.apple.com]]
```

And obviously, it is up to the wiki users to make the judgment and not to submit a query including the above query component. However, what if the above query is actually submitted? In fact, if you submit it to `Special:Ask` page in wikicompany, you in fact can get some result back!

By default, if a given property’s datatype has no natural ordering, Semantic MediaWiki engine will simply apply the alphabetical order to the values of the given property. Therefore, in the above query example, the first returned company page has a Home page property given by <http://www.aroyh.com>, which is indeed alphabetically larger than www.apple.com.

Another important point about comparators is related to the search for a property value that happens to start with <. For instance, one of the query components could be `[[propertyName::
]]`, with `
` being the value to search for. In this case, we can insert a space after `::` to prevent the engine from interpreting the symbol as a comparator:

```
[[propertyName:: <br>]]
```

- String comparison

Semantic MediaWiki also provides a like comparator “~”, which only works for properties of `Type:String`. More specifically, in a like condition, you can use `*` wildcard to match any sequence of characters and `?` to match any single character. For example,

```
[[Category:Companies]][[Competitor::microsoft]]
[[Address::~~*California*]]
```

will try to find those companies that treat Microsoft as their competitor, and each one of these companies is located in California. Note that this string comparison feature is disabled by default, so if your query does not work, you might want to contact your admin person to check the settings.

9.3.2.2 Direct Wiki Query: Advanced Search

We have covered the basics of the query language used by Semantic MediaWiki; in this section, we will discuss some advanced features of this language.

- Union of query results

Recall in the previous section, among the queries we have created, if a given query involves more than one constraint on properties or categories, these constraints are understood by the wiki engine as having a logical AND relationship. Therefore, we are selecting pages that satisfy these conditions at the same time. In some cases, we will need to look for a logical OR relationship among these constraints.

In Semantic MediaWiki, one way to implement a logical OR relationship is to use the OR operator. For example,

```
[[Competitor::microsoft]] OR [[Competitor::google]]
```

The OR operator is used to take the union of two queries: the query on the left side of OR and the query on the right side of OR. Therefore, this query tries to find all the companies that treat either Microsoft or Google (or both) as their competitors. To understand this query, you can try the query on the left first, then the query on the right, and finally try the whole query. You will be able to see the results clearly.

Another operator provided by Semantic MediaWiki is the || operator, which is used to express a logical OR relationship among values, pages, and category names. For instance, the above query can be rewritten by using the || operator as follows:

```
[[Competitor::Microsoft|google]]
```

You can easily confirm that this query gives exactly the same result as the previous query which uses OR operator.

Clearly, using || operator provides a more concise form of the same query. However, there are cases where using OR operator is a must, especially for those queries that involve different properties. For example, the following query tries to find those companies that have provided either a phone number or a home page, or both:

```
[[Phone::+]] OR [[Homepage::+]]
```

Finally, note that operator || can be used not only with property values but also with categories, such as the following query:

```
[[Category:Companies|Planet]]
```


- Getting pages from a given namespace

Sometimes, we need to get all the pages from a given namespace, and this is done by specifying the namespace and a wildcard operator. For example, the following query will return every page from `Help` namespace:

```
[[Help:+]]
```

Since `main` namespace does not really have a prefix, to select all the pages from `main` namespace, you have to use the following query:

```
[[[:+]]
```

What if we want to see all the categories currently created in `Category` namespace? You might want to use the following query:

```
[[Category:+]]
```

which actually returns every single page in the wiki which has been assigned a category value! Obviously, it is not what we want. In fact, the following query will do the trick:

```
[[[:Category:+]]
```

Note that a `:` is needed in front of the namespace to avoid confusion.

- Subqueries

Assume we want to find those companies that partner with any company that treats Microsoft as its competitor. This could be a useful query for the marketing department of Microsoft: the companies found by this query could become potential customers or partners of Microsoft.

To start, we can execute the following query first:

```
[[Category:Companies]][[Competitor::microsoft]]
```

which will find all the companies that treat Microsoft as their competitor. And at the time of this writing, these companies are in the returned set: Apple Computer, Opera Software, and TIBCO.

Next, we can execute the following query:

```
[[Partner::Apple Computer|Opera Software|TIBCO]]
```

which obviously accomplishes our goal. At the time of executing this query, only Showtime Networks is returned.

However, this query has taken two separate steps to finish. In fact, we can use the first query as a subquery within the second query to directly obtain the result

set. More specifically, we need to use `<q>` and `</q>` to enclose the subquery as follows:

```
[[Partner::<q>[[Category:Companies]]
[[Competitor::microsoft]]</q>]]
```

This will give us exactly the same result.

Up to this point, we have discussed some of the query features provided by Semantic MediaWiki, just to show the benefit offered by the added semantics. There are other query techniques you can use, such as templates and inline queries, which we will leave for you to explore. With what you have seen in this section, these techniques should be straightforward to understand.

9.3.2.3 Displaying Information

As you have seen from the previous section, query result in Semantic MediaWiki is defined as a set of pages, and each page in the result set must satisfy the query conditions. Semantic MediaWiki engine simply displays the titles of the result pages, and the user has to click a page to see more information on the page, such as the page’s property values or category values.

This is where the `Additional printouts` window on the `Special:Ask` page can be useful. We can enter the so-called printout statements into this window to show the property and category values that we are interested in.

There are two basic things to know about printout statements:

- All the printout statements must start with a question mark ?
- For a page contained in the query result set, if some printout statement has no value for this page, an empty field will simply be printed.

Let us see some examples. The following query

```
[[Sector::software]][[Region::California]]
[[Employees::>1000]]
```

with the printout statement

```
?Employees
```

will print out the values of `Employees` property of all the companies that are in software sector located in California region and also have no less than 1,000 employees. Note that the result is shown in a table column that is labeled by the name of the property. To save space, the result page is not included here, but you can definitely try it out to see the result.

You can in fact enter more than one printout statements in `Additional printouts` window to make the query result contain more information. For example,

```
?Employees
?Competitor
```

will add one more column called `Competitor` in the result table. As we have mentioned earlier, if a given page in the result set does not contain, for example, a value for `Competitor` property, an empty field will simply be used in the table. Also, note that the above two printout statements have to be on different lines in `Additional printouts` window; otherwise, they will not be recognized by the wiki engine.

Be aware that you can change the label in the output table. For example,

```
?Employees = Number of Employees
```

This will change the label from the default `Employees` to the new one `Number of Employees`, which is a very useful feature for the users.

Last but not the least, note that wildcards are often used together with printout statements to ensure that all the property value printouts have a non-empty value. For example, the following query

```
[[Category:Companies]][[Sector::software]]
[[Region::California]][[Employees::+]]
```

together with a printout statement

```
?Employees
```

has expressed this request: tell me the number of employees of each one of those companies that are in software business and also located in California. Because of this particular combination of query and the printout statement, the `Employee` column in the output table will never be empty, and that is exactly what we wanted.

9.3.3 *Inferencing*

Before we discuss the inferencing capability provided by semantic wiki engine, let us first understand the difference between semantic search and semantic inferencing.

In semantic wiki, semantic search is to find the requested pages based on the semantic annotations added by the users. It requires that semantic information be entered onto the page document ahead of the time when the search is conducted. On the other hand, semantic inferencing refers to the fact that users can retrieve

information that was not added explicitly by the users but derived by the semantic wiki engine.

Let us look at one example. Suppose in wikicompany, one user has created a page about a company called `companyA`, and he has also added some semantic markups into the page about `companyA`. However, the page is not marked with any category information: it does not have the tag `Category:Companies` associated with it. Why this could happen? Let us say our user simply forgot to do it.

The same user has also created a page for a different company called `companyB`. He has marked this page with `Category:Companies` information, and he has also added some semantic information to this page. Among others, one of the added semantic information is as follows:

```
[[competitor::companyA]]
```

meaning that `companyB` sees `companyA` as its competitor.

Now assume that the wiki engine settings in wikicompany site have disabled the semantic inferencing capability of the engine. For all the semantic search that involves `Category:Companies`, `companyA` will never be included in the result set, even its on-page semantic markups do satisfy the rest of the search requirements. This happens simply because the user has not marked `companyA` has a category value of `Category:Companies`, therefore the wiki engine is not able to realize that `companyA` is a company.

At this point, let us change the settings of the wiki engine, so the semantic inferencing capability is now set to `ON`. At the moment the engine scans the page of `companyB`, it finds the markup information which specifies that `companyA` is a competitor of `companyB`. This information will trigger the following inferencing process:

- Property `competitor` can only take instance of `Category:Companies` as its value (assume the wiki engine knows this by reading some ontology).
- Now property `competitor` has taken `companyA` as it value.
- Therefore `companyA` must be an instance of `Category:Companies`.

Now, the query result will be quite different: `companyA` will show up in at least some of the result sets.

Note that the user has not explicitly stated the fact that `companyA` is a `Category:Companies` instance; the wiki engine has been able to derive this fact and use it in related searches. Obviously, without semantic inferencing, only semantic search will not be able to accomplish the same result.

It is also worth mentioning that for a wiki user, query result is simply results – he does not know or cannot tell whether the result has come from some explicitly added semantic information or from some intelligent reasoning based on the ontology and instance data.

With all these being said, let us go back to the topic of inferencing capability in semantic wikis. The truth is that semantic wiki engine normally offers very

limited support for complex semantic knowledge representation and inferencing. Using Semantic MediaWiki engine as an example, we will briefly discuss inferencing capability offered by semantic wiki for the rest of this section. Note that in many of the wiki sites (including wikicompany), the inferencing capability is often disabled, therefore, we will not be able to see any examples.

The inferencing capability is based on two main hierarchies: the category hierarchy and the property hierarchy.

Semantic inferencing based on category hierarchy is quite straightforward: any search that involves a specific category will return all pages that belong to any subcategory of the given category.

For instance, in wikicompany site, *Networking* is currently created as a category, representing any company that is specialized in networking, and it has nothing to do with *Companies* category. However, networking company is a company indeed, therefore a better design is to make *Networking* category a subcategory of *Companies*. To do so, we can add the following markup to the page that represents *Networking* category, i.e., the page named `Category:Networking`:

```
[[Category:Companies]]
```

Once this is done, any company page that is marked as `Category:Networking` will be included in the following search:

```
[[Category:Companies]]
```

even if it is not marked as `[[Category:Companies]]`. In fact, we can simply understand the above search as follows:

```
[[Category:Companies]] OR [[Category:Networking]] OR
[[Category:otherSubCategory]]
```

Also, note that for any page, we should always mark it with the most specific category and depend on the category hierarchy-based inferencing to include the page into searching results. Otherwise, if we list all the related categories from the root until the most specific one, we will potentially have a long list of categories to maintain. This could become a nightmare especially when we have a fairly large category system.

The inferencing capability based on property hierarchy is quite similar. More specifically, a query that specifies a property name will return a page which contains that property's sub-property (direct or indirect) as part of its markup information even if the specified root property does not show up in that page. I will leave this to yourself to come up with examples. However, one thing to remember: `printout` statements do not perform any inferencing at all. For example, you could have a `printout` statement like this:

```
?propertyA
```

and for a specific page in the result set, if it contains only a sub-property of `propertyA`, this value field for this page will show up to be empty. The wiki engine will return only the values of the properties that are explicitly added onto the page.

At the time of this writing, inferencing based on category and property hierarchies are the only two inferencing mechanisms supported by the wiki engine. This decision is in fact a direct result from wiki engine's scalability concern: supporting transitivity properties, inverse properties, domain and range restrictions, etc. simply means a quickly growing computing complexity. Let me hope that at the time you are reading this chapter, we have a better inferencing engine in place.

At this point, we have covered most of the main features about semantic searching and inferencing. It is quite impressive to realize the fact that with the added semantic information, a semantic wiki engine can handle queries that will otherwise be completely impossible in traditional wiki sites. Indeed, take any query request from the previous section, and in order to find the results in a traditional wiki site, you probably have to spend days to read through thousands of pages!

So what is happening behind the scene? How is the added semantic information being used to deliver the performance like we have seen? The rest of this chapter will answer these questions.

9.4 Where Is the Semantics?

At this point, we have built enough background about semantic wiki, and it is time to ask the following questions:

- What ontology is in use when we add semantics to the wiki pages?
Semantic annotation in general requires the existence of at least one ontology. We use the terms (classes and properties) defined in the ontology to mark up the content. In semantic wiki, we mark up the pages in an ad hoc way: we either reuse some existing property or we invent a new one, then we assign a value to this property, and that is all we have done. The user who marks up the page may not even know the concept of ontology. So for the curious minds among us: where is the ontology? How is the semantics formally defined?
- Is ontology reuse possible between different semantic wikis? and similarly, is knowledge aggregation possible between different semantic wikis?
The Semantic Web is powerful and interesting, largely due to the fact that the data on the Web is linked. By the same token, if two different Wikis have some overlap in their domains, is it possible to aggregate the knowledge from both wikis?

Questions like these can be listed more. To gain more insight into semantic wiki, we will explore the answers in this section. The answers to these questions will point us to a brand new direction, as discussed in the last section of this chapter.

9.4.1 SWiVT: an Upper Ontology for Semantic Wiki

It turns out that there is indeed an ontology developed by the Semantic MediaWiki project team, and its name is *semantic wiki vocabulary and terminology* (SWiVT). This ontology is developed by using OWL DL, and it includes the most basic terms involved in the markup metadata model used by Semantic MediaWiki. Since the terms from SWiVT can be used in any wiki site powered by Semantic MediaWiki, SWiVT can be viewed as an upper ontology for semantic Wiki sites.

In this section, we will take a look at this ontology by covering only its main terms. With the discussion here, it should be a fairly easy task if you decide to explore SWiVT more on your own. Also, note that at the time of this writing, SWiVT is the only built-in ontology that Semantic MediaWiki uses.

Before we get into the details, the following are the namespace abbreviations that you should know, as summarized in Table 9.2.

- `swivt:Subject`

This class represents all things that could be described by wiki pages. In other words, all things that anyone may ever want to talk about. The following shows its definition:

```
<owl:Class rdf:ID="Subject">
  <rdfs:label xml:lang="en">Subject</rdfs:label>
  <rdfs:comment xml:lang="en">
    An individual entity that is described on a wiki page.
    Can be basically anything.
  </rdfs:comment>
</owl:Class>
```

Using the namespace abbreviations in Table 9.2, the complete URI for this class is given as

<http://semantic-mediawiki.org/swivt/1.0#Subject>

Table 9.2 Namespaces and their abbreviations

Namespace	Abbreviations
http://www.w3.org/1999/02/22-rdf-syntax-ns#	rdf
http://www.w3.org/2000/01/rdf-schema#	rdfs
http://www.w3.org/2002/07/owl#	owl
http://semantic-mediawiki.org/swivt/1.0#	swivt
http://wikicompany.org/wiki/Special:URIResolver/	wiki
http://wikicompany.org/wiki/Special:URIResolver/Property-3A	property
http://wikicompany.org/wiki/	wikiurl

- `swiwt:Wikipage`

Obviously, an instance of `swiwt:Subject` represents an individual topic, which requires a concrete wiki page to describe it. Class `swiwt:Wikipage` is defined to represent the page itself:

```
<owl:Class rdf:ID="Wikipage">
  <rdfs:label xml:lang="en">Wikipage</rdfs:label>
  <rdfs:comment xml:lang="en">A page in a wiki</rdfs:comment>
</owl:Class>
```

- `swiwt:page`

To establish the link between an instance of `swiwt:Subject` and a concrete `swiwt:Wikipage` which describes the subject, a property called `swiwt:page` is created:

```
<owl:AnnotationProperty rdf:ID="page">
  <rdfs:label xml:lang="en">Page</rdfs:label>
  <rdfs:comment xml:lang="en">
    Connects a swiwt:Subject (URI) with the according
    swiwt:Wikipage (URL).
  </rdfs:comment>
</owl:AnnotationProperty>
```

Note that SWiVT ontology is written in OWL DL. Therefore, `swiwt:page` as an annotation property does not have any domain/range constraints specified. The same is true for all other annotation properties defined in SWiVT ontology.

- `swiwt:Type`

We have seen the importance of datatype in Semantic MediaWiki engine, and datatype is heavily used when marking up the information on a given page. To formally describe datatype system, SWiVT first defines the `swiwt:Type` class:

```
<owl:Class rdf:ID="Type"/>
```

This class is used as the base class for different datatypes in the wiki. Using the datatype information of a given property, the wiki engine is able to understand pragmatically how to represent the property value on the page while storing the semantic information available for other application.

- `swiwt:BuiltinType`

Class `swiwt:BuiltinType` is used to represent a set of built-in datatypes offered by Semantic MediaWiki:

```
<owl:Class rdf:ID="BuiltinType">
  <rdfs:label xml:lang="en">Builtin type</rdfs:label>
```



```
<rdfs:subClassOf rdf:resource="#Type" />
</owl:Class>
```

As an example, here is the definition of `Page` type, which is created as one instance of `swikt:BuiltInType` class:

```
<swikt:BuiltInType rdf:ID="PageType">
  <rdfs:label xml:lang="en">Page</rdfs:label>
</swikt:BuiltInType>
```

Other built-in types are all defined similarly. For example, `swikt:StringType`, `swikt:NumberType`, and `swikt:TextType`. Note that compared to `swikt:StringType`, `swikt:TextType` can be of any length.

- `swikt:CustomType`

Semantic MediaWiki allows the users to create their own customer datatypes, and `swikt:CustomType` is defined to represent these customized datatypes:

```
<owl:Class rdf:ID="CustomType">
  <rdfs:subClassOf rdf:resource="#Type" />
</owl:Class>
```

9.4.2 Understanding OWL/RDF Exports

Given the fact that we have an ontology (SWiVT) defining the semantics for the metadata model used in Semantic MediaWiki, a question comes to mind is, why we have not used it when marking up the pages?

The answer is quite simple: for ordinary wiki users, they don't have to see and understand this ontology when adding markups to the pages. However, it is very useful when the semantic content on a given page is exported.

Let us go back to our example about Apple Computer in `wikicompany`. When the wiki engine scans the page for Apple Computer, the user-added semantic markups are collected and parsed, and an RDF file is generated to represent the semantic content on this page. This RDF document can be exported for machine to understand and use.

To get this document, open the page for Apple Computer in `wikicompany`, then move down to the very bottom of the page. At the top of the Factbox, you will see a link called `RDF feed`, click this link, the exported RDF file will appear.

Note that at the time when you read this book, the exported file you will see probably will be different from what I obtained when this writing is done (as shown in List 9.1). So refer to List 9.1 when reading this section. However, the content in this section will still help you to understand the exported data no matter how the exported data may change.

List 9.1 RDF output of Apple Computer page

```

1: <?xml version="1.0" encoding="UTF-8"?>
2: <!DOCTYPE rdf:RDF[
3:   <!ENTITY rdf 'http://www.w3.org/1999/02/22-rdf-syntax-ns#'>
4:   <!ENTITY rdfs 'http://www.w3.org/2000/01/rdf-schema#'>
5:   <!ENTITY owl 'http://www.w3.org/2002/07/owl#'>
6:   <!ENTITY swivt 'http://semantic-mediawiki.org/swivt/1.0#'>
7:   <!ENTITY
7a:     wiki 'http://wikicompany.org/wiki/Special:URIResolver/'>
8:   <!ENTITY property
8a:     'http://wikicompany.org/wiki/Special:URIResolver/
8b:     Property-3A'>
9:   <!ENTITY wikiurl 'http://wikicompany.org/wiki/'>
10: ]>
11:
12: <rdf:RDF
13:   xmlns:rdf="&rdf;"
14:   xmlns:rdfs="&rdfs;"
15:   xmlns:owl="&owl;"
16:   xmlns:swivt="&swivt;"
17:   xmlns:wiki="&wiki;"
18:   xmlns:property="&property;">
19:   <!-- Ontology header -->
20:
21:   <owl:Ontology rdf:about="">
22:     <swivt:creationDate rdf:datatype=
22a:       "http://www.w3.org/2001/XMLSchema#dateTime">
22b:       2009-01-09T19:07:24-06:00
22c:     </swivt:creationDate>
23:     <owl:imports rdf:resource=
23a:       "http://semantic-mediawiki.org/swivt/1.0" />
24:   </owl:Ontology>
25:   <!-- exported page data -->
26:   <swivt:Subject rdf:about="&wiki;Apple_Computer">
27:     <rdfs:label>Apple Computer</rdfs:label>
28:     <swivt:page rdf:resource="&wikiurl;Apple_Computer"/>
29:
30:     <rdfs:isDefinedBy rdf:resource=
30a:       "&wikiurl;Special:ExportRDF/Apple_Computer"/>
31:     <rdf:type rdf:resource="&wiki;Category-3ACompanies"/>
32:     <property:Address rdf:datatype=
32a:       "http://www.w3.org/2001/XMLSchema#string">
32b:       1 Infinite Loop, Cupertino
32c:       California 95014, United States

```

```

32d: </property:Address>
33: <property:Competitor rdf:resource="&wiki;Microsoft"/>
34: <property:Coordinates rdf:datatype=
34a:     "http://www.w3.org/2001/XMLSchema#string">
34b:     37°19'49.98"N, 122°1'46.92"W
34c: </property:Coordinates>
35: <property:Employees rdf:datatype=
35a:     "http://www.w3.org/2001/XMLSchema#double">
35b:     13500
35c: </property:Employees>
36: <property:Exchange rdf:datatype=
36a:     "http://www.w3.org/2001/XMLSchema#string">
36b:     NASDAQ
36c: </property:Exchange>
37:
38: <property:Fax rdf:datatype=
38a:     "http://www.w3.org/2001/XMLSchema#string">
38b: </property:Fax>
39: <property:Founding rdf:resource="&wiki;1976"/>
40: <property:Fullname rdf:datatype=
40a:     "http://www.w3.org/2001/XMLSchema#string">
40b:     Apple Computer, Inc.
40c: </property:Fullname>
41: <property:Homepage rdf:resource="http://www.apple.com"/>
42: <property:Phone rdf:datatype=
42a:     "http://www.w3.org/2001/XMLSchema#string">
42b:     +1-800-676-2775
42c: </property:Phone>
43: <property:Region rdf:datatype=
43a:     "http://www.w3.org/2001/XMLSchema#string">
43b:     Cupertino
43c: </property:Region>
44: <property:Region rdf:datatype=
44a:     "http://www.w3.org/2001/XMLSchema#string">
44b:     California
44c: </property:Region>
45:
46: <property:Region rdf:datatype=
46a:     "http://www.w3.org/2001/XMLSchema#string">
46b:     United States
46c: </property:Region>
47: <property:Region rdf:datatype=
47a:     "http://www.w3.org/2001/XMLSchema#string">
47b:     World
47c: </property:Region>

```

```

48:      <property:Sector rdf:datatype=
48a:          "http://www.w3.org/2001/XMLSchema#string">
48b:          Desktop
48c:      </property:Sector>
49:      <property:Sector rdf:datatype=
49a:          "http://www.w3.org/2001/XMLSchema#string">
49b:          software
49c:      </property:Sector>
50:      <property:Sector rdf:datatype=
50a:          "http://www.w3.org/2001/XMLSchema#string">
50b:          Computer
50c:      </property:Sector>
51:      <property:Sector rdf:datatype=
51a:          "http://www.w3.org/2001/XMLSchema#string">
51b:          hardware
51c:      </property:Sector>
52:
53:      <property:Sector rdf:datatype=
53a:          "http://www.w3.org/2001/XMLSchema#string">
53b:          media
53c:      </property:Sector>
54:      <property:Sector rdf:datatype=
54a:          "http://www.w3.org/2001/XMLSchema#string">
54b:          music
54c:      </property:Sector>
55:      <property:Sector rdf:datatype=
55a:          "http://www.w3.org/2001/XMLSchema#string">
55b:          online
55c:      </property:Sector>
56:      <property:Sector rdf:datatype=
56a:          "http://www.w3.org/2001/XMLSchema#string">
56b:          store
56c:      </property:Sector>
57:      <property:Ticker rdf:datatype=
57a:          "http://www.w3.org/2001/XMLSchema#string">
57b:          NASDAQ.AAPL
57c:      </property:Ticker>
58: </swivt:Subject>
59:
60: <!-- auxilliary definitions -->
61: <owl:DatatypeProperty rdf:about="&property;Ticker">
62:   <rdfs:label>Ticker</rdfs:label>
63:   <swivt:page rdf:resource="&wikiurl;Property:Ticker"/>
64:   <rdfs:isDefinedBy rdf:resource=
64a:       "&wikiurl;Special:ExportRDF/Property:Ticker"/>

```

```

65:   </owl:DatatypeProperty>
66:   <owl:DatatypeProperty rdf:about="&property;Sector">
67:     <rdfs:label>Sector</rdfs:label>
68:
69:     <swikt:page rdf:resource="&wikiurl;Property:Sector" />
70:     <rdfs:isDefinedBy rdf:resource=
71a:       "&wikiurl;Special:ExportRDF/Property:Sector" />
71:   </owl:DatatypeProperty>
72:   <owl:DatatypeProperty rdf:about="&property;Region">
73:     <rdfs:label>Region</rdfs:label>
74:     <swikt:page rdf:resource="&wikiurl;Property:Region" />
75:     <rdfs:isDefinedBy rdf:resource=
76a:       "&wikiurl;Special:ExportRDF/Property:Region" />
76:   </owl:DatatypeProperty>
77:
78:   <owl:DatatypeProperty rdf:about="&property;Phone">
79:     <rdfs:label>Phone</rdfs:label>
80:     <swikt:page rdf:resource="&wikiurl;Property:Phone" />
81:     <rdfs:isDefinedBy rdf:resource=
82a:       "&wikiurl;Special:ExportRDF/Property:Phone" />
82:   </owl:DatatypeProperty>
83:   <owl:ObjectProperty rdf:about="&property;Homepage">
84:     <rdfs:label>Homepage</rdfs:label>
85:     <swikt:page rdf:resource="&wikiurl;Property:Homepage" />
86:
87:     <rdfs:isDefinedBy rdf:resource=
88a:       "&wikiurl;Special:ExportRDF/Property:Homepage" />
88:   </owl:ObjectProperty>
89:   <owl:DatatypeProperty rdf:about="&property;Fullname">
90:     <rdfs:label>Fullname</rdfs:label>
91:     <swikt:page rdf:resource="&wikiurl;Property:Fullname" />
92:     <rdfs:isDefinedBy rdf:resource=
93a:       "&wikiurl;Special:ExportRDF/Property:Fullname" />
93:   </owl:DatatypeProperty>
94:   <swikt:Subject rdf:about="&wiki;1976">
95:
96:     <rdfs:label>1976</rdfs:label>
97:     <swikt:page rdf:resource="&wikiurl;1976" />
98:     <rdfs:isDefinedBy rdf:resource=
99a:       "&wikiurl;Special:ExportRDF/1976" />
99:   </swikt:Subject>
100:  <owl:ObjectProperty rdf:about="&property;Founding">
101:    <rdfs:label>Founding</rdfs:label>
102:    <swikt:page rdf:resource="&wikiurl;Property:Founding" />
103:    <rdfs:isDefinedBy rdf:resource=

```

```

103a:         "&wikiurl;Special:ExportRDF/Property:Founding" />
104:
105:     </owl:ObjectProperty>
106:     <owl:DatatypeProperty rdf:about="&property;Fax">
107:         <rdfs:label>Fax</rdfs:label>
108:         <swikt:page rdf:resource="&wikiurl;Property:Fax" />
109:         <rdfs:isDefinedBy rdf:resource=
109a:             "&wikiurl;Special:ExportRDF/Property:Fax" />
110:     </owl:DatatypeProperty>
111:     <owl:DatatypeProperty rdf:about="&property;Exchange">
112:         <rdfs:label>Exchange</rdfs:label>
113:
114:         <swikt:page rdf:resource="&wikiurl;Property:Exchange" />
115:         <rdfs:isDefinedBy rdf:resource=
115a:             "&wikiurl;Special:ExportRDF/Property:Exchange" />
116:     </owl:DatatypeProperty>
117:     <owl:DatatypeProperty rdf:about="&property;Employees">
118:         <rdfs:label>Employees</rdfs:label>
119:         <swikt:page rdf:resource="&wikiurl;Property:Employees" />
120:         <rdfs:isDefinedBy rdf:resource=
120a:             "&wikiurl;Special:ExportRDF/Property:Employees" />
121:     </owl:DatatypeProperty>
122:
123:     <owl:DatatypeProperty rdf:about="&property;Coordinates">
124:         <rdfs:label>Coordinates</rdfs:label>
125:         <swikt:page
125a:             rdf:resource="&wikiurl;Property:Coordinates" />
126:         <rdfs:isDefinedBy rdf:resource=
126a:             "&wikiurl;Special:ExportRDF/Property:Coordinates" />
127:     </owl:DatatypeProperty>
128:     <swikt:Subject rdf:about="&wiki;Microsoft">
129:         <rdfs:label>Microsoft</rdfs:label>
130:         <swikt:page rdf:resource="&wikiurl;Microsoft" />
131:
132:         <rdfs:isDefinedBy rdf:resource=
132a:             "&wikiurl;Special:ExportRDF/Microsoft" />
133:     </swikt:Subject>
134:     <owl:ObjectProperty rdf:about="&property;Competitor">
135:         <rdfs:label>Competitor</rdfs:label>
136:         <swikt:page
136a:             rdf:resource="&wikiurl;Property:Competitor" />
137:         <rdfs:isDefinedBy rdf:resource=
137a:             "&wikiurl;Special:ExportRDF/Property:Competitor" />
138:     </owl:ObjectProperty>
139:     <owl:DatatypeProperty rdf:about="&property;Address">

```

```

140:
141:     <rdfs:label>Address</rdfs:label>
142:     <swivt:page rdf:resource="&wikiurl;Property:Address" />
143:     <rdfs:isDefinedBy rdf:resource=
143a:         "&wikiurl;Special:ExportRDF/Property:Address" />
144: </owl:DatatypeProperty>
145: <owl:Class rdf:about="&wiki;Category-3ACompanies">
146:     <rdfs:label>Companies</rdfs:label>
147:     <swivt:page rdf:resource="&wikiurl;Category:Companies" />
148:     <rdfs:isDefinedBy rdf:resource=
148a:         "&wikiurl;Special:ExportRDF/Category:Companies" />
149:
150: </owl:Class>
151: <!-- References to the SWiVT Ontology,
151a:     see http://semantic-mediawiki.org/swivt/ -->
152: <owl:AnnotationProperty rdf:about="&swivt;page">
153:     <rdfs:isDefinedBy rdf:resource=
153a:         "http://semantic-mediawiki.org/swivt/1.0" />
154: </owl:AnnotationProperty>
155: <owl:AnnotationProperty rdf:about="&swivt;creationDate">
156:     <rdfs:isDefinedBy rdf:resource=
156a:         "http://semantic-mediawiki.org/swivt/1.0" />
157: </owl:AnnotationProperty>
158: <owl:Class rdf:about="&swivt;subject">
159:
160:     <rdfs:isDefinedBy rdf:resource=
160a:         "http://semantic-mediawiki.org/swivt/1.0" />
161: </owl:Class>
162: <!-- Created by Semantic MediaWiki,
162a:     http://semantic-mediawiki.org -->
163: </rdf:RDF>

```

Now, let us take a look at the exported RDF file. First of all, with the discussion of the upper ontology in the previous section, it is easy to understand the content presented by lines 152–158, which is just a reiteration of some of the main classes defined in SWiVT ontology.

Since SWiVT is only an upper ontology, it is therefore up to the wiki site to add its own application-specific class definitions. One such class definition is seen in lines 145–150, and it has the following URI:

```
wiki:Category-3ACompanies
```

The full URI can be obtained by replacing the `wiki` prefix (see Table 9.2):

```
http://wikicompany.org/wiki/Special:URIResolver/Category-3ACompanies
```

Note that two properties are used on `wiki:Category-3ACompanies` class; they are `rdfs:label` and `swikt:page`. `rdfs:label` has `Companies` as its value, and `swikt:page` has the following individual document as its value:

```
http://wikicompany.org/wiki/Category:Companies
```

If you follow this URL, you actually land on `Category:Companies` page; the page category `Companies` is in `Category` namespace. Remember `Companies` also represents the category of the page for Apple Computer.

In general, each category in the `Category` namespace will be mapped to an application-specific class defined by the wiki site. When generating RDF document for a given page, the entire page will be represented as an instance of the class that corresponds to the category of the page.

With an application-specific class type representing a given wiki page, a set of properties will also be needed in order to describe the formal semantics of the page. For example, lines 117–121 define a property that we are familiar with: `Employees` property, which is identified by the following URI:

```
property:Employees
```

Its full URI can be obtained by replacing the `property` prefix (see Table 9.2):

```
http://wikicompany.org/wiki/Special:URIResolver/Property-3AEmployees
```

This property has the following details:

```
property:Employees rdf:type owl:DatatypeProperty.
property:Employees rdfs:label "Employees".
property:Employees swikt:page
    <http://wikicompany.org/wiki/Property:Employees>.
```

Similarly, if you follow the URL given as the value of `swikt:page` property, you will land on `Property:Employees` page, which describes `Employees` property, and this page itself is collected in the `Property` namespace.

Another property is the `Competitor` property, defined by lines 134–138. It has the following URI:

```
property:Competitor
```

with the following details:

```
property:Competitor rdf:type owl:ObjectProperty.
property:Competitor rdfs:label "Competitor".
property:Competitor swikt:page
    <http://wikicompany.org/wiki/Property:Competitor>.
```

Again, the value of `swikt:page` property points to `Property:Competitor` page, which describes `Competitor` property in the `Property` namespace.

Table 9.3 Property URIs and their line numbers

Property URI	Line numbers
<code>property:Ticker</code>	61–65
<code>property:Sector</code>	66–71
<code>property:Region</code>	72–76
<code>property:Phone</code>	78–82
<code>property:Homepage</code>	83–88
<code>property:Fullname</code>	89–93
<code>property:Founding</code>	100–105
<code>property:Fax</code>	106–110
<code>property:Exchange</code>	111–116
<code>property:Coordinates</code>	123–127
<code>property:Address</code>	139–144

Besides the above two examples, the exported RDF document has defined quite a few other properties. Table 9.3 summarizes these properties and the line numbers where they are defined.

In general, each property in the `Property` namespace will be mapped to an application-specific property defined by the wiki site. When generating RDF document for a given page, the properties used by the user when annotating the page will be instantiated and included in the exported RDF file, therefore become part of the semantic information.

We are now able to easily understand the rest of List 9.1. First off, line 26 creates an instance of class `swikt:Subject` to represent Apple Computer, which has the following URI:

```
wiki:Apple_Computer
```

Its full URI can be obtained by replacing the `wiki` prefix (see Table 9.2):

```
http://wikicompany.org/wiki/Special:URIResolver/Apple_Computer
```

It is also declared as an instance of `wiki:Category-3Acompanies` class (line 31). This is where the existing category system is used in the semantic wiki.

Lines 32–57 include all the properties that have been used to describe Apple Computer, together with their values and datatypes. For example, line 35 creates an instance of `Employees` property, its value is 13500, with a datatype defined as <http://www.w3.org/2001/XMLSchema#double>. Clearly, this piece of data in the exported RDF file corresponds to the markup for the employee information on Apple Computer’s page.

It is interesting to note that the definition of `Employees` property does not make use of `rdfs:range` property to put constraints on its possible values. So how does the wiki engine get its value when it generates the RDF file? The following summary not only helps us to answer this question but also gives us a better understanding about semantic wiki as a whole.

1. Wiki engine loads a page.
2. Wiki engine parses the annotated information on this page. For each property, the engine creates a definition using OWL DL, and these definitions will be included in the exported RDF file.
3. For each property, the engine tries to load a page from `Property` namespace, which provides a description about this property if it exists in the namespace.
4. If the engine loads the page successfully, it will search a special property called `Property:Has type` within the page. The value of this property will be used to create the property instance in the exported RDF file.
5. If the engine cannot find the page or cannot find `Property:Has type` property, the underlying property is then assumed to have a `Page` type, and the rest will be the same as described in step 4.

Another example is in line 33, which creates an instance of `Competitor` property. This property use another object, i.e. `wiki:Microsoft`, as its value. Note that lines 128–133 give the definition of `wiki:Microsoft`, which is defined as an instance of `swikt:Subject`, and it has the following URI:

```
http://wikicompany.org/wiki/Special:URIResolver/Microsoft
```

Obviously, there is a page in this wiki that describes Microsoft as well. In addition, just like Apple Computer page, when its RDF document is generated, Microsoft page will be mapped to an instance of both `wiki:Category-3Acompanies` class and `swikt:Subject` class, and it should have the same URI as above to make sure that it represents exactly the same Microsoft as used here.

To confirm this, go to wikicompany, open up the page for Microsoft, and click `RDF feed` to request the exported RDF file. As you can tell, an instance is created in the output file to represent Microsoft page, and the URI of that instance is exactly the same as the one given in line 128 of List 9.1.

By the same token, you should also realize the fact that property definition has to be consistent as well. For example, `Employees` property appears on Apple Computer page, and it also appears on Microsoft page. When exporting the RDF file for Apple Computer page, wiki engine has defined `Employees` property to have the following URI (line 117):

```
http://wikicompany.org/wiki/Special:URIResolver/Property-3AEmployees
```

When exporting the RDF file for Microsoft page, wiki engine will have to define `Employees` property again. Furthermore, it will guarantee that the URI for this property will be exactly the same as the above URI so as to make sure that this `Employees` property is the same as that `Employees` property. Again, you can easily confirm this by asking the RDF feed on Microsoft page.

The above may seem obvious, but it is necessary to make sure a complete and consistent semantics has been defined, which is further responsible for many of the powerful searching features we have seen on the semantic wiki sites.

At this point, we have finished our discussion about ontologies and RDF exports in a semantic wiki site. The powerful search features provided by semantic wiki can be viewed as an application that is directly built upon the ontology and generated RDF documents. With what we have learned here, it is your turn now to come up with other application examples that work on the exported RDF files.

9.4.3 *Importing Ontology: a Bridge to Outside World*

With our understanding so far, we can summarize the following about semantic wiki:

- A given wiki site always uses its own ontology, which is created by mapping the category information and the property names to their appropriate OWL terms, as we have discussed in Sect. 9.4.2.
- Classes and properties defined in this ontology are therefore local to the wiki and are invisible to the outside world.

For instance, within Semantic MediaWiki, the following rule is used to come up with the URI for a class:

```
http://wiki_site_name/wiki/Special:URIResolver/Category-3ACategoryName
```

and similarly, the URI of a property has the following format:

```
http://wiki_site_name/wiki/Special:URIResolver/Property-3APropertyName
```

Recall that in wikicompany, class `Companies` has the following URI:

```
http://wikicompany.org/wiki/Special:URIResolver/Category-3ACompanies
```

and property `Competitor` has an URI that looks like

```
http://wikicompany.org/wiki/Special:URIResolver/Property-3ACompetitor
```

This is all neat and clean, and it also explains why when exporting the RDF files for different pages, wiki engine is able to make sure that the same property and class have the same URI.

However, this “locality” implies the fact that there are no links to the outside world, and it is therefore difficult for outside tools or applications to work with the exported RDF data. And as we know, data will be most useful when they are linked together.

One solution is to reuse existing ontologies whenever it is possible, which will improve compatibility with other tools. Fortunately, Semantic MediaWiki engine does provide a mechanism to accomplish this. In this section, we will use FOAF ontology as an example to take a brief look at how this is done.

To reuse FOAF ontology in a given wiki site, the first step is to make sure that this ontology is visible to the site. To accomplish this, a user with administrator status will have to create a specific page that has a unique name just for the purpose of reusing ontologies. More specifically, this page will have to be in MediaWiki namespace with a prefix `smw_import_`. For our case, since we want to reuse FOAF ontology, this page can have the following name:

```
MediaWiki:smw_import_foaf
```

At the time of this writing, wikicompany site has not yet imported any outside ontology. Therefore, to see a real example of this page, we will have to go to another wiki site called semanticWeb wiki:

```
http://semanticweb.org/wiki/Main_Page
```

which is also a semantic wiki powered by Semantic MediaWiki.

Now, we can open up the query page from the following URL:

```
http://semanticweb.org/wiki/Special:Ask
```

and type in the following query:

```
[[MediaWiki:+]]
```

which asks for all the pages in MediaWiki namespace. Once the result is back, we should see a page that has a name called `MediaWiki:smw_import_foaf`.

Open up this page, we see the following lines on this page (note that the line numbers are added for explanation purpose):

```
1: http://xmlns.com/foaf/0.1/|[http://www.foaf-project.org/
1a: Friend Of A Friend]
2: name|Type:String
3: homepage|Type:URL
4: mbox|Type:Email
4: mbox_sha1sum|Type:String
5: depiction|Type:URL
6: phone|Type:String
7: Person|Category
8: Organization|Category
9: knows|Type:Page
10:member|Type:Page
```

Line 1 maps the last part of the page name to a real namespace (the string from the beginning of line 1 until the | sign). In our case, `foaf` is the last part of the page name, and it therefore maps to the following namespace:

```
http://xmlns.com/foaf/0.1/
```

which is exactly the namespace used by FOAF ontology. The string after the | sign provides a human-readable name for the ontology and a link that user can click to get more information about the imported ontology.

Note that not every single term (class or property) defined in the ontology is automatically imported. In fact, we have to explicitly specify which term will get imported and once it is imported, how will it be used in the wiki. For example, line 7 says `foaf:Person` will be imported, and the text after `|` sign specifies that `Person` will be used as a class (a `Category` maps to a class, as discussed in Sect. 9.4.2). Similarly, line 2 says `foaf:name` will be imported, and it will be used as a property with `String` as its type. You can read the rest of the page in the same way, and for this ontology, only nine terms (two classes and seven properties) have been imported.

The second step is to modify the pages of the corresponding categories and properties in the wiki to state the facts that these categories and properties are the same as the corresponding imported terms.

For example, in `semanticWeb` wiki, we need to specify `Category:Person` is the same as `foaf:Person`, and `foaf:Person` should be used whenever `Category:Person` shows up in any RDF exported file. To do so, we need to modify the page of `Category:Person` in `Category` namespace by adding the following line into the page:

```
[[imported from:=foaf:Person]]
```

which will do the trick: the wiki engine, when exporting RDF files, will use `foaf:Person` as the class whenever it sees `Category:Person`. Again, note that you can find `Category:Person` page at the following URL:

<http://semanticweb.org/wiki/Category:Person>

Similarly, to map a property to an imported property term, we need to make change to the property page. For example, we can change `Property:Name` page to add the fact that `Property:Name` is represented by FOAF term `foaf:name`. To do so, we can open the page of `Property:Name` from the following URL:

<http://semanticweb.org/wiki/Property:Name>

then we add the following line into the page:

```
[[imported from:=foaf:name]]
```

Remember to remove `Type:Has` type statement if it exists on the page, since page `MediaWiki:smw_import_foaf` has already specified the type for every imported item, as you have seen earlier.

Also note that when annotating the page, we don't have to use `foaf:name`. Instead, we should continue to use the property name as it is defined earlier. The wiki engine will find and replace all the mapped terms when it exports the RDF files.

The above is the basic flow for reusing existing ontologies, and it is included here for you to get some basic understanding about this topic. Given the rapid development in semantic wiki area and since ontology reuse promotes the Linked Data idea, more improvement on this issue is expected.

9.5 The Power of the Semantic Web

Now it is the time to answer the question we have in mind for quite a while. The semantic markup data users have added on each page (in `main` namespace) has made some big difference when searching for the information from the wiki. So what is happening behind the scene? How is the added markup information used by the wiki engine?

It turns out that semantic annotation is just a small part of the whole story, and the major operation happens inside the wiki engine and can be summarized as follows:

1. For a given user page in `main` namespace, wiki engine collects and parses the added semantic information to produce an RDF document that represents a machine-readable version of the page.
2. Wiki engine saves the RDF document into a database called semantic store.
3. Wiki engine moves on to the next user page, and repeat steps 1 and 2. If there is no other user page found, stop.

We have seen an example of RDF document in Sect. 9.4.2 already: the RDF file generated for Apple Computer. This RDF document can be expressed by a graph shown in Fig. 9.7 (note that it is not a complete graph, but it shows the main idea).

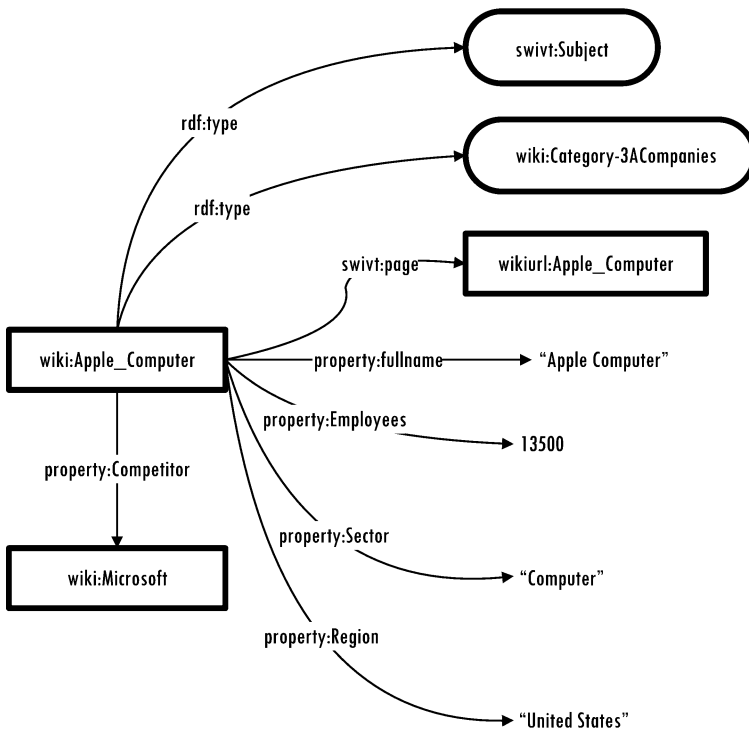


Fig. 9.7 RDF model for Apple Computer is expressed as a graph

Now, for each user page, a graph like Fig. 9.7 is created and saved in a database. With links among these graphs, they all together form a collection of linked data inside the wiki, and any query submitted by the user is in fact run against this dataset. This is why your query can provide accurate and relevant results all the time – something a traditional wiki can hardly match.

Figure 9.7 also lets you visualize the link between graphs. More specifically, `wiki:Apple_Computer` is the root node and instance `wiki:Microsoft` shows up in this graph as a value of `Competitor` property. Meanwhile, in the graph that represents Microsoft page, the same instance will be the root node.

Also note the difference between semantic search discussed in this chapter and full text search used in traditional wiki sites. Semantic search is conducted against a Web of linked data; the result of semantic search is answers, not pages. However, full text search is based on the keyword indexing system, and the result is a set of pages, which a user has to sift through for answers.

9.6 Use Semantic MediaWiki to Build Your Own Semantic Wiki

This section provides some brief guidelines if you are planning to build your own semantic wiki site.

First you need to understand that semantic wiki engine itself is not a semantic wiki site; it is a software that provides a framework for building a semantic wiki site. There are quite a few semantic wiki engines available to use, so you need to decide first which engine to use.

If you decide to use Semantic MediaWiki as the engine, and since Semantic MediaWiki is an extension to MediaWiki engine, you need to install a MediaWiki first. In addition, you need to make sure that it works well before you install Semantic MediaWiki.

After MediaWiki is working, you can install Semantic MediaWiki on top of it. As an extension to MediaWiki, Semantic MediaWiki requires very little configuration work to make the system work, and you will always have chance to configure and modify the engine in the course of using it.

Finally, understand that once you have a semantic wiki, it is the added markups that make it work. And remember to get help from this page on Semantic MediaWiki's official site; it always contains the most up-to-date information for you to work on your own site:

http://semantic-mediawiki.org/wiki/Help:Administrator_manual

9.7 Summary

In this chapter, we have learned semantic wiki, another example of the Semantic Web technologies at work.

First off, understand that semantic markup information in a semantic wiki is added manually by the users:

- A user can mark up both links and text on a given page.
- Category information and properties are used to mark up the page. It is not necessary for a user to understand ontology or RDF in order to add markup information.
- User can add new categories and create new properties, but reusing existing categories and properties is always recommended.

The added markup information is used by semantic wiki engine to facilitate semantic search in the wiki. To conduct search, a special search language is used.

And behind the scene, the following has happened to make semantic search possible:

- The semantic markup on each page is collected and parsed by the wiki engine; an RDF document is generated to represent the page.
- This is repeated for all the pages, and all the generated RDF documents therefore form a linked dataset, which is used for the semantic search.

Also understand the following about this dataset:

- SWiVT, an upper ontology for all the wiki site, is created so that we can have a high level vocabulary.
- For each specific wiki site, new classes and properties are defined. More specifically, categories are mapped to classes, and page properties are mapped to properties.
- When generating RDF document for each page, both SWiVT and the new classes and properties are used.

Finally, understand that the impressive searching capability provided by semantic wiki engine can hardly be matched by any traditional wiki site, and all is the result of some simple markup that a user is willing to add.

Chapter 10

DBpedia

At the end of [Chap. 7](#), we have discussed the topic about semantic markup. More specifically, it is possible to automatically generate markup documents for some Web content, especially when there is pre-existing structured information contained in these content.

This chapter will provide one such example, and this is the popular DBpedia project. In fact, it is important to understand DBpedia, not only as an example of automatically generating structured data from Web content, but also because of its key position in the Web of Linked Data, as we will see in [Chap. 11](#).

10.1 Introduction to DBpedia

10.1.1 From Manual Markup to Automatic Generation of Annotation

As we have learned by now, the classic view about the Semantic Web is to add semantic annotations to each page by using RDF data model and ontologies so that the added information can be processed by a machine.

We have discussed semantic wiki in the previous chapter. We understand that in order to make a semantic wiki site work, the user of the wiki has to manually enter the semantic markups at the first place. As an example that we have studied, Semantic MediaWiki has modified its original markup language, therefore when editing the wiki page, the user can easily annotate the page at the same time. Furthermore, the benefit of the added semantics is shown by the wiki's ability to answer complex questions, as we have seen in the previous chapter.

However, in general, manually adding semantic annotations for a large-scale application is quite difficult. For instance, it requires the availability of widely accepted ontologies, and it requires manually marking up millions of pages, which is simply not practical or at least formidably expensive. In addition, what about the new pages that are generated each day? How do we require an ordinary user to conquer the learning curve and go through the extra steps to mark up the page?

The reason why the manual approach is successful when applied to semantic wiki sites is mainly due to the uniqueness of the wiki sites themselves. More specifically, these sites normally have limited scopes and quite often are only used internally by some organizations. It is therefore not necessary to have some standard ontologies built before hand, and a home-grown ontology is often good enough for the goal. Also, the markup language provided by a semantic wiki engine is quite simple and not much more complex than or different from the original wikitext.

Obviously, not every application can offer a favorable environment to manual approach. To overcome its difficulty, another approach has become popular in recent years, where the semantic annotation information is automatically generated. Instead of independently adding semantic markups to the current Web document, this approach tries to derive semantic information automatically from the existing structured information contained in the Web document.

The main attraction of this automatic approach is the fact that it does not require much manual work, therefore quite scalable. However, it does have to deal with the imperfectness of information on each page, and how well it works heavily depends upon how much structured information contained in a given page.

In this chapter, we are going to study one example system that is built solely by using this automatic approach: the DBpedia project. Once you finish this chapter, you will have examples of both approaches, which should be valuable for your future development work.

Note that these two approaches can also benefit from each other. For example, the more manual annotations there are, the more precise the automatic approach will be, since it can directly take advantage of the existing structured information. Similarly, automatically generating the semantic information can minimize the need for manual approach, and combining these two approaches can sometimes be the best solution at your hand.

10.1.2 From Wikipedia to DBpedia

The most successful and popular wiki by far is probably Wikipedia, the largest online encyclopedia created and maintained by a globally distributed author community. At the time of this writing, Wikipedia appears in more than 251 different languages, with the English version containing more than 3.1 million articles. You can access Wikipedia from the following URL:

`http://en.wikipedia.org/wiki/Main_Page`

And you will be amazed by how much information it can provide.

However, as we have discussed in previous chapter, similar to any other traditional wiki site, using Wikipedia means reading it. It is up to you to digest all the information and to search through pages to find what you are looking for.

The solution is also similar to semantic wiki sites: use the Semantic Web technologies to make better use of the vast amount of knowledge in Wikipedia. This

time, however, instead of adding semantic markup to each wiki page, an agent has been developed to extract existing structured information from each page. And furthermore:

- The extracted information will take the form of an RDF data graph, and this graph will be the corresponding machine-readable page of the original wiki page.
- Repeating this extraction process for each page in Wikipedia will build a huge collection of RDF data graphs, which forms a large RDF dataset.
- This RDF dataset can be viewed as Wikipedia's machine-readable version, with the original Wikipedia remains as the human-readable one.
- Since all the RDF graphs in this RDF dataset share the same set of ontologies, they therefore share precisely defined semantics, meaning that we can query against this dataset and find what we want much easier.

The above is the outline of an idea about how to transform Wikipedia to make it more useful. It was originally proposed by researchers and developers from University of Leipzig, Freie Universität Berlin and OpenLink Software. The initial release of this RDF dataset was in January 2007, and the dataset is called *DBpedia*. The exact same idea has since then grown into a project called *DBpedia project*, as described in the following URL:

<http://dbpedia.org/About>

And here is the official definition of DBpedia, taken directly from the above official Web site:

DBpedia is a community effort to extract structured information from Wikipedia and to make this information available on the Web. DBpedia allows you to ask sophisticated queries against Wikipedia, and to link other data sets on the Web to Wikipedia data.

The rest of this chapter will present DBpedia project in detail. Here is some quick summary of DBpedia dataset at the time of this writing:

- Its latest release is DBpedia 3.5.1 (28 April 2010).
- It describes more than 3.4 million things, including at least 312,000 persons, 413,000 places, 94,000 music albums, 49,000 films, 140,000 organizations, just to name a few.
- It has 5,543,000 links to external Web pages, 4,887,000 external links to other RDF datasets, and 565,000 Wikipedia categories.

I am sure at the time you are reading this chapter, the above numbers will change. It will be interesting to make a comparison to see how fast the dataset grows. Yet, the basic techniques behind it should remain the same, and that would be the topic for the rest of this chapter.

10.1.3 The Look and Feel of DBpedia: Page Redirect

Before we start to understand the automatic extraction of the semantic information, it will be helpful to get a basic feeling of DBpedia: how does it look like and how are we going to access it?

As far as an user is concerned, DBpedia is essentially a huge RDF dataset. And there are two ways to access it:

1. Use a Web browser to view different RDF graphs contained in DBpedia dataset or,
2. Use a SPARQL endpoint to query against DBpedia dataset with the goal of discovering information with much greater ease.

The second way of accessing DBpedia is perhaps the one that DBpedia project has intended for us to do. However, using a Web browser to access a specific RDF graph contained in the dataset feels like using DBpedia as if it were another version of the original Wikipedia, and it would be interesting to those curious minds. And by comparing the two pages – the original one from Wikipedia and the generated RDF graph from DBpedia – we can also learn a lot about DBpedia itself.

This section will concentrate on the first way of accessing DBpedia, and the second way will be covered in detail in later sections.

Let us use Swiss tennis player Roger Federer as an example. First, let us see how he is described in Wikipedia. Open Wikipedia from the following URL:

`http://www.wikipedia.org/`

And type *Roger Federer* in the search box, also make sure *English* is the selected language by using the language selection drop-down list. Once you click the continue button, you will be taken to the page as shown in Fig. 10.1.

And note that this page has the following URL:

`http://en.wikipedia.org/wiki/Roger_Federer`

Now to get to its DBpedia equivalent page, replace the following prefix in the above URL:

`http://en.wikipedia.org/wiki/`

with this one:

`http://dbpedia.org/resource/`

and you get the following URL:

`http://dbpedia.org/resource/Roger_Federer`

which is the corresponding machine-readable DBpedia page for Roger Federer.



Fig. 10.1 Roger Federer’s wiki page in Wikipedia

Now enter this URL into your Web browser. Instead of seeing this exciting new DBpedia equivalent page, your browser will redirect you to the following URL:

`http://dbpedia.org/page/Roger_Federer`

And there, you will see the corresponding RDF graph for Roger Federer displayed as an HTML page, as shown in Fig. 10.2. So what has happened?

We will cover the reason in the next chapter, but here is a quick answer to this question. The page

`http://dbpedia.org/resource/Roger_Federer`

in fact represents a generated RDF data file that is intended for machine to read, not for human eyes to enjoy. Therefore, it will not display as well as a traditional

About: Roger Federer

An Entity of Type : [person](#), from Named Graph : <http://dbpedia.org>, within Data Space : <dbpedia.org>

Roger Federer (born 8 August 1981) is a Swiss professional tennis player. As of 8 March 2010, he is ranked world number 1 by the Association of Tennis Professionals (ATP), having previously held the number one position for a record 237 consecutive weeks. Many sports analysts, tennis critics, former and current players consider Federer to be the greatest tennis player of all time. Federer has won 16 Grand Slam singles titles, more than any other male player.

Property	Value
dbpedia-owl:abstract	<ul style="list-style-type: none"> Roger Federer ist ein Schweizer Tennisspieler. In seiner bisherigen Karriere konnte er die Rekordanzahl von 16 Grand-Slam-Turnieren im Einzel gewinnen und beendete die Jahre 2004, 2005, 2006, 2007 und 2009 an der Spitze der Tennis-Weltrangliste. Insgesamt gewann Federer bisher 62 Titel im Einzel sowie acht im Doppel. Federer ist der einzige Spieler, der dreimal in seiner Karriere drei Grand-Slam-Titel in einer Saison gewinnen konnte. Dies gelang ihm 2004, 2006 und 2007. Er ist einer von sechs Spielern, die im Laufe ihrer Karriere bei allen vier Grand-Slam-Turnieren erfolgreich waren. Als zweitem Spieler (seit 1922) neben Bjorn Borg gelang es dem Schweizer, fünf Mal in Folge das Tennisturnier von Wimbledon für sich zu entscheiden. Zudem ist er der einzige Spieler der Open Era, der fünf Mal in Folge die US Open gewinnen konnte. Federer wurde in den Jahren 2005, 2006, 2007 und 2008 jeweils zum Weltsportler des Jahres gewählt. Bereits zu aktiven Zeiten wird Federer von nahezu allen Experten zu den besten Tennisspielern in der Geschichte dieses Sports gezählt und oftmals auch als bester Spieler aller Zeiten bezeichnet. Roger Federer es un tenista profesional suizo considerado por la mayoría de los grandes del tenis mundial y la prensa especializada, como el mejor jugador de todos los tiempos. Actualmente ocupa el número 1 del ranking de la ATP. Es el único, de los jugadores en actividad, que ha ganado los cuatro torneos del Grand Slam. Comenzó a practicar tenis a los tres años, a los ocho entró en el Tennisclub TC Old Boys, logrando, ya en su etapa junior, grandes resultados, al finalizar la temporada de 1998 en el primer lugar. Aunque en el circuito profesional no comenzó a destacar hasta la temporada de 2001, en apenas tres años, a partir del 2 de febrero de 2004, Federer se situó como número 1 del mundo en el ranking de la ATP, posición que mantuvo durante un tiempo récord de 237 semanas consecutivas hasta el 18 de agosto de 2008. Acumula 275 semanas como número 1 del mundo al lunes 22 de marzo de 2010, ocupando el segundo lugar en la historia en semanas al tope del circuito, sólo por detrás de Pete Sampras con 286. Es dueño de numerosos récords en el circuito, ganador de 16 coronas de Grand Slam, 4 ATP World Tour Finals y 16 Torneos ATP Masters 1000, además de su reconocida habilidad para jugar en todas las superficies, condición poco común que le ha permitido sumar victorias en todas y cada una de ellas. El 5 de julio de 2009 se convirtió en el jugador con más torneos del Grand Slam ganados, tras ganar su sexto Wimbledon y superar la plus marca de Pete Sampras (14 torneos). Debido a su exitosa trayectoria, Federer ha ganado el premio Laureus World Sportsman of the Year en cuatro años consecutivos (2005–2008) y fue nombrado tenista de la década por la ITF. Roger Federer (born 8 August 1981) is a Swiss professional tennis player. As of 8 March 2010, he is ranked world number 1 by the Association of Tennis Professionals (ATP), having previously held the number one position for a record 237 consecutive weeks. Many sports analysts, tennis critics, former and current players consider Federer to be the greatest tennis player of all time. Federer has won 16 Grand Slam singles titles, more than any other male player.

<http://dbpedia.org/ontology/abstract>

Fig. 10.2 Generated RDF graph for Roger Federer, displayed as an HTML page

Web page in an ordinary HTML browser. Yet in order to give back what has been requested by its user, DBpedia has implemented a HTTP mechanism called *content negotiation* (details in [Chap. 11](#)), so your browser will be re-directed to the following page:

http://dbpedia.org/page/Roger_Federer

which then presents the page as shown in [Fig. 10.2](#).

As you can see, this page is mainly a long summary of property–value pairs, which are presented in a table format. You may also find this page not as readable as its original Wikipedia version. However, it is also quite amazing that everything on this page is automatically extracted from the original text. We will come back to

this page again, but for now, what has been described here is the simplest way to access DBpedia.

And as a summary, any page in Wikipedia has the following URL:

```
http://en.wikipedia.org/wiki/Page_Name
```

And you can always replace the prefix and use the following URL to access its corresponding DBpedia equivalent page:

```
http://dbpedia.org/resource/Page_Name
```

10.2 Semantics in DBpedia

Before we get into the exciting world of using SPARQL to query against DBpedia, we need to understand how the meaningful information is extracted from the corresponding Wikipedia pages, and this is the goal of this section.

10.2.1 Infobox Template

The key idea of DBpedia is to automatically extract structured information from existing wiki pages without the need to make any change to them. As we all know, a wiki page is simply a page of text. So where is the structured information?

The answer lies in a special type of templates called *infoboxes*. Let us take a look at these templates first. Note that the goal of this section is to show you the fact that the information contained in infobox templates is the main source for structured information extraction, and not to discuss how to create and populate an infobox. If you are completely new to infobox, it might be helpful to check out its basic concept from Wikipedia's `help` page.

Templates in Wikipedia are originally introduced mainly for layout purposes, and infobox template is one particular type of these templates. When used on a page, it provides summary information about the subject that is being discussed on the given page, and the direct benefit to the user is to save time; if you don't have time to read the long wiki page, you can simply read this infobox to get its main point.

For the wiki site itself, infobox offers several benefits. First, since it is created from a template, similar subjects on different pages will all have a uniform look and a common format. Second, to change the display style and common texts in infoboxes, one does not have to go through each one of these infoboxes. Instead, one can simply modify the style and common texts from a well-controlled central place (the template page), and all the displayed infoboxes will be changed.

To see the code behind a given infobox, for example, the infobox on Roger Federer's page, you can simply click the `view source` tab on the top of the page (see Fig. 10.1).

List 10.1 shows part of the code for the infobox on Roger Federer's wiki page. Note that the actual image of the infobox is not included here, since we are mainly interested in code behind the infobox. Again, remember at the time of reading this chapter, this infobox may as well be changed, but the basic idea is still the same. Also, in order for us to read it easily, List 10.1 has been edited slightly.

List 10.1 Infobox on Roger Federer's wiki page

```

{{Infobox Tennis player
|playername = Roger Federer
|image = [[File:Roger Federer (26 June 2009, Wimbledon) 2
new.jpg|200px|]]
|caption = Wimbledon 2009
|country = [[Switzerland]]
|nickname= ' 'Swiss Maestro' ' <ref>...</ref><br />
          ' 'Federer Express' ' </br>
          ' 'Fed Express' ' </br>
          ' 'FedEx' ' <ref>...</ref><br/>
|residence = [[Wollerau]], [[Switzerland]]
|datebirth = {{birth date and age|df=yes|1981|08|08}}
|placebirth = [[Basel]], [[Switzerland]]
|height = {{height|m=1.86}}
|weight = {{convert|85.0|kg|lb st|abbr=on}}<ref>...</ref>
|turnedpro = 1998<ref>...</ref>
|plays = Right-handed; one-handed backhand
|careerprizemoney = [[US]] 53,362,068<br />* [[ATP Tour
          records#Earnings|All-time leader in earnings]]
|singlesrecord = 678-161 (80.8%)<ref>...</ref>
|singlestitles = 61
|highestsinglesranking = No. ' ' '1' ' ' (February 2, 2004)
|currentsinglesranking = No. ' ' '1' ' ' (July 6, 2009)
|AustralianOpenresult = ' ' 'W' ' ' (
    [[2004 Australian Open - Men's Singles|2004]],
    [[2006 Australian Open - Men's Singles|2006]],
    [[2007 Australian Open - Men's Singles|2007]])
|FrenchOpenresult = ' ' 'W' ' ' (
    [[2009 French Open - Men's Singles|2009]])
|Wimbledonresult = ' ' 'W' ' ' (
    [[2003 Wimbledon Championships - Men's Singles|2003]],
    [[2004 Wimbledon Championships - Men's Singles|2004]],
    [[2005 Wimbledon Championships - Men's Singles|2005]],
    [[2006 Wimbledon Championships - Men's Singles|2006]],
    [[2007 Wimbledon Championships - Men's Singles|2007]],
    [[2009 Wimbledon Championships - Men's Singles|2009]])
|USOpenresult = ' ' 'W' ' ' (

```



```

[[2004 U.S. Open - Men's Singles|2004]],
[[2005 U.S. Open - Men's Singles|2005]],
[[2006 U.S. Open - Men's Singles|2006]],
[[2007 U.S. Open - Men's Singles|2007]],
[[2008 U.S. Open - Men's Singles|2008]])
|Othertournaments = Yes
|MastersCupresult = ' ' 'W' ' ' (
    [[2003 Tennis Masters Cup#Singles|2003]],
    [[2004 Tennis Masters Cup#Singles|2004]],
    [[2006 Tennis Masters Cup#Singles|2006]],
    [[2007 Tennis Masters Cup#Singles|2007]])
|Olympicsresult = ' ' 'SF' ' ' (
    {{OlympicEvent|Tennis|2000 Summer|title=2000|
        subcategory=Men's Singles}})
|doublesrecord = 112-72 (60.8%)
|doublestitles = 8
|OthertournamentsDoubles = yes
|grandslamsdoublesresults= yes
|AustralianOpenDoublesresult = 3R (2003)
|FrenchOpenDoublesresult = 1R (2000)
|WimbledonDoublesresult = QF (2000)
|USOpenDoublesresult = 3R (2002)
|OlympicsDoublesresult = [[Image:Gold medal.svg|20px]]
    ' ' 'Gold Medal' ' ' ({{OlympicEvent|Tennis|2008 Summer|
        title=2008|subcategory=Men's Doubles}})
|highestdoublesranking = No. 24 (9 June 2003)
|updated = 24 November 2009}}

```

Now, besides admiring Roger Federer's amazing career achievements, we should not miss the most important thing we see from this infobox: an infobox is simply a collection of property–value pairs.

The infobox in List 10.1 is created by the page authors who have used an infobox template designed for athletes, which belongs to `People` category. There are infobox templates for a large number of other categories as well. For example, `Place`, `Music`, `Movie`, `Education`, `Organization`, just to name a few. All these infoboxes, although from different category templates, will share the same style; each one of them is a collection of property–value pairs.

This property–value pair style should suggest the creation of RDF statements, with each statement mapped to one such pair in the infobox. More specifically, for a given pair, the property name maps to the predicate of a statement, and the property value maps to the object of that statement.

What about the subject of these RDF statements? Note that the whole collection of property–value pairs in a given infobox is used to describe the subject of that given page. Therefore, all the RDF statements should share the exact same subject,

and the subject of the current wiki page naturally becomes the resource described by the generated RDF statements.

And this is the basic idea behind DBpedia's automatic information extraction.

10.2.2 Creating DBpedia Ontology

10.2.2.1 The Need for Ontology

As we have learned, RDF statements use classes and properties to describe resources, and these classes and properties are defined in some given ontologies. It should be clear to us that when using the same set of ontologies, distributed RDF graphs are able to share the same precisely defined semantics, and when linked together, they can provide new non-trivial facts that are valuable to us. Therefore, whenever we discuss a collection of RDF documents, the first question we should ask is, what are the ontologies used by these RDF documents?

Note that, however, it is perfectly legal to create RDF statements without using any ontology at all. The result is that the resources described by these statements will not have any precisely defined meanings. In addition, they will not be able to participate in any reasoning process or take advantage of any benefit from aggregation with other resources. These RDF statements will simply remain isolated with fairly limited value.

In fact, generating RDF statements without using ontology was a major drawback in the early versions of DBpedia's extractor. To have a better idea about this, let us take a look at one such example.

Again, go back to the page for Roger Federer. List 10.2 shows part of the infobox on his page in Wikipedia.

List 10.2 Name, birthday, and birthplace information from Roger Federer's infobox

```
|playername = Roger Federer
|datebirth = {{birth date and age|df=yes|1981|08|08}}
|placebirth = [[Basel]], [[Switzerland]]
```

One earlier version of DBpedia's extractor, when parsing this infobox, would simply turn the attribute names contained in the infobox into the predicates of the generated RDF statements. For example, for Federer's name attribute, the predicate would have the following name:

```
dbprop:playername
```

where `dbprop` is the abbreviation of `http://dbpedia.org/property/`. and for his birth date and birth place attributes, the corresponding RDF predicates would look like.

```
dbprop:datebirth
dbprop:placebirth
```

Other property–value pairs would be processed similarly.

Now, take a look at the infobox of another person, Tim Berners-Lee. Part of his infobox is shown in List 10.3.

List 10.3 Name, birthday and birthplace information from Berners-Lee’s infobox

```
| name = Tim Berners-Lee
| birth_date = {{birth date and age|1955|6|8|df=y}}
| birth_place = [[London]], [[UK]]
```

Similarly, the extractor would use the following property names as the predicates of the generated RDF statements:

```
dbprop:name
dbprop:birth_date
dbprop:birth_place
```

Since there is no formal ontology shared by these statements, there will be no way for the machine to know the following fact:

- Both Roger Federer and Tim Berners-Lee are resources whose class type is Person.
- `dbprop:name` is the same as `dbprop:playername`.
- `dbprop:birth_date` is the same as `dbprop:datebirth`.
- `dbprop:birth_place` is the same as `dbprop:placebirth`.

And without knowing the above, as far as any application is concerned, the generated statements are just a collection of alphabetic strings.

The conclusion is that whenever RDF statements are generated, ontologies should be used. And more specifically to the case of DBpedia, we need formal definitions of classes and properties. The attributes in infobox templates will be mapped to these classes and properties when RDF statements about the page subject are generated.

Fortunately, the above idea has been implemented by the new extractor starting from DBpedia Release 3.2. More specifically,

- it first creates an instance of some class type defined in the ontology to represent the subject of the current page;
- it then maps each attribute extracted from the infobox to a property that is defined in the ontology and can also be used to describe the given subject instance, and
- the extracted property value will become the object of the created RDF statement.

We are going to study a real example of the generated RDF graph, but at this point, let us take a look at the ontology that is being used.

The ontology used by the new extractor is simply called *DBpedia ontology*; it is based on OWL and it forms the structural backbone of DBpedia. Its features can be summarized as follows (Jentzsch 2009):

- It is a shallow, cross-domain ontology.
- It is manually created, based on the most commonly used infobox templates within Wikipedia. More specifically,
- from 685 most frequently used templates, 205 ontology classes are defined, and
- from 2,800 template properties, 1,200 ontology properties are created.

To access this ontology, you can visit DBpedia's official Web site and find the link to this ontology. At the time of this writing, this link is given as follows:

<http://wiki.dbpedia.org/Ontology>

The rest of this section will focus on how this ontology is developed.

10.2.2.2 Mapping Infobox Templates to Classes

First off, each Wikipedia's infobox template is carefully and manually mapped to a class defined in DBpedia ontology.

For example, one such infobox template is the `Tennis player` template, which has been manually mapped to the following class:

<http://dbpedia.org/ontology/TennisPlayer>

Currently, Wikipedia has about 685 infobox templates, and these templates are mapped to about 205 classes defined in DBpedia ontology. Table 10.1 shows more example mappings.

The following reverse-engineering steps are used to come up with the information contained in Table 10.1, and they are listed here so that you can explore the mapping on your own if you need to. Note that we have used the city of Berlin as our example to describe these steps.

Table 10.1 Mapping Wikipedia's infobox templates to classes defined in DBpedia ontology

Wikipedia infobox template	DBpedia class mapping	Example page
Tennis player	<code>dbclass:TennisPlayer</code>	Roger Federer
Officeholder	<code>dbclass:officeHolder</code>	Bill Clinton
Person	<code>dbclass:Person</code>	Tim Berners-Lee
German Bundesland	<code>dbclass:City</code>	Berlin
Film	<code>dbclass:Film</code>	Forrest Gump
Company	<code>dbclass:Company</code>	Ford Motor Company
University	<code>dbclass:University</code>	Tsinghua University

prefix `dbclass:` <<http://dbpedia.org/ontology/>>

Step 1. Go to the page about city of Berlin in Wikipedia.

This page is located in the following URL:

```
http://en.wikipedia.org/wiki/Berlin
```

Step 2. After you land on the page about Berlin, click `edit this page` link to see the infobox code. In our example, you can see the infobox has a template called `German Bundesland`.

Step 3. Open up Berlin's corresponding page in DBpedia.

The following link will be able to take you to the DBpedia Berlin page:

```
http://dbpedia.org/resource/Berlin
```

Step 4. When you reach the DBpedia Berlin page, click the RDF icon on the upper right hand, and this will take you to the RDF file generated by the extractor (more on this later).

In our example, this takes you to the following page:

```
http://dbpedia.org/data/Berlin.rdf
```

Step 5. Confirm the above file does exist at the above URL. In other words, you should be able to open the above URL without any trouble. Now open up a SPARQL endpoint, and conduct the query as shown in List 10.4.

List 10.4 SPARQL query to check the class type for city Berlin

```
select distinct ?value
from <http://dbpedia.org/data/Berlin.rdf>
where
{
  <http://dbpedia.org/resource/Berlin> rdf:type ?value .
}
order by asc(?value)
```

This will show you all the class types that Berlin as a resource belongs to. And you can see the most specific class type is the following:

```
http://dbpedia.org/ontology/City
```

and this is how we know the fact that infobox template `German Bundesland` has been mapped to class <http://dbpedia.org/ontology/City> as shown in Table 10.1. You can repeat the above steps for other mappings shown in Table 10.1. For a given infobox template that is not included in Table 10.1, you can find its corresponding class type by following the above steps as well.

To get more understanding of this ontology, let us go back to Roger Federer's wiki page again. His wiki page has an infobox template called `Tennis player`, and as discussed earlier, this has been manually mapped to `TennisPlayer` class defined in DBpedia ontology. List 10.5 shows the definition.

List 10.5 Definition of `TennisPlayer` class in DBpedia ontology

```
<owl:Class rdf:about="http://dbpedia.org/ontology/TennisPlayer">
  <rdfs:label xml:lang="en">Tennis Player</rdfs:label>
  <rdfs:subClassOf
    rdf:resource="http://dbpedia.org/ontology/Athlete" />
</owl:Class>
```

As shown in List 10.5, class `TennisPlayer` is a sub-class of `Athlete`, whose definition is shown in List 10.6.

List 10.6 Definition of `Athlete` class in DBpedia ontology

```
<owl:Class rdf:about="http://dbpedia.org/ontology/Athlete">
  <rdfs:label xml:lang="en">Athlete</rdfs:label>
  <rdfs:subClassOf
    rdf:resource="http://dbpedia.org/ontology/Person" />
</owl:Class>
```

And similarly, class `Person` is defined in List 10.7.

List 10.7 Definition of `Person` class in DBpedia ontology

```
<owl:Class rdf:about="http://dbpedia.org/ontology/Person">
  <rdfs:label xml:lang="en">Person</rdfs:label>
  <rdfs:subClassOf
    rdf:resource="http://www.w3.org/2002/07/owl#Thing" />
</owl:Class>
```

Therefore, `Person` is the top-level class. By following the same route, you can get a good understanding about every class defined in DBpedia ontology.

10.2.2.3 Mapping Infobox Template Attributes to Properties

DBpedia ontology also includes a set of properties, which are created by another important mapping that is also manually implemented. More specifically, for a given Wikipedia infobox template, the attributes used in the template are carefully mapped to a set of properties defined in DBpedia ontology, and these properties all have the template's corresponding ontology class as their `rdfs:domain`. For now, about 2,800 template properties have been mapped to about 1,200 ontology properties.

For example, to see the properties that can be used on `TennisPlayer` class, we can start from its base class, namely `Person`. Since `TennisPlayer` is a sub-class

of `Person`, all the properties that can be used on a `Person` instance can also be used to describe any instance of `TennisPlayer`.

The SPARQL query in List 10.8 can be used to find all these properties.

List 10.8 SPARQL query to find all the properties defined for `Person` class

```
prefix dbpediaOnt: <http://dbpedia.org/ontology/>
select distinct ?propertyName
from <http://downloads.dbpedia.org/3.4/dbpedia_3.4.owl>
where
{
  ?propertyName rdfs:domain dbpediaOnt:Person .
}
```

Note the following link in List 10.8:

http://downloads.dbpedia.org/3.4/dbpedia_3.4.owl

which is the URL location for DBpedia ontology. And List 10.9 shows part of the results.

List 10.9 Some of the properties defined for `Person` class

```
<http://dbpedia.org/ontology/Person/otherName>
<http://dbpedia.org/ontology/Person/birthName>
<http://dbpedia.org/ontology/Person/birthDate>
<http://dbpedia.org/ontology/Person/birthPlace>
<http://dbpedia.org/ontology/title>
<http://dbpedia.org/ontology/party>
<http://dbpedia.org/ontology/child>
<http://dbpedia.org/ontology/spouse>
<http://dbpedia.org/ontology/partner>
<http://dbpedia.org/ontology/father>
<http://dbpedia.org/ontology/mother>
```

Similarly, repeat the query shown in List 10.8, but change the class name to `Athlete`, so the query will tell us all the properties defined for class `Athlete`. List 10.10 shows some of the properties that can be used to describe an `Athlete` instance.

List 10.10 Some of the properties defined for `Athlete` class

```
<http://dbpedia.org/ontology/currentNumber>
<http://dbpedia.org/ontology/currentPosition>
<http://dbpedia.org/ontology/currentTeam>
<http://dbpedia.org/ontology/formerTeam>
```

And List 10.11 shows the ones for `TennisPlayer`. Again, you can obtain these two lists by modifying the query given in List 10.8.

List 10.11 Some of the properties defined for `TennisPlayer` class

```
<http://dbpedia.org/ontology/careerprizemoney>
<http://dbpedia.org/ontology/plays>
```

Take a quick look at the properties contained in Lists 10.9, 10.10, and 10.11. We will see some of them in use when the extractor tries to describe Roger Federer as a resource of type `TennisPlayer` in the next section.

Now going back to Tim Berners-Lee’s example. Without even studying the infobox template on his wiki page, we are sure he will be another instance of class `Person` or its sub-class. Therefore, he will share lots of properties with the instance that identifies Roger Federer. The situation where same properties have different names, as we have described earlier, will not happen again.

Finally, you can always follow what we have done here to understand DBpedia ontology, which will be very helpful when you need to conduct SPARQL queries, as we will show in the sections to come.

10.2.3 Infobox Extraction Methods

Now we have reached the point where we are ready to see how the extractor works. In general, the extractor visits a wiki page, parses the infobox template on the page, and generates an RDF document that describes the content of the given page.

In real practice, it is much more complex than this. More specifically, Wikipedia’s infobox template system has evolved over time without a centralized coordination, and the following two situations happen quite often:

- Different communities use different templates to describe the same type of things. For example, to describe Roger Federer, a `Tennis player` template is used. Yet to describe Pete Sampras (another famous tennis player), a `Tennis biography` template is used. We will see more about this in later sections.
- Different templates use different names for the same attribute. For example, to describe Roger Federer, `datebirth` and `placebirth` are used, and when it comes to describe Tim Berners-Lee, `birth_date` and `birth_place` are used.

Other things similar to the above can also happen. In general, one of the main reasons for the above is that it is difficult to guarantee that all the Wikipedia editors will strictly follow the recommendations given on the page that describes a template.

As a result, DBpedia project team has decided to use two different extraction approaches in parallel: a generic approach and a mapping-based approach, which will be discussed in the next two sections.

10.2.3.1 Generic Infobox Extraction Method

The generic infobox extraction method is quite straightforward and can be described as follows:

- For a given Wikipedia page, a corresponding DBpedia URI is created, which has the following form:

`http://dbpedia.org/resource/Page_Name`

- The above DBpedia URI will be used as the URI identifier for the subject.
- The predicate URI is created by concatenating the following namespace fragment and the name of the infobox attribute:

`http://dbpedia.org/property/`

- Object is created from the attribute value and will be post-processed in order to generate a suitable URI identifier or a simple literal value.
- Repeat this for each attribute–value pair until all are processed.

The above process will be repeated for each page in Wikipedia. The advantage is that this approach can completely cover all infoboxes and their attributes. The disadvantage is the fact that synonymous attribute names are not resolved. Therefore, some ambiguity always exists, and SPARQL query will be difficult to construct. In addition, there is no formal ontology involved, meaning that no application can make any inferencing based on the generated RDF statements.

10.2.3.2 Mapping-Based Infobox Extraction Method

The main difference between the mapping-based extraction approach and the generic extraction approach that we have discussed above is that the mapping-based approach makes full use of the DBpedia ontology. More specifically, two types of mapping are included:

- *Template-to-class mapping*: infobox template types are mapped to classes. As we have discussed, at this point, 685 templates are mapped to 205 ontology classes.
- *Attribute-to-property mapping*: properties from within the templates are mapped to ontology properties. At this point, 2,800 template properties are mapped to 1,200 ontology properties.

And to implement these mappings, fine-tuned rules are created to help parsing infobox attribute names and values. The following is a rundown of the steps the extractor uses to generate the RDF graph for a given page:

- For a given Wikipedia page, its on-page infobox template type is retrieved by the extractor.
- Based on the *template-to-class* mapping rule, the extractor is able to find its corresponding class type defined in DBpedia ontology.

- The extractor then creates an RDF resource as an instance of this class, and this resource will also be the subject of all the future RDF statements for this page. In addition, it has the following URI as its identifier:

`http://dbpedia.org/resource/Page_Name`

- The extractor now parses the attribute–value pairs found in the infobox on the current page. More specifically, the *attribute-to-property* mapping rules have to be applied to map each attribute to the appropriate property defined in the ontology. Each one of such mapping will create a predicate URI.
- For each attribute value, a corresponding object will be created to represent the value and will be post-processed in order to generate either a suitable URI identifier or a simple literal value.
- Repeat this for each attribute–value pair until all are processed.

The above are the basic steps used to generate an RDF document for a given page. Today, this extraction approach is the one that is in use, and it has been repeated for a large portion of Wikipedia. The generated machine-readable dataset is called DBpedia, as you have known.

Note that to save space, we are not going to include examples here and list the generated RDF statements. If you would like to, you can go to the page and get the generated RDF file easily. For example, the generated DBpedia page for Roger Federer can be accessed from the following URL:

`http://dbpedia.org/page/Roger_Federer`

and at the bottom of the page, you will find different sterilization formats of the generated RDF document.

The advantage of this mapping-based approach is obvious; the RDF documents are based on ontologies, therefore, SPARQL queries are much easier to construct, any given application can now conduct inference on the datasets.

The main disadvantage is that this method cannot provide a complete coverage on the Wikipedia pages. For example, it can only cover the infobox templates that have been mapped to the ontology. However, with more and more mapping being built, more coverage can be easily implemented.

Finally, note that the quality of DBpedia heavily depends on this extractor, which further depends on the mapping rules and the DBpedia ontology itself. At the time of your reading, this algorithm will almost certain be improved, but the idea as described above will likely remain the same.

10.3 Accessing DBpedia Dataset

DBpedia is a huge collection of RDF graphs, with precisely defined semantics. And of course, there are different paths one can use when it comes to interaction with DBpedia. However, it is difficult for first-time users to use portions of this

infrastructure without any guidance. Actually, given the fact that data gathering and exposure via RDF standards is making constant progress, the related user interfaces, documentation, data presentation, and general tutorial for users are still surprisingly limited.

In this section, we will discuss three different methods you can use to interact with DBpedia, and they will serve as a starting point for you. With this starting point, you will find your journey with DBpedia much easier and enjoyable.

10.3.1 Using SPARQL to Query DBpedia

10.3.1.1 SPARQL Endpoints for DBpedia

As you might have guessed, you can use SPARQL to query DBpedia. In fact, DBpedia provides a public SPARQL endpoint you can use:

```
http://dbpedia.org/sparql
```

In practice, this endpoint is generally used directly by remote agents. We therefore will access the endpoint via a SPARQL viewer, and you can find it at

```
http://dbpedia.org/snorql/
```

Figure 10.3 shows the opening page of this SPARQL explorer.

As you can see, to make your query construction easier, a set of namespace shortcuts have been provided to you, as shown in List 10.12.

List 10.12 Pre-defined namespaces used in SPARQL explorer for DBpedia

```
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX : <http://dbpedia.org/resource/>
PREFIX dbpedia2: <http://dbpedia.org/property/>
PREFIX dbpedia: <http://dbpedia.org/>
PREFIX skos: <http://www.w3.org/2004/02/skos/core#>
```

In fact, we would like to replace the shortcut for `dbpedia` with the following one:

```
PREFIX dbpedia: <http://dbpedia.org/ontology/>
```

which is in fact more useful than the one given in List 10.12. For the rest of this section, all our examples have the same prefix definitions as above, and we therefore will not include them in our queries.

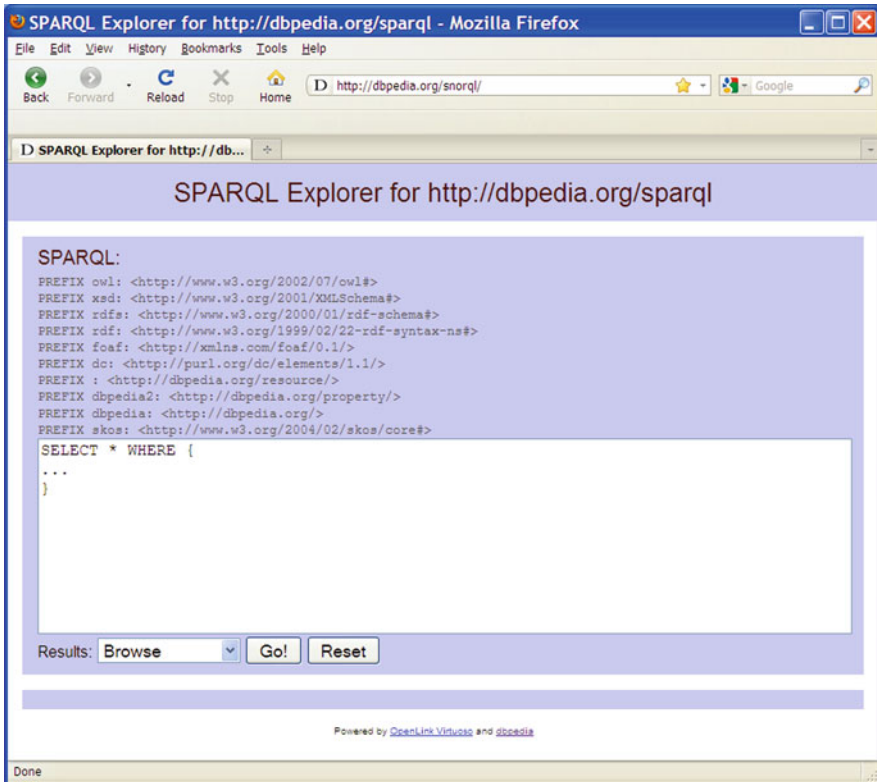


Fig. 10.3 DBpedia's SPARQL endpoint

10.3.1.2 Examples of Using SPARQL to Access DBpedia

Obviously, to start using SPARQL to query DBpedia dataset, you have to know the resource name of the subject you are interested in, its class type, and some properties that can be used to describe the resource.

To get to know the subject's corresponding resource name, follow this simple rule: if the subject has the following page in Wikipedia:

<http://en.wikipedia.org/wiki/Name>

its resource name will then be given by the following URI:

<http://dbpedia.org/resource/Name>

To know which class type this resource belongs to, the easiest way is to use the query as shown in List 10.13, again using Roger Federer as our favorite example:

List 10.13 SPARQL query to understand the class type of the resource identifying Federer

```
SELECT * WHERE {
    :Roger_Federer a ?class_type.
}
```

In order not to make this chapter take too much space, the query result is normally not included unless it is necessary to do so. Also, including the result may not be that useful because of the dynamic nature of DBpedia dataset.

Now, the query shown in List 10.13 does tell us the class type of the resource that represents Roger Federer. For our immediate purpose, we will remember the most specific one: `dbpedia:TennisPlayer`.

List 10.14 is another simple query we can use to find what properties the extractor has used to describe Roger Federer.

List 10.14 SPARQL query to find out all the properties used to describe Federer

```
SELECT * WHERE {
    :Roger_Federer ?propertyName ?propertyValue.
}
```

With these two queries (in fact, you can use the query in List 10.14 alone), you can gain some basic knowledge about your subject of interest, and you can start more interesting queries from here.

For example, List 10.15 shows all the tennis players who are from the same country as Federer is.

List 10.15 Find all the tennis players who are from the same country as Federer

```
SELECT ?someone ?birthPlace
WHERE {
    :Roger_Federer dbpedia:birthPlace ?birthPlace.
    ?someone a dbpedia:TennisPlayer.
    ?someone dbpedia:birthPlace ?birthPlace.
}
```

At the time of my writing, I got nine players back, including Federer himself. Compare this to what you get when you are reading this chapter, and it is interesting to see the growth of the DBpedia.

The query in List 10.16 tries to find those tennis players who have also won all the grand slams that Federer has won.

List 10.16 Find all those players who also won all the grand slams that Federer has won

```
SELECT * WHERE {
    :Roger_Federer dbpedia2:australianopenresult ?aussie_result.
```

```

:Roger_Federer dbpedia2:usopenresult ?us_result.
:Roger_Federer dbpedia2:wimbledonresult ?wimbeldon_result.
:Roger_Federer dbpedia2:frenchopenresult ?frenchopen_result.
?player a dbpedia:TennisPlayer.
?player dbpedia2:australianopenresult ?aussie_result.
?player dbpedia2:usopenresult ?us_result.
?player dbpedia2:wimbledonresult ?wimbeldon_result.
?player dbpedia2:frenchopenresult ?frenchopen_result.
}

```

As we know, so far, Federer has won all four grand slams in his career, and List 10.16 is looking for other players who have also won all the four titles. Table 10.2 shows the query result.

Clearly, this query does not return Pete Sampras as one of the players who have won all the grand slams titles that Federer has won. If you have followed tennis world even vaguely, you might know the reason: Sampras has never won French Open in his career, that is why he is not included in the query result.

Now, we can change the query by eliminating the French Open result, as shown in List 10.17.

List 10.17 Change the query in List 10.16 to exclude French Open result

```

SELECT * WHERE {
:Roger_Federer dbpedia2:australianopenresult ?aussie_result.
:Roger_Federer dbpedia2:usopenresult ?us_result.
:Roger_Federer dbpedia2:wimbledonresult ?wimbeldon_result.
?player a dbpedia:TennisPlayer.
?player dbpedia2:australianopenresult ?aussie_result.
?player dbpedia2:usopenresult ?us_result.
?player dbpedia2:wimbledonresult ?wimbeldon_result.
}

```

Table 10.2 Players who have also won all the grand slams that Federer has won

aussie_result	us_result	wimbeldon_result	frenchopen_result	player name
"W"@en	"W"@en	"W"@en	"W"@en	:Chris_Evert
"W"@en	"W"@en	"W"@en	"W"@en	:Steffi_Graf
"W"@en	"W"@en	"W"@en	"W"@en	:Rod_Laver
"W"@en	"W"@en	"W"@en	"W"@en	:Andre_Agassi
"W"@en	"W"@en	"W"@en	"W"@en	:Roger_Federer
"W"@en	"W"@en	"W"@en	"W"@en	:Billie_Jean_King
"W"@en	"W"@en	"W"@en	"W"@en	:Roy_Emerson
"W"@en	"W"@en	"W"@en	"W"@en	:Martina_Navratilova

And we do get more players back this time, but we still cannot find Pete Sampras in the result set. However, we know he did win all the other three titles. So what is wrong?

The reason is the resource that represents Pete Sampras

```
http://dbpedia.org/resource/Pete\_Sampras
```

has not been created as an instance of class `dbpedia:TennisPlayer`, since the infobox template used on his wiki page is not the tennis player template. Clearly, this is another example showing the importance of ontology.

You can further explore the DBpedia dataset by using SPARQL queries, and you can try some other areas that you like. For example, movies or music. In addition, try to think about how you can accomplish the same thing in Wikipedia. For example, try to accomplish what List 10.16 has accomplished in Wikipedia. Obviously, you have to sift through lots of pages for tennis players. It is very likely you will stop and decide to find something better to do.

10.3.2 Direct Download of DBpedia Datasets

Another way to access DBpedia is to directly download its RDF dumps. One reason of doing this is that you can then build your application on the datasets that you have downloaded. Since the datasets are on your local machine, your application will run faster, therefore easier to test.

To access this download page, visit the following URL:

```
http://wiki.dbpedia.org/Downloads351
```

Since currently the most recent release of DBpedia is 3.5.1, the download site has a 351 suffix. At the time of your reading, this will be changed for sure, so the link to the download page will also be changed. However, you can always find the latest download link on DBpedia's home page.

10.3.2.1 The Wikipedia Datasets

The first dataset you can download contains the original Wikipedia files, i.e., a copy of all Wikipedia wikis in the form of wikitext source and a copy of all pages from all Wikipedia wikis in HTML format. Obviously, these are the input materials for the DBpedia extractor, and they are offered here as a foundation for your own research, or for building your own applications.

10.3.2.2 DBpedia Core Datasets

The second part of the downloadable files are the so-called core datasets. They are machine-readable datasets generated by DBpedia project itself. To generate these datasets, a complete DBpedia dataset is first created by the DBpedia extractor; it is then sliced into several parts based on triple predicate. The resulting parts are the datasets you see, and each dataset is offered in the form of N-triples. We will not be

able to cover all the datasets here, but the discussion should be detailed enough for you to continue your own explore of these datasets.

The first dataset is the DBpedia ontology dataset. It is offered here so that you can download and make use of this ontology in your own applications. The main advantage of this ontology is that it is created from a central knowledge resource, and it is not domain specific. It is written in OWL and should be easily understood. List 10.18 shows part of this ontology.

List 10.18 A portion of the DBpedia ontology

```
<?xml version="1.0" encoding="UTF-8"?>
<rdf:RDF
  xmlns = "http://dbpedia.org/ontology/"
  xml:base="http://dbpedia.org/ontology/"
  xmlns:owl="http://www.w3.org/2002/07/owl#"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"

  <owl:Ontology rdf:about="">
    <owl:versionInfo xml:lang="de">
      Version 3.4 2009-10-05
    </owl:versionInfo>
  </owl:Ontology>

  <owl:Class
    rdf:about="http://dbpedia.org/ontology/PopulatedPlace">
    <rdfs:label xml:lang="en">Populated Place</rdfs:label>
    <rdfs:subClassOf
      rdf:resource="http://dbpedia.org/ontology/Place"/>
  </owl:Class>

  <owl:Class rdf:about="http://dbpedia.org/ontology/Place">
    <rdfs:label xml:lang="en">Place</rdfs:label>
    <rdfs:subClassOf
      rdf:resource="http://www.w3.org/2002/07/owl#Thing"/>
  </owl:Class>

  <owl:Class rdf:about="http://dbpedia.org/ontology/Country">
    <rdfs:label xml:lang="en">Country</rdfs:label>
    <rdfs:subClassOf
      rdf:resource="http://dbpedia.org/ontology/PopulatedPlace"/>
  </owl:Class>

  <owl:Class rdf:about="http://dbpedia.org/ontology/Area">
    <rdfs:label xml:lang="en">Area</rdfs:label>
```



```

<rdfs:subClassOf
  rdf:resource="http://dbpedia.org/ontology/PopulatedPlace"/>
</owl:Class>

```

The second core dataset is the Ontology Type dataset. This dataset includes all the resources covered by DBpedia and their related types. For example, List 10.19 shows two triples you will find in this file.

List 10.19 Example triples included in Ontology Types dataset

```

<http://dbpedia.org/resource/Roger_Federer>
<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
<http://dbpedia.org/ontology/TennisPlayer>.

```

```

<http://dbpedia.org/resource/Tim_Berners-Lee>
<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
<http://dbpedia.org/ontology/Person>.

```

With this dataset and the DBpedia ontology, different levels of reasoning can start to take place. For example, based on the following statements:

```

<http://dbpedia.org/resource/Roger_Federer>
<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
<http://dbpedia.org/ontology/TennisPlayer>.

```

```

<http://dbpedia.org/ontology/TennisPlayer>
<http://www.w3.org/2000/01/rdf-schema#subClassOf>
<http://dbpedia.org/ontology/Athlete>.

```

```

<http://dbpedia.org/ontology/Athlete>
<http://www.w3.org/2000/01/rdf-schema#subClassOf>
<http://dbpedia.org/ontology/Person>.

```

An application now understands that Roger Federer is not only a `TennisPlayer` but also an `Athlete` and a `Person`. Note that the above first statement comes from the Ontology Types dataset, and the other two statements are from the DBpedia Ontology dataset.

The next dataset is the Ontology Infobox Properties dataset, where all the properties and property values of all resources are collected. List 10.20 shows some example content you will find in this dataset.

List 10.20 Example content in Ontology Infoboxes dataset

```

<http://dbpedia.org/resource/Roger_Federer>
<http://xmlns.com/foaf/0.1/homepage>
<http://www.rogerfederer.com/>.

```

```
<http://dbpedia.org/resource/Roger_Federer>
<http://dbpedia.org/ontology/country>
<http://dbpedia.org/resource/Switzerland>.
```

```
<http://dbpedia.org/resource/Roger_Federer>
<http://dbpedia.org/ontology/birthDate>
"1981-08-08"^^xsd:date.
```

```
<http://dbpedia.org/resource/Roger_Federer>
<http://dbpedia.org/ontology/plays>
"Right-handed; one-handed backhand".
```

If you are developing your own application, this dataset will become a main source from where you will expect to get most of the machine-readable data. In fact, this dataset is further sliced into a collection of more detailed datasets based on triple predicate, and the following is a brief rundown of these generated datasets. Again, we only discuss a few, and you can understand the rest accordingly.

Titles dataset is about `rdfs:label` property, therefore contains triples as follows:

```
<http://dbpedia.org/resource/Roger_Federer>
<http://www.w3.org/2000/01/rdf-schema#label>
"Roger Federer"@en.
```

```
<http://dbpedia.org/resource/Tim_Berners-Lee>
<http://www.w3.org/2000/01/rdf-schema#label>
"Tim Berners-Lee"@en.
```

Home pages dataset is about `foaf:homepage` property, therefore contains triples as follows:

```
<http://dbpedia.org/resource/Roger_Federer>
<http://xmlns.com/foaf/0.1/homepage>
<http://www.rogerfederer.com/>.
```

```
<http://dbpedia.org/resource/Tim_Berners-Lee>
<http://xmlns.com/foaf/0.1/homepage>
<http://www.w3.org/People/Berners-Lee/>.
```

Finally, Persondata dataset is all about personal information, therefore contains predicates such as `foaf:name`, `foaf:givenname` and `foaf:surname`. The following are some triple examples from this dataset:

```
<http://dbpedia.org/resource/Roger_Federer>
<http://xmlns.com/foaf/0.1/name>
"Roger Federer".
```

```
<http://dbpedia.org/resource/Roger_Federer>
<http://xmlns.com/foaf/0.1/givenname>
"Roger"@de.
```

```
<http://dbpedia.org/resource/Roger_Federer>
<http://xmlns.com/foaf/0.1/surname>
"Federer"@de.
```

10.3.2.3 Extended Datasets

Besides the core datasets we have discussed above, DBpedia has also created a collection of extended datasets, which provide links to those datasets that are outside of DBpedia. We will have more understanding about the reasons behind these extended datasets when we finish the next chapter. For now, let us briefly examine these extended datasets to get some basic understanding. Again, we will not cover all of them, and what you will learn here will be enough for you to continue on your own.

The first one we would like to cover is the Links to Wikicompany dataset. Clearly, there will be quite a lot of statements in DBpedia datasets created about companies. And as we have learned from the last chapter, Wikicompany is a semantic wiki site about companies. Therefore, links between these two seems to be helpful. For example, List 10.21 shows some statements taken from this dataset.

List 10.21 Example statements taken from Links to Wikicompany dataset

```
<http://www4.wiwiss.fu-berlin.de/wikicompany/resource/ABC_News_Now>
<http://www.w3.org/2002/07/owl#sameAs>
<http://dbpedia.org/resource/ABC_News_Now>.
```

```
<http://www4.wiwiss.fu-berlin.de/wikicompany/resource/ACCBank>
<http://www.w3.org/2002/07/owl#sameAs>
<http://dbpedia.org/resource/ACCBank>.
```

As shown in List 10.21, *ABC News Now*, the popular 24-h news network, is identified in DBpedia by the following URI:

```
http://dbpedia.org/resource/ABC_News_Now
```

and in Wikicompany, the same resource has the following URI:

```
http://www4.wiwiss.fu-berlin.de/wikicompany/resource/ABC_News_Now
```

And knowing this fact is very useful. For example, one thing we can do, among many others, is to aggregate the information from both sites so as to learn more about this news broadcasting network.

This example has captured the main idea behind these extended datasets provided by DBpedia project. Let us take a look at one more example.

Another extended dataset is called Links to RDF Bookmashup. It maps DBpedia books to the books identified by the RDF Bookmashup project developed by Freie Universität Berlin.¹ RDF Bookmashup assigns URIs to books, authors, reviews and online bookstores, and purchase offers. Whenever a book title is submitted, the mashup queries Amazon API for information about the book and Google Base API for purchase offers from different bookstores that sell the book. The aggregated information is returned in RDF format which can be understood by applications.

List 10.22 shows some example statements from this extended dataset.

List 10.22 Example statements from Links to RDF Bookmashup dataset

```
<http://dbpedia.org/resource/Honour_Among_Thieves>
<http://www.w3.org/2002/07/owl#sameAs>
<http://www4.wiwiss.fu-berlin.de/bookmashup/books/9780330419031>.
```

```
<http://dbpedia.org/resource/Honour_Among_Thieves>
<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
<http://dbpedia.org/class/Book>.
```

```
<http://dbpedia.org/resource/Acquainted_With_the_Night>
<http://www.w3.org/2002/07/owl#sameAs>
<http://www4.wiwiss.fu-berlin.de/bookmashup/books/0002006391>.
```

```
<http://dbpedia.org/resource/Acquainted_With_the_Night>
<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
<http://dbpedia.org/class/Book>.
```

10.3.3 Access DBpedia as Linked Data

Finally, we can access DBpedia datasets as part of Linked Data. Linked Data will be covered in the next chapter, and you will see more on this topic until then. For now, understanding two important aspects of DBpedia project will prepare you well.

The first thing to understand is that DBpedia's resource identifiers are set up in such a way that DBpedia dataset server will deliver different documents based on different requests. More specifically, the following URI that identifies Roger Federer

http://dbpedia.org/resource/Roger_Federer

¹<http://www4.wiwiss.fu-berlin.de/bizer/bookmashup/>

will return RDF descriptions when accessed by Semantic Web applications and will return HTML content for the same information if accessed by traditional Web or human users. This is the so-called content negotiation mechanism that we will discuss in detail in the next chapter.

Second, to facilitate URI reuse, DBpedia should be consulted whenever you are ready to describe a resource in the world. Since DBpedia is the machine-readable version of Wikipedia, it is therefore possible that DBpedia has already created a URI for the resource you intend to describe. For example, Nikon D300 camera that we have been working with in the earlier chapters of this book has the following DBpedia URI:

`http://dbpedia.org/resource/Nikon_D300`

As we will see in the next chapter, using URIs created by DBpedia project not only promotes URI reuse but also helps to create more Linked Data. In fact, Linked Data publishers often try to find DBpedia resource URIs to build more links.

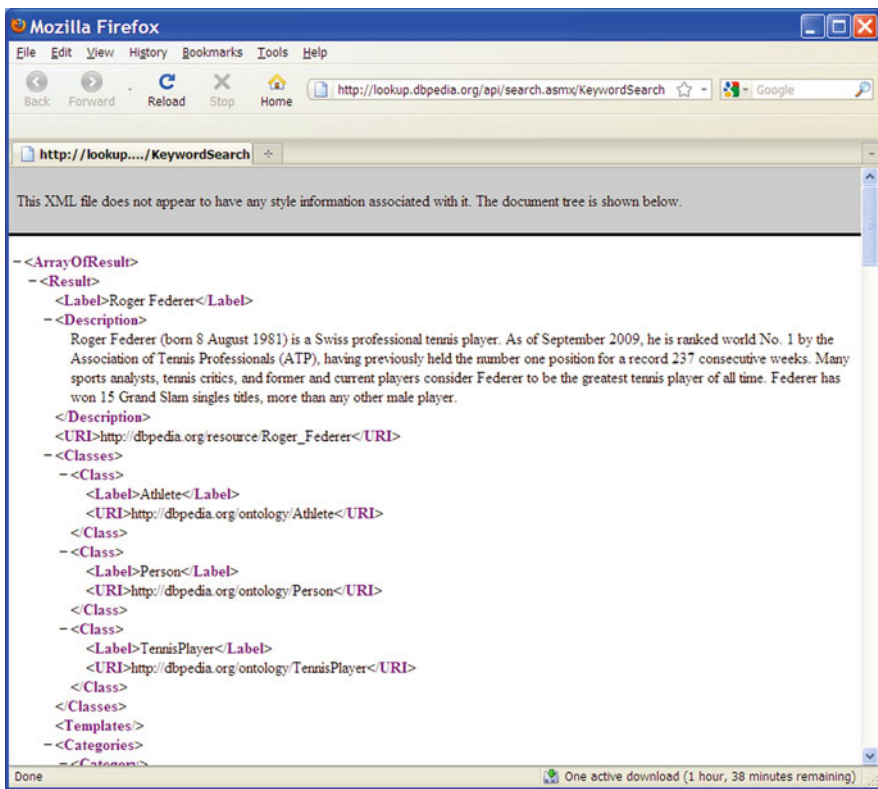


Fig. 10.4 Use DBpedia’s URI lookup service to find URIs of Roger Federer

In order to make it easy to discover these related URIs, DBpedia provides a lookup service that returns DBpedia URIs for a given set of keywords. This service can be accessed from the following URL:

`http://lookup.dbpedia.org/api/search.asmx`

and not only human users can directly access it, it can also be used as a Web service. The submitted keywords will be used to compare against the `rdfs:label` property of a given resource, and the most likely matches will be returned. Fig. 10.4 shows the results when using “Roger Federer” as keywords.

As shown in Fig. 10.4, http://dbpedia.org/resource/Roger_Federer is returned as the first choice. Therefore, if you were to search a URI that identifies Federer, you would be easily finding it.

10.4 Summary

In this chapter, we have learned DBpedia. It is not only another example of the Semantic Web technologies at work, but also a key component on the Web of Linked Data, as we will see in the next chapter.

First off, understand the following about DBpedia:

- It is a machine-readable version of Wikipedia.
- It is automatically generated by processing the pre-existing structured information on each page in Wikipedia.
- Also, understand that in order to make DBpedia machine readable, the following has been implemented by the DBpedia project team:
 - A DBpedia ontology is defined by manually mapping infobox templates to classes and template attributes to properties.
 - A DBpedia extractor is developed to process the infobox on a given Wikipedia page, an RDF document that represents the page is generated, and terms from the DBpedia ontology are used when generating this RDF document.
 - The DBpedia extractor has visited a good portion of Wikipedia; the result is the DBpedia dataset, which is machine readable.

Finally, understand different ways of accessing DBpedia:

- DBpedia can be accessed by using a Web browser.
- DBpedia can be accessed by using a SPARQL endpoint.
- DBpedia can be accessed as part of the Linked Data.

Reference

Jentzsch A (2009) DBpedia – extracting structured data from Wikipedia. Presentation at Semantic Web in Bibliotheken (SWIB2009), Cologne, Germany

Chapter 11

Linked Open Data

In [Chap. 9](#) we have studied semantic wiki, where semantic information is manually added to the Web content. In [Chap. 10](#), we have studied DBpedia project, where semantic documents are automatically generated. As we have discussed in [Chap. 7](#), besides annotating the pages manually or generating the markup documents automatically, there is indeed another solution: to create a machine-readable Web all from the scratch.

The idea is simple: if we start to publish machine-readable data, such as RDF documents on the Web, and somehow make all these documents connected to each other, then we will be creating a Linked Data Web that can be processed by machines.

This is the idea behind the Linked Open Data (LOD) project, the topic of this chapter.

11.1 The Concept of Linked Data and Its Basic Rules

In recent years, the concept of *Linked Data*, and the so-called Web of Linked Data, has attracted tremendous attention from both the academic world and real application world. In this section, we will examine the concept of Linked Data and its basic rules. What we will learn here from this section will provide a solid foundation for the rest of this chapter.

11.1.1 The Concept of Linked Data

The concept of Linked Data was originally proposed by Tim Berners-Lee in his 2006 Web architecture note.¹ Technically speaking, Linked Data refers to data published on the Web in such a way that it is machine readable, its meaning is explicitly defined, it is linked to other external datasets, and it can in turn be linked to from external datasets as well. Conceptually, Linked Data refers to a set of best practices for publishing and connecting structured data on the Web.

¹<http://www.w3.org/DesignIssues/LinkedData.html>

The connection between Linked Data and the Semantic Web is quite obvious: publishing and consuming machine-readable data is the center for both of these concepts. In fact, in recent years, Linked Data and the Semantic Web have become two concepts that are interchangeable. After finishing this chapter, you will reach your own conclusion regarding the relationship between Linked Data and the Semantic Web.

In practice, the basic idea of Linked Data is quite straightforward and can be summarized as follows:

- use the RDF data model to publish structured data on the Web and
- use RDF links to interlink data from different data sources.

Applying these two simple tenets repeatedly leads to the creation of a Web of Data that machine can read and understand. This Web of Data, at this point, can be understood as one realization of the Semantic Web. The Semantic Web, therefore, can be viewed as created by the linked structured data on the Web.

Given the fact that Linked Data is also referred to as the *Web of Linked Data*, it is then intuitive to believe that it must share lots of common traits exhibited by the traditional Web. This is a true intuition, yet for every single one of these traits, the Web of Linked Data is profoundly different from the Web of document.

Let us take a look at this comparison, which will certainly give us more understanding about Linked Data and the Semantic Web. Note that at this point, some of the comparisons may not make perfect sense to you, but rest assured that they will become clear after you have finished the whole chapter.

- On the traditional Web, anyone can publish anything at his/her will, at any time.

The same is true for the Linked Data Web: anyone, at any time, can publish anything on the Web of Linked Data, except that the published documents have to be RDF documents. In other words, these documents are for machines to use, not for human eyes.

- To access the traditional Web, we use Web browsers.

The same is true for the Web of Linked Data. However, since the Web of Linked Data is created by publishing RDF documents, we use Linked Data browsers that can understand RDF documents and can follow the RDF links to navigate between different data sources. Traditional Web browsers, on the other hand, are designed to handle HTML documents, and they will not be the best choices when it comes to accessing the Web of Linked Data.

- Traditional Web is interesting since everything on the Web is linked together.

The same is true for the Web of Linked Data. An important fact, however, is that under the hood, the HTML documents contained by the traditional Web are connected by un-typed hyperlinks. For the Web of Linked Data, rather than simply connecting documents, it uses RDF model to make *typed links* that connect

arbitrary things in the world. The result is that we can then build much smarter applications as we will see in the later part of this chapter.

- Traditional Web can provide structured data which can be consumed by Web-based applications.

This is especially true with more and more APIs being published by major players on the Web. For example, eBay, Amazon, Google all have published their APIs. Web applications that consume these APIs are collectively named as *mashups*, and they do offer quite impressive Web experiences to their users. On the other hand, under the Web of Linked Data, mashups are called *semantic mashups*, and they can be developed in a much more scalable and efficient way. More importantly, they have the ability to grow dynamically upon unbounded datasets, and that is what makes them much more useful than traditional mashups. Again, details will be covered in later sections.

Before we move on, understand that the technical foundation for the Web of Linked Data is not something we have to create from the ground up. To its very bottom, the Web of Linked Data is a big collection of RDF triples, where the subject of any triple is a URI reference in the namespace of one dataset, and the object of the triple is a URI reference in the namespace of another. In addition, by employing HTTP URIs to identify resources, HTTP protocol as retrieval mechanism and RDF data model to represent resource descriptions, Linked Data is directly built upon the general architecture of the Web – a solid foundation that has been tested for more than 20 years.

Furthermore, what we have learned so far, such as RDF model, RDF Schema, OWL, and SPARQL, all these technical components will find their usages in the world of Linked Data.

11.1.2 How Big Is the Web of Linked Data and the LOD Project

The most accurate way to calculate the size of the Web of Linked Data is to use a crawler to count the number of RDF triples that it has collected when traveling on the Web of Linked Data. This is quite a challenging task, and given the fact that some of the RDF triples are generated dynamically, we therefore have to run the crawler repeatedly in order to get the most recent count.

However, the size of the Web of Data can be estimated based on the dataset statistics collected by the LOD community in the ESW Wiki.² According to these statistics, the Web of Data, on 4 May 2010, consists of 13.1 billion RDF triples, which are interlinked by around 142 million RDF links (as of 29 September 2009). Note the majority of these triples are generated by the so-called wrappers, which are utility applications responsible for generating RDF statements from existing

²<http://esw.w3.org/topic/TaskForces/CommunityProjects/LinkingOpenData/DataSets/Statistic>,
<http://esw.w3.org/topic/TaskForces/CommunityProjects/LinkingOpenData/DataSets/LinkStatistics>

relational database tables, and only a small portion of these triples are generated manually.

The Linking Open Data Community Project has been focusing on the idea and implementation of the Web of Data for the last several years. It was originally sponsored by W3C Semantic Web Education and Outreach Group, and you can find more information about this group from this URL:

<http://www.w3.org/2001/sw/sweo/>

For the rest of this chapter, we will mainly examine the Linked Data project from technical perspective; you can always find more information on the project from the following Web sites:

- Linking Open Data project wiki home page:

<http://esw.w3.org/SweoIG/TaskForces/CommunityProjects/LinkingOpenData>

- Linked Data at the ESW Wiki page:

<http://esw.w3.org/topic/LinkedData>

- Linked Data Community Web site:

<http://linkeddata.org/>

11.1.3 The Basic Rules of Linked Data

The basic idea of Linked Data is to use RDF model to publish structured data on the Web and also use RDF links to interlink data from different data sources.

In practice, to make sure the above idea is carefully and correctly followed when constructing the Web of Linked Data, four basic rules are further proposed by Tim Berners-Lee in his 2006 Web architecture note:

Rule 1. Use URIs as names for things.

Rule 2. Use HTTP URIs so that a client (machine or human reader) can look up these names.

Rule 3. When someone looks up a URI, useful information should be provided.

Rule 4. Include links to other URIs, so that a client can discover more things.

The first rule is obvious, and it is also what we have been doing all the time: for a given resource or concept, we should use a unique and universal name to identify it. This simple rule eliminates the following two ambiguities on the traditional Web: (1) same name (word) in different documents can refer to completely different resources or concepts and (2) a given resource or concept can be represented by different names (words) in different documents.

The second rule simply puts one more constraint on the first rule by specifying that not only should we use URIs to represent objects and concepts, but we should also only use HTTP URIs.

The reason behind this rule is quite obvious. To make sure that data publishers can come up with identifiers that are indeed globally unique without involving any centralized management, the easiest way is to use HTTP URIs, since the domain part of these URIs can automatically guarantee their uniqueness. In addition, HTTP URIs naturally suggest to the clients that these URIs can be directly used as a means of accessing information about the resources over the Web.

The third rule further strengthens the second rule: if the client is dereferencing a given URI in a Web browser, there should always be some useful information returned back to the client. In fact, at the early days of the Semantic Web, this was not always true: when a given URI was used in a browser, there might or might not be any information coming back at all. We will see more details on this rule later.

The last rule is to make sure the Linked Data world will grow into a real Web: without the links, it will not be a Web of data. In fact, the real interesting thing happens only when the data are linked together and the unexpected fact is discovered by exploring the links.

Finally, note that the above are just the rules of the Web of Linked Data; breaking these rules does not destroy anything. However, without these rules, the data will not be able to provide anything that is interesting.

Now that we have all the background information and we have also learned all the rules, let us take a detailed look into the world of Linked Data. In the next two sections, we will first study how exactly to publish RDF data on the Web; we will then explore different ways to link these data together on the Web.

11.2 Publishing RDF Data on the Web

RDF data are the building blocks of Linked Data. To publishing RDF data on the Web means to follow these steps:

- identifying things by using URIs;
- choosing vocabularies for RDF data;
- producing RDF statements to describe the things;
- creating RDF links to other RDF datasets; and finally
- serving your RDF triples on the Web.

Let us study each one of them in detail.

11.2.1 Identifying Things with URIs

11.2.1.1 Web Document, Information Resource, and URI

To begin with, URI is not something new, and for most of us, a URI represents a Web document. For example, the following URI:

`http://www.liyangyu.com/`

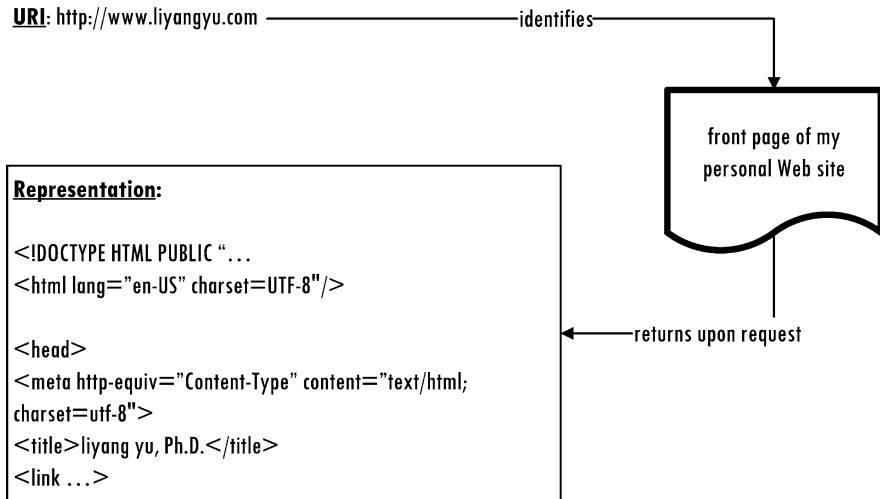


Fig. 11.1 URI/URL for information resource

represents the front page of my personal Web site. This page, like everything else on the traditional Web, is a Web document. We often call the above URI a URL, and as far as Web document is concerned, URL and URI are interchangeable: URL is a special type of URI; it tells us the location of the given Web document. In other words, if a user types in the above URL (URI) into a Web browser, the front page of my Web site will be returned.

Recall that a Web document is defined as something that has a URI and can return representations of the identified resource in response to HTTP requests. The returned representations can take quite a few formats including HTML, JPEG, or RDF, just to name a few.

In recent years, Web documents have a new name: *information resources*. More precisely, everything we find on the traditional document Web, such as documents, images (and other media files) are information resources. In other words, information resources are the resources that satisfy the following two conditions:

- can be identified by URIs;
- can return representations when the identified resources are requested by the users.

Figure 11.1 shows the above concept.

Currently on the Web, to request the representations of a given Web document, clients and servers use HTTP to communicate. For example, the following could be the request sent to the server:

```
GET / HTTP/1.1
Host: www.liyangyu.com
Connection: close
```

```
User-Agent: Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1)
Accept-Encoding: gzip
Accept-Charset: ISO-8859-1,UTF-8;q=0.7,*;q=0.7
Cache-Control: no
Accept-Language: de,en;q=0.7,en-us;q=0.3
```

And the server will answer with a response header, which tells the client whether the request has been successful, and if successful, the content (representation) will follow the response header.

Let us go back to our basic question in this section: what URIs should we use to identify things in the world? At this point, we can come up with part of the answer: for all the information resources, we can simply use the good old URLs as their URIs to uniquely identify them.

Now, what URIs should we use for the rest of the things (resources) in the world?

11.2.1.2 Non-information Resources and Their URIs

Except for the information resources, the rest of the resources in the world are called *non-information resources*. In general, non-information resources include all the real-world objects that exist outside the Web, such as people, places, concepts, ideas, anything you can imagine, and anything you want to talk about.

To come up with URIs that can be used to identify these non-information resources, there are two important rules proposed by W3C Interest Group.³ Let us use some examples to understand them.

Let us say I want to come up with a URI to represent myself (a non-information resource). Since I already have a personal Web site, www.liyangyu.com, could I then use the following URI to identify myself?

```
http://www.liyangyu.com/
```

This idea is quite intuitive, given the fact that the Web document at the above location does describe me and the URI itself is also unique. However, this clearly confuses a person with a Web document. For any user, the first question that comes to mind will be, does this URI represent this person's home page, or does it represent him as a person?

If we do use the above URI to identify myself, it is then likely that part of my FOAF file would look like the following:

```
<rdf:Description rdf:about="http://www.liyangyu.com/">
  <foaf:name>liyang yu</foaf:name>
  <foaf:title>Dr</foaf:title>
  <foaf:givenname>liyang</foaf:givenname>
  <foaf:family_name>yu</foaf:family_name>
  <foaf:mbox rdf:resource="liyang910@yahoo.com"/>
```

³Cool URIs for the Semantic Web, W3C Interest Group Note 03 December 2008 (<http://www.w3.org/TR/cooluris/>).

Now, if this URI represents my home page, then how could a home page have `foaf:name`, and how could it also have a `foaf:mbox`? On the other hand, if this URI does represent a person named Liyang Yu, then the above FOAF document in general seems to be describing a home page which has a Web address given by www.liyangyu.com.

All these said, it seems to be clear that I need another unambiguous URI to represent myself. And this gives the first rule summarized by W3C Interest Group:

Be unambiguous: There should be no confusion between identifiers for Web documents and identifiers for other resources. URIs are meant to identify only one of them, so one URI cannot stand for both a Web and a real-world object.

Now let us say I have already come up with a URI to represent myself, for example,

```
http://www.liyangyu.com/foaf.rdf#liyang
```

then what happens if the above URI is dereferenced in a browser – do we get anything back at all? If yes, what do we get back?

For information resources, we get one possible form of representation back, could be a HTML page, for example. For non-information resources, based on the following rule proposed by W3C Interest Group, when their URIs are used in a browser, related information should be retrieved as follows:

Be on the Web: Given only a URI, machines and people should be able to retrieve a description about the resource identified by the URI from the Web. Such a look-up mechanism is important to establish shared understanding of what a URI identifies. Machines should get RDF data and humans should get a readable representation, such as HTML. The standard Web transfer protocol, HTTP, should be used.

This rule makes it clear that for URIs identifying non-information resources, some descriptions should be returned to the clients. However, it does not specify any details for implementation purpose.

It turns out in the world of Linked Data, the implementation of this rule also dictates how the URIs for non-information resources are constructed. Let us cover the details next.

11.2.1.3 URIs for Non-information Resources: 303 URIs and Content Negotiation

The first solution is to use the so-called *303 URIs* to represent non-information resources. The basic idea is to create a URI for a given non-information resource, and when a client posts a request using this URI, the server will return the special HTTP status code 303 *See Other*. This not only indicates the fact that the requested resource is not a regular Web document, but also further redirects the client to some other document which provides information about the thing identified by this URI. By doing so, we will be able to satisfy the above two rules and also avoid the ambiguity between the real-world object and the non-information resource that represents it.

As a side note, if the server answers the request using a status code in the 200 range, such as 200 OK, it is then clear that the given URI represents a normal Web document or information resource.

Now, in case where a 303 See Other status code is returned, which document should the server redirect its client to? This depends on the request from the client. If the client is an RDF-enabled browser (or some applications that understands RDF model), it will more likely prefer a URI which points to an RDF document. If the browser is a traditional HTML browser (or the client is a human reader), it will then more likely prefer a URI that points to a HTML document. In other words, when sending the request, the client will include information in the HTTP header to indicate what type of representation it prefers. The server will inspect this header to return a new URI that links to the appropriate response. This process is called *content negotiation*.

It is now a common practice that for a given real-world resource, we can often have three URIs for it. For example, for myself as a non-information resource, the following three URIs will be in use:

- a URI that identifies myself as a non-information resource:

`http://www.liyangyu.com/resource/liyang`

- a URI that identifies a Web document which has an RDF/XML representation describing myself. This URI will be returned when a client prefers an RDF description:

`http://www.liyangyu.com/data/liyang`

- a URI identifies a Web document that has a HTML representation describing myself. This URI will be returned when a client prefers a HTML document:

`http://www.liyangyu.com/page/liyang`

And the first URI,

`http://www.liyangyu.com/resource/liyang`

is often the one that is seen by the outside world as my URI.

The above schema for constructing URIs for non-information resources is also viewed as the best practice by the Linked Data community. Another example is the following three URIs about Berlin, as seen in DBpedia project:

- a URI that is used as the identifier for Berlin:

`http://dbpedia.org/resource/Berlin`

- a URI that identifies a representation in HTML format (for human readers):

`http://dbpedia.org/page/Berlin`

- a URI that identifies a representation in RDF/XML format (for machines):

`http://dbpedia.org/data/Berlin`

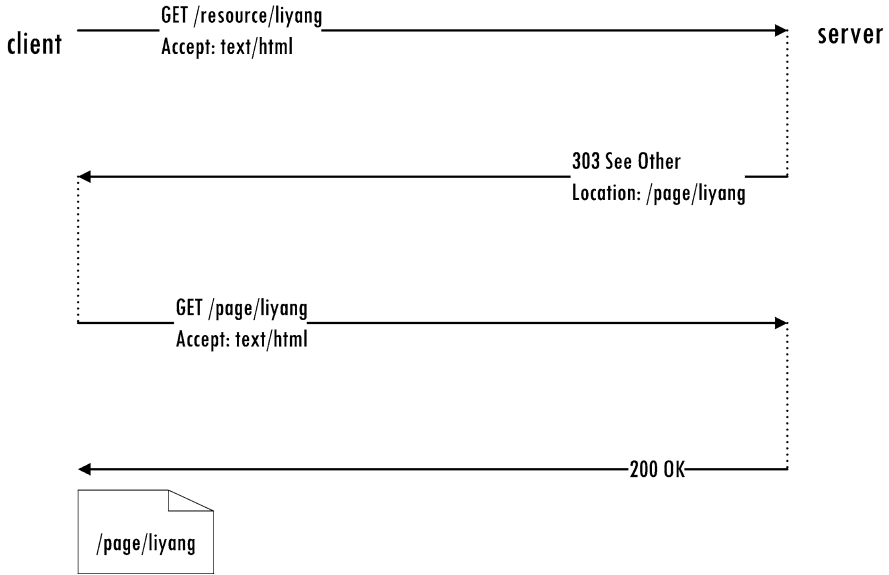


Fig. 11.2 Example of content negotiation

It is now clear that for a given resource, there could be multiple content types for its representation, such as HTML format and RDF/XML format as seen above. Figure 11.2 shows the process of content negotiation, using my own URI as an example.

The following steps show the interaction between the server and a client:

- Client is requesting a HTML Web document:

```

GET /resource/liyang HTTP/1.1
Host: www.liyangyu.com
Accept: text/html

```

- Server’s response header should include the following fields:

```

HTTP/1.1 303 See Other
Location: http://www.liyangyu.com/page/liyang

```

- Client is requesting a machine-readable document for the resource:

```

GET /resource/liyang HTTP/1.1
Host: www.liyangyu.com
Accept: application/rdf+xml

```

- Server’s response header should include the following fields:

```

HTTP/1.1 303 See Other
Location: http://www.liyangyu.com/data/liyang

```


As a summary, 303 URIs require content negotiation when they are used in a browser to retrieve their descriptions. Furthermore, content negotiation requires at least two HTTP round-trips to the server to retrieve the desired document. However, 303 URIs eliminate the ambiguity between information and non-information resources, therefore they provide a uniform and consistent way of representing resource in the real world.

11.2.1.4 URIs for Non-information Resources: Hash URIs

A *hash URI* is a URI that contains a fragment, i.e., the part that is separated from the rest of the URI by a hash symbol (“#”). For example, the following is a hash URI to identify myself as a resource:

```
http://www.liyangyu.com/foaf.rdf#liyang
```

and *liyang* (to the right of #) is the fragment part of this URI.

Hash URI provides an alternative choice when it comes to identifying non-information resources. The reason behind this solution is related to the HTTP protocol itself.

More specifically, when a hash URI is used in a browser, the HTTP protocol requires the fragment part to be stripped off before sending the URI to the server. For example, if you dereference the above URI into a Web browser and also monitor the request sent out to the server, you will see the following lines in the request:

```
GET /foaf.rdf HTTP/1.1  
Host: www.liyangyu.com
```

Clearly, the fragment part is gone. Instead of retrieving this URI,

```
http://www.liyangyu.com/foaf.rdf#liyang
```

the client is in fact requesting this one:

```
http://www.liyangyu.com/foaf.rdf
```

In other words, a URI that includes a hash fragment cannot be retrieved directly, therefore it does not identify a Web document at all. As a result, any URI including a fragment part is a URI that identifies a non-information resource, thus the ambiguity is avoided.

Now that there is no ambiguity associated with a hash URI, what should be served if the URI is dereferenced in a browser? Since we know the fragment part will be taken off by the browser, we can simply serve a document (either human readable or machine readable) at the resulting URI which does not have the fragment part. Again using the following as the example,

```
http://www.liyangyu.com/foaf.rdf#liyang
```

we can then serve an RDF document identified by the URI:

```
http://www.liyangyu.com/foaf.rdf
```

Note that there is no need for any content negotiation, which is probably the main reason why hash URIs look attractive to us.

Hash URI does have its own downside. Consider the following three URIs:

```
http://www.liyangyu.com/foaf.rdf#liyang
http://www.liyangyu.com/foaf.rdf#connie
http://www.liyangyu.com/foaf.rdf#ding
```

which represent three different resources. However, using any one of them in a browser will send a single request to this common URI:

```
http://www.liyangyu.com/foaf.rdf
```

and if someone is only interested in `#connie`, still the whole document will have to be returned. Obviously, using hash URIs lacks the flexibility of configuring a response for each individual resource.

It is also worth mentioning that even when hash URIs are in used, we can still use content negotiation if we want to serve both HTML and RDF representations for the resources identified by the URIs. For example,

- Client is requesting a HTML Web document for the following resource,

```
http://www.liyangyu.com/foaf.rdf#liyang
```

and you will see these lines in the request:

```
GET /foaf.rdf HTTP/1.1
Host: www.liyangyu.com
Accept: text/html
```

- Response header from the server should include the following fields:

```
HTTP/1.1 303 See Other
Location: http://www.liyangyu.com/foaf.html
```

Note that we assume there is a HTML file called `foaf.html` which includes some HTML representations of the given resource.

- Now client is requesting machine-readable document for the resource:

```
GET /foaf.rdf HTTP/1.1
Host: www.liyangyu.com
Accept: application/rdf+xml
```

- Response header from the server should include the following fields:

```
HTTP/1.1 303 See Other
Location: http://www.liyangyu.com/foaf.rdf
```

And similarly, the following two hash URIs,

```
http://www.liyangyu.com/foaf.rdf#connie
http://www.liyangyu.com/foaf.rdf#ding
```

will have exactly the same content negotiation process, since their fragment parts will be taken off by the browser before sending them out to the server.

11.2.1.5 URIs for Non-information Resources: 303 URIs vs. Hash URIs

Now that we have introduced both 303 URIs and hash URIs, the next question is about when a 303 URI should be used and when a hash URI should be used. Table 11.1 briefly summarizes the advantages and disadvantages of both URIs.

The following is a simple guideline. Once you have more experience working with the Linked Data and the Semantic Web, you will be able to add more to it:

- For ontologies that are created by using RDF Schema and OWL, it is preferred to use hash URIs to represent all the terms defined in the ontology, and frequent access of this ontology will not generate lots of network redirects.
- If you need a quicker and easier way of publishing Linked Data or small and stable datasets of RDF resource files, hash URI should be the choice.
- Other than the above, 303 URIs should be used to identify non-information resources if possible.

11.2.1.6 URI Aliases

When it comes to identifying things with URIs, one obvious fact we have noticed so far is the lack of centralized control of any kind. In fact, anyone can talk about any resource and come up with a URI to represent that resource. It is therefore quite possible that different users happen to talk about the same non-information resource. Furthermore, since they are not aware of each other’s work, they create different URIs to identify the same resource or concept. Since all these URIs are created to identify the same resource or concept, they are called *URI aliases*.

It is commonly suggested that when you plan to publish RDF statements about a given resource, you should try to find at least some of the URI aliases for this resource first. If you can find one or multiple URIs for the resource, by all means reuse one of them, create your own if only you have very strong reason to do so. And in which case, you should use `owl:sameAs` to link it to at least one existing

Table 11.1 303 URI vs. Hash URI: advantages and disadvantages

	303 URI	Hash URI
Advantages	Provides the flexibility of configuring redirect targets for each resource Provides the flexibility of changing/updating these targets easily and freely, at any given time	Does not require content negotiation, therefore reduces the number of HTTP round-trips Since content negotiation is not required, publishing Linked Data is easier and quicker
Disadvantages	Requires two round-trips for each use of a given URI	All the resource descriptions have to be collected in one file

URI. Certainly, you can create your own URI if you cannot find any existing ones at all.

Now, how do you find the URI aliases for the given resource? At the time of this writing, there are some tools available on the Web. Let us use one example to see how these tools can help us.

Assume that we want to publish some RDF statements about Roger Federer, the tennis player who holds the most grand slam titles at current time. Since he is such a well-known figure, it is safe to assume that we are not the first one who would like to say something about him. Therefore, there should be at least one URI identifying him, if not more.

A good starting place where we can search for these URI aliases is the *Sindice* Web site. You can access this Web site here:

<http://sindice.com/>

More specifically, Sindice can be viewed as a Semantic Web search engine, and it was originally created at DERI (Digital Enterprise Research Institute) as a research project. Its main idea is to index the Semantic Web documents over the Web, so for a given URI, it can search its datasets and further tell us which dataset has mentioned this given URI.

To us, a more useful feature of Sindice is when searching its datasets, Sindice not only accepts URIs, but also takes keywords. When it accepts keywords, it will find all the URIs that either describe or closely match the given keywords first, then it will locate all the datasets that contain these URIs. This is what we need when we want to know if there are any existing URIs identifying Roger Federer. Figure 11.3 shows the query session.

And Fig. 11.4 shows the Sindice search result.

The first result in Fig. 11.4 shows a URI identifying Roger Federer (we know this by noticing the file type of this result, i.e., an RDF document), and this URI is given as follows:

http://dbpedia.org/resource/Roger_Federer

To collect other URI aliases identifying Roger Federer, we can continue to use Sindice. However, another tool, called *sameAs*, can also be very helpful. You can find sameAs by accessing its Web site:

<http://www.sameas.org/>

It will help us to find the URI aliases for a given URI. In our case, our search is shown in Fig. 11.5, and Fig. 11.6 shows the result:

Clearly, at the time of this writing, there are about 23 URIs identifying Roger Federer, as shown in Fig. 11.6. It is now up to us to pick one of these URIs so we can publish something about Roger Federer.

As you can tell, these two Web sites are very helpful on finding existing URIs. In fact, www.sameas.org even provides a link to www.sindice.com, as shown in Fig. 11.5. You can either directly enter a URI in the <sameAs> box to search

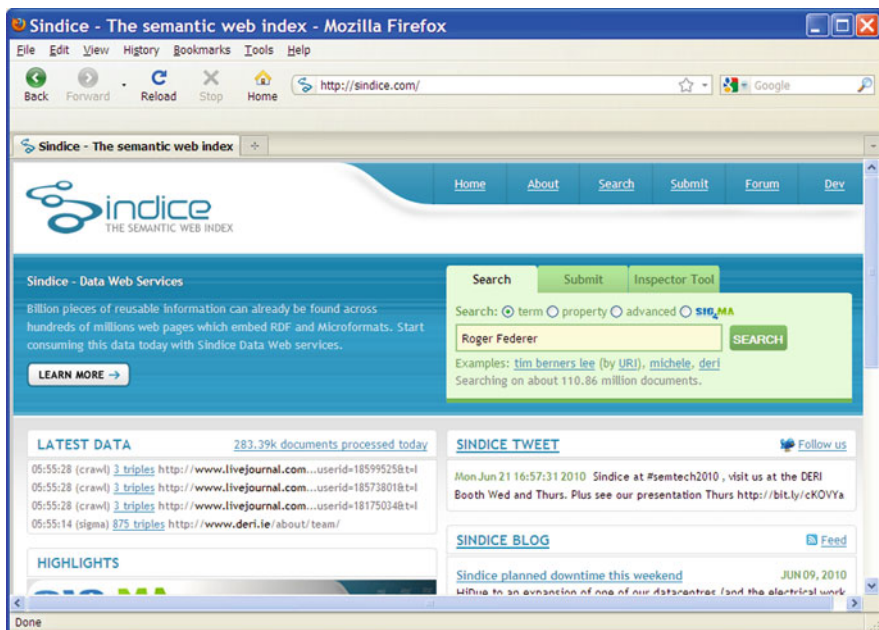


Fig. 11.3 A Sindice search session (search for Roger Federer)

for its URI aliases or you can use Sindice first by entering the keywords in the Sindice box.

Recall the lookup service we have discussed in Chap. 10 about DBpedia – it is another service we can use to locate URIs that are created by DBpedia for a given resource. See Fig. 10.4 and Sect. 10.3.3 for details.

At this point, we have briefly discussed about URI aliases. With the development of the Semantic Web, let us hope that there will better and better solutions out there, which will greatly facilitate the reuse of URIs.

11.2.2 Choosing Vocabularies for RDF Data

By now, you should understand that when publishing RDF statements, you should always try to use terms defined in one or more ontologies. For example, the predicate of an RDF statement should always be a URI that comes from the ontologies you are using. In addition, it is recommended that instead of inventing your own ontology, you should always use the terms from well-known existing ontologies. Reusing ontologies will make it possible for clients to understand your data and further process your data, therefore the data you have published can easily become part of the Web of Linked Data.

At this point, there is already a good collection of some well-known ontologies covering multiple application domains. You can find this collection at the Linking

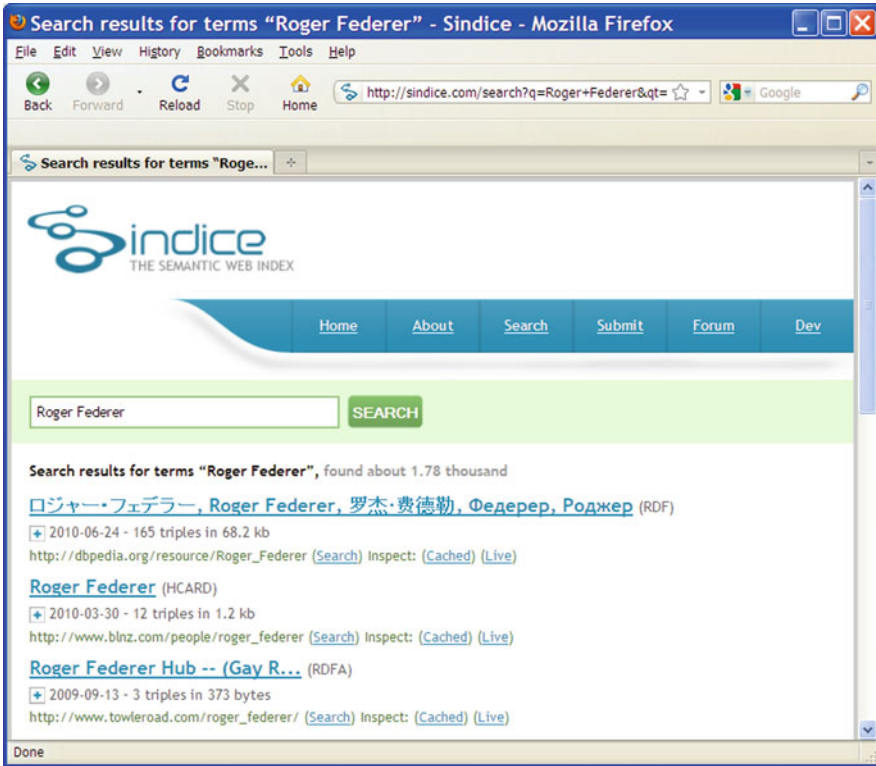


Fig. 11.4 Search results from Fig. 11.3

Open Data project wiki home page (see Sect. 11.1.2) and make sure to check back often for updates. The following is a short list, just to name a few:

- Friend-of-a-Friend (FOAF): terms for describing people;
- Dublin Core (DC): terms for general metadata attributes;
- Semantically Interlinked Online Communities (SIOC): terms for describing online communities;
- Description of a Project (DOAP): terms for describing projects;
- Music Ontology: terms for describing artists, albums, and tracks;
- Review Vocabulary: terms for representing reviews.

In case you do need to create your own ontology, it is still important to make use of the terms that are defined in these well-known ontologies. In fact, some of the ontologies given above, such as the Music Ontology, make use of the terms defined in other ontologies. For example, List 11.1 is taken from the Music Ontology, and it shows the definition of class SoloMusicArtist.

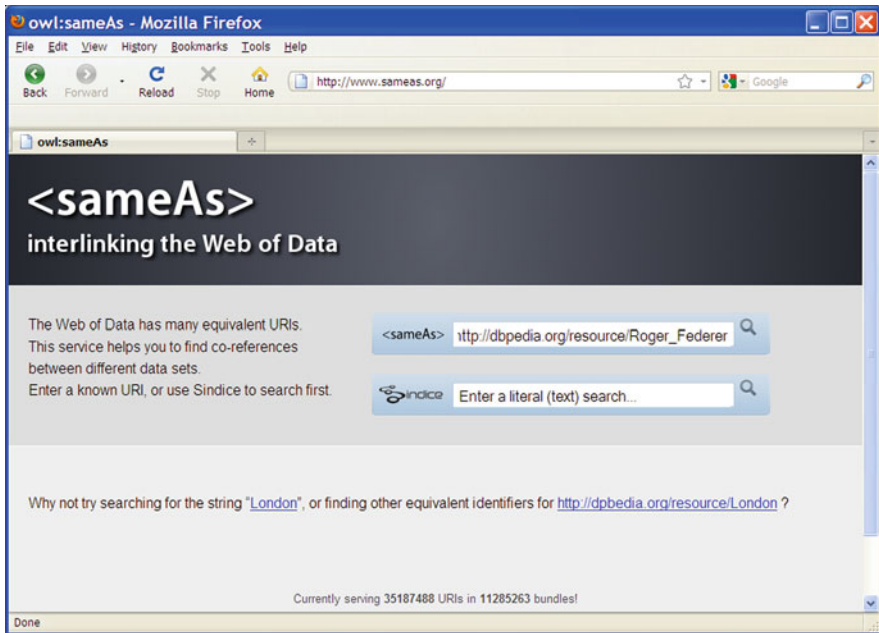


Fig. 11.5 Use sameAs to find URI aliases

List 11.1 Part of the Music Ontology

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE rdf:RDF [
  <!ENTITY dc 'http://purl.org/dc/elements/1.1/'>
  <!ENTITY mo 'http://purl.org/ontology/mo/'>
  <!ENTITY ns1 'http://www.w3.org/2003/06/sw-vocab-status/ns# '>
  <!ENTITY owl 'http://www.w3.org/2002/07/owl# '>
  <!ENTITY rdf 'http://www.w3.org/1999/02/22-rdf-syntax-ns# '>
  <!ENTITY rdfs 'http://www.w3.org/2000/01/rdf-schema# '>
  <!ENTITY xsd 'http://www.w3.org/2001/XMLSchema# '>
]>

<rdf:RDF
  xmlns:dc="&dc;"
  xmlns:mo="&mo;"
  xmlns:ns1="&ns1;"
  xmlns:owl="&owl;"
  xmlns:rdf="&rdf;"
  xmlns:rdfs="&rdfs;"
  xmlns:xsd="&xsd;"
>
...

```

```

<rdfs:Class rdf:about="&mo;SoloMusicArtist"
  mo:level="1"
  rdfs:label="SoloMusicArtist"
  ns1:term_status="stable">
  <rdfs:subClassOf rdf:resource="&mo;MusicArtist"/>
  <rdfs:subClassOf
    rdf:resource="http://xmlns.com/foaf/0.1/Person"/>
  <rdf:type rdf:resource="&owl;Class"/>
  <rdfs:comment>Single person whose musical creative work shows
sensitivity and imagination.
  </rdfs:comment>
  <rdfs:isDefinedBy rdf:resource="&mo;"/>
</rdfs:Class>
...

```

Note that SoloMusicArtist is defined as a sub-class of foaf:Person. Therefore, if a given client sees the following:

```

<rdf:Description
  rdf:about="http://zitgist.com/music/artist/79239441-bfd5-4981-
a70c-55c3f15c1287">
  <rdf:type
    rdf:resource="http://purl.org/ontology/mo/SoloMusicArtist"/>
</rdf:Description>

```

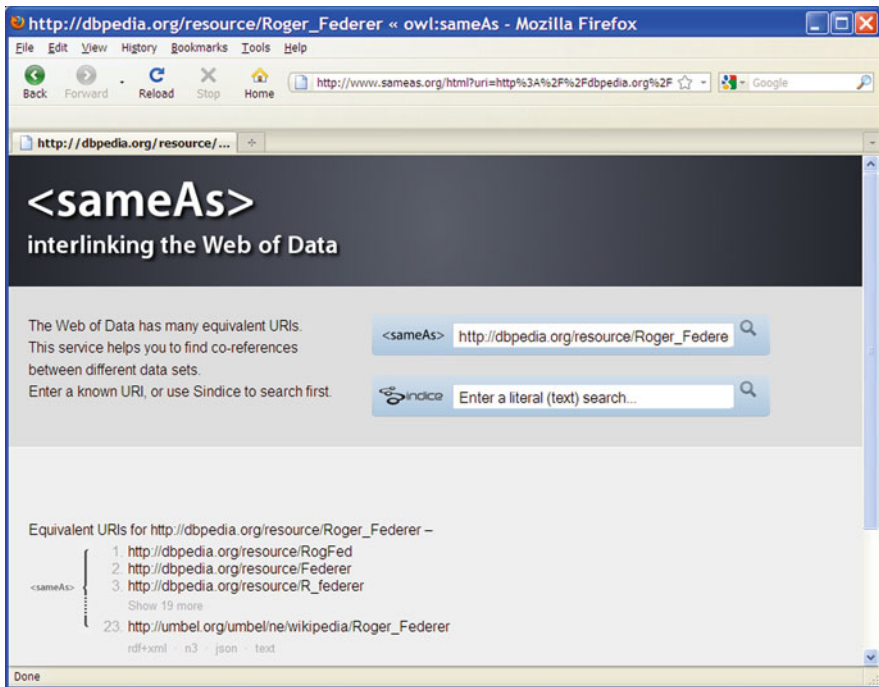


Fig. 11.6 sameAs search result

it will know the real-world resource identified by this URI,

```
http://zitgist.com/music/artist/79239441-bfd5-4981-a70c-55c3f15c1287
```

must be an instance of `foaf:Person`. If this client is not interested in any instance of `foaf:Person`, it can safely disregard any RDF statements that are related to this resource. Clearly, this reasoning is possible only when the authors of the Music Ontology have decided to make use of the terms defined in the FOAF ontology.

Creating ontology, like any other design work, requires not only knowledge, but also experience. It is always helpful to learn how other ontologies are created, so check out the ontologies listed above. After reading and understanding how these ontologies are designed and coded, you will be surprised to see how much you have learned. Also, with the knowledge you have gained, it is more likely that you will be doing a solid job when creating your own.

11.2.3 Creating Links to Other RDF Data

Now that you have come up with the URIs, and you have the terms from the ontologies to use, you can go ahead to make your statements about the world. There is only one thing you need to remember: you need to make links to other RDF datasets so your statements can participate in the Linked Data cloud.

In this section, we discuss the basic language constructs and ways you can use to add these links.

11.2.3.1 Basic Language Constructs to Create Links

Let us start with a simpler case: you are creating a FOAF document. The easiest way to make links in this case is to use `foaf:knows`, as we have shown in [Chap. 7](#). Using `foaf:knows` will not only make sure you can join the “circle of trust”, but also put your data into the Linked Data cloud.

In fact, besides `foaf:knows`, there are couple other FOAF terms you can use to create links. Let us take a look at some examples:

- Use **`foaf:interest`** to show your interest:

For example,

```
<rdf:RDF
  xmlns:dc="http://purl.org/dc/terms/"
  xmlns:foaf="http://xmlns.com/foaf/0.1/"
  <!-- other namespace definitions -->
>
<foaf:interest>
  <rdf:Description
    rdf:about="http://dbpedia.org/resource/Photography">
```

```

    <dc:title>photography</dc:title>
  </rdf:Description>
</foaf:interest>

```

This will link you to the world of photography as defined in DBpedia. And as you know, DBpedia is a major component of the Linked Data cloud.

- Use `foaf:base_near` to show where you are located:

For example,

```

<rdf:RDF
  xmlns:foaf="http://xmlns.com/foaf/0.1/"
  <!-- other namespace definitions -->
>
  <rdf:Description
    rdf:about="http://www.liyangyu.com/foaf.rdf#liyang">
    <foaf:name>liyang yu</foaf:name>
    <foaf:base_near
      rdf:resource="http://dbpedia.org/resource/Beijing"/>
    <!-- other descriptions I may want -->
  </rdf:Description>
</rdf:RDF>

```

This will link you to Beijing, the capital city of China, which is represented by DBpedia as <http://dbpedia.org/resource/Beijing>. Again, this is good enough for putting you into the Linked Data cloud.

With the above two examples, you understand that there are different FOAF terms you can use to link to the Web of Linked Data. We will leave it to you to discover other FOAF terms that can be used besides the above two examples.

For a more general case, at least two properties should be considered when making links: `rdfs:seeAlso` and `owl:sameAs`.

`rdfs:seeAlso` is defined in W3C's RDFS vocabulary, and it is used to indicate the fact that another resource might provide additional information about the subject resource. It therefore can be used to link the current RDF document into the Linked Data world.

In addition, note that `rdfs:domain` of `rdfs:seeAlso` is `rdfs:Resource`, and `rdfs:range` of `rdfs:seeAlso` is also `rdfs:Resource`. As a result, this property is entirely domain-neutral, and works for people, companies, documents, etc.

List 11.2 shows one simple example of using `rdfs:seeAlso`.

List 11.2 Use `rdfs:seeAlso` to create link

```

<rdf:Description
  rdf:about="http://www.liyangyu.com/foaf.rdf#liyang">
  <foaf:name>liyang yu</foaf:name>
  <foaf:title>Dr</foaf:title>

```

```

<foaf:givenname>liyang</foaf:givenname>
<!-- other descriptions here -->
<rdfs:seeAlso>
  <rdf:Description>
    rdf:about="http://www.liyangyu.com/people/connie.rdf">
  </rdf:Description>
</rdfs:seeAlso>
</rdf:Description>
</rdf:RDF>

```

`rdfs:seeAlso` property in List 11.2 says that you can find more information about resource `http://www.liyangyu.com/foaf.rdf#liyang` from another RDF document (`connie.rdf`). A given client can follow this link to download `connie.rdf` and expect to be able to parse this file and collect more information about the current resource.

This simple example in fact raises a very interesting question: when we build our application, is it safe to assume that the value of `rdfs:seeAlso` property will always be a document that can be parsed as RDF/XML?

Unfortunately, the answer is no. As we have discussed, the formal definition of `rdfs:seeAlso` is couched in very neutral terms, allowing a wide variety of document types. You could certainly reference a JPEG or PDF or HTML document with `rdfs:seeAlso`, which are not RDF documents at all. Therefore, an application should always account for all these possibilities when following the `rdfs:seeAlso` link.

Sometimes, it is a good idea to explicitly indicate that `rdfs:seeAlso` property is indeed used to reference a document that is in RDF/XML format. List 11.3 shows how this can be implemented.

List 11.3 Use `rdfs:seeAlso` together with `dc:format` to provide more information

```

<rdf:Description>
  rdf:about="http://www.liyangyu.com/foaf.rdf#liyang">
  <foaf:name>liyang yu</foaf:name>
  <foaf:title>Dr</foaf:title>
  <foaf:givenname>liyang</foaf:givenname>
  <!-- other descriptions here -->
  <rdfs:seeAlso>
    <rdf:Description>
      rdf:about="http://www.liyangyu.com/people/connie.rdf">
      <dc:format>application/rdf+xml</dc:format>
    </rdf:Description>
  </rdfs:seeAlso>
</rdf:Description>
</rdf:RDF>

```

Another useful feature about `rdfs:seeAlso` is that it is often used as a typed link, which can be very helpful to clients. List 11.4 shows one example of a typed link specified using `rdfs:seeAlso`.

List 11.4 `rdfs:seeAlso` used with typed link

```
<rdf:Description
  rdf:about="http://www.liyangyu.com/foaf.rdf#liyang">
  <foaf:name>liyang yu</foaf:name>
  <foaf:title>Dr</foaf:title>
  <foaf:givenname>liyang</foaf:givenname>
  <!-- other descriptions here -->
  <rdfs:seeAlso>
    <rdf:Description
      rdf:about="http://www.liyangyu.com/publication/liyang">
      <rdf:type
        rdf:resource="http://example.org/someAuthorClassDefinition"/>
      <dc:format>application/rdf+xml</dc:format>
    </rdf:Description>
  </rdfs:seeAlso>
  <rdfs:seeAlso>
    <rdf:Description
      rdf:about="http://www.liyangyu.com/publication/yu_cv.rdf">
      <rdf:type
        rdf:resource="http://example.org/someResumeClassDefinition"/>
      <dc:format>application/rdf+xml</dc:format>
    </rdf:Description>
  </rdfs:seeAlso>
</rdf:Description>
</rdf:RDF>
```

Imagine an application that is only interested in publications (not CVs). This typed link will help the application to eliminate the second `rdfs:seeAlso`, but only concentrate on the first one.

`owl:sameAs` is not something new either. It is defined by OWL to state that two URI references refer to the same individual. It is now frequently used by Linked Data publishers to create links between datasets. For example, Tim Berners-Lee, in his own FOAF file, has been using the following URI to identify himself:

`http://www.w3.org/People/Berners-Lee/card#i`

and he also uses the following four `owl:sameAs` properties to state that the individual identified by the above URI is the same individual as identified by these URIs:

```
<owl:sameAs rdf:resource="http://identi.ca/user/45563"/>
<owl:sameAs
rdf:resource="http://www.advogato.org/person/timbl/foaf.rdf#me"/>
<owl:sameAs rdf:resource=
"http://www4.wiwiw.fu-berlin.de/bookmashup/persons/Tim+Berners-
Lee"/>
```

```
<owl:sameAs rdf:resource=
  "http://www4.wiwiss.fu-berlin.de/dblp/resource/person/100007"/>
```

and clearly, some of these URIs can be used to link his FOAF document into the Linked Data cloud. In fact, at this point, the last two URIs are indeed used for this purpose.

`owl:sameAs` can certainly be used in other RDF documents. Generally speaking, when instances of different classes refer to the same individual, these instances can be identified and linked together by using `owl:sameAs` property. This directly supports the idea that the same individual can be seen in different context as entirely different entities, and by linking these entities together, we can discover the unexpected facts that are both interesting and helpful to us.

The above discussion has listed some basic language constructs we can use to create links. In practice, when it comes to creating links in RDF documents, there are two methods: creating the links manually or generating the links automatically. Let us briefly discuss these two methods before we close this section.

11.2.3.2 Creating Links Manually

Manually creating links is quite intuitive, yet it does require you to be familiar with the published and well-known linked datasets out there, therefore you can pick your linking targets. In particular, the following steps are normally followed when creating links manually in your RDF document:

- Understand the available linked datasets.

This can be done by studying the currently available datasets published and organized by experts in the field. For example, as of July 2009, Richard Cyganiak has published the *LOD Cloud* as shown in Fig. 11.7. The updated version can be accessed from this location:

<http://linkeddata.org/images-and-posters>

And if you access this Linked Data collection at the above location, you can actually click each dataset and start to explore that particular dataset. This will help you to get an overview of all the datasets that are available today, and you can also select the dataset(s) that you wish to link into.

- Find the URIs as your linking targets.

Once you have selected the datasets to link into, you can then search in these datasets to find the URIs that you want to link to. Most datasets provide a search interface, such as a SPARQL endpoint, so you can locate the appropriate URI references for your purpose. If there is no search interface provided, you can always use Linked Data browsers to explore the dataset, as we will discuss in a later section.

With the above two steps, you can successfully create your links. Let us take a look at a simple example.

This will successfully put my own small FOAF document into the Web of Linked Data.

In some cases, you can use a relatively direct way to find the URI reference that you can use to create your links. For example, without selecting any datasets, we can directly search the phrase “the Semantic Web” in `Sindice.com`. We can easily find a list of URIs that have been created to identify this concept, including the URI coined by DBpedia.

11.2.3.3 Creating Links Automatically

Compared to manually creating links, generating links automatically is certainly more efficient and more scalable, and it is always the preferred method if possible. However, at the time of this writing, there is still a lack of good and easy-to-use tools to automatically generate RDF links. In most cases, dataset-specific algorithms have to be designed to accomplish the task. In this section, we will briefly discuss this topic so as to give you some basic idea along this direction.

A collection of often used algorithms is the so-called *pattern-based algorithms*. This group of algorithms take advantage of the fact that for a specific domain, there may exist some generally accepted naming pattern, which could be useful for generating links.

For example, in the publication domain, if ISBN is included as part of the URI that is used to identify a book, such as the case in the RDF Book Mashup dataset, then a link can be created with ease. More specifically, DBpedia can locate all the wiki pages for books, and if a given wiki page has an ISBN number included, this number is used to search among the URIs used by RDF Book Mashup dataset. When a match is found, an `owl:sameAs` link will be created to link the URI of the book in DBpedia to the corresponding RDF Book Mashup URI. This algorithm has helped to generate at least 9000 links between DBpedia and RDF Book Mashup dataset.

In cases where no common identifiers can be found across datasets, more complex algorithms have to be designed based on the characteristics of the given datasets. For example, many geographic places appear in Geonames⁴ dataset as well as in DBpedia dataset. To make the two sets of URIs representing these places link together, the Geonames team has designed a property-based algorithm to automatically generate links. More specifically, properties such as latitude, longitude, country, and population are taken into account, and a link will be created if all these properties show some similarity as defined by the team. This algorithm has generated about 70,500 links between the datasets.

As a summary, automatic generation of links is possible for some specific domain, or with a specifically designed algorithm. When it is used properly, it is much more scalable than the manual method.

⁴<http://www.geonames.org/ontology/>

11.2.4 Serving Information as Linked Data

11.2.4.1 Minimum Requirements for Being Linked Open Data

Before we can put our data onto the Web, we need to make sure it satisfies some minimal requirements in order to be qualified as “Linked Data on the Web”:

1. If you have created any new URI representing non-information resource, this new URI has to be dereferenceable in the following sense:
 - your Web server must be able to recognize the MIME-type `application/rdf+xml`;
 - your Web server has to implement the 303 redirect as described in Sect. 11.2.1.3. In other words, your Web server should be able to return a HTTP response containing a HTTP redirect to a document that satisfies the client’s need (either an `rdf+xml` document or a `html+text` document, for example);
 - if implementing 303 redirect on your Web server is not your plan, your new URIs have to be hash URIs as we have discussed in Sect. 11.2.1.5.
2. You should include links to other data sources, so a client can continue its navigation when it visits your data file. These links can be viewed as outbound links.
3. You should also make sure there are external RDF links pointing at URIs contained in your data file, so the open Linked Data cloud can find your data. These links can be viewed as the inbound links.

At this point, these requirements should look fairly straightforward. The following are some technical details that you should be aware of.

First off, you need to make sure your Web server is able to recognize the `rdf+xml` as a MIME type. Obviously, this is necessary since once you have published your data into the Linked Data cloud, different clients will start to ask for `rdf+xml` files from your server. In addition, this is a must if you are using hash URIs to identify real-world resources.

A popular tool we can use for this purpose is called `cURL`,⁵ which provides a command-line HTTP client that communicates with a given server. It is therefore able to help us to check whether a URI supports some given requirements, such as understanding `rdf+xml` as a MIME type, supporting 303 redirects and content negotiation, just to name a few.

To get this free tool, go to this place:

<http://curl.haxx.se/download.html>

and on this page, you will find different packages for different platforms. For windows users, you can find the download here:

<http://curl.haxx.se/download.html#Win32>

⁵<http://curl.haxx.se/>

Once you have downloaded the package, you can extract it to a location of your choice, and you should be able to find `curl.exe` in that location. You can then start to test whether a given server is able to recognize the `rdf+xml` MIME type.

For testing purpose, we can request my own URI as follows:

```
curl -I http://www.liyangyu.com/foaf.rdf#liyang
```

Note that the `-I` parameter has to be used here (refer to `cURL`'s documentation for details). Once we submit the above line, the server sends back the content type and other HTTP headers along with the response. For this example, the following is part of the result:

```
HTTP/1.1 200 OK
Last-Modified: Tue, 11 Aug 2009 02:49:10 GMT
Accept-Ranges: bytes
Content-Length: 1152
Content-Type: application/rdf+xml
Connection: close
```

The important line is the `Content-Type` header. We see the file is served as `application/rdf+xml`, just as it should be. If we were to see `text/plain` here or if the `Content-Type` header was missing, the server configuration would have to be changed.

When it comes to fixing the problem, it does depend on the server. Use Apache as an example, the fix is simple: just add the following line to `httpd.conf` file, or to a `.htaccess` file in the Web server's directory where the RDF files are located:

```
AddType application/rdf+xml.rdf
```

That is it. And since you are on it, you might as well go ahead and add the following two lines to make sure your Web server can recognize two more RDF syntaxes, i.e., N3 and Turtle:

```
AddType text/rdf+n3;charset=utf-8.n3
AddType application/x-turtle.ttl
```

Now, when it comes to configuring your Web server to implement 303 redirect and furthermore content negotiation, it is unfortunately not all that easy. This process depends heavily on your particular Web server and its local configuration; it is some times quite common that you may not even have the access rights that are needed to make the configuration changes. Therefore, we will not cover this in detail, but remember, it is one step that is needed to publish Linked Data on the Web and it is not hard at all if you have the full access to your server.

With all these said, let us take a look at one example showing how to publish Linked Data on the Web.

11.2.4.2 Example: Publishing Linked Data on the Web

A good starting point is to publish our own FOAF files as Linked Data on the Web. Let us start with my own FOAF file. To satisfy the minimal requirements that we have discussed above, we can follow these steps:

Step 1. Check whether our Web server is configured to return the correct MIME type when serving `rdf/xml` files.

Let us assume this step has been done correctly or you can always follow the previous discussion to make sure your Web server is configured properly.

Step 2. Since we are not going to configure our Web server to implement 303 redirect and content negotiation, we decide to use Hash URI to identify myself:

```
http://www.liyangyu.com/foaf.rdf#liyang
```

Again, when a client attempts to dereference this URI, the hash fragment (`#liyang`) will be taken off by the client before it sends the URI to the server. The resulting URI is therefore given by the following:

```
http://www.liyangyu.com/foaf.rdf
```

Now, all we need to do is to make sure that we put the RDF file, `foaf.rdf`, at the right location on the server, so a client submitting the above URI will be able to look into the response and find the RDF file successfully.

In this example, `foaf.rdf` file should be located in the root directory on our server, and it could look like something as shown in List 11.5.

List 11.5 My own FOAF document

```
1: <?xml version="1.0" encoding="UTF-8"?>
2: <rdf:RDF
3:   xml:lang="en"
4:   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
5:   xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
6:   xmlns:foaf="http://xmlns.com/foaf/0.1/">
7:
8: <rdf:Description
9:   rdf:about="http://www.liyangyu.com/foaf.rdf#liyang">
10:   <foaf:name>liyang yu</foaf:name>
11:   <foaf:title>Dr</foaf:title>
12:   <foaf:givenname>liyang</foaf:givenname>
13:   <foaf:family_name>yu</foaf:family_name>
14:   <foaf:mbox_sha1sum>1613a9c3ec8b18271a8fe1f79537a7b08803d896
15:   </foaf:mbox_sha1sum>
16:   <foaf:homepage rdf:resource="http://www.liyangyu.com"/>
17:   <foaf:workplaceHomepage
18:     rdf:resource="http://www.delta.com"/>
19:   <rdf:type rdf:resource="http://xmlns.com/foaf/0.1/Person"/>
20: </rdf:Description>
21: </rdf:RDF>
```

Step 3. Make sure you have outbound links.

We can add some outbound links to the existing linked datasets. As shown in List 11.6, properties `<foaf:knows>` (lines 18–24) and `<foaf:topic_interest>` (line 26) are used to add two outbound links. This will ensure any client visiting my FOAF document can continue its journey into the Linked Data cloud.

List 11.6 My FOAF document with outbound links

```

1: <?xml version="1.0" encoding="UTF-8"?>
2: <rdf:RDF
3:     xml:lang="en"
4:     xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
5:     xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
6:     xmlns:foaf="http://xmlns.com/foaf/0.1/">
7:
8: <rdf:Description
9:     rdf:about="http://www.liyangyu.com/foaf.rdf#liyang">
10:    <foaf:name>liyang yu</foaf:name>
11:    <foaf:title>Dr</foaf:title>
12:    <foaf:givenname>liyang</foaf:givenname>
13:    <foaf:family_name>yu</foaf:family_name>
14:    <foaf:mbox_sha1sum>1613a9c3ec8b18271a8fe1f79537a7b08803d896
15:    <foaf:homepage rdf:resource="http://www.liyangyu.com"/>
16:    <foaf:workplaceHomepage
17:        rdf:resource="http://www.delta.com"/>
18:    <rdf:type rdf:resource="http://xmlns.com/foaf/0.1/Person"/>
19:
20:    <foaf:knows>
21:        <!-- the following is for testing purpose -->
22:        <foaf:Person>
23:            <foaf:mbox
24:                rdf:resource="mailto:libby.miller@bristol.ac.uk"/>
25:            <foaf:homepage
26:                rdf:resource="http://www.ilrt.bris.ac.uk/~ecemm/">
27:        </foaf:Person>
28:    </foaf:knows>
29:
30:    <foaf:topic_interest
31:        rdf:resource="http://dbpedia.org/resource/Semantic_Web"/>
32:
33: </rdf:Description>
34: </rdf:RDF>

```

Step 4. Make sure you have inbound links.

This step is to make sure my FOAF document can be discovered by the outside world. The details about this can be found in [Chap. 7](#), refer back to that chapter if you need to.

After these four steps, we are ready to upload my FOAF document onto the server at the right location and claim success. However, for those curious minds, how do

we know it is published as Linked Data correctly based on the given standards? Is there a way to check this?

The answer is yes, and let us now take a look at how to make sure we have done everything correctly.

11.2.4.3 Make Sure You Have Done It Right

Just as we have validators for checking RDF documents including RDF instance files and OWL ontologies, we also have Linked Data validator which can be used to confirm whether some structured data are published correctly as Linked Data, based on the current best practices as we have been discussing in this chapter.

Here is one such tool you can use. It is called Vapour and you can access this service from this location:

`http://vapour.sourceforge.net/`

and Fig. 11.8 shows this page.

Let us again use my own FOAF document as shown in List 11.6 as the example, and we will validate whether this FOAF document is published correctly as Linked

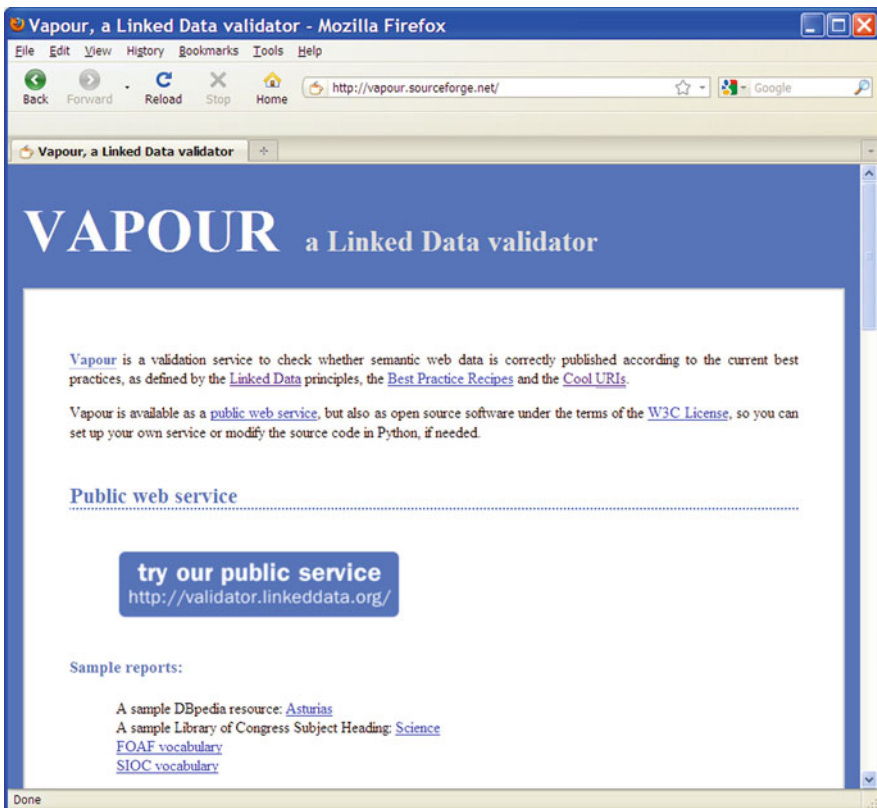


Fig. 11.8 Vapour: a Linked Data validator

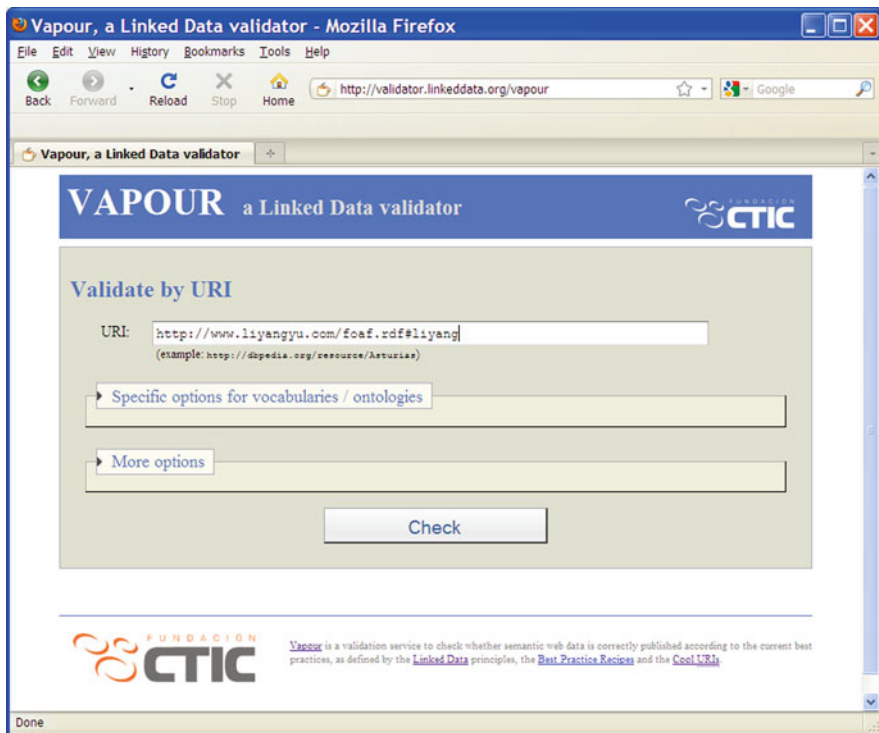


Fig. 11.9 Check my own URI to make sure it is published correctly as Linked Data

Data. To do so, click try our public service link in Fig. 11.8 and enter the following URI as shown in Fig. 11.9:

`http://www.liyangyu.com/foaf.rdf#liyang`

Once we click Check button, we get the result as shown in Fig. 11.10. Clearly, all tests are passed, meaning that my FOAF document is indeed published correctly as Linked Data.

You can also see more details on the same page if you try this test out, including dereferencing resource URI with and without content negotiation. As a summary, it is always a good idea to use a validation service when you publish Linked Data on the Web, to make sure your data will participate in the loop successfully.

11.3 The Consumption of Linked Data

Now that we understand how the Web of Linked Data is built, the next step is to study what to do with it. In general, this involves discovering Linked Data, accessing Linked Data, and building applications that run on top of the Web of Linked Data, as summarized below:



Fig. 11.10 Validating result from Fig. 11.9

- Discovery of Linked Data

For a given resource in the world, for example, a city or a tennis player, how do we know this resource has already been a subject of Linked Data? Is there any Linked Data search engine that crawls the Web of Linked Data by following links between data sources, and therefore provides answers to our questions?

- Accessing Linked Data

We use Web browsers to access our current Web, the Web of documents. For the Web of Linked Data, do we have similar Linked Data browsers that we can use to

access the Web of Linked Data? If we have indeed discovered some Linked Data that we are interested in, how can we start from there? And by using Linked Data browser, can we start browsing in one data source and then navigate along links into related data sources?

- Applications built upon Linked Data

Given the fact that the Web of Linked Data is built for machine to read and understand, we can go beyond discovery and accessing the Web of Linked Data and create new applications built upon the Web of Linked Data. Compared to Web 2.0 mashups, Linked Data applications offer much more flexibility and completeness in their operations, as we will see later in this chapter.

11.3.1 Discover Specific Target on the Linked Data Web

In the world of traditional hypertext Web, discovery almost exclusively means using one of the major search engines to find the information you are interested in. Search engines are therefore the places where the navigation process begins.

For the Web of Linked Data, the same is true: we need search engines that can work on the Web of Linked Data and therefore can provide us with a tool to make our discovery.

It will not be too surprising if you are seeing a different look-and-feel from the search engines that work on the Web of Linked Data. After all, the Web of Linked Data is quite different from our traditional Web of documents. In fact, Semantic Web search engines are mostly geared toward the needs of applications, not that of human eyes. Nevertheless, some researchers and developers have designed search engines that have a similar look-and-feel as the traditional search engines, and this breed of search engines can be very useful to at least some user groups.

In this section, we will cover both these types, with the goal of discovering Linked Data on the Web. To make things easier, we will start from those Semantic Web search engines that look familiar to our human eyes.

11.3.1.1 Semantic Web Search Engine for Human Eyes

First off, remember that these kinds of search engines are Semantic Web search engines from their roots. Instead of crawling the Web of document and indexing each document, these search engines crawl the Web of Linked Data by following their RDF links, and prepare their indexations based on the Web of Linked Data.

Falcons is a good example of this type of search engine. Falcons represents “Finding, Aligning and Learning ontologies, ultimately for Capturing knowledge via ONtology-driven approcheS,” and it is developed by the Institute of Web Science (IWS), Southeast University of China. You can access it from the following link:

<http://iws.seu.edu.cn/services/falcons/>

The first thing to note about Falcons is that it provides a keyword-based search service, i.e., the user is presented with a search box, where keywords related to the topics in mind can be entered. Falcons then reacts by returning a list of results that may be related to the topic. Clearly, this closely mimics the same look-and-feel offered by current market leaders such as Google and Yahoo!.

Let us say we want to discover if there is any Linked Data about tennis player Roger Federer. Obviously, if Roger Federer is indeed mentioned in the Web of Linked Data, he has to be some instance of a given class. For example, he could be an instance of some class such as `Person` defined in some ontology. With this in mind, we should use the `Object` search in Falcons, and enter Roger Federer in the search box. This will tell Falcons that the results we are searching for has to contain “Roger Federer” as keywords and should be coming from some instance data, not class or type definitions.

Once we submit this query, Falcons responses by returning a list of results. When presenting the results, for each object (instance data), Falcons shows its title, label, comment, image, page, type, and URI, if applicable. Clearly, for the Web of Linked Data, type and URI are all important since type identifies the class of this instance data, and URI uniquely identifies the instance. For our example, the first result is a good hit: the URI is given by

http://dbpedia.org/resource/Roger_Federer

and its type is `Person`. Clearly, the above URI comes from DBpedia, and we know already that DBpedia is a key component of Linked Data. Therefore, just based on the very first result, we know we have discovered some Linked Data for tennis player Roger Federer, which can be a good start point for whatever we plan to do next.

Note that Falcons also provides a `Type` pane together with the search result, and this is in fact a very useful feature. Recall that when we first started our search for any Linked Data related to Roger Federer, we can only say that if this data exists, it has to be some instance data of some class type. However, we don’t really know what exactly this class type is, except that the correct type should be something like `Person` or `Athlete`.

Now, the `Type` pane on the result page shows all the types that are found in the results. Note that the initial search will focus on “Any type”, therefore it does not put any further constraints on the type at all. Once we have the initial results back, we can further narrow down the type by clicking a specific type in the `Type` pane. For example, we can click `Person` in the `Type` pane, telling Falcons that we believe Roger Federer should be an instance of some `Person` class. Once we do this, all the sub-classes of type `Person` are now summarized in `Type` pane, and you can continue to narrow down your search. Therefore, `Object` search can be guided by recommended concepts, and we can further refine search results by selecting object types.

Besides `Object` search tab as we have discussed above, Falcons provides two more search tabs: `Concept` and `Document`. `Concept` search is not much related to discovering Linked Data on the Web, it is more suited for locating classes and

properties defined in ontologies that are published on the Web. It is quite useful if you want to find classes and properties that you can reuse, instead of inventing them again.

Document search gives you a more traditional search engine experience, especially the look-and-feel of the search results. If you search for Roger Federer, any RDF document that contains these words will be returned as part of the result list, be these search items in the instance data or the class or property definitions. Although not quite efficient, this search can also be used to discover Linked Data on the Web.

11.3.1.2 Semantic Web Search Engine for Applications

We have made use of Sindice search engine in previous sections, with the goal of discovering if there is any URI existing for some real-world object that we would like to talk about. For example, if we want to say something about Roger Federer, we can use Sindice to discover the URI for him, as shown in Figs. 11.3 and 11.4.

Sindice search engine therefore can also be used as a tool to discover Linked Data on the Web. When used by human users, it has a similar look-and-feel as Falcons does: a certain number of keywords can be provided to Sindice, and Sindice will return RDF documents on the Web which contains these keywords.

Furthermore, if you know the URI for some real-world object, you can search it in Sindice, and Sindice will return all the RDF documents on the Web which contains this given URI. For example, Fig. 11.11 shows the result from Sindice when we search for the URI of Roger Federer.

Clearly, these RDF documents are all linked to some extent since they all have the URI of Roger Federer in their triples.

By far, Sindice feels much like Falcons. As human users, we can use both to discover Linked Data on the Web. However, there is more to Sindice: it can be used by applications as well.

We have not yet discussed Linked Data applications at this point. However, it is quite intuitive to realize that the first thing each Linked Data application will have to do is to somehow harvest some Linked Data before it can do anything interesting with the data. As a result, each Linked Data application will have to implement its own crawling and indexing component, just to find the interested Linked Data. Clearly, moving this common infrastructure for crawling and indexing the Web of Linked Data to a search engine that each individual Linked Data application can then use will be a much better and cleaner design.

This is the rationale behind the design and implementation of Sindice's Data Web Services API and also the reason why we claim Sindice can be a search engine used by applications. More specifically, each application, by using the Sindice API, can query Sindice's collection and receive a set of links that point to those potentially relevant RDF documents, which can then be processed by the application to create other interesting results.

At the time of this writing, Sindice API is still in early beta version and is experiencing rapid changes and developments. Therefore, we are not going to show any

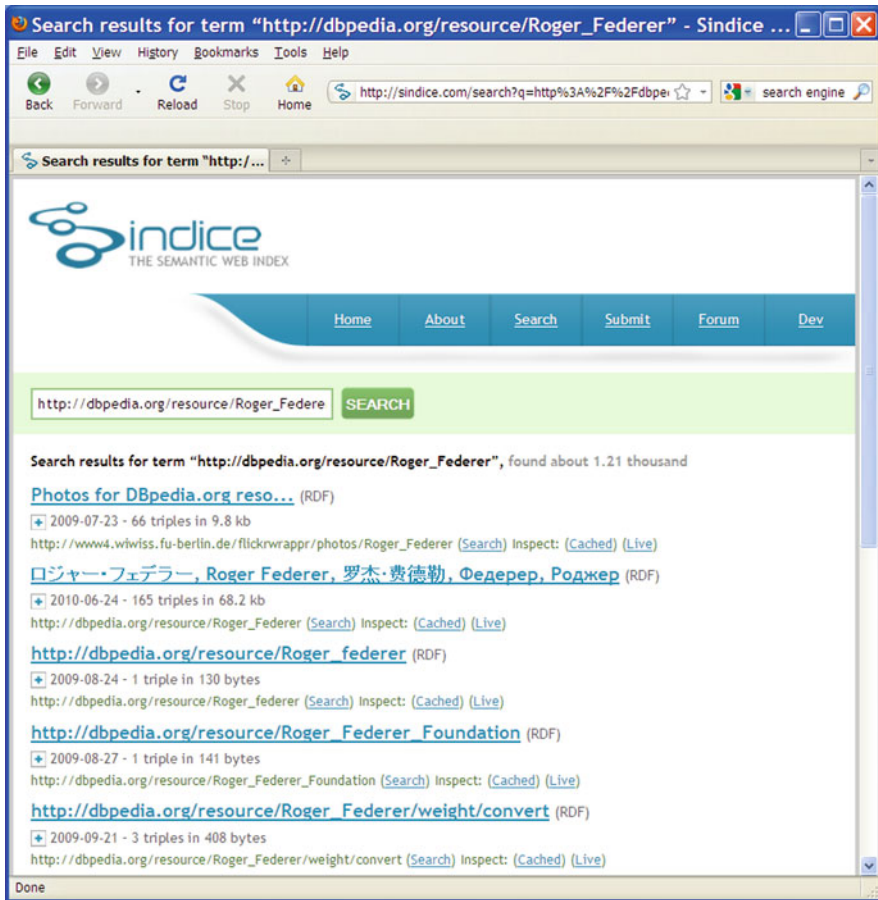


Fig. 11.11 Sindice results when searching for the URI of Roger Federer

concrete examples here, but the basic idea as we have discussed above will not change.

Before we conclude this section, let us very briefly discuss two more Semantic Web search engines, just to give you a flavor of other choices when it comes to discovering Linked Data on the Web:

- SWSE

SWSE (Semantic Web Search Engine) is developed by DERI Ireland and can provide search capabilities more suitable toward human users. It accepts keyword-based search and further offers access to its underlying data store via SPARQL query language. At this point, you can access SWSE from this URL:

<http://swse.deri.org/>

Also, note that similar to Sindice, SWSE is more related to search for instance data, not types and properties.

- Swoogle

Swoogle is developed by UMBC Ebiquity Research group, which consists of faculty and students from the Department of Computer Science and Electrical Engineering (CSEE) of University of Maryland, Baltimore County (UMBC). Unlike Sindice or SWSE, Swoogle is designed to search ontologies that related to the concepts provided by its users. Swoogle also provides Web services to the public users, which can be used by applications that are built on top of the Linked Data Web. At this point, Swoogle can be accessed at this location:

`http://swoogle.umbc.edu/`

11.3.2 Accessing the Web of Linked Data

Accessing the Web of Linked Data has two different meanings. First, human users can access it in a way that is similar to what has been done in traditional Web of documents, i.e., Linked Data browsers can be used to manually navigate from one data source to another. Second, applications that are built to understand Linked Data can access the Linked Data Web and further accomplish different requirements from us. For example, the so-called Follow-Your-Nose method can be used by applications to browse the Web of Linked Data. In this section, we will take a closer look at both these methods.

11.3.2.1 Using a Linked Data Browser

As human users, we can access the Web of Linked Data manually. This normally requires us to discover a specific piece of Linked Data on the Web first, which is then used as the starting point for further navigation on the Web of Linked Data.

As traditional Web browsers allow us to access the Web by following hyper-text links, their counterparts in the Web of Linked Data, the so-called Linked Data browsers, allow us to navigate from the starting data source to the next data source. This process can go on by following links that are expressed and coded as RDF triples, and that is all there is to it when it comes to manually accessing the Web of Linked Data.

Therefore, the key component in this process is the Linked Data browser. To learn how to manually access the Linked Data Web is to learn how to use one of these browsers.

In recent years, there are quite a few browsers that have been developed and deployed for public use. To find a list of these browsers, you can visit the W3C's ESW Wiki page,⁶ which is also updated quite frequently.

⁶<http://esw.w3.org/topic/TaskForces/CommunityProjects/LinkingOpenData/SemWebClients>

In this section, we will take the Sig.ma browser as an example to show you how Linked Data browsers can be used to access the Web of Linked Data.

Sig.ma is built on top of Sindice, which provides the data needs for Sig.ma. To some extent, Sig.ma acts much like Sindice's front-end GUI. You can access Sig.ma at this location:

`http://sig.ma/`

And its main page is shown in Fig. 11.12.

To start using it, simply enter the keywords you want to search in the input box and hit the SEARCH button, quite like using Sindice as a search engine. Obviously, this step is to find some entry point to access the Web of Linked Data.

Once you hit the SEARCH button, Sig.ma starts its work by doing the following steps:

1. select 20 data sources from the Web of Linked Data based on the keywords you have entered;
2. aggregate the information contained in these data sources; and
3. present the aggregated information and the data sources back to the user.

Note the first step is accomplished by using the underlying Sindice search engine to carry out a keyword-based search in the Web of Linked Data to select the relevant

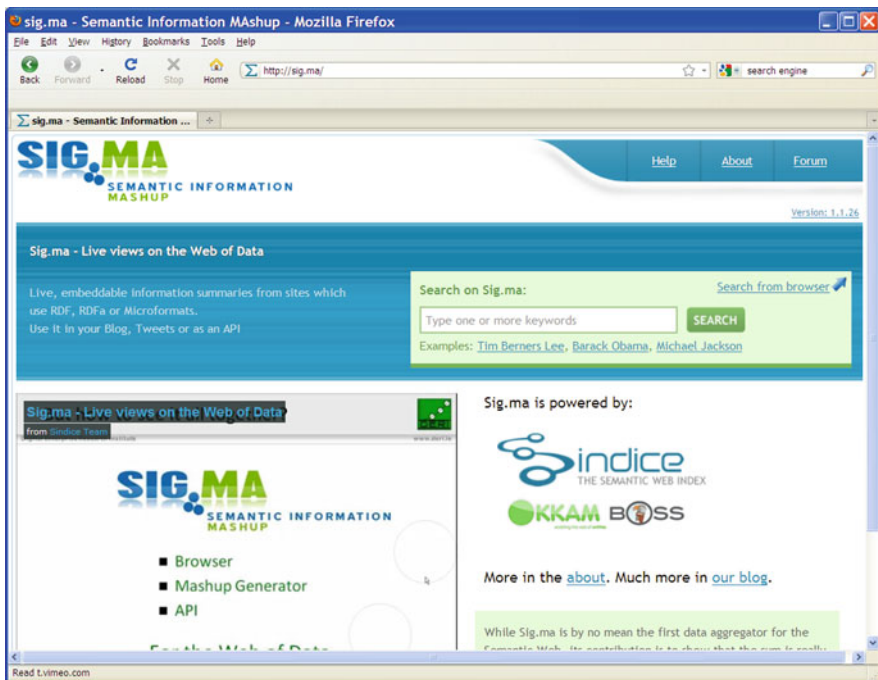


Fig. 11.12 Sig.ma: a Linked Data Web browser

data sources. If there are less than 20 data sources that are considered to be relevant to the given keywords, then whatever that are available will be included in the result set. If there are more than 20 data sources that are relevant, you can add the other data sources later on, as will be discussed soon.

Aggregating over the selected data sources essentially means to collect everything each data source says about the resource or concept represented by the keywords you have provided. And since the data sources are all taken from the Web of Linked Data, aggregating different data sources can be done easily.

Once the first two steps are done, Sig.ma presents the results back to the user by dividing the screen into the left pane and the right pane. The left pane shows the aggregated information, which is called the “sigma” for this search. The right pane shows the data sources based on which the sigma is obtained.

Let us use one example to see how it works. This time, instead of searching for Roger Federer, I will search for myself. The reason being that if we were to search Roger Federer, there would be too many datasets to be included. To show you how to use Sig.ma, a search that does not yield too many results is better.

Now, enter the keywords “liyang yu” in the search box (remember to include them in a pair of double quotes). Once you hit the SEARCH button, you will be presented with the result page as shown in Fig. 11.13 (notice if you are trying it, it is likely that what you see is not the same as what we printed here, and the reason is obvious).

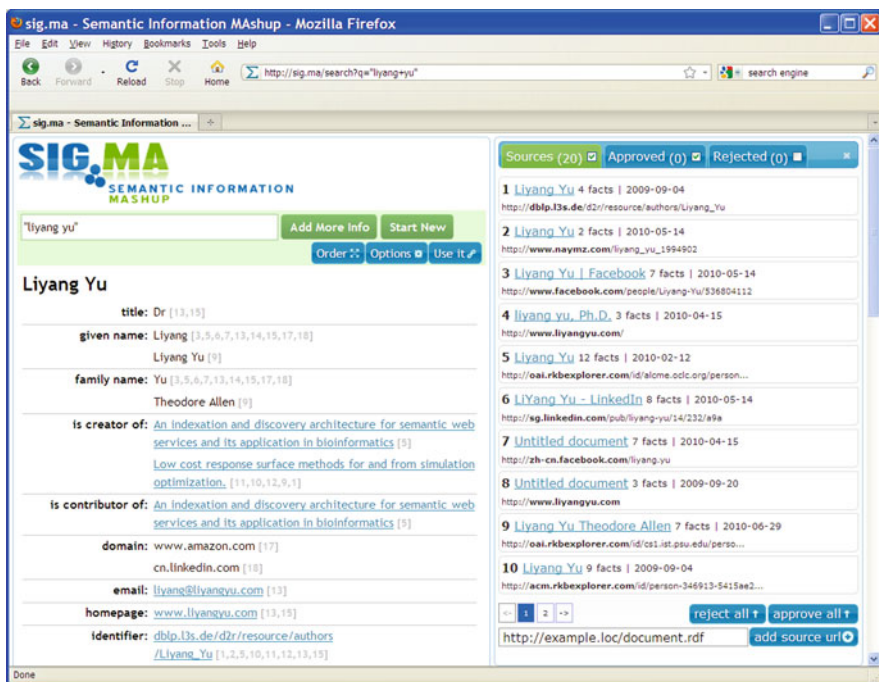


Fig. 11.13 Using Sig.ma to search for “liyang yu”

Let us first take a look at the left pane, i.e., the sigma of this search. It is quite different from what you would have seen if you had used a Google search. More specifically, Google search simply gives you back a list of links that point to a collection of Web pages, with each one of them containing the keyword “liyong yu.” Google itself is not able to tell you the fact such as, on a given Web page, the word Liyong shows up as a given name, and on some other page, the word Yu shows up as a family name, so on and so forth.

For Sig.ma, however, this is not difficult at all. It knows not only that the word Liyong is a given name and the word Yu is a family name, but quite a lot more. For example, it can tell the string `liyong910@yahoo.com` represents my e-mail address. Clearly, since Sig.ma is built upon a Web of Linked Data, it is therefore able to present the result in a way that seems like the machine is able to understand all the data sources it has encountered during its search.

Now since the sigma pane shows the result of data aggregation, it is certainly useful to include the data sources that have been used to obtain the current sigma. This is the right pane, as shown in the Fig. 11.13. For discussion purpose, let us call it the source pane.

It is very easy for Sig.ma to trace the data source for each information segment in the sigma. For example, if you hover your mouse over the given name “Liyong”, at least four data sources in the source pane will be highlighted, telling us that this information is included in all these four data sources, as shown in Fig. 11.14.

In fact, each data source in the source pane has a number associated with it, indicating how many facts are collected from this particular data source. For example, data source number 4

`http://www.liyongyu.com/`

has contributed three facts to the current sigma, as shown in Fig. 11.13. To see a detailed list of these three facts, hover your mouse over this document; the facts from this document will be highlighted in the sigma pane, as shown in Fig. 11.15.

Note that there is a pop-up menu showing up when you hover the mouse over data source number 4 (see Fig. 11.15). The first selection in this pop-up menu is called `solo`, which is a very useful tool: if you click `solo`, the current sigma will show only the facts that are collected from this data source, as seen in Fig. 11.16.

To go back to the complete list of facts, simply click `unsolo`, as you can easily tell.

Another useful feature of Sig.ma is the ability to approve and reject data sources. Recall the fact that search in Sig.ma is based on keyword matching, i.e., when it scans a given RDF data source, it looks for the keywords in that document. The keywords themselves can appear in a comment, a label, or the string value of a given subject. They can also show up in a URI that identifies a subject, a predicate, or an object. As the developers of Sig.ma have pointed out, since very simple strategies have been on purpose chosen at this stage to filter data source candidate, it is quite possible that a given data source is in fact not the data source you are looking for.

In the case where a given data source should not be included in the sigma, you can simply click the `reject` button from the pop-up menu in the source pane (you need to hover the mouse over the selected source data document). Once this is done, all

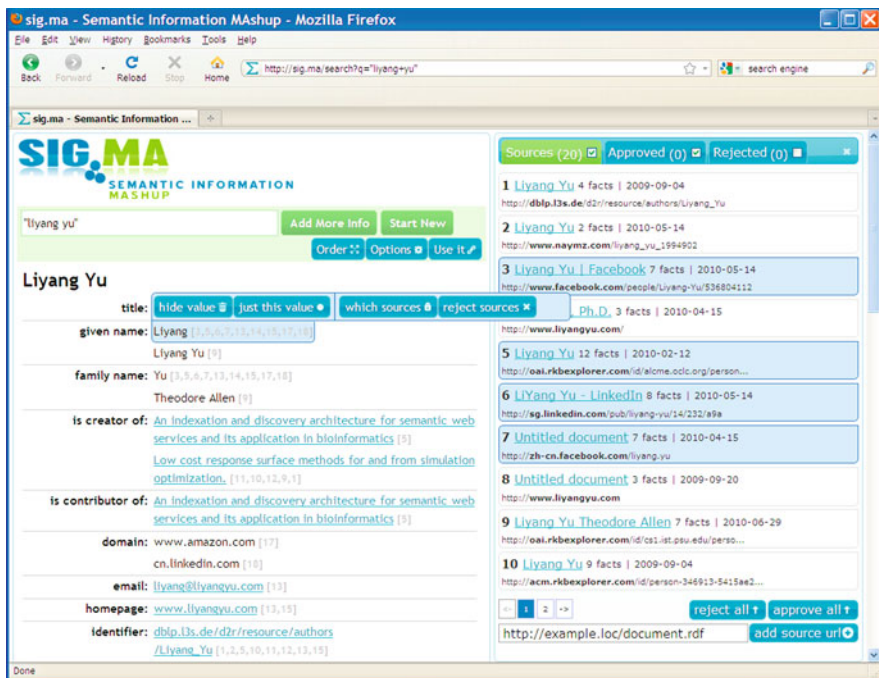


Fig. 11.14 Hovering the mouse over Liyang will show the data sources from where this information is obtained

the facts in the current sigma will be removed and the data source will be removed as well. For example, go back to Fig. 11.13; we can reject document number 4 by easily following these steps.

It is now easy to understand the approve selection. Clicking approve for a given data resource means that this data source is highly relevant to the search and should stay in the source pane at all times.

Besides rejecting and/or approving the data source files contained in the current source pane, I can click Add More Info button in the sigma pane to ask Sig.ma to search more datasets. Once more data sources have been added, I can start to filter them again by applying the same rejecting/approving process until all the data sources are stable in the source pane.

Figure 11.17 shows my final sigma. As you can tell, I have rejected altogether 12 data sources to reach this sigma.

In fact, my final sigma presents a set of entry points to the Web of Linked Data, since each one of the corresponding data sources in the source pane contains links to the Web of Linked Data. It is now time to start our navigation by following these links.

Let us get back to Fig. 11.17. Note the label `topic interest:` and its value `Semantic Web`. Clicking this link brings us to a brand new sigma as shown in Fig. 11.18.

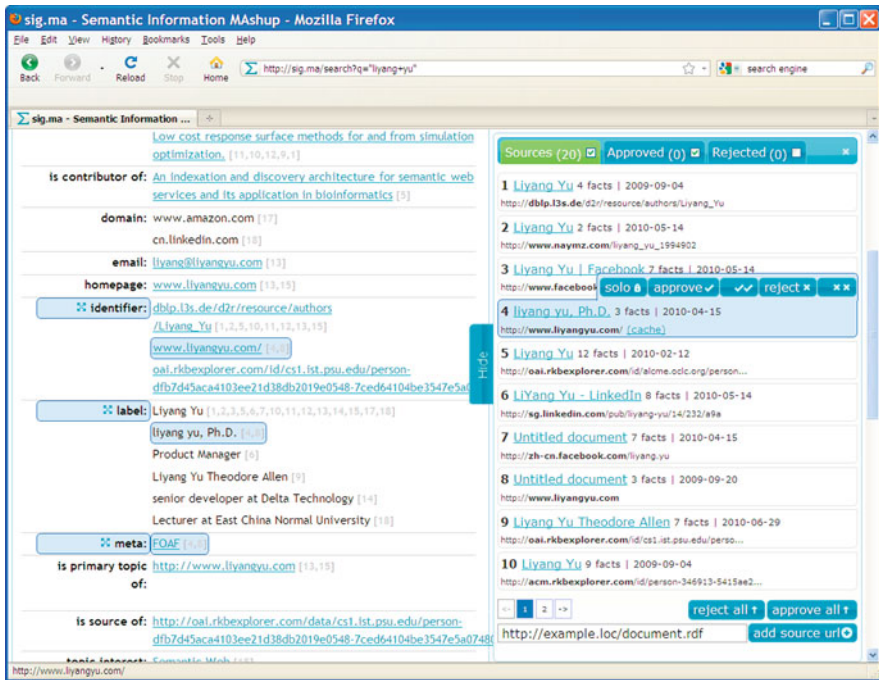


Fig. 11.15 Hover mouse over a data source document, all the facts from this document will be highlighted

As you can tell, the sigma about Semantic Web opens another new entry to the Linked Data Web for you to explore. For example, you can do the following:

- you can start to explore Jena Semantic Web Framework;
- you can start to read more about Resource Description Framework (RDF);
- you can start to understand OWL;
- and more.

I will leave this to you to continue, and Sig.ma is an excellent tool to search and access the Web of Linked Data.

Finally, note that we have only covered some basic functionality provided by Sig.ma. Sig.ma was first released on 22 July 2009, and given the fact that it is experiencing constant development and improvement, at the time when you are reading this book, you will likely see a different version of Sig.ma. However, the basics should remain the same.

11.3.2.2 Using SPARQL Endpoints

We have discussed how to use Linked Data browsers to access the Web of Linked Data manually in the previous section. While it is a useful way to explore the Linked

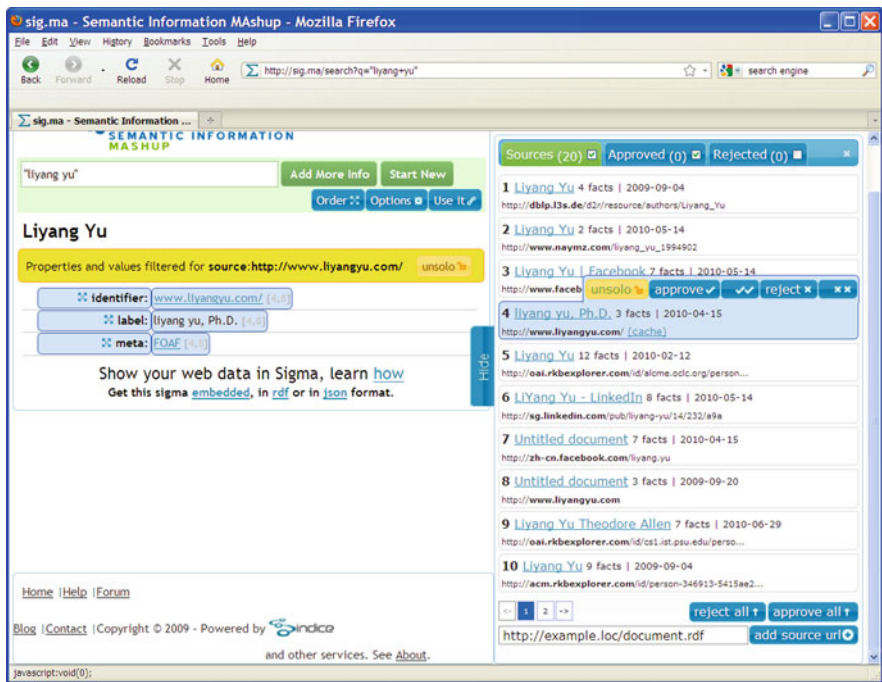


Fig. 11.16 Click solo will show only the facts collected from the selected data source in the sigma pane

Data, it does not take full advantage of the fact that the Web of Linked Data contains structured data, which means that we can actually access the Linked Data by using a query language.

In this section, we will use SPARQL to access the Web of Linked Data. In fact, this is not something completely new to us. In Chap. 10, we have discussed how to use SPARQL to access DBpedia. We will expand the same idea and show you how to use SPARQL to access the Web of Linked Data in general.

A good start point is the current Linked Data cloud presented in Fig. 11.7. Again, the benefit of accessing it online is that you can get the version that is clickable: when you click a dataset, it directly takes you to the home site of that dataset.

Now, let us say that we want to understand more about Musicbrainz, which at this point we know nothing about. Clicking this dataset takes us to the home site of Musicbrainz. On its home site, we can find the following SPARQL endpoint (note that not all the datasets provide SPARQL endpoints):

```
http://dbtune.org/musicbrainz/snorql/
```

and opening this endpoint will give us the query interface supported by the dataset.

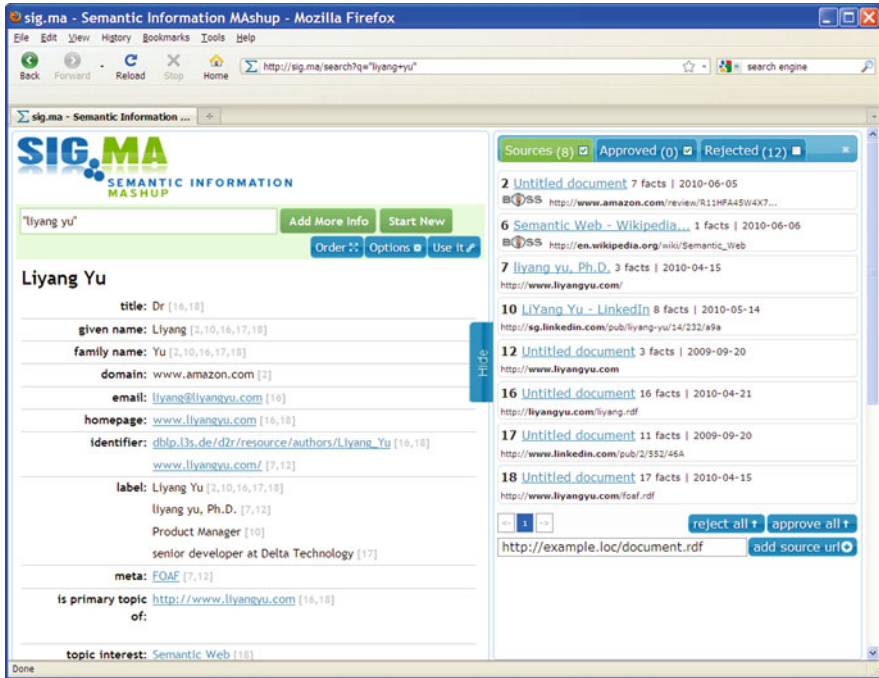


Fig. 11.17 My final sigma after rejecting 12 data sources

Now, to explore this dataset, or rather, to explore any given dataset, we can always start from two general queries. The first query is given below:

```
SELECT DISTINCT ?concept
WHERE
{ [] a ?concept }
```

This shows all classes that are used in a given dataset. Note that there might be a large number of classes used, and some of them might look unfamiliar to you. However, this query can give you some feeling about what the given dataset is all about.

Let us try this query on the Musicbrainz dataset. Enter the above query in the query box, but change the query so it looks like this:

```
SELECT DISTINCT ?concept
WHERE
{ [] a ?concept }
LIMIT 10
```

Adding LIMIT 10 is to make sure the query can be executed in a reasonable amount of time. You can change it to another integer number if you prefer, such as 20.

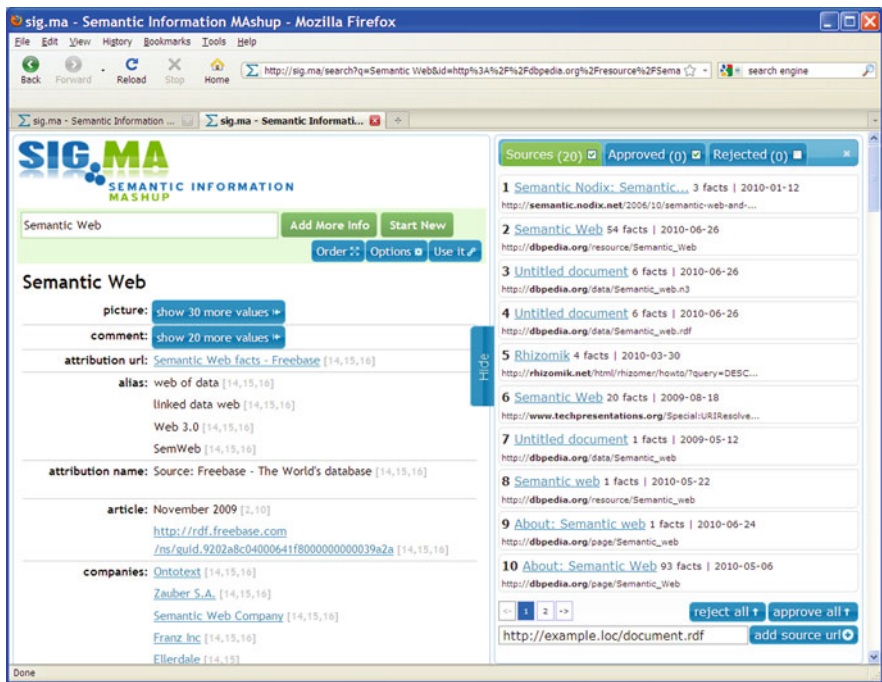


Fig. 11.18 Clicking Semantic Web from Fig. 11.17 will take us to this new sigma

Once you submit the query, you should get some results back. For example, part of the classes I got is shown as follows:

```

bio:Birth
bio:Death
db:vocab/puidjoin
db:vocab/l_label_track
db:vocab/lt_artist_label
db:vocab/lt_artist_artist
lingvoj:LinguisticSystem
mo:MusicArtist
mo:Performance
mo:Release
mo:Record

```

Again, by the time you are reading this book, you could get different results back.

Now, the above class list will give us some basic idea about what is covered in this dataset. For example, this dataset is more about some music artists, their albums, their performance, so on and so forth.

The second useful query is similar to the first one. It asks all the properties that are included in a given dataset:

```
SELECT DISTINCT ?property
WHERE
{?sub ?property ?obj}
```

Again, you might want to use it together with `LIMIT 10` constraint, just to make sure the performance of the endpoint is acceptable:

```
SELECT DISTINCT ?property
WHERE
{?sub ?property ?obj}
LIMIT 10
```

The following is the result:

```
rdfs:label
db:vocab/puidjoin_puid
db:vocab/puidjoin_usecount
db:vocab/puidjoin_id
db:vocab/puidjoin_track
rdf:type
db:vocab/l_label_track_enddate
db:vocab/l_label_track_link_type
db:vocab/l_label_track_begindate
db:vocab/l_label_track_modpending
```

As you can tell, the above two queries are very useful when you know nothing about the dataset. In fact, some SPARQL endpoints have included these two queries for you as your default starting point.

After these two general queries, it is up to you to continue your exploration. In most cases, what you will be doing depends on the results from these two queries. For example, for the Musicbrainz dataset, I am interested in `mo:MusicArtist` class, and I want to find who is a member of this class. To do so, I will use the following query:

```
SELECT ?artist
WHERE
{?artist a <http://purl.org/ontology/mo/MusicArtist> }
LIMIT 10
```

And I got 10 instances of `mo:MusicArtist` back. One of them is the following:

```
db:artist/0002260a-b298-48cc-9895-52c9425796b7
```

To know more about this instance, I continue to execute the following query:

```
SELECT ?property ?hasValue ?isValueOf
WHERE {
  { <http://dbtune.org/musicbrainz/resource/artist/
    0002260a-b298-48cc-9895-52c9425796b7> ?property ?hasValue }
  UNION
  { ?isValueOf ?property
    <http://dbtune.org/musicbrainz/resource/artist/
    0002260a-b298-48cc-9895-52c9425796b7> }
}
```

This query will find everything that has been said about this artist. Once you execute the query, you will get the name of the artist, the label, etc. Obviously, we can continue like this by following a number of different directions, and at some point, we will find ourselves moving on to explore other datasets.

The point is clear: besides using Semantic Web browsers or Semantic Web search engines to access the Web of Linked Data, it is also very useful and efficient to access it by using SPARQL queries. After all, search engines will point you to a set of documents that might contain the answer, but SPARQL queries can directly give you the answer you need.

11.3.2.3 Accessing the Linked Data Web Programmatically

The most significant difference between our current Web and the Web of Linked Data is the fact that the Web of Linked Data is processable by machine. Given this, it is certainly possible to access the Web of Linked Data programmatically, and it has already been the backbone of many Linked Data applications (as we will see in the next section).

Different applications may implement different ways of accessing the Web of Linked Data. However, two basic methods of accessing the Linked Data Web should be understood: one is referred to as Follow-Your-Nose method, the other is about issuing SPARQL queries within your application by using supporting tools.

The best way to learn these two methods is by going through some examples. Since these two methods are quite generic, they are discussed in [Chap. 14](#); you can find working example for each method there. For now, we will move on with the discussion of Linked Data applications.

11.4 Linked Data Application

Discovering and accessing Linked Data is only the first step, our ultimate goal is to build applications that make use of Linked Data. In this section, we present one popular example to show you how Linked Data can be used.

11.4.1 Linked Data Application Example: Revyu

11.4.1.1 Revyu: An Overview

Revyu is a Web site that everyone can login to review and rate anything in the world. However, it is not just another review site; it is developed by using the Semantic Web technologies and standards, and by following Linked Data principles and best practices. More importantly, it also consumes Linked Data from the Web to enhance its user experience.

Revyu is implemented in PHP and runs on a regular Apache Web server. It can be accessed at this location:

<http://revyu.com/>

A registered user can review and rate things by filling out a Web form, which does not require any knowledge of the Semantic Web. Once finished, the user can submit this review form and the review will show up at the site.

This does not sound too much different from other review sites at all. However, lots of things will then happen inside Revyu. To understand all these, we first need to understand one fact: every review created in Revyu is also expressed as an RDF graph, besides its normal look-and-feel on the Web.

Let us look at one example. From Revyu home page, click `Search Things` link to search for the movie *Broken Flowers*, for which a review has been created as an example by Tom Heath, the creator of Revyu. Figure 11.19 shows the review page of the movie *Broken Flowers*.

On this page, click the link which identifies the reviewer. In this case, the link reads as `by tom on 30 Jan 2007`. Once you click this link, you will land on the page as shown in Fig. 11.20.

On the right side of the page, you will find a link called `RDF Metadata for this Review of Broken Flowers` (the right-hand side of Fig. 11.20). Clicking this link will take us to the RDF format of this review.

With the understanding that every review in Revyu has its RDF representation, let us now take a look at what will happen inside Revyu when a review is submitted by a user.

- All things represented in Revyu are assigned with URIs.

At the moment a review is submitted, the reviewer (i.e., the user), the review the user creates, and the resource being reviewed are all assigned with URIs. Also, the tags used when reviewing the resource are assigned with URIs as well (will discuss tags later this section).

Note that Revyu is designed to follow the four basic principles of Linked Data discussed early in this chapter. To see this, we can open the RDF document which represents the review of *Broken Flowers* and locate the URI Revyu has assigned to Tom Heath:

<http://revyu.com/people/tom>

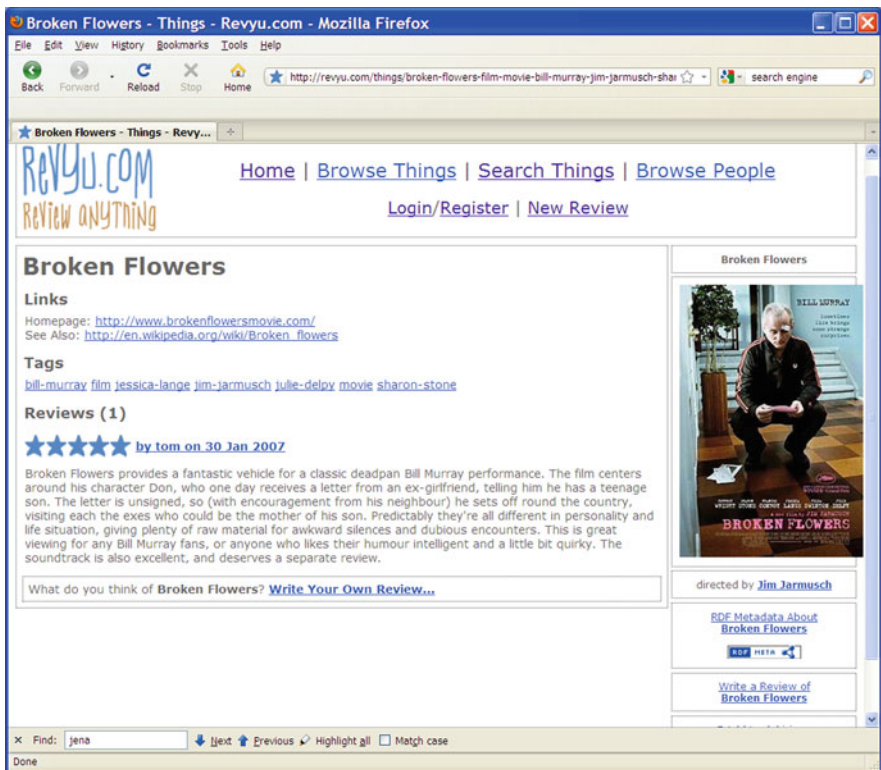


Fig. 11.19 Review page for the movie Broken Flowers

Since this URI represents a non-information resource, if it is dereferenced, our Internet browser should receive a HTTP 303 See Other response. Furthermore, our browser should also receive a URL pointing to a document that describes the resource, in this case, Tom Heath.

To test this, let us paste the above URI into our Web browser, and we will be taken to another URL given as below:

```
http://revyu.com/people/tom/about/html
```

which contains a HTML description about Tom.

In fact, content negotiation is also supported by Revyu: if a user agent asks for HTML format, it will receive a HTML document located at the above URL, and if it asks for an RDF format, it will receive an RDF description located at this URL:

```
http://revyu.com/people/tom/about/rdf
```

- Tags are used to create links to the datasets on the Web of Linked Data.

Obviously, the collection of reviewed items is at the center of any review site. As we have discussed, every item in this collection has been assigned a URI by

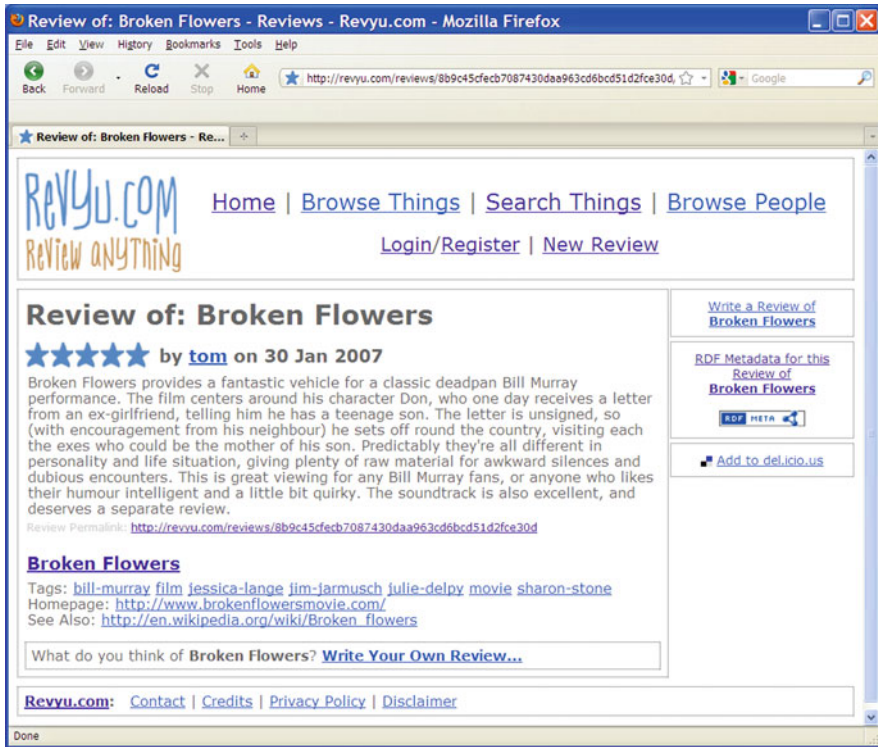


Fig. 11.20 Reviewer page of tom

Revyu. However, an isolated URI will not be of much value unless one of the following two (or both) can happen:

- it is associated with another resource URI contained in another dataset, by using `owl:sameAs` or `rdfs:seeAlso` property;
- it is associated with a type information (a class defined in an ontology), so some application can perform reasoning on this URI.

Clearly, asking the user of Revyu to accomplish either one of these conditions is not feasible: not only has the user to understand the Semantic Web technologies and standards, but also there has to be ontologies readily available which can provide sufficient coverage to any arbitrary item that may receive a review.

The solution taken by the Revyu designers is to use tags. In particular, it is up to the user to associate keyword tags to the item being reviewed. With this tag information, Revyu is then responsible for deriving type information and linking the item to a certain resource described by another dataset.

Currently two domains are covered by Revyu: books and films. More specifically, when Revyu recognizes that a new item is tagged as book, it will examine every Web link provided by the reviewer at the time the review is submitted. For example, the

reviewer may have provided a link from Amazon which contains some information about the book. When examining this link, Revyu parses the Web document downloaded from the link and attempts to extract an ISBN number embedded in the document. If Revyu can find an ISBN number, it will conclude that the reviewed item is indeed a book and will assert a corresponding `rdf:type` statement in the generated RDF statements.

As one example, List 11.7 shows some generated RDF statements for the reviewed book titled *The Unwritten Rules of Ph.D. Research*. The reviewer has provided the related link from Amazon, which contains the ISBN number (line 23). Based on this information, line 26 has been added by Revyu to establish the type information for this item.

List 11.7 RDF statements generated by Revyu (a book review)

```

1: <?xml version="1.0" encoding="UTF-8" ?>
2: <rdf:RDF
3:   xml:base="http://revyu.com/"
4:   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
5:   xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
6:   xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
7:   xmlns:owl="http://www.w3.org/2002/07/owl#"
8:   xmlns:dc="http://purl.org/dc/elements/1.1/"
9:   xmlns:dcterms="http://purl.org/dc/terms/"
10:  xmlns:vcard="http://www.w3.org/2001/vcard-rdf/3.0#"
11:  xmlns:foaf="http://xmlns.com/foaf/0.1/"
12:  xmlns:rev="http://purl.org/stuff/rev#"
13:  xmlns:tag=
13a:    "http://www.holygoat.co.uk/owl/redwood/0.1/tags/"
14:  xmlns:ns1="http://www.hackcraft.net/bookrdf/vocab/0_1/"
15:
16: <rdf:Description
16a:   rdf:about="things/the-unwritten-rules-of-phd-research">
17:   <rev:hasReview rdf:resource=
17a:     "reviews/82825d6cec2a2267c541848397e1605ab0042af0"/>
18:   <tag:tag rdf:resource=
18a:     "taggings/82825d6cec2a2267c541848397e1605ab0042af0"/>
19: </rdf:Description>
20:
21: <owl:Thing rdf:about=
21a:   "things/the-unwritten-rules-of-phd-research">
22:   <rdfs:label>The Unwritten Rules of Phd Research, by Gordon
22a:     Rugg and Marian Petre </rdfs:label>
23:   <rdfs:seeAlso rdf:resource=
23a:     "http://www.amazon.co.uk/Unwritten-Rules-Phd-
23b:     Research/dp/0335213448/" />
24:   <foaf:homepage rdf:resource=
24a:     "http://mcgraw-hill.co.uk/openup/unwrittenrules/" />
25:   <owl:sameAs rdf:resource=
25a:     "http://www4.wiwiss.fu-berlin.de/bookmashup/
25b:     books/0335213448"/>

```

```

26:   <rdf:type rdf:resource=
26a:     "http://www.hackcraft.net/bookrdf/vocab/0_1/Book"/>
27: </owl:Thing>
28:
29: <rdf:Description rdf:about=
29a:   "taggings/82825d6cec2a2267c541848397e1605ab0042af0">
30: <rdfs:label>A bundle of Tags associated with this Thing, de
30a:   fining when they were added and by whom</rdfs:label>

31: </rdf:Description>
32:
33: </rdf:RDF>

```

If a reviewed item is tagged as movie or film, Revyu will issue a query against DBpedia's SPARQL endpoint with the goal of finding any instance data whose type is given by `yago:Film` and also has the same name as the reviewed item. If this is successful, Revyu will conclude that the reviewed item is indeed a movie, and an `rdf:type` statement will be generated. For example, once the review for movie *Broken Flowers* is submitted, Revyu is able to confirm that this item is the movie named *Broken Flowers*, and List 11.8 shows the RDF statements generated for the item.

List 11.8 RDF statements generated by Revyu (a movie review)

```

1: <?xml version="1.0" encoding="UTF-8" ?>
2: <rdf:RDF
3:   xml:base="http://revyu.com/"
4:   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
5:   xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
6:   xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
7:   xmlns:owl="http://www.w3.org/2002/07/owl#"
8:   xmlns:dc="http://purl.org/dc/elements/1.1/"
9:   xmlns:dcterms="http://purl.org/dc/terms/"
10:  xmlns:vcard="http://www.w3.org/2001/vcard-rdf/3.0#"
11:  xmlns:foaf="http://xmlns.com/foaf/0.1/"
12:  xmlns:rev="http://purl.org/stuff/rev#"
13:  xmlns:tag=
13a:    "http://www.holygoat.co.uk/owl/redwood/0.1/tags/"
14:  xmlns:ns1=
14a:    "http://www.csd.abdn.ac.uk/~ggrimnes/dev/imdb/IMDB#">
15:
16: <rdf:Description rdf:about=
16a:  "things/broken-flowers-film-movie-bill-murray-jim-jarmusch-
16b:  sharon">
17:   <rev:hasReview rdf:resource=
17a:     "reviews/8b9c45cfecb7087430daa963cd6bcd51d2f30d"/>
18:   <tag:tag rdf:resource=
18a:     "taggings/8b9c45cfecb7087430daa963cd6bcd51d2f30d"/>
19: </rdf:Description>
20:

```

```

21: <owl:Thing rdf:about=
21a:   "things/broken-flowers-film-movie-bill-murray-
21b:     jim-jarmusch-sharon">
22:   <rdfs:label>Broken Flowers</rdfs:label>
23:   <rdfs:seeAlso rdf:resource=
23a:     "http://en.wikipedia.org/wiki/Broken_flowers"/>
24:   <foaf:homepage rdf:resource=
24a:     "http://www.brokenflowersmovie.com/" />
25:
26:   <owl:sameAs rdf:resource=
26a:     "http://dbpedia.org/resource/Broken_Flowers"/>
27:   <rdf:type rdf:resource=
27a:     "http://www.csd.abdn.ac.uk/~ggrimnes/dev/imdb/IMDB#Movie"/>
28: </owl:Thing>
29:
30: <rdf:Description rdf:about=
30a:   "taggings/8b9c45cfecb7087430daa963cd6bcd51d2fce30d">
31:   <rdfs:label>A bundle of Tags associated with this Thing,
31a:     defining when they were added and
31b:     by whom</rdfs:label>
32: </rdf:Description>
33:
34: </rdf:RDF>

```

As you can see, line 27 is added to identify the type of the reviewed item.

11.4.1.2 Revyu: Why It Is Different

Revyu is different from other review sites. For books and movies, Revyu assigns a URI to the item being reviewed and also generates an RDF document to represent the review itself. Furthermore, Revyu searches against existing linked datasets and automatically creates links to these external datasets whenever possible. For example, line 25 of List 11.7 and line 26 of List 11.8 are the links to other datasets. As a result, these links have turned both these two RDF documents into newly produced Linked Data on the Linked Data Web.

In fact, changing the submitted review into structured data not only adds new elements to the Linked Data Web, but also makes it much easier for any application that attempts to consume the review results.

For example, to get the review for a given item, all the application has to do is to query the Revyu dataset by issuing SPARQL query via Revyu's SPARQL interface, and the result is an RDF document that can be easily processed. Compared to Amazon, where the review data has to be obtained by using its own APIs (Amazon Web services), the benefit is quite obvious. We will come back to this point later in this chapter.

Besides producing new Linked Data, Revyu also consumes existing Linked Data on the Web to enhance its user experience.

To see this, let us go back to line 25 of List 11.7 and line 26 of List 11.8. Since these statements are links that point to other linked datasets, one can simply apply

the Follow-Your-Nose method to retrieve additional information about the reviewed item. In fact, this is exactly what Revyu has done. For example, by following the link on line 26 of List 11.8, Revyu was able to obtain this movie's entry in DBpedia, which contains the URI of the film's promotional poster and the name of the director, etc. All this additional information has been displayed on the page about this film, as shown in Fig. 11.19.

Clearly, this automatic consumption of the existing Linked Data has greatly enhanced the value of the whole site, without requiring this information to be manually entered by the reviewer.

Similarly, Revyu can fetch more information about the book from RDF Book Mashup dataset (line 25 of List 11.7), such as the book cover and author information, which is also displayed on the Revyu page about the book. Again, all this does not ask any extra work from the reviewer, but is very valuable to anyone who is reading these reviews.

Furthermore, the same idea can be applied to a reviewer. Recall each reviewer is assigned an URI, and a simple RDF document is created for this reviewer. If a given reviewer has an existing FOAF document, an `rdfs:seeAlso` statement will be included in the RDF description. For example, List 11.9 shows the RDF document created for me by Revyu.

List 11.9 RDF statements generated by Revyu for a reviewer

```

1: <?xml version="1.0" encoding="UTF-8" ?>
2: <rdf:RDF
3:   xml:base="http://revyu.com/"
4:   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
5:   xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
6:   xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
7:   xmlns:owl="http://www.w3.org/2002/07/owl#"
8:   xmlns:dc="http://purl.org/dc/elements/1.1/"
9:   xmlns:dcterms="http://purl.org/dc/terms/"
10:  xmlns:vcard="http://www.w3.org/2001/vcard-rdf/3.0#"
11:  xmlns:foaf="http://xmlns.com/foaf/0.1/"
12:  xmlns:rev="http://purl.org/stuff/rev#"
13:  xmlns:tag=
13a:    "http://www.holygoat.co.uk/owl/redwood/0.1/tags/">
14:
15: <foaf:Person rdf:about="people/liyang">
16:   <foaf:mbox_sha1sum>
16a:     1613a9c3ec8b18271a8fe1f79537a7b08803d896
16b:   </foaf:mbox_sha1sum>
17:   <foaf:nick>liyang</foaf:nick>
18:   <foaf:made rdf:resource=
18a:     "reviews/cbe1fd43cf7de69ee0530fe65593d6d77d03daed"/>
19:   <foaf:made rdf:resource=
19a:     "reviews/436f699d347d433315507923664cf567fe872a59"/>
20:   <rdfs:seeAlso
20a:     rdf:resource="http://www.liyangyu.com/foaf.rdf"/>

```

```
21: </foaf:Person>  
22:  
23: </rdf:RDF>
```

Revyu will dereference the URI on line 20 and query the resulting FOAF document for information such as my photo, location, home page, and interests. This information then automatically shows up at my profile page without me entering them at all.

As a summary, Revyu is a simple and elegant application that makes use of existing Linked Data to enhance its user experience without asking extra work from its users. Although it is not a large-scale Linked Data application, it does show us the benefits offered by the Web of Linked Data.

11.4.2 Web 2.0 Mashups vs. Linked Data Mashups

In the previous section, you have seen an interesting application that consumes Linked Data on the Web. However, up to now, consuming Linked Data on the Web in a large scale still remains an open question, and the business value of Linked Data Web can be better appreciated only through these large-scale applications.

However, researchers and developers in the field of the Semantic Web are still very optimistic about its future, and one of the reasons is based on the comparison of the so-called Web 2.0 mashup to semantic mashup. In fact, if Web 2.0 mashups can continue to remain in high demand in the environment of traditional Web, semantic mashups under the concept of the Semantic Web will sooner or later be the real data mashup tool that everyone will use, simple because it is much more easier, much more efficient, and much more scalable. The rest of this section will explain this conclusion in detail.

Exactly what is a mashup? In a very simple sentence, a *mashup* is a Web application that collects structured data produced by third parties through APIs offered by these parties and processes the data in some way and then represents the data back to the user in a form that differs from its original look-and-feel. Normally, a mashup application will either enhance the visual presentation of the data or offer added value to its users by combining the data from different sources or both. This concept is more related to Web 2.0, where more and more Web sites expose their data via their APIs.

A typical mashup could be something like this: a shopbot can be coded to retrieve the price of a given product (such as a camera with a specific make and model number) from Amazon.com by accessing its published APIs. At the same time, the same shopbot can also retrieve the price of the same product from BestBuy.com (assuming BestBuy has also published their APIs), and these two prices can be compared and returned to the user so the user can decide where to buy the product. The shopbot can even retrieve the prices from these two vendors periodically, so the user can see the change of these prices over a certain amount of time, and therefore

can buy the product when its price is going down and reaches a relatively stable stage. Here the added value is obvious, and we can expand this shopbot in many ways, such as including more vendors and more products.

This all sounds correct and feasible. However, when you really set off to construct such a shopbot, you will soon discover its limitations:

- poor scalability of the method itself

Since different vendors publish different APIs, this is a constant learning process. You will have to learn each set of APIs, and once a new vendor is available, you will have to learn a new set of APIs again. Therefore, the construction of such a mashup is not scalable and its maintenance will be quite expensive as well.

- limited coverage at most

Obviously, the shopbot only understands the APIs that you have coded for it to understand, it cannot do any simple explore on its own. The data coverage is therefore very limited and any decision based on this shopbot will probably not be optimal either.

- lost links to channel back to the data providers

Once the data are retrieved and consumed by the shopbot, the link between the shopbot and the original data provider is lost; a user cannot link back to the original data providing site. Even in the case where we have decided to put some links channeling back to the data providers, these links are shallow links at their best, and they will not be able to link back to the precise locations of those particular data components. In addition, a mashup site supported by this shopbot only shows the price. What if the original site offers some free gifts if you buy it now? If there were a link back to this particular product, the user might be able to catch this offer. Even more importantly, the links that channel back to the original data provider mean more incoming traffic, which means significant chances for some potential business value.

Now, with all the above being said, let us take a look at what would be the case if a mashup application is developed under the environment of Linked Data Web. In fact, Revyu is such a mashup: it retrieves data from external Web sites (DBpedia, RDF Book Mashup, etc.) to enhance its user experience, a typical way that a mashup should work. More specifically,

- good scalability of the method itself

Under the Web of Linked Data, structured data are expressed by using RDF graphs and standards, which is the only set of standards across all the sites, and there is no specific APIs for each site to expose its structure data. Therefore, constructing the mashup and maintaining the mashup is quite scalable, there is no need for constant learning of new APIs.

- unbounded coverage of datasets

Obviously, Web 2.0 mashups work against a fixed set of data sources; Linked Data applications operate on top of an unbound, global data space. This enables them to deliver more complete answers as new data sources appear on the Web.

- crucial links to channel back to the data providers

In Linked Data mashups, all the items (resources) are identified by URIs, each of which may be minted and controlled by the data provider. If a user looks up one of these URIs, the user may be channeled back to the original data provider; it is then up to the data provider to publish appropriate content to further direct the incoming traffic. This linking-back capability is a key difference between Web 2.0 mashups and Linked Data mashups, and this is where the potential business value could be.

Besides the above, Linked Data mashups also offer a chance to their users to chain up almost unlimited resources. More specifically, each item in the mashup is identified by a URI, which can be linked to other resources in other datasets, and the links themselves are also typed. As a result, you can choose to follow a specific link and visit a specific resource, which further takes you to other resources in other datasets, so on and so forth. As this point, you should be able to appreciate the value of this unlimited linkage, without the need of much explanation at all.

11.5 Summary

In this chapter, we have learned Linked Open Data. It is another example of the Semantic Web technologies at work, and it is quite different from other examples we have learned. Instead of adding semantics to the current Web (either manually or automatically), its idea is to create a machine-readable Web all from the scratch. It is therefore also called the Web of Linked Data, or the Linked Data Web.

First off, understand the following about Linked Open Data:

- its basic concept and basic principles;
- its relationship to the classic view of the Semantic Web, i.e., it can be viewed as an implementation of the vision of the Semantic Web.

Second, understand the two major topics about Linked Open Data: how to publish Linked Data on the Web and how to consume Linked Data on the Web.

About how to publish Linked Data on the Web, you need to understand the following:

- how to mint URIs for resources, what is the difference between 303 URIs and hash URIs;
- how to create links to other datasets, and how to make sure your data is published as Linked Data, i.e., what are the minimal requirements of being Linked Data on the Web;
- remember to use a validator to make sure you have done everything correctly.

About how to consume the Linked Data, you need to understand the following:

- consuming Linked Data means discovering Linked Data on the Web, accessing the Web of Linked Data, and building applications on top of the Web of Linked Data;
- there are Semantic Web search engines you can use to discover Linked Data on the Web;
- there are Semantic Web browsers you can use to access the Web of Linked Data manually;
- you can also use SPARQL endpoints to access the Web of Linked Data, in addition, you can programmatically access the Web of Linked Data from within your applications;
- the ultimate goal is to create powerful applications that make use of the Web of Linked Data.

Finally, to show you how to build applications on top of the Web of Linked Data, we have included Revyu as one such example. Make sure you understand how Revyu makes use of the Linked Data on the Web, and more importantly, hope this can serve as a hint to you, so you can come up with possible applications of your own to show the power of the Web of Linked Data.

Chapter 12

Building the Foundation for Development on the Semantic Web

Finally, with what you have learned from this book, you are now ready to start your own development on the Semantic Web.

To better prepare you for this work, we will present an overview in this chapter that covers two major topics. With the knowledge presented in this chapter, your future development work will start with a solid foundation.

The first topic is about available development tools for the Semantic Web, including frameworks, reasoners, ontology engineering environments, and other related tools. As a developer who works on the Semantic Web, understanding the available tools is simply a must.

The second topic covers some guidelines about the development methodologies. In general, building applications on the Semantic Web is different from building applications that run on the traditional Web and requires its own development strategies. With a clear understanding of the methodologies, your design and development work will be more productive and your applications will be more scalable and more maintainable.

12.1 Development Tools for the Semantic Web

12.1.1 Frameworks for the Semantic Web Applications

12.1.1.1 What Is a Framework and Why We Need It?

A *framework* in general can be understood as a software environment designed to support future development work. It is often created for a specific development domain and normally contains a set of common and reusable building blocks so that developers can use, extend, or customize for their specific business logic. With the help from such a framework, developers do not have to start from scratch each time an application is developed.

More specifically, for development work on the Semantic Web, the main features of a framework may include the following:

- core support for RDF, RDFS, and OWL;
- inference capabilities for both RDFS ontologies and OWL ontologies;

- support for SPARQL query;
- the handling of persistent RDF models, with the ability to scale efficiently to large datasets.

This list is growing with more and more development frameworks becoming available. For developers, a Semantic Web development framework can at least provide the following benefits:

- It provides developers with the implementation of common tasks in the form of reusable code, therefore less repeated work and less bugs.

For example, there is a set of common operations that have to be implemented for probably every single application on the Semantic Web. These common tasks include reading/parsing a given RDF model, understanding an RDF model, and inferencing based on ontology model handling, just to name a few. Since a framework provides the support for these common tasks, developers can focus on the specific business logic, and it is more likely that they can deliver more reliable code.

- It makes it easier to work with complex technologies such as the Semantic Web technologies.

By now, you have seen all the major technical components for the Semantic Web. Clearly, there are a lot to learn even before you can get started with your own development. With the help from a development framework, not only it is easier to put what you have learned into practice but also you will gain deeper understanding about the technology itself. You will get more feelings on this when you finish the last several chapters of this book – for those chapters, you will have a chance to use a concrete framework to develop several Semantic Web applications.

- It forces consistency within the team, even across platforms.

This directly follows from the fact that quite a lot of the common tasks are built by reusable code and are accessed through a set of common “wrappers.” This not only forces the consistency but also makes testing and debugging tasks much easier, even if you are not the one who wrote the code at the first place.

- It promotes design patterns, standards, and policies.

This may not seem quite obvious at this point; you will see more on this in Sect. [12.2.1.3](#).

With this being said, let us take a look at some popular frameworks. Note that we are not going to cover the usage details of these frameworks, since each one of them will require a separate chapter to cover. We will however give you an overview of each framework so that you will have a set of choices when it comes to your own development work.

12.1.1.2 Jena

Jena (<http://jena.sourceforge.net/>) is a free, open-source Java platform for applications on the Semantic Web. It was originally developed by Brian McBride from Hewlett-Packard Laboratories (HPL). Jena 1 was originally released in 2001, and Jena 2 was first released in August 2003. Its latest version, Jena 2.6.2, was released on 16 October 2009.

Jena is now believed to be the most used Java toolkit for building applications on the Semantic Web. It is also the leading Java toolkit referenced in academic papers and conferences.

Jena's comprehensive support for Semantic Web application development is quite obvious, given its following components:

- an RDF API;
- an OWL API, which can also be used as RDFS API;
- reading and writing RDF in RDF/XML, N3, and N-triples formats;
- in-memory and persistent storage of RDF models;
- SPARQL query engine, and a
- rule-based inference engine.

Throughout this book, Jena will be used as our example development framework. A detailed description about Jena can be found in [Chap. 13](#) as well.

12.1.1.3 Sesame

Sesame (<http://www.openrdf.org/>) is an open-source Java framework for storage and querying of RDF datasets. Sesame was originally developed as a research prototype for an EU research project called On-To-Knowledge, and it is currently developed as a community project with developers participating from around the globe. Its latest version, Sesame 2.3.1, was released on 1 February 2010.

Sesame has the following components:

- the *RDF Model*, which defines interfaces and implementation for all basic RDF entities;
- the *Repository API* (built upon the RDF Model), a higher level API that offers a large number of developer-oriented methods for handling RDF data, including RDFS reasoning support; and
- a *HTTP server* (built on top of Repository API), which consists of a number of Java Servlets that implement a protocol for accessing Sesame repositories over HTTP.

In general, Sesame's focus is on the RDF data storage and query, but without much support for OWL and related inferencing tools.

12.1.1.4 Virtuoso

Virtuoso (<http://virtuoso.openlinksw.com/>) is also called *Virtuoso Universal Server*. Essentially, it is a database engine that combines the functionality of traditional

RDBMS, ORDBMS, RDF, XML, free-text, Web application server, and file server into a single server product package. Its latest version, V6.1.1, was released on 31 March 2010, and it can be downloaded freely for Linux and various Unix platforms. A Windows binary distribution is also available. Note that there are also commercial editions of Virtuoso, with Virtual Database Engine and Data Clustering as their extra contents.

As far as the Semantic Web is concerned, Virtuoso has the following support:

- It can be used as a RDF Triple Store. One can load N3, Turtle, and RDF/XML files into a Virtuoso hosted named graph using Virtuoso SQL functions.
- It includes a number of metadata extractors for a range of known data formats, such as microformats. These metadata extractors enable automatic triple generation and storage in its own RDF Triple Store.
- It supports SPARQL statements, and these SPARQL statements can be written inside SQL statements. In other words, any ODBC, JDBC, .net, or OLE/DB application can simply make SPARQL queries just as if they were all SQL queries.
- It supports reasoning based on RDFS and OWL. Note that there are differences between the open-source versions and closed-source ones; you need to check the documents for more details.
- Its API support includes Jena, Sesame, and Redland; all are available via the Client Connectivity Kit in the form of Virtuoso libraries.
- It supports Java, .NET bound languages, C/C++, PHP, Python and Perl.

As you can tell, Virtuoso is a product with a very board scope, not only in the Semantic Web world but also in data management in general. Although this book focuses on Jena, Virtuoso can be another good choice for development work on the Semantic Web.

12.1.1.5 Redland

Redland (<http://librdf.org/>) is a set of free C libraries that provide support for RDF. Its latest release was on 15 February 2010. Redland offers the following support:

- Redland RDF library provides a C API that works with client application.
- *Raptor* as the RDF parser library deals with reading RDF/XML and N-triples into RDF triples. It is an independent piece from Redland, but required by Redland.
- *Rasqal* (pronounced *rascal*) as the RDF Syntax and Query Library for Redland is responsible for executing RDF queries with SPARQL.
- Redland Language Bindings for APIs support C#, Java, Objective-C, Perl, PHP, Python, Ruby and Tcl.

Although Redland does not provide a strong support for reasoning and inferencing, it does work with C language. When speed is a major concern, Redland framework can be the choice.

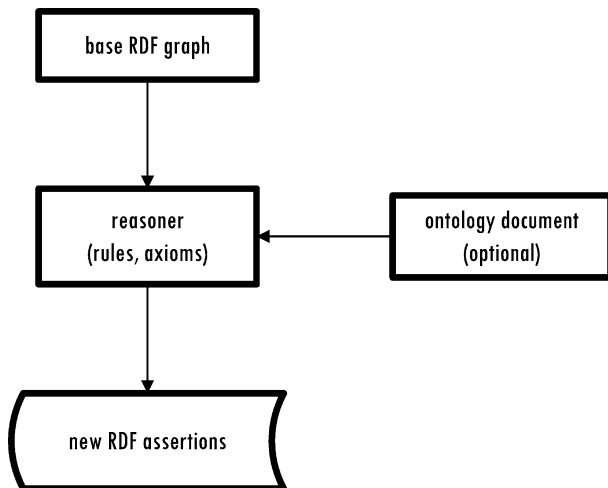


Fig. 12.1 Basic structure of a reasoner on the Semantic Web

12.1.2 Reasoners for the Semantic Web Applications

12.1.2.1 What Is a Reasoner and Why We Need It?

To put it simple, a Semantic Web *reasoner* is a software that can perform reasoning tasks for applications on the Semantic Web, typically based on RDFS or OWL ontologies.

Note that reasoning refers to the process of deriving facts that are not explicitly expressed by the given ontology documents and the instance documents. In [Chaps. 4 and 5](#), we have seen quite a few examples of the reasoning power provided by RDFS and OWL ontologies. The inferencing process implemented in those examples is the work done by a reasoner.

Without touching the theoretical details of reasoning process, [Fig. 12.1](#) shows the basic structure of the inferencing process.

As [Fig. 12.1](#) shows, a reasoner is used to derive additional RDF statements which are entailed from the given base RDF graph together with any optional ontology information. The reasoner works by employing its own rules, axioms, and appropriate chaining methods (forward/backward chaining, for example).

When a given application on the Semantic Web needs a reasoner, it is usually not the best choice to write one by yourself. You should take advantage of an existing reasoner to accomplish your task. In fact, some popular development frameworks have already included reasoners for us to use, and there are also stand-alone reasoners that can be easily plugged into our applications. Let us take a look at these choices in the next few sections.

12.1.2.2 Pellet

Pellet (<http://clarkparsia.com/pellet>) is an OWL 2 reasoner for Java. It is freely downloadable, and its latest release, Pellet 2.1, was announced on 1 April 2010. It supports the following main reasoning functionalities:

- qualified cardinality restrictions;
- complex sub-property axioms (between a property chain and a property);
- local reflexivity restrictions;
- reflexive, irreflexive, symmetric, and anti-symmetric properties;
- disjoint properties;
- negative property assertions;
- vocabulary sharing (punning) between individuals, classes, and properties;
- user-defined data ranges.

Besides the above, Pellet provides all the standard inference services that you can find from a traditional OWL DL reasoner, such as ontology consistency checking, classification, and realization (finding the most specific classes that an individual belongs to).

Pellet itself is not embedded in any development framework. The following are some of the common ways to access Pellet's reasoning capabilities:

- a Web-based demonstration page called OWLSight¹;
- a command line program (included in the distribution package);
- a set of programmatic API that can be used in a stand-alone application;
- the reasoner interfaces with the Manchester OWL API and Jena. Therefore, you can use Pellet in your applications developed by Jena;
- direct integration with the Protégé ontology editor.

12.1.2.3 RacerPro

RacerPro (<http://www.racer-systems.com/>) is an OWL reasoner and inference server for the Semantic Web. RACER stands for Renamed ABox and Concept Expression Reasoner. RacerPro is the commercial name of the software. At this point, RacerPro 2.0 is the latest release.

RacerPro can process OWL Lite as well as OWL DL documents. Some major reasoning capabilities supported by RacerPro includes the following:

- Check the consistency of an OWL ontology and a set of data descriptions.
- Find implicit sub-class relationships induced by the declaration in the ontology.
- Find synonyms for resources (including both classes and instance names).

¹You can find this page at <http://pellet.owldl.com/owlsight/>

As a query server, RacerPro supports *incremental query answering* for information retrieval tasks (retrieve the next n results of a query). In addition, it supports the adaptive use of computational resource: answers that require few computational resources are delivered first, and user applications can decide whether computing all answers is worth the effort.

RacerPro is not embedded in any development framework. It offers the following deployment options a given application can choose from:

- back-end network server application;
- file processor launched from command line interface;
- when loaded as object code, it can be part of a user application as well (Java API provided).

12.1.2.4 Jena

We have discussed Jena as a development framework in Sect. 12.1.1.2. In fact, Jena framework also has several reasoners embedded:

- *Jena's RDFS reasoner*. This reasoner supports most of the RDFS entailments described by the RDFS standard (see Chap. 4).
- *Jena's OWL reasoner*. This is the second major set of reasoners supplied with Jena. This set includes a default OWL reasoner and two small/faster configurations. Each of the configurations is intended to be a sound implementation of a subset of OWL Full, but none of them is complete.

To use the complete OWL reasoning, an external reasoner such as Pellet can be used, as we have described earlier.

12.1.2.5 Virtuoso

Finally, we would like to mention Virtuoso again, which also offers an OWL reasoner. Based on its official document,

- it supports `owl:sameAs`, `rdfs:subClassOf`, and `rdfs:subPropertyOf`, which are sufficient for many purposes;
- `owl:sameAs`, `owl:equivalentClass` and `owl:equivalentProperty` are considered when determining sub-class or sub-property relations;
- for version 6.1.0, `owl:TransitiveProperty`, `owl:SymmetricalProperty`, and `owl:inverseOf` have also been added.

Finally, Virtuoso defaults to using backward chaining, but if desired, forward chaining may be forced.

12.1.3 Ontology Engineering Environments

12.1.3.1 What Is an Ontology Engineering Environment and Why We Need It?

The vision of the Semantic Web and its applications is characterized by a large number of ontologies. Without ontology, the Semantic Web will not exist.

However, ontology creation and development is never an easy task, mostly due to the following facts:

- A given ontology often aims to cover a whole domain by using a set of classes, properties and by summarizing their relationships. Since the domain knowledge is often complex, the resulting ontology is also complex and large in scale.
- Ontology designers/developers have different skill levels, different cultural and social backgrounds, and different understanding of the needs from potential applications.
- Ontology development requires the knowledge of ontology presentation languages (such as RDFS and OWL) which can be difficult to grasp for many domain experts, who are the main drivers behind a give ontology.
- The real world changes quickly, and ontologies that represent knowledge in the real world also have to be dynamically updated to keep up with the changes and new requirements. This makes ontology development/maintenance harder as well.

To make ontology development easier, a special group of support tools are created, and they can be used to build a new ontology from scratch. In addition to their common editing and browsing functions, they also provide support to ontology documentation, ontology export (to different formats and different ontology languages), and ontology import (from different formats and different languages). Some tools even have some inferencing capabilities. These group of tools are often referred to as *ontology engineering environment*.

The benefits offered by a ontology engineering environment are quite obvious. For example, as a basic component of any given ontology engineering environment, a graphical ontology editor can make the common tasks easy to accomplish. For any given ontology, either new or existing, these common tasks include adding new concepts, properties, relations, and constraints. With the help from ontology editor's graphical user interface, domain experts will not have to be efficient on a given ontology representation language. In addition, graphical ontology editor can help the developer to visualize and organize the overall conceptual structure of the ontology. With this help, designers and developers can discover logical inconsistencies or potential problems with the given ontology, such as not being able to fully cover the needs of foreseeable applications in the specific domain. In fact, some graphical ontology editors do have the ability to help developers and designers to reconcile logical or semantic inconsistencies among the concepts and properties defined in the ontology.

Other functions offered by an ontology engineering environment can also be very useful. For example, inferencing capability is offered to help the ontology evaluation and refinement phase. Another example is annotation using the ontology. For this function, usually a graphical user interface is provided to facilitate the annotation process. This not only is used to mark up a given document but also can be used to examine the completeness of a given ontology. Note that these functionalities are provided by employing the so-called plug-in architecture to make the tool easily extensible and customizable by the users, as we will see in the coming discussions.

Ontology engineering and development is a complex topic and there are in fact many books available just to cover this area. With the basic knowledge we have, together with OWL/RDFS as ontology language choices, you can explore more on your own. For now, let us look at some example environments we can use in our own development work.

12.1.3.2 Protégé

Protégé (<http://protege.stanford.edu/>) is currently the leading environment for ontology development. It is a free, open-source ontology editor and knowledge-base framework written in Java.

Protégé was originally developed at Stanford University; it is now supported by a community of developers and academic, government, and corporate users. Its latest release was Protégé 4.1 on 4 March 2010.

Protégé supports two main ways of modeling ontologies: the Protégé–OWL editor and the Protégé–Frames editor. The Protégé–Frames editor enables users to build ontologies in accordance with the Open Knowledge Base Connectivity (OKBC) protocol, whilst the Protégé–OWL editor enables the users to build ontologies for the Semantic Web, in particular by using the W3C standards such as RDFS and OWL. Therefore, we will focus more on its Protégé–OWL editor in this section.

It is important to note that Protégé–OWL editors in Protégé 4.x versions only support OWL 2.0. If OWL 1.0 and RDFS are needed, Protégé–OWL editors from Protégé 3.x versions should be selected. The following main features are therefore a combination of all these previous versions:

- create ontologies by using RDFS and OWL 1.0 (version 3.x only);
- create ontologies by using OWL 2.0 (version 4.x only);
- load and save OWL and RDFS ontologies. With version 3.x, OWL and RDFS files are accessible via Protégé–OWL API. However, with version 4.x, only OWL files are accessible by using OWL API, developed by the University of Manchester, and this API is different from the Protégé–OWL API;
- edit and visualize classes, properties, and relations;
- define logical class characteristics as OWL expressions;
- edit OWL individuals for semantic markup;
- execute reasoners via direct access. For example, with version 3.x, direct connection to Pellet reasoner can be established. With version 4.x, direct connection to other DL reasoners besides Pellet can also be used.

Besides the above main features, Protégé can be extended by way of a *plug-in architecture*. More specifically, plug-ins can be used to change and extend the behavior of Protégé, and in fact, Protégé itself is written as a collection of plug-ins. The advantage is that these plug-ins can be replaced individually or as a whole to completely alter the interface and behavior of Protégé.

For developers, the Protégé Programming Development Kit (PDK) is a set of documentation and examples that describes and explains how to develop and install plug-in extensions for Protégé. If you need to use plug-ins to change Protégé, the PDK document is where you should start. Also, note that from version 4.x, Protégé's plug-in framework has been switched to the more industry standard technology, OSGi,² which allows for any type of plug-in extension. Currently, a large set of plug-ins are available, which are mainly developed either in-house or by the Protégé community.

Finally, Protégé supports a Java-based API for building applications on the Semantic Web. This Protégé API is often referred to as Protégé–OWL API; it is an open-source Java library for OWL and RDFS. More specifically, this API provides classes and methods to load and save OWL files, to query and manipulate OWL data models, and to conduct reasonings. It can either be used to develop components that are executed inside Protégé–OWL editor's user interface as plug-ins, or be used to develop external stand-alone applications.

12.1.3.3 NeOn

NeOn is a project involving 14 European partners and co-funded by the European Commission's Sixty Framework Program. It started in March 2006 and had a duration of 4 years.³ The goal of this project is to advance the state of the art in using ontologies for large-scale semantic applications, and the NeOn Toolkit,⁴ an ontology engineering environment, is one of the core outcomes of the NeOn project.

At this point, NeOn Toolkit is available for download from its community Web site, and its latest major version, v2.3, was released on 17 February 2010. Its main features include the following:

- supports OWL 2 specification;
- provides NeOn OWL editor, which can be used for creating and maintaining ontologies written in OWL.

The NeOn OWL editor has three components: Ontology Navigator, Individual panel, and Entity Properties panel. Using these components, a user can accomplish common ontology development tasks such as defining, creating, and modifying classes, properties, and their relationships in a given ontology. In addition, users can use these components as ontology management tool; tasks such as navigating between ontologies, visualizing a given ontology, inspecting related instances of a given class can also be accomplished.

²<http://www.osgi.org/Main/HomePage>

³<http://www.neon-project.org>

⁴<http://www.neon-toolkit.org/>

Note that NeOn toolkit is implemented as a set of Eclipse plug-ins. Similar to Protégé, it has a large user base. With its foundation on the Eclipse plug-in architecture, developers can build additional services and components and add them into the current NeOn toolkit.

12.1.3.4 TopBraid Composer

TopBraid Composer (<http://www.topquadrant.com/index.html>) is a visual modeling environment for creating and managing ontologies using the Semantic Web standards such as RDFS and OWL.

TopBraid Composer has three versions available: Free Edition, Standard Edition and Maestro Edition. Each of them offers different packages of support, and you can get onto the official Web site to check out the details to see which one fits your needs.

TopBraid Composer is implemented as an Eclipse plug-in, and it is built by using Jena API. It has four major functions: ontology editing, ontology refactoring, ontology reasoning, and ontology visualization. Its main features can be summarized as follows:

- *Overall features.* TopBraid Composer supports both RDFS and OWL (configurable as RDFS only and/or OWL only) and supports SPARQL queries as well. It can import from or export to a variety of data formats including RDBs, XML, and Excel; it also offers published APIs for custom extensions and building of Eclipse-based applications on the Semantic Web.
- *Ontology editing.* Users can create classes and define the relationships between classes. Similarly, users can define properties and relationships between properties. Furthermore, users can create instances based on the specified ontology. These common tasks can be accomplished by using either a form-based editor or an editor with graphical user interface.
- *Ontology refactoring.* User can move classes, properties, and instances between models, clone classes, properties, and instances. TopBraid Composer can synchronize name changes across multiple imported models; it also has a complete log of all changes and can roll back changes if necessary. It supports the conversion between RDFS and OWL.
- *Ontology reasoning.* TopBraid Composer interfaces with a collection of inference engines, including Pellet and Jena Rule Engine. It provides the ability to convert the inferred statements into assertions and also supports for debugging of inferences with explanations.
- *Ontology visualization.* TopBraid Composer offers UML-like graphical notations, as well as graph visualization and editing. It also provides tree-like views describing relationships between classes and properties.

As you can tell, as an ontology engineering environment, TopBraid Composer offers relatively complete and impressive support. Note that at the time of this writing, TopBraid Composer is not supporting OWL 2 yet.

12.1.4 Other Tools: Search Engines for the Semantic Web

Before we move on to the next section, let us talk about search engines on the Semantic Web. They are important for your development for a variety of reasons. For example, you might want to check if there are existing ontologies that may satisfy your needs. And again, it is always good to reuse any existing URIs for the resources your development work might involve, and a search engine for the Semantic Web will help to find these URIs.

When it comes to discovering resources on the Semantic Web, Sindice, Falcon, and Sig.ma are all quite useful. We have presented detailed descriptions for all these three search engines in this book, and you can find more about each one of them in [Chap. 11](#). More specifically, [Sect. 11.3.2.1](#) covers Sig.ma, [Sect. 11.2.1.6](#) covers Sindice, and finally, [Sect. 11.3.1.1](#) covers Falcon.

12.1.5 Where to Find More?

With the quick development around the Semantic Web, it is not easy to keep up with the latest releases and new arrivals. To find more, the W3C Semantic Web wiki is a good starting place:

`http://www.w3.org/2001/sw/wiki/Tools`

which also has links to other useful resources online. You should be checking back to this page for updates and more information frequently.

12.2 Semantic Web Application Development Methodology

In this section, we will discuss another important topic: methodological guidelines for building Semantic Web applications.

Applications on the Semantic Web, after all, are Web applications. Therefore, existing software engineering methodologies and well-known design patterns for Web application development are still applicable. However, considering the uniqueness of Semantic Web applications, such as they are built upon ontologies and the concept of Linked Data, what changes should be made to the existing methodologies? What are the new or improved design patterns that developers should be aware of? In this section, we will answer these questions; so when it comes to your own development work, you have the necessary methodological support available.

12.2.1 From Domain Models to Ontology-Driven Architecture

12.2.1.1 Domain Models and MVC Architecture

In our daily life as software developers, whether we are working on stand-alone systems or Web-based applications, we all have more or less heard and used the word *model*, and quite often, we use it in the context of *domain models*.

For a software system, a domain model represents the related concepts and data structures from an application domain. It also encodes the knowledge that is driving the application's behavior. To successfully complete a given application development project, developers quite often have to spend long hours, working with domain experts, to learn and understand the underlying domain model. They will express the domain model by using a set of classes and a description about the interactions among these classes, for example, by means of a collection of UML design diagrams.

After a few iterations with the domain experts, developers will settle down with their view of the domain model, which will become the centerpiece of the application system. At this point, developers may have also started with the design of the user interface components, which not only validate the fulfillment of the user requirements but also reinforce the correct understanding about the domain model.

It is certainly possible that the domain model can change. In which case, the data structure may have to be updated, which may or may not trigger a change of the implementation of those user interface components.

Finally at some point, the domain model will become relatively stable and does not change much. It is then desirable that we can somehow reuse it in our later development work. For example, we might want to grow the system so that it can offer more functionalities to the user or somehow make part of the domain model available to other systems so that some of the data elements can be accessed by the outside world.

To address all these issues, a well-known architecture is proposed. It is the so-called *model-view-controller* (MVC) architecture. More specifically,

- *Model*. The model represents domain-specific data and the business rules that specify how the data should be accessed and updated.
- *View*. The view renders the contents of the model to the end user, in the form of user interface elements. It is possible that for a single model component, there are multiple views (for different level of users, for example).
- *Controller*. The controller acts like a broker between the model and the view. For example, it accepts the user inputs from the view and translates the user requests into appropriate calls to the business objects in the model.

The controller is the key component that separates the domain model from the view. More specifically, in a stand-alone application, user interactions can be button clicks or menu selections, and in a Web-based application, the user requests will be mapped to `GET` and `POST` HTTP requests. Once the user request is submitted by the controller, the actions performed by the model include activating business processes which will result in state change of the model. Finally, based on the user requests and the outcome of the model actions, the controller responds by selecting an appropriate view to the user.

It is important to note that both the view and the controller depend on the model; however, the model depends on none of them. This is the key benefit of the MVC architecture and it allows us to accomplish the following:

- The model can be built and tested independent of the visual presentation.
- If there is a change in the domain model, such as a change in the business logic, the change can be easily implemented and can be transparent to the end users if necessary.
- Any change on the user interface does not generally affect the model implementation.
- The model can be reused and shared for other applications and target platforms.

For these reasons, MVC architecture is quite successful and widely accepted. Today, most Web-based applications are built by employing this architecture.

And how is this related to building applications on the Semantic Web? As we have pointed out earlier, applications on the Semantic Web are after all still Web-based applications. Therefore, MVC architecture remains to be a good choice. However, given the uniqueness of Semantic Web applications, a mere MVC solution might not be enough. Let us talk about this more in the next two sections.

12.2.1.2 The Uniqueness of Semantic Web Application Development

The most obvious uniqueness about applications on the Semantic Web is the fact that they are using ontologies and RDF models. With the help from ontologies and RDF models, the level of efficiency and scalability that can be reached by Semantic Web applications cannot be matched by traditional Web applications.

Let us go back to our task of collecting reviews about digital SLR cameras on the Web (see [Chap. 2](#)). We hope to have a soft agent to do this for us, so we don't have to read them one by one. Furthermore, we would like to have the report automatically generated based on the collected reviews so that we can run this application and produce an updated report as often as we want.

Although we have not discussed this task more since after [Chap. 2](#), however from what you have learned, the solution should be clear. Let us take a look at some details here.

During the course of this book, we have developed a small camera ontology, which is mainly for illustration purpose and is far from being of any practical value. Let us assume that a real camera ontology has been defined by some standardization group of the camera industry, and it is widely accepted and also published at a fixed URL as OWL file.

A camera ontology like this will allow the reviewers over the world to publish metadata about their reviews, and this metadata is exactly what a soft agent will collect. The collected metadata information will later on be used to make conclusions about a given camera.

More specifically, a reviewer can take one of the following two alternatives:

- A reviewer can publish metadata using RDFa.

In this case, a reviewer continues to write his/her review via a HTML page, but he/she will also mark up the content by using RDFa, with the terms defined in

the camera ontology. This is a simpler choice since the reviewer does not have to generate another separate page for the review.

- A reviewer can publish the metadata using RDF document.

In this case, a reviewer publishes the review via HTML page; meanwhile, he/she will also release an RDF document that expresses the same review. Again, this RDF document will be created by using the terms contained in the camera ontology.

Now, our agent can go out and collect all the available reviews. It is able to understand both RDFa markups and separate RDF documents. The final result is a set of collected RDF statements stored in an RDF data store that we have chosen to use.

Once the collection process is finished, we can start to issue SPARQL queries against the RDF data store to get the report we want. For example, our report can have the answers to the following questions:

- What is the most often used performance measurement?
- What is the average rating for a given camera model?
- What is the most popular camera model being reviewed?

This step is quite flexible and sometimes even requires some imagination. For instance, if a reviewer has included personal information in the published metadata (such as his/her location, contact information, profession), we can even start to understand the customer group for each camera model.

Clearly, all the components we have mentioned above, the agent and the SPARQL query interface, can be packaged together into a Web-based application. With this application, one simple button click will activate the agent so that it can start the collecting process, and other GUI components can be used to generate the report.

A user of this Web application does not have to know the technologies that make it possible. However, as developers, we understand how important the camera ontology is in this whole process.

In fact, for any application on the Semantic Web, the ontology *is* the domain model. In addition, understand that for a traditional Web application based on MVC architecture, the domain model maps to a set of classes that are used to code the domain knowledge, and quite often it is expressed in UML format. For an application built on the Semantic Web, the domain model maps to an ontology that is domain specific, and there is no direct map to a set of classes established yet (more on this in the next section).

Another important fact is that if the domain model is expressed as an ontology, it can be more easily shared and reused. For example, the camera ontology we have discussed above can be easily reused in a totally different application.

To see this, imagine we are building a ShopBot that can help a user to buy cameras. If camera retailers can mark up their catalogs by using the same camera ontology and publish the markup on their own Web sites, our ShopBot will be able to collect them. Once the collection is done, our ShopBot could easily find a retailer that offers a camera that satisfies the user's requirement. Furthermore, with

the help from the reasoning power offered by the camera ontology, our ShopBot can automatically suggest potential choices that are not obvious at the first look. You will see such a ShopBot example in the last chapter of this book.

Obviously, collecting the reviews and shopping for a camera are quite different applications; however, they are sharing and reusing the same camera ontology. In fact, they can start to share data with each other, and by aggregating the shared data elements, we can start to understand how the reviews can affect the sales of cameras.

To some extent, application development on the Semantic Web is getting easier: developers first discover sharable ontologies as the domain model and then wire the ontologies together with remaining object-oriented components for user interface and control components.

In fact, this design concept has been taking shape for the last several years, and it is now being recognized as the *ontology-driven architecture*. We will take a closer look at this architecture in the next section.

12.2.1.3 Ontology-Driven Software Development

As we have discussed earlier, a traditional application based on MVC architecture normally has its domain model mapped to classes that are defined by mainstream programming languages such as Java or C#. The related domain knowledge is coded into these classes, which are also responsible for common tasks such as communicating with databases and other resources.

For an application that runs on the Semantic Web, its domain model is expressed by ontologies. Could we again simply map all the RDFS or OWL classes into object-oriented classes and continue to build the system as we would have done traditionally? The answer is no.

The key to understand ontology-driven design is to understand the following facts about a given ontology:

- Properties in ontology are independent of specific classes.
- Instances can have multiple types, and they can change their types as a result of classification.
- Classes can be dynamically defined at run time as well.

These key differences tell us, that in order to fully exploit the weak typing and flexibility offered by the RDFS/OWL ontology, we will have to map RDFS/OWL classes onto runtime objects so that classes defined in the ontology will become instances of some object-oriented classes. This is the key idea of the ontology-driven software development method.

For example, Fig. 12.2 shows one such mapping.

As shown in Fig. 12.2, the object-oriented model that represents ontologies in our code is designed to contain classes which represent RDFS/OWL classes, RDFS/OWL properties, and RDF individuals separately, where RDF individuals are instances of the RDFS/OWL classes or RDFS/OWL properties. And obviously, all these three classes, namely, `OWLClass`, `OWLProperty`, and `RDFIndividual` are sub-classes of class `Resource`.

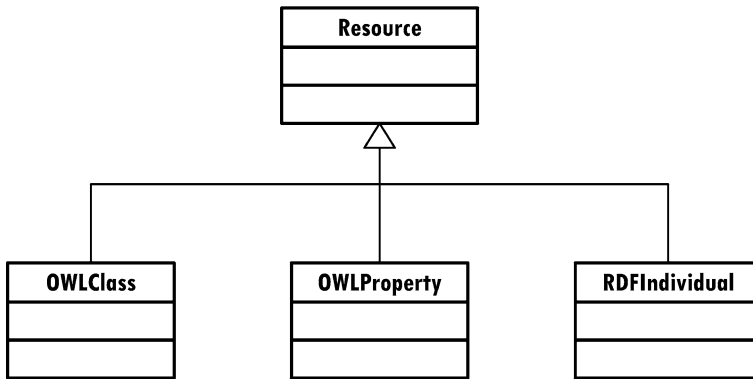


Fig. 12.2 A simple mapping of RDFS/OWL classes to runtime objects

With this design, our application can load the ontology into this object model and start to build logic on top of it. For example,

- since RDFS/OWL classes are now instances of the class `OWLClass`, it is possible to add and modify RDFS/OWL classes at the run time;
- since RDFS/OWL properties are now instances of the class `OWLProperty`, it is possible to assign property values to any resource dynamically and also possible to query any property value for any resource dynamically;
- since individuals are now instances of the class `RDFIndividual`, it is possible to change their types dynamically.

Obviously, the key is to represent the object types as objects, which provides the flexibility we need. For those of you who are familiar with design patterns, you probably have recognized this pattern already: it is known as the *dynamic object model* pattern.

Now, when it comes to real development work, how do we implement the dynamic object model pattern? In fact, as developers, you don't have to worry about this. Most application development frameworks for the Semantic Web have implemented this pattern for you already, and clearly, this is one of the benefits offered by using a development framework (see Sect. 12.1.1.1).

For example, Jena provides a dynamic object model in which OWL classes, properties, and individuals are stored using generic Java classes such as `OntClass` and `Individual`. If you are using other development frameworks, you will be able to see similar patterns implemented for you by the framework.

Note that there are also some disadvantages associated with dynamic object model pattern. For instance, references to ontology objects are established only through names (i.e., Strings), making code hard to maintain and test. In addition, as the ontology may change at design time, existing code may become inconsistent with the changing ontology.

One solution is to reflect the ontology concepts with custom classes so that the ontological structure is exploited at compile time. This also allows developers to attach access methods to these classes, leading to cleaner object-oriented design patterns. We will not go into details at this point, but in your development work, if you see `reflect` used, you should be able to understand the reason behind it.

12.2.1.4 Further Discussions

So far in this book, it is probably obvious to you that lots of effort have been devoted to defining standards (earlier standards such as RDF, RDFS, and OWL 1, latest standards such as OWL 2 and SPARQL 1.1, for example) and creating appropriate tool support. Work on development methodologies for Semantic Web applications is still in its infancy, probably due to the fact that this field is rather new and few people have experience in the development of real-world applications.

This is of course going to change. With more and more real-world applications emerging on the Web, there will be more experience gathered, and developers will be offered with more guidelines. What we have presented here in this section is only a start and only serve as a basic guideline to your development work. It is up to you also to discover more and share more with your fellow developers.

12.2.2 An Ontology Development Methodology Proposed by Noy and McGuinness

It is obvious from the previous discussion how important ontologies are for any given application on the Semantic Web. In fact, when developing applications on the Semantic Web, a significant portion of the effort has to be spent on developing ontologies. These ontologies must be consistent, generally useful, and formally correct. In addition, they must be extensible and may become very large.

In this section, we will present a summary of an ontology development method proposed by Noy and McGuinness (2001). Over the years, these steps are followed by many applications and research projects, and are widely referenced in a variety of research papers as well.

Understand that there is no single correct ontology design methodology that everyone can use, and it is also impossible to cover all the issues that an ontology developer may need to consider. In most cases, it is up to you to come up with the best solution for your specific application domain.

12.2.2.1 Basic Tasks and Fundamental Rules

According to Noy and McGuinness, the basic task of developing an ontology include the following:

- define classes;
- arrange the classes into a class hierarchy;

- define properties and allowed values for the properties;
- create instances and specify values for the properties for the instances.

In essence, Noy and McGuinness propose an iterative approach to ontology development. More specifically, an initial ontology is created in a rough first pass; it is then revised and refined, with details provided and filled. They summarize the fundamental rules of ontology development as follows:

- There is no such thing as the “correct way to model a domain.” The solution almost always depends on the application that we have in mind and extensions we can anticipate.
- Ontology development should be an iterative process.
- Concepts in the ontology should be close to objects and relationships, where objects can be either physical or logical. Also, these are likely to be nouns (objects) or verbs (relationships) in sentences that describe the domain.

Noy and McGuinness emphasize the fact that ontology development is iterative. The initial version of the ontology should be tested and evaluated in applications, and should be discussed with domain experts. Similarly, the revised version should be put back to the cycle for more fine-tuning.

With this in mind, let us now take a look at the basic development steps proposed by Noy and McGuinness.

12.2.2.2 Basic Steps of Ontology Development

Noy and McGuinness suggest that ontology development follows these steps:

Step 1. Determine the domain and scope of the ontology

To effectively accomplish this step, developers should answer these basic questions:

- What is the domain the ontology will cover?
- For what purpose the ontology is going to be used?
- For what types of questions the ontology should be able to provide answers?
- Who will use and maintain the ontology?

Take our camera ontology as one example. If the ontology is going to be used by camera retailers, pricing information should be included. On the other hand, if the ontology is used only for performance review, pricing information could be optional. Therefore, understanding the application we have in mind and anticipating what kinds of questions should be answered by using the ontology are quite important questions to answer.

Step 2. Consider reusing existing ontologies

As we have discussed throughout the book, reusing existing ontology is always a good choice when it is appropriate to do so. And as pointed out

by Noy and McGuinness, if one of the requirements is to make sure our system can communicate with other applications that have already committed to particular ontologies, reusing these ontologies is a must.

Step 3. Enumerate important terms in the ontology

This is the step before defining classes and class hierarchy. Noy and McGuinness suggest that one create a comprehensive list of the terms in the given domain, without worrying about overlap between concepts they represent, relations among the terms, or any properties that the concepts may have. The goal is to make sure all the important terms are included, since once we get into the details of defining classes and class hierarchy, it is easier to focus on the details and overlook the overall completeness of the ontology.

Step 4. Define classes and the class hierarchy

In general, three approaches can be used when defining classes and class hierarchy.

The first one is called *top-down* approach, where the definition process starts with the definition of the most general classes and continues to the subsequent specialization of the classes.

The second one is *bottom-up* approach, which is the exact opposite of top-down approach. When using this approach, developers start with the most specific classes and move on to more general ones.

The last approach is the *combination* approach. As its name suggests, it combines the above two approaches and developers can switch between the two approaches when defining classes and class hierarchy.

Whichever approach is chosen, it is important to understand that none of these three methods is inherently better than any of the others. It depends strongly on the personal view of the domain. In reality, the combination approach is often the easiest for many ontology developers.

Step 5. Define the properties of classes

To define properties, Noy and McGuinness suggest that the developer considers the following types of properties:

- “Intrinsic” properties, which represent those inherent characteristics of a given class. For example, in camera ontology, `shutter speed` would be an intrinsic property of a given camera.
- “Extrinsic” properties, which represent those characteristics that are not inherent. For instance, the `model` of a given camera.
- Parts. For example, a camera has a `body` and a `lens`.
- Relationships to other individuals. For example, the `manufacturer` of a given camera and the `owner` of a given camera.

Step 6. Add constraints to the properties

First off, note that when Noy and McGuinness published their paper (Noy and McGuinness 2001), properties were called slots and property constraints

were called facets. You may still see these terminologies today in some literatures.

Once we finish adding properties to the ontology, we need to consider property constraints such as cardinality constraints and value type constraints, just to name a few. The constraints you can use and add also depend on the ontology language you use. For example, OWL 2 offers much more constraint constructs than does RDFS, as you have learned in this book.

Note that for a given property, its domain and range information is also defined in this step.

Step 7. Create instances

This step is the last step, also an optional step. In other words, an ontology document does not have to include instance definition. For instance, FOAF ontology does not have any instance defined.

In case you would like to include instances into the ontology document, the procedure of creating instances normally has the following steps:

- choose a class;
- create an individual instance of that class; and
- define values for its properties.

At this point, we have summarized the basic steps when it comes to ontology development. Obviously, steps 4 and 5 are the most important steps as well as the most flexible ones. In the next two sections, we will discuss more about these two steps.

12.2.2.3 Other Considerations

Noy and McGuinness have also discussed things to look out for and errors that are easy to make when defining classes and a class hierarchy. In this section, we summarize their findings which are quite useful in real work.

- A class hierarchy represents an *is-a* relation.

To decide whether your class hierarchy is correct, an easy way is to see if a given sub-class and its root class has an *is-a* relation. For example, a `Digital camera` is a `Camera`, therefore, `Digital` class is a sub-class of `Camera` class. Also, once you have this *is-a* relation in your mind, you will be able to avoid some common mistakes, such as specifying a single camera as a sub-class of all cameras. Obviously, a `Camera` is not a `Cameras`.

- A sub-class relationship is transitive.

Another way to validate your class hierarchy is to remember the fact that a sub-class relationship is transitive. In other words, if `A` is a sub-class of `B` and `B` is sub-class of `C`, `A` will be a sub-class of `C`. You can always apply this rule to check if your class hierarchy is correct.

- How many sub-classes a class should have?

First off, note that a class does not have to have sub-class at all. In case it does, there are no hard rules specifying how many direct sub-classes it should have. However, many well-structured ontologies have between two and a dozen direct sub-classes. And the two guidelines are: (1) if a class has only one direct sub-class, there may be a modeling problem or the ontology is not complete and (2) if there are more than a dozen sub-classes for a given class, then additional intermediate categories may be necessary.

- When should we introduce a new class?

During ontology modeling, to represent some knowledge, we sometimes have to decide whether to introduce a new class or to add a new property to an existing class. The rules of thumb summarized by Noy and McGuinness can be stated as follows: a new sub-class should have additional properties that its super class does not have, or have new property value defined for a given property, or participate in different relationships than its super class.

Note that in practice, the decision of whether to model a specific distinction as a property value or as a new class will also depend on the scope of the domain and the task at hand.

For example, in our camera ontology, we can introduce a new class called `Digital` to represent digital camera and also a new class called `Film` to represent film camera. Furthermore, these two classes are sub-classes of `Camera` class. Another solution is to add a new property called `cameraMedia`, which can have values such as `digital` and `film`. Which one is a good solution for us?

The answer usually depends on the scope of the ontology and the potential applications that we have in mind. More specifically, if `Digital` and `Film` classes are very important in our domain and they play significant roles in our future applications, it will be a good idea to make these separate classes instead of a single property value on their super class. This is especially true when it is likely that we will need to specify new properties for each one of these classes.

On the other hand, if a camera has only marginal importance in our domain and whether or not the camera is a digital camera or a traditional film camera does not have any significant implications; it will then be a good choice to add a new property to the `Camera` class.

- For a given concept, when should we model it as a class, and when should we model it as an instance?

This is a very common question in real development work. For example, Nikon D200 is a digital camera made by Nikon. If we have a class named `Digital` representing digital cameras, we can now model Nikon D200 either as a sub-class of `Digital` or as an individual instance of `Digital`.

To make this decision, we need to decide the lowest level of granularity in our design. This decision is in turn determined by the potential applications of the ontology. For example, if we model Nikon D200 as a sub-class of `Digital` class (name it `Nikon_D200`), we can then model one particular Nikon D200 camera

offered by a specific retailer as an instance of the `Nikon_D200` class. If we model Nikon D200 as an instance of `Digital` class, this instance will represent all the Nikon D200 cameras in the world, and any particular Nikon D200 camera will not be identified (if we want to, we need to add some property, such as `retailer`, for example). As you can tell, different design decisions have a direct impact on our future applications.

The bottom line is that individual instances are the most specific concepts represented in a given ontology.

- If possible, we should always specify disjoint classes.

It is possible that in a given domain, several given classes are disjoint. For example, `Digital` class represents the collection of all digital cameras, and `Film` class represents all the traditional film cameras. These two classes are disjoint. In other words, a given camera cannot be both digital camera and film camera.

Ontology language such as OWL allows us to specify that several classes are disjoint. In real practice, we should always do so for the disjointed classes. This will enable the system to validate the ontology better. For example, if we create a new class that is a sub-class of both `Digital` and `Film`, a modeling error can be flagged by the system.

At this point, we have covered the main ontology design guidelines proposed by Noy and McGuinness. For any more details, you can find their original paper to continue your study. The main point to remember is that ontology design is a creative process and there is no single correct ontology for any domain. Also, the potential applications have a direct impact on ontology design and only by using the ontology in the related applications, we can further assess the quality of the ontology.

12.3 Summary

In this chapter, we have presented an overview of development on the Semantic Web. This overview has covered two major topics: the development tools and development methodologies you can use.

Understand the following about the development tools:

- There is a collection of development tools you can use for your development work on the Semantic Web.
- This collection includes development frameworks, ontology reasoners, ontology engineering environments, and other related tools.
- Understand different tools, for example, their functionalities and their limitations so that you will be able to pick the most suitable tools for your specific application.

Understand the following about development methodologies:

- The uniqueness of development work on the Semantic Web.
- The so-called ontology-driven development methodology, why it is suitable for the development work on the Semantic Web.
- The ontology development method proposed by Noy and McGuinness, including the basic rules, basic steps, and related considerations.

Reference

Noy NF, McGuinness DL (2001) Ontology development 101: a guide to creating your first ontology. Stanford Knowledge Systems Laboratory Technical Report KSL-01-05 and Stanford Medical Informatics Technical Report SMI-2001-0880

Chapter 13

Jena: A Framework for Development on the Semantic Web

Part of the previous chapter has presented an overview of available development frameworks you can use. This chapter will focus on Jena as a concrete example as well as our main development environment.

In this chapter, we will write quite a few examples, starting from `HelloWorld` to a set of basic tasks that you will likely encounter for any of your application on the Semantic Web, including RDF model operations, persistent RDF graph handling, and inferencing capabilities. The goal is not only to show you how to use Jena as a development framework but also to add some working Java classes into your own tool collection so that you can reuse them in your future development work.

Note that Jena is a programmer's API for Java Semantic Web applications. This chapter therefore assumes you are familiar with Java programming language.

13.1 Jena: A Semantic Web Framework for Java

13.1.1 What Is Jena and What It Can Do for Us?

At this point, we have not developed any application on the Semantic Web yet. However, based on what we have learned about the Semantic Web, it is not difficult for us to realize that any Semantic Web application will probably have to be able to handle the following common tasks:

- read/parse RDF documents
- write/create RDF documents
- navigate/search through an RDF graph
- query an RDF dataset by using SPARQL
- inference using OWL ontologies

This is certainly not a complete list, but they are probably needed to make even a simple application work. Fortunately, these standard items can be developed and assembled into a library that we can use, so our attention can be focused on the business logic when developing specific Semantic Web applications.

As we have discussed in [Chap. 12](#), there are quite a few such development tools available for us to use. In this book, we are going to use Jena as our development

tool. If you are using other frameworks, what you will learn here will also be helpful.

There are two steps we have to cover in order to use Jena API in our development. The first step is to download Jena package, and the second step is to set up a Java development environment that will be able to make use of Jena package. These two steps will be covered in detail in this section.

In this book, we are going to use Eclipse as our Java development environment. Other environments are available and can also be selected based on your needs. Also, note that the version numbers of Jena and Eclipse in this book will probably be different from the ones you have, but again this will not matter, since the basic steps of setting up the development environment should remain the same.

13.1.2 Getting Jena Package

To access Jena, go to the following page:

`http://jena.sourceforge.net/`

which links to the homesite of Jena, as shown in Fig. 13.1.

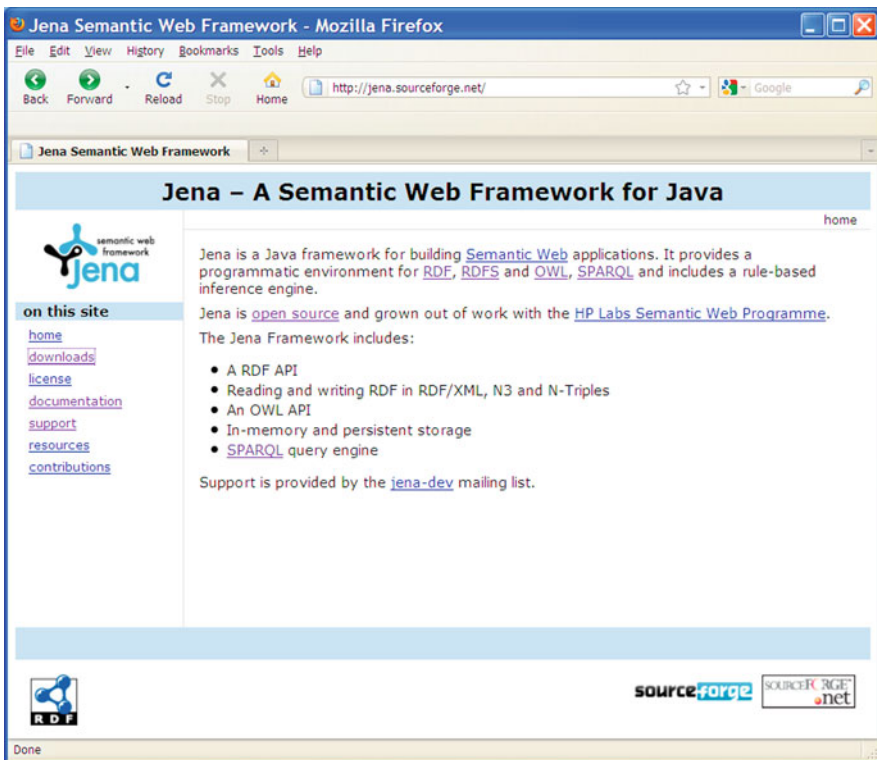


Fig. 13.1 Jena's homesite

To download Jena package, click `download` link on the left pane of this home page, and you will land on `sourceforge.net`'s download page. Note that in this book, we use Jena 2.6.0 as our example package, and as far as our applications will need, other version number should be the same. You can therefore choose to download Jena 2.6.0 or any version that is higher.

Once the download is done, you will find a zip file named `Jena-2.6.0` on your local hard drive. Unzip this file to create a Jena package directory on your hard drive. For example, I have saved it as follows:

```
C:\liyayang\DevApp\Jena-2.6.0
```

Note that Jena is a library for writing applications based on RDF and OWL documents. It is only used in your application code as a collection of APIs, and there is no GUI of any kind for Jena. The core components of Jena are stored in its `\lib` directory as shown in List 13.1.

List 13.1 Jena kernel library (Jena 2.6.0)

```
Directory of C:\liyayang\DevApp\Jena-2.6.0\lib
```

```
05/12/2009  03:04 PM    <DIR>          .
05/12/2009  03:04 PM    <DIR>          ..
05/12/2009  03:04 PM                236,733  arq-extra.jar
05/12/2009  03:04 PM                1,341,800 arq.jar
05/12/2009  03:04 PM                3,147,374 icu4j_3_4.jar
05/12/2009  03:04 PM                131,393  iri.jar
05/12/2009  03:04 PM                1,992,688 jena.jar
05/12/2009  03:04 PM                1,354,547 jenatest.jar
05/12/2009  03:04 PM                 34,638  json.jar
05/12/2009  03:04 PM                198,940  junit-4.5.jar
05/12/2009  03:04 PM                358,085  log4j-1.2.12.jar
05/12/2009  03:04 PM                665,064  lucene-core-2.3.1.jar
05/12/2009  03:04 PM                22,338  slf4j-api-1.5.6.jar
05/12/2009  03:04 PM                 9,678  slf4j-log4j12-1.5.6.jar
05/12/2009  03:04 PM                 26,518  stax-api-1.0.jar
05/12/2009  03:04 PM                 473,187  wstx-asl-3.0.0.jar
05/12/2009  03:04 PM                1,203,860 xercesImpl.jar
                15 File(s)          11,196,843 bytes
                2 Dir(s)          5,545,816,064 bytes free
```

To make use of the Jena library, it is important to add all the above `.jar` files to your `classpath` variable.

Using Windows XP as example, `classpath` variable is contained in two categories: `System Variables` category and `User Variables` category. It is normally enough that you make change to the one contained in `User Variables` category.

To do so, find `My Computer` icon on your desktop, right click it, and then click `Properties`, which will bring up `System Properties` window. On this window,

click **Advanced** tab and then click **Environment Variables** button, which will bring you to the window where you can edit `classpath` variable contained in **User Variables** category.

Now, set `classpath` variable to include all the following, as shown in List 13.2.

List 13.2 Jar files to be added to your `classpath` variable

```
JENA_INSTALL_DIR\arq-extra.jar
JENA_INSTALL_DIR\arq.jar
JENA_INSTALL_DIR\icu4j_3_4.jar
JENA_INSTALL_DIR\iri.jar
JENA_INSTALL_DIR\jena.jar
JENA_INSTALL_DIR\jenatest.jar
JENA_INSTALL_DIR\json.jar
JENA_INSTALL_DIR\junit-4.5.jar
JENA_INSTALL_DIR\log4j-1.2.12.jar
JENA_INSTALL_DIR\lucene-core-2.3.1.jar
JENA_INSTALL_DIR\slf4j-api-1.5.6.jar
JENA_INSTALL_DIR\slf4j-log4j12-1.5.6.jar
JENA_INSTALL_DIR\stax-api-1.0.jar
JENA_INSTALL_DIR\wstx-asl-3.0.0.jar
JENA_INSTALL_DIR\xercesImpl.jar
```

where `JENA_INSTALL_DIR` is where you have installed (unzipped) your Jena system. In my case, the above becomes the ones in List 13.3.

List 13.3 Jar files to be added, with specific path name

```
C:\liyang\DevApp\Jena-2.6.0\lib\arq-extra.jar
C:\liyang\DevApp\Jena-2.6.0\lib\arq.jar
C:\liyang\DevApp\Jena-2.6.0\lib\icu4j_3_4.jar
C:\liyang\DevApp\Jena-2.6.0\lib\iri.jar
C:\liyang\DevApp\Jena-2.6.0\lib\jena.jar
C:\liyang\DevApp\Jena-2.6.0\lib\jenatest.jar
C:\liyang\DevApp\Jena-2.6.0\lib\json.jar
C:\liyang\DevApp\Jena-2.6.0\lib\junit-4.5.jar
C:\liyang\DevApp\Jena-2.6.0\lib\log4j-1.2.12.jar
C:\liyang\DevApp\Jena-2.6.0\lib\lucene-core-2.3.1.jar
C:\liyang\DevApp\Jena-2.6.0\lib\slf4j-api-1.5.6.jar
C:\liyang\DevApp\Jena-2.6.0\lib\slf4j-log4j12-1.5.6.jar
C:\liyang\DevApp\Jena-2.6.0\lib\stax-api-1.0.jar
C:\liyang\DevApp\Jena-2.6.0\lib\wstx-asl-3.0.0.jar
C:\liyang\DevApp\Jena-2.6.0\lib\xercesImpl.jar
```

And you should substitute `JENA_INSTALL_DIR` variable with the location where you have installed your own copy of Jena.

Now to test whether you have done everything correctly, fire up a command line window, navigate to the location where you have installed your Jena package, and type `test.bat`. Waiting for about 80 seconds, you should see a screen as shown in Fig. 13.2.

If you can successfully finish this, your setup work is done and you are ready to use Jena in your development work.

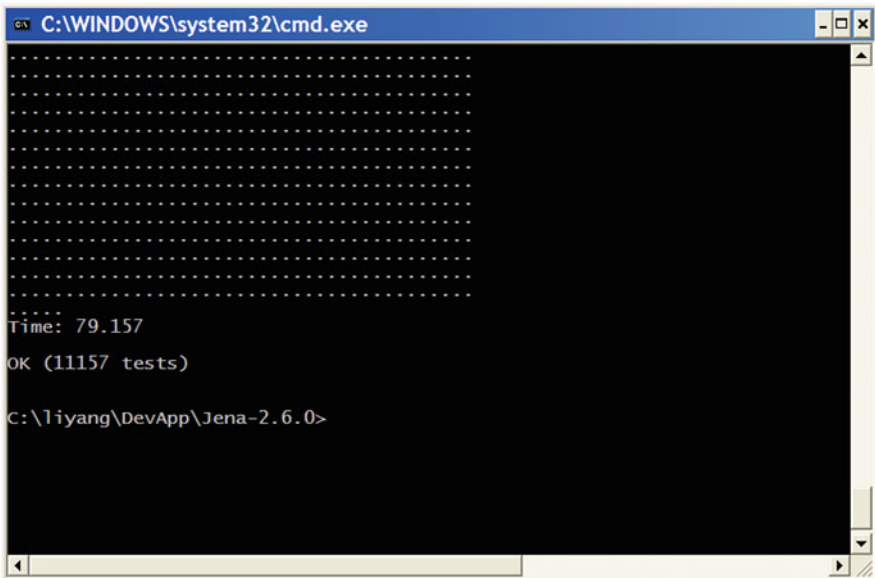


Fig. 13.2 Testing your Jena settings

13.1.3 Using Jena in Your Projects

Only setting up the Jena environment is not really enough, your project has to make use of the Jena package. This section assumes you are using Eclipse as your Java development tool and shows you how to use Jena framework in your Eclipse projects.

13.1.3.1 Using Jena in Eclipse

As far as Eclipse is concerned, the difference between a plain Java project and a Java project that uses Jena library is that Eclipse has to know where to find the Jena library files that the project refers to. Once it can locate the library files, it will be able to load the related class definitions from the library as the necessary supporting code to our project.

One way to accomplish this is to create a `lib` directory in our project workspace, copy Jena related library files to this `lib` directory, and then add this `lib` directory

into our project’s build path. This will work; however, a user library is a better solution to use.

In Eclipse, a user library is a user-defined library (a collection of jar files) that one can reference from any project. In other words, once we have configured a user library, we can use it in multiple different projects. Furthermore, if Jena releases a new updated version, updating the user library once will guarantee that all the projects using this library will all see the newly updated version. If we had created a library under each specific project workspace, we would have to copy the new version to every workspace that makes use of Jena.

To configure a user library, open up Eclipse and select `Window` from the menu bar. From the drop-down menu list, select `preferences`, which will bring up the `Preferences` dialogue window. In this window, open `Java` on the left navigation tree and then open `Build Path`. Once `Build Path` is open, select `User Libraries` as shown in Fig. 13.3.

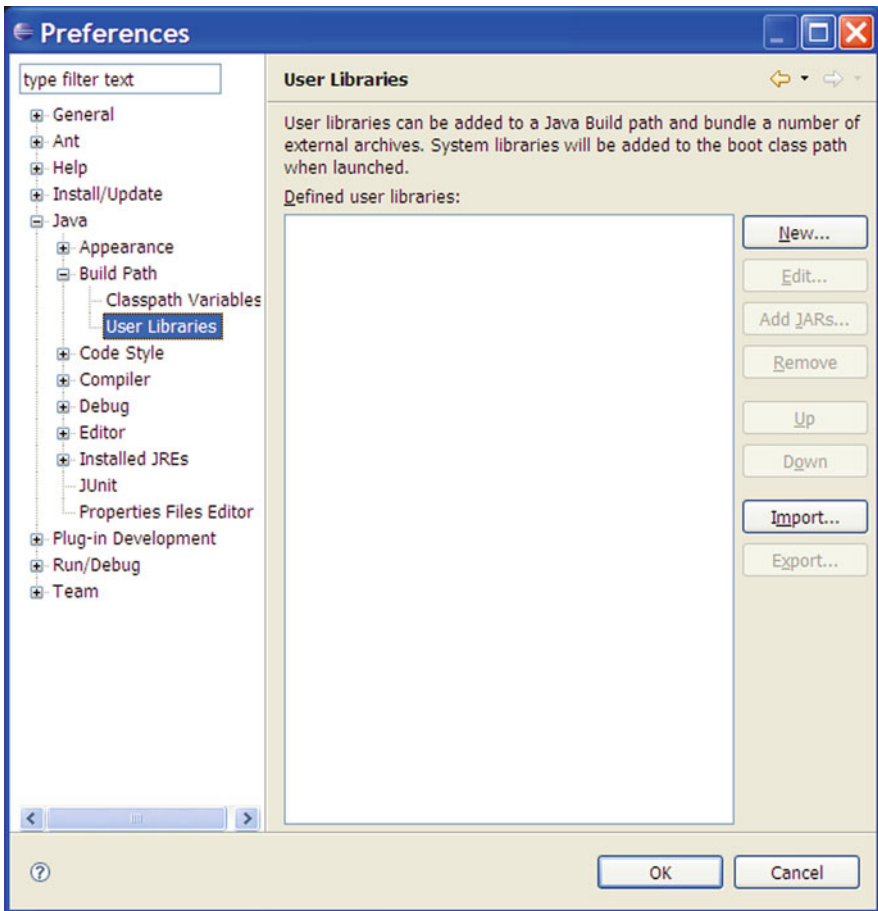


Fig. 13.3 Setup Jena framework as a user library in Eclipse

Now click `New` to create a new user library. To do so, in the pop-up window, enter `jena` as the library name, click `OK` to close this window. At this point, click `Add JARs...` will bring up a `JAR Selection` window. You can then navigate to the location where you have installed Jena (for me, this is `C:\liyang\DevApp\Jena-2.6.0\lib`), and select all of the `.jar` files in the `lib` directory, which will be enough for our project. Once you click `Open`, you should see the user library is correctly created, as shown in Fig. 13.4:

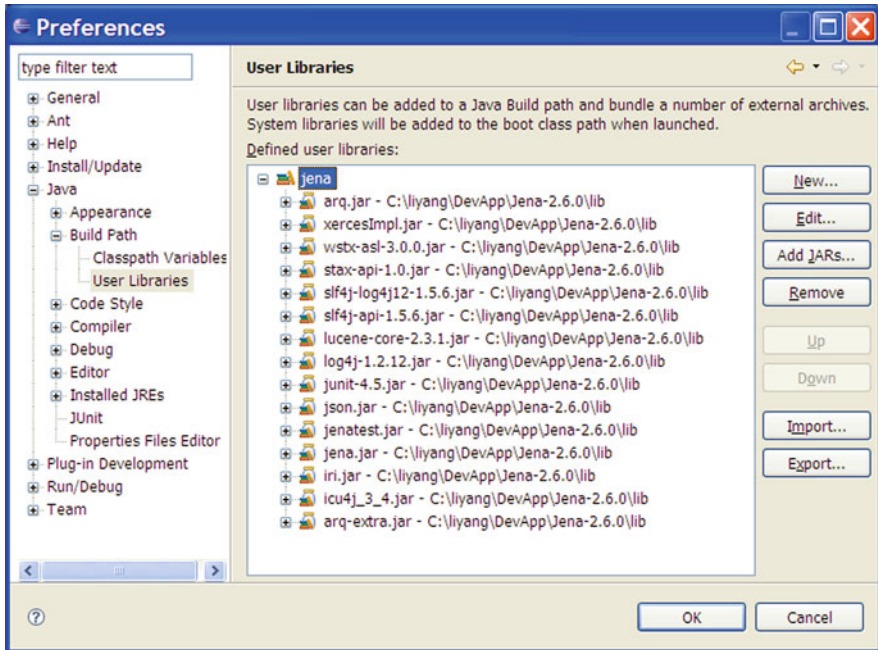


Fig. 13.4 Setup Jena framework as a user library in Eclipse (continued)

This is quite similar to the concept of symbolic link that we are familiar with when using Unix platform. The user library we just created simply contains a collection of links that point to those `*.jar` files in `lib/` directory under my Jena install directory; nothing is copied to my Eclipse workspace at all.

Now that we have configured a user library, we can start to use it in our project by adding this library to the build path of our project. To show how to do this, we do need to create a new project, and since it will be our very first real programming project in this book, we will call it `Hello World` project, and we will cover the details in the next section.

13.1.3.2 Hello World! from Semantic Web Application

We all know the importance of `Hello World` example, and we add it here so that you can see how to build a project that makes use of the Jena Semantic Web framework.

Our `HelloWorld` example will work like this: we will create a simple RDF document which contains only one RDF statement, and this statement has the following subject, property and object:

```
subject: http://example.org/test/message  
property: http://example.org/test/says  
object: Hello World!
```

Let us fire up Eclipse, create a new project called `HelloWorld`, and also define an empty class called `HelloWorld.java`.

Now, enter the class definition as shown in List 13.4, and it is fine if you currently don't understand the code at all.

List 13.4 `HelloWorld.java`

```
1: public class HelloWorld {  
2:  
3:     static private String nameSpace = "http://example.org/test/";  
4:  
5:     public static void main(String[] args) {  
6:         Model model = ModelFactory.createDefaultModel();  
7:  
8:         Resource subject =  
8a:             model.createResource(nameSpace + "message");  
9:         Property property =  
9a:             model.createProperty(nameSpace + "says");  
10:         subject.addProperty(property,  
10a:             "Hello World!", XSDDatatype.XSDstring);  
11:  
12:         model.write(System.out);  
13:     }  
14:  
15: }
```

Once you have entered the definition shown in List 13.4, Eclipse shows all the error signals. For example, it does not recognize `Model`, `ModelFactory`, `Resource` and `Property`. Clearly, these definitions are provided by the Jena package that we would like to use, and they are currently not visible to Eclipse. We now need to tell Eclipse that the user library we have just created contains the definitions that it needs, and we do this by adding the user library to the build path of our project.

To do so, right click the `HelloWorld` project node in `Project Explorer` window to bring up the project's pop-up menu. From this menu, select `Properties`, which will bring up `Properties for HelloWorld` window. On this window, navigate to `Java Build Path` and select `Libraries` tab, as shown in Fig. 13.5.

In this window, clicking `Add Library` will bring up the dialog window as shown in Fig. 13.6.

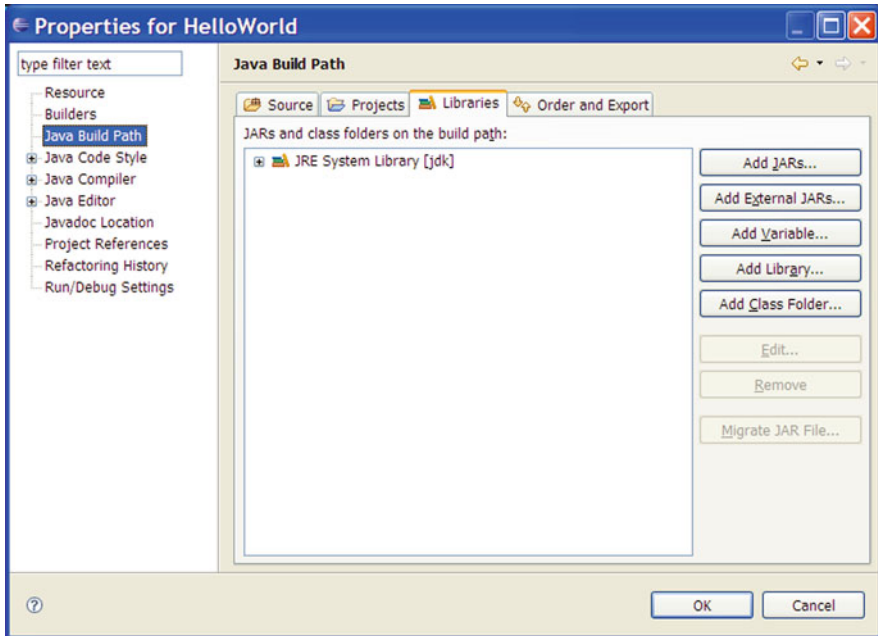


Fig. 13.5 Using Jena framework as a user library

Highlight `User Library`, and click `Next`, select `jena` as the user library to be added, and click `Finish` to add the user library into the build path, as shown in Fig. 13.7.

Click `OK` in `Properties for HelloWorld` window to finish this task. You will also note that `jena` user library shows up in `Project Explorer` window correctly.

However, the error signals still do not disappear. In fact, Eclipse is now waiting for us to use the appropriate `import` statements so that it will be able to find the definitions for class `Model`, `ModelFactory`, `Resource` and `Property`.

In fact, this could be fairly difficult for us: as beginners, we don't know where these definitions can be found in the library either. Yet the good news is, since we have used the correct user library, all you need to do is to click the error symbol (the red x on the left margin), and Eclipse will show you the right `import` statement to use. Try this out, and if you have done everything correctly, all the errors should be gone, and you should find the following `import` statements are being used, as shown in List 13.5.

List 13.5 `Import` statements in our `HelloWorld` project

```
import com.hp.hpl.jena.datatypes.xsd.XSDDatatype;
import com.hp.hpl.jena.rdf.model.Model;
```

```
import com.hp.hpl.jena.rdf.model.ModelFactory;
import com.hp.hpl.jena.rdf.model.Property;
import com.hp.hpl.jena.rdf.model.Resource;
```

Now, run the project, you should be able to see the result as shown in List 13.6.

List 13.6 HelloWorld output

```
<rdf:RDF
  xmlns:j.0="http://example.org/test/"
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#" >
  <rdf:Description rdf:about="http://example.org/test/message">
    <j.0:says rdf:datatype=
      "http://www.w3.org/2001/XMLSchema#string">
      Hello World!
    </j.0:says>
  </rdf:Description>
</rdf:RDF>
```

And congratulations – this is your first Semantic Web application developed using Jena.

Last thing before we move on: the above RDF model has only one statement, and we can make it look much better: change line 12 on List 13.4 to make it look like

```
12: model.write(System.out, "Turtle");
```

and run the project again, you will get a better output as shown in List 13.7.

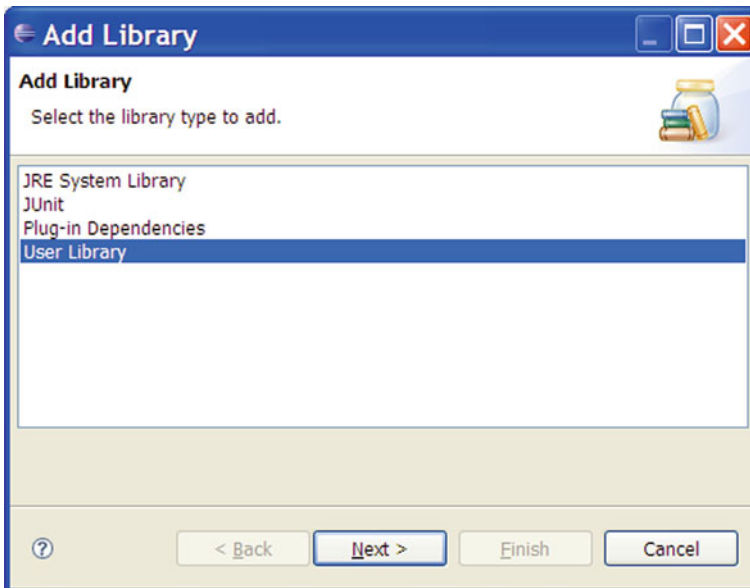


Fig. 13.6 Using Jena framework as a user library (continued)

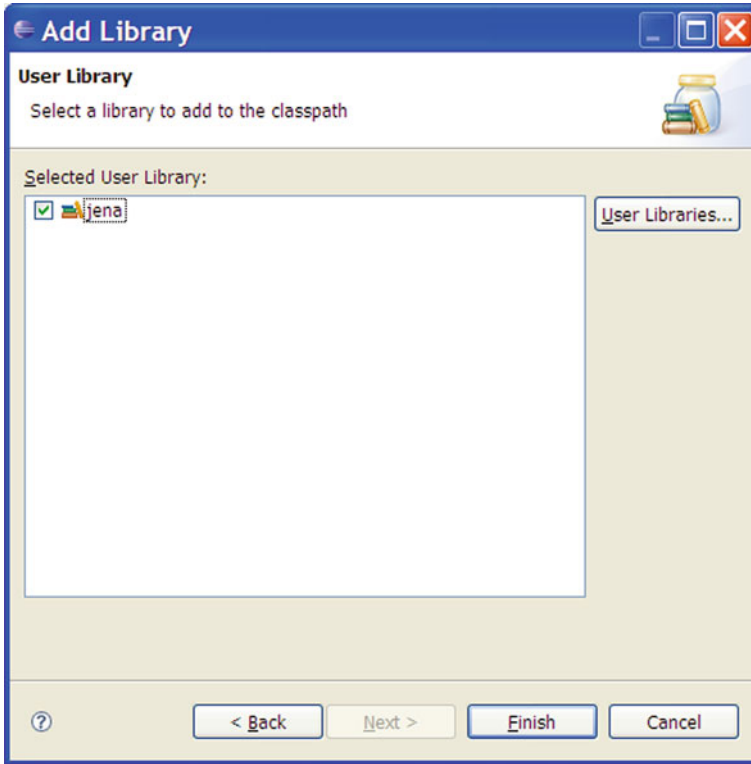


Fig. 13.7 Using Jena framework as a user library (final step)

List 13.7 A better output from HelloWorld project

```
<http://example.org/test/message>
  <http://example.org/test/says>
    "Hello World!"^^<http://www.w3.org/2001/XMLSchema#string>.
```

13.2 Basic RDF Model Operations

In this section, we will use Jena to accomplish some basic functionalities. The goal is to get you familiar with Jena. To make things simpler, we will be using in-memory RDF models in this section, meaning that we will either create the RDF model in memory or read it into memory from a given URL or a file system. Persistent RDF models will be covered in the next section.

13.2.1 Creating an RDF Model

In this section, we will create an empty RDF model from scratch and then add RDF statements to it. For the purpose of testing, we are going to create a model that represents my own FOAF document as shown in List 13.8 (note that it has been changed a little bit to make it easier to work with).

List 13.8 My FOAF document

```

1: <?xml version="1.0" encoding="UTF-8"?>
2: <rdf:RDF
3:     xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
4:     xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
5:     xmlns:foaf="http://xmlns.com/foaf/0.1/">
6:
7:   <rdf:Description
8:     rdf:about="http://www.liyangyu.com/foaf.rdf#liyang">
9:     <foaf:name>liyang yu</foaf:name>
10:    <foaf:title>Dr</foaf:title>
11:    <foaf:givenname>liyang</foaf:givenname>
12:    <foaf:family_name>yu</foaf:family_name>
13:    <foaf:mbox rdf:resource="mailto:liyang910@yahoo.com"/>
14:    <foaf:homepage rdf:resource="http://www.liyangyu.com"/>
15:    <foaf:workplaceHomepage
16:      rdf:resource="http://www.delta.com"/>
17:    <rdf:type
18:      rdf:resource="http://xmlns.com/foaf/0.1/Person"/>
19:
20:    <foaf:knows>
21:      <!-- the following is for testing purpose -->
22:      <foaf:Person>
23:        <foaf:mbox
24:          rdf:resource="mailto:libby.miller@bristol.ac.uk"/>
25:        <foaf:homepage
26:          rdf:resource="http://www.ilrt.bris.ac.uk/~ecemm"/>
27:      </foaf:Person>
28:    </foaf:knows>
29:
30:    <foaf:topic_interest
31:      rdf:resource="http://dbpedia.org/resource/Semantic_Web"/>
32:  </rdf:Description>
33: </rdf:RDF>

```

And the source code to accomplish this is shown in List 13.9.

List 13.9 Create a new RDF model and add statements to it

```

1: import java.io.PrintWriter;
2: import com.hp.hpl.jena.rdf.model.Model;
3: import com.hp.hpl.jena.rdf.model.ModelFactory;

```

```

4: import com.hp.hpl.jena.rdf.model.Resource;
5: import com.hp.hpl.jena.sparql.vocabulary.FOAF;
6: import com.hp.hpl.jena.vocabulary.RDF;
7: import com.hp.hpl.jena.vocabulary.RDFS;
8:
9: public class MyFOAFModel {
10:
11:     public static void main(String[] args) {
12:
13:         Model model = ModelFactory.createDefaultModel();
14:         model.setNsPrefix("rdfs",RDFS.getURI());
15:         model.setNsPrefix("foaf",FOAF.getURI());
16:
17:         Resource subject = model.createResource
17a:             ("http://www.liyangyu.com/foaf.rdf#liyang");
18:
19:         subject.addProperty(FOAF.name,"liyang yu");
20:         subject.addProperty(FOAF.title,"Dr");
21:         subject.addProperty(FOAF.givenname,"liyang");
22:         subject.addProperty(FOAF.family_name,"yu");
23:         subject.addProperty(FOAF.mbox,
23a:             model.createResource("mailto:liyang910@yahoo.com"));
24:         subject.addProperty(FOAF.homepage,
24a:             model.createResource("http://www.liyangyu.com"));
25:         subject.addProperty(FOAF.workplaceHomepage,
25a:             model.createResource("http://www.delta.com"));
26:         subject.addProperty(FOAF.topic_interest,
26a:             model.createResource
26b:             ("http://dbpedia.org/resource/Semantic_Web"));
27:         subject.addProperty(RDF.type,FOAF.Person);
28:
29:         Resource blankSubject = model.createResource();
30:         blankSubject.addProperty(RDF.type,FOAF.Person);
31:         blankSubject.addProperty(FOAF.mbox,
31a:             model.createResource
31b:             ("mailto:libby.miller@bristol.ac.uk"));
32:         blankSubject.addProperty(FOAF.homepage,
32a:             model.createResource
32b:             ("http://www.ilrt.bris.ac.uk/~ecemm/"));
33:         subject.addProperty(FOAF.knows,blankSubject);
34:
35:         model.write(System.out);
36:     }
37:
38: }

```

First thing to remember is that in Jena's world, `ModelFactory` class is the preferred way when it comes to creating different types of RDF models. For our purpose in this example, we want an empty, in-memory model, therefore `ModelFactory.createDefaultModel()` is the method to call, as shown in line

13 of List 13.9. This method returns an instance of class `Model`, which represents the empty RDF model we have just created. At this point, we can start to add statements into this model.

To add a statement into an RDF model, the first thing to do is to create a statement. In Jena, the subject of a statement is always represented by an instance of `Resource` class, the predicate is represented by an instance of `Property` class, and the object is either a `Resource` instance or a literal value, which is represented by an instance of `Literal` class. All of these classes, namely `Resource` class, `Property` class, and `Literal` class, share a common interface called `RDFNode`. At this point, you should be able to tell how Jena's class hierarchy maps to the related concepts in RDF world.

Now, to create a statement, we create the subject first. One way to do this is to call `createResource()` method provided by `Model` class, as shown in line 17, and we pass in the URI of the subject so that it can be created with the given URI as its identifier.

Once we have the subject, we can create a statement by calling method `Resource.addProperty()`. This method directly creates a statement in the model with the `Resource` as its subject. The method takes two parameters, a `Property` instance representing the predicate of the statement and the statement's object. Note that `addProperty()` method is overloaded in multiple forms, and one overload takes an `RDFNode` as its object, so a `Resource` or a `Literal` can be used. There are also other overloads that take a literal represented by a Java primitive or a string, as we will see next.

With this said, line 19 should be easy to understand: it directly inserts a statement into the model with subject as its subject, `foaf:name` as its property, and string literal "liyong yu" as its object. Note that Jena provides support for some popular vocabularies, and FOAF ontology is one of these ontologies. In this case, `FOAF.name` returns a `Property` instance that represents `foaf:name` property.

As a side note, if Jena did not support FOAF ontology, line 19 could have been written as two separate lines as given below:

```
Property nameProperty =
    model.createProperty("http://xmlns.com/foaf/0.1/name");
subject.addProperty(nameProperty, "liyong yu");
```

or a more concise form would look like

```
subject.addProperty
(model.createProperty("http://xmlns.com/foaf/0.1/name"),
 "liyong yu");
```

Similarly, lines 20–22 are all easy to understand. Line 20 inserts a statement saying our subject has a `foaf:title` property whose value is "Dr," line 21 adds a statement saying our subject has a `foaf:givenname` property whose value is "liyong," and line 22 maps to a statement saying our

subject has a `foaf:family_name` property whose value is “yu.” And obviously, the statements created by lines 19–22 are corresponding to lines 8–11 in List 13.8.

Note that lines 19–22 are all adding statements whose objects are having string literals as their values. Lines 23–26, on the other hand, are using resources as the values of their objects. For example, line 23 inserts a statement which has another resource as its object, and again, `createResource()` is called to create this resource. Also, statements created by lines 23–26 map to lines 12–14 and 25 in List 13.8.

Line 27 is also quite straightforward: it creates a statement that maps to line 15 in List 13.8. Again, since Jena supports FOAF and RDF vocabulary, `RDF.type` returns `rdf:type` property and `FOAF.Person` returns `foaf:Person` resource.

Lines 29–32 can be understood together with lines 19–22 in List 13.8, which defines a blank node that represents an instance of `foaf:Person` class. To create this blank node, `createResource()` method is called without any parameters being passed in, as shown in line 29.

Once this blank resource is created, we can add statements that use this resource as their subject. Line 30 inserts a statement saying this blank resource is an instance of `foaf:Person` and line 31 says this bland node has a `foaf:mbox` property who is using another resource as its value. Finally, line 32 adds a statement about its `foaf:homepage` property. Again, lines 29–32 map to lines 19–22 in List 13.8.

Line 33 in List 13.9 inserts into our model the last statement which expresses the fact that the subject `foaf:the` knows the above blank node, and at this point, it should be quite easy to understand as well.

Now, we have finished creating a simple in-memory RDF model that represents the document shown in List 13.8. To see this model, line 35 is used, which simply prints out the model on the screen so that we can take a look.

If you run the java class shown in List 13.9, you should get the output as shown in List 13.10.

List 13.10 Output generated by List 13.9

```

1: <rdf:RDF
2:     xmlns:foaf="http://xmlns.com/foaf/0.1/"
3:     xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
4:     xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#" >
5:   <rdf:Description
5a:     rdf:about="http://www.liyangyu.com/foaf.rdf#liyang">
6:     <foaf:name>liyang yu</foaf:name>
7:     <foaf:mbox rdf:resource="mailto:liyang910@yahoo.com"/>
8:     <rdf:type
8a:     rdf:resource="http://xmlns.com/foaf/0.1/Person"/>
9:     <foaf:family_name>yu</foaf:family_name>
10:    <foaf:knows rdf:nodeID="A0"/>
11:    <foaf:givenname>liyang</foaf:givenname>
12:    <foaf:title>Dr</foaf:title>

```

```

13:   <foaf:topic_interest rdf:resource=
13a:       "http://dbpedia.org/resource/Semantic_Web"/>
14:   <foaf:homepage rdf:resource="http://www.liyangyu.com"/>
15:   <foaf:workplaceHomepage
15a:       rdf:resource="http://www.delta.com"/>
16: </rdf:Description>
17: <rdf:Description rdf:nodeID="A0">
18:   <foaf:homepage
18a:       rdf:resource="http://www.ilrt.bris.ac.uk/~ecemm"/>
19:   <foaf:mbox
19a:       rdf:resource="mailto:libby.miller@bristol.ac.uk"/>
20:   <rdf:type
20a:       rdf:resource="http://xmlns.com/foaf/0.1/Person"/>
21: </rdf:Description>
22: </rdf:RDF>

```

Note that List 13.10 is not the best output that we can have. For one thing, the blank node has been assigned a node ID (line 17), so it is not blank anymore. In fact, a better output can be obtained by replacing line 35 with the following line which makes use of RDF/XML-ABBREV parameter:

```
model.write(new PrintWriter(System.out), "RDF/XML-ABBREV");
```

And now, run the code again, and you will see the output as shown in List 13.11.

List 13.11 A better output generated from List 13.9

```

1: <rdf:RDF
2:   xmlns:foaf="http://xmlns.com/foaf/0.1/"
3:   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
4:   xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
5:   <foaf:Person
5a:     rdf:about="http://www.liyangyu.com/foaf.rdf#liyang">
6:     <foaf:name>liyang yu</foaf:name>
7:     <foaf:mbox rdf:resource="mailto:liyang910@yahoo.com"/>
8:     <foaf:knows>
9:       <foaf:Person>
10:        <foaf:homepage
10a:          rdf:resource="http://www.ilrt.bris.ac.uk/~ecemm"/>
11:        <foaf:mbox
11a:          rdf:resource="mailto:libby.miller@bristol.ac.uk"/>
12:      </foaf:Person>
13:    </foaf:knows>
14:    <foaf:family_name>yu</foaf:family_name>
15:    <foaf:givenname>liyang</foaf:givenname>
16:    <foaf:title>Dr</foaf:title>
17:    <foaf:topic_interest
17a:      rdf:resource="http://dbpedia.org/resource/Semantic_Web"/>
18:    <foaf:homepage rdf:resource="http://www.liyangyu.com"/>

```



```

19:     <foaf:workplaceHomepage
19a:         rdf:resource="http://www.delta.com"/>
20: </foaf:Person>
21: </rdf:RDF>

```

Before we move on, note that we have been using `addProperty()` method to create statements. In fact, statements can also be created directly on the model by calling `Model.createStatement()` with the subject, predicate, and object of the triple. For example,

```

Statement statement =
    model.createStatement(mySubject, myProperty, myObject);

```

However, a main difference between these two methods of creating statements is that creating a statement in this way doesn't add it into the model. If you want to add it into the model, call `Model.add()` with the created statement:

```

// but remember to add the created statement to the model
model.add(statement);

```

At this point, we are done using Jena to create a simple RDF model. We have seen the use of important classes such as `ModelFactory`, `Model`, `Resource` and `Property`; we have also seen how to create resources and properties and how to insert statements into an existing RDF model.

Obviously, creating RDF models like what we have done here is not quite scalable. A large RDF model will simply require too much coding work and maintenance work; there has to be other ways to build RDF models.

In real practice, most of the RDF documents are generated automatically, from a given database table, for instance. The generated RDF documents can then be read into memory for more processing work. Therefore, learning how to read in an RDF model is also important, and let us cover this in the next section.

13.2.2 Reading an RDF Model

Compared to creating an RDF model as we have discussed in the previous section, reading a given RDF document into memory is probably a more frequently used operation. As we will see in the coming sections and chapters of this book, for many applications, we often need to read a certain RDF document located at a given URL into memory before we can do anything about it. This can be understood as downloading a machine-readable document from the Web.

In the case where there is no state persistence necessary, implementing this download action is quite straightforward. List 13.12 shows how to read my FOAF document from the following URL:

```
http://liyangyu.com/foaf.rdf
```

and to show we have correctly read the document, we also write it out in Turtle format.

List 13.12 Reading an RDF document from a given URL

```

1: package test;
2:
3: import com.hp.hpl.jena.rdf.model.Model;
4: import com.hp.hpl.jena.rdf.model.ModelFactory;
5: import com.hp.hpl.jena.util.FileManager;
6:
7: public class ReadRDFModel {
8:
9:     public static final String MY_FOAF_FILE =
9a:         "http://liyangyu.com/foaf.rdf";
10:
11:     public static void main( String[] args ) {
12:
13:         Model model = ModelFactory.createDefaultModel();
14:         model.read(MY_FOAF_FILE);
15:         model.write(System.out, "N3");
16:     }
17: }

```

List 13.12 makes use of a basic `Model` form that is created by calling `createDefaultModel()` method. This model uses an in-memory storage model and has no inference or any other reasoning power. In the case where we do need to have inference capabilities, we will have to create the model in some other ways, as we will see in a later section. Also note that line 15 uses `N3` as the format parameter since `Turtle` is a subset of `Notation 3 (N3)`. List 13.13 shows the output from List 13.12.

List 13.13 Output generated from List 13.12

```

@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .

<http://www.liyangyu.com/foaf.rdf#liyang>
  a foaf:Person ;
  foaf:family_name "yu"@en ;
  foaf:givenname "liyang"@en ;
  foaf:homepage <http://www.liyangyu.com> ;
  foaf:knows
    [ a foaf:Person ;
      foaf:homepage <http://www.ilrt.bris.ac.uk/~ecemm/> ;
      foaf:mbox <mailto:libby.miller@bristol.ac.uk>
    ] ;
  foaf:mbox_sha1sum
    "1613a9c3ec8b18271a8fe1f79537a7b08803d896"@en ;
  foaf:name "liyang yu"@en ;
  foaf:title "Dr"@en ;

```

```
foaf:topic_interest
    <http://dbpedia.org/resource/Semantic_Web> ;
foaf:workplaceHomepage
    <http://www.delta.com> .
```

It is also possible to load an RDF document into memory from a local file system. As shown by line 15 of List 13.14, `FileManager` class is used to finish the task.

List 13.14 Read an RDF document from local file system

```
1: package test;
2:
3: import com.hp.hpl.jena.rdf.model.Model;
4: import com.hp.hpl.jena.rdf.model.ModelFactory;
5: import com.hp.hpl.jena.util.FileManager;
6:
7: public class ReadRDFModel {
8:
9:     public static final String MY_FOAF_FILE =
9a:         "c:/liyang/myWebsite/currentPage/foaf.rdf";
10:     // public static final String MY_FOAF_FILE =
10a:     //     "http://liyangyu.com/foaf.rdf";
11:
12:     public static void main( String[] args ) {
13:
14:         Model model = ModelFactory.createDefaultModel();
15:         FileManager.get().readModel(model,MY_FOAF_FILE);
16:         // model.read(MY_FOAF_FILE);
17:         model.write(System.out, "N3");
18:     }
19: }
```

13.2.3 Understanding an RDF Model

Now that we have an RDF model in memory, we can continue to use Jena's library to know more about it. For example, knowing the answers to the following questions can be helpful:

- What types (classes) are used in this model?
- For each type used, what instances are created/included in this model?
- What are the namespaces used in this model?

You probably can list more questions here. For now, List 13.15 shows the code we can use to understand a given RDF document.

List 13.15 Understanding an RDF document

```

1: package test;
2:
3: import java.util.Iterator;
4: import java.util.Map;
5:
6: import com.hp.hpl.jena.rdf.model.Model;
7: import com.hp.hpl.jena.rdf.model.ModelFactory;
8: import com.hp.hpl.jena.rdf.model.NodeIterator;
9: import com.hp.hpl.jena.rdf.model.ResIterator;
10: import com.hp.hpl.jena.rdf.model.Resource;
11: import com.hp.hpl.jena.vocabulary.RDF;
12:
13: public class ReadRDFModel {
14:
15:     public static final String RDF_FILE =
15a:         "http://liyanguy.com/foaf.rdf";
16:
17:     public static void main( String[] args ) {
18:
19:         Model model = ModelFactory.createDefaultModel();
20:         model.read(RDF_FILE);
21:         // model.write(System.out, "N3");
22:
23:         // show all the namespaces in the model
24:         Iterator prefixNsPairs =
24a:             model.getNsPrefixMap().entrySet().iterator();
25:         while ( prefixNsPairs.hasNext() ) {
26:             Map.Entry entry = (Map.Entry) prefixNsPairs.next();
27:             System.out.print("prefix:" + entry.getKey());
28:             System.out.println(", namespace:" + entry.getValue());
29:         }
30:
31:         // show all the classes and their instances
32:         System.out.println("the following types/classes have
32a:         been used in this RDF document(with their instances):");
33:         NodeIterator classes =
33a:             model.listObjectsOfProperty(RDF.type);
34:         while ( classes.hasNext() ) {
35:             Resource typeRes = (Resource) classes.next();
36:             System.out.println("(class/type)" + typeRes.getURI());
37:             ResIterator resources =
37a:                 model.listResourcesWithProperty(RDF.type, typeRes);
38:             while ( resources.hasNext() ) {
39:                 Resource instanceRes = resources.nextResource();
40:                 if ( instanceRes.isAnon() ) {
41:                     System.out.println(" [anonymous instance] " +
41a:                                     instanceRes.getId());
42:                 } else {

```

```

43:             System.out.println(" [instance] " +
43a:                                     instanceRes.getURI());
44:         }
45:     }
46: }
47:
48: }
49: }

```

Line 15 specifies the RDF document we want to read, and lines 19 and 20 actually read the RDF document into our in-memory model. Lines 23–29 show a summary of the namespaces (and their prefixes) that have been used in this document. Method `getNsPrefixMap()` (line 24) is the key when it comes to namespaces. This method returns a collection of key–value pairs for each one of these pairs, the key being the prefix and the value being the namespace. As you see, line 27 retrieves the prefix and line 28 retrieves the namespace itself.

To find all the types/classes that are referenced in this model, method `listObjectsOfProperty()` is used (line 33). This method takes a property as its input parameter (in this case, this property is given by `RDF.type`) and it visits all the statements in the model and tries to match this pattern:

```
subject rdf:type object
```

Obviously, any statement following this pattern is there to assert the type of a given resource (represented by `subject`), so the `object` component must represent a class definition.

Once executed, method `listObjectsOfProperty()` returns a group of types (this group will have only one member if only one class is ever used in the whole model). For each class in this group, we try to find all the resources that are instances of this class. This is done by calling `listResourcesWithProperty()` method in line 37. This method takes a property instance and object type as its input parameters, and in our case, `RDF.type` is the property instance and `typeRes` represents the object type. It then tries to match all the statements that have the following pattern:

```
subject rdf:type typeRes
```

Once a match is found, the `subject` is collected and becomes one of the returned resources when the call is finished. Our code then lists out all these resources, together with their type information (lines 38–44).

Note that it could be true that a given resource is represented by a blank node, yet it is still an instance of a given type. To take this into account, we use `isAnon()` method to test if a given resource is a blank node (line 40). If a resource is represented by a blank node, method `getId()` is called to get its identifier; otherwise, method `getURI()` is used (line 43).

Now, let us run this code against my FOAF document, and List 13.16 shows the result.

List 13.16 Output generated from List 13.15 with my FOAF document

```

1: prefix:rdfs, namespace:http://www.w3.org/2000/01/rdf-schema#
2: prefix:rdf,
2a:     namespace:http://www.w3.org/1999/02/22-rdf-syntax-ns#
3: prefix:foaf, namespace:http://xmlns.com/foaf/0.1/
4: the following types/classes have been used in this RDF
4a:  document (with their in-stances):
5: (class/type) http://xmlns.com/foaf/0.1/Person
6:  [anonymous instance] -6aa9a0b:12454881133:-8000
7:  [instance] http://www.liyangyu.com/foaf.rdf#liyng

```

As you can see, lines 1–3 show the namespaces and their prefixes that are referenced by my FOAF document, line 5 shows the only class (`foaf:Person`) used in this document, and lines 6 and 7 list the two instances that actually have the type `foaf:Person`. Therefore, we have now learned that this document describes some instances whose type is `foaf:Person`, and it has also included two such instances.

Let us try some other file that has more content than my simply FOAF document. The following RDF document seems to be a good choice:

```
http://dbpedia.org/data/Roger_Federer.rdf
```

You do need to change line 15 of List 13.15 to make it look like

```
public static final String RDF_FILE =
    "http://dbpedia.org/data/Roger_Federer.rdf";
```

And List 13.17 shows part of the result.

List 13.17 Understanding Federer’s RDF document generated by DBpedia

```

prefix:dbpprop, namespace:http://dbpedia.org/property/
prefix:dbpedia-owl, namespace:http://dbpedia.org/ontology/
prefix:dc, namespace:http://purl.org/dc/elements/1.1/
prefix:rdfs, namespace:http://www.w3.org/2000/01/rdf-schema#
prefix:rdf, namespace:http://www.w3.org/1999/02/22-rdf-syntax-ns#
prefix:foaf, namespace:http://xmlns.com/foaf/0.1/
prefix:owl, namespace:http://www.w3.org/2002/07/owl#
prefix:skos, namespace:http://www.w3.org/2004/02/skos/core#
the following types/classes have been used in this RDF document
(with their instances):
(class/type) http://dbpedia.org/ontology/Person
  [instance] http://dbpedia.org/resource/Roger_Federer
(class/type) http://dbpedia.org/class/yago/USOpenChampions
  [instance] http://dbpedia.org/resource/Roger_Federer
(class/type)
http://dbpedia.org/class/yago/AustralianOpenChampions
  [instance] http://dbpedia.org/resource/Roger_Federer
(class/type) http://dbpedia.org/ontology/Athlete
  [instance] http://dbpedia.org/resource/Roger_Federer
(class/type) http://dbpedia.org/ontology/TennisPlayer
  [instance] http://dbpedia.org/resource/Roger_Federer

```

```
(class/type) http://dbpedia.org/class/yago/LivingPeople
 [instance] http://dbpedia.org/resource/Roger_Federer
(class/type) http://xmlns.com/foaf/0.1/Person
 [instance] http://dbpedia.org/resource/Roger_Federer
```

Based on this output, without reading the RDF document itself, we have obtained quite some information about it already.

Besides understanding the type information for the resources, we can also inspect the properties defined for them. Among all the properties, the following three are of particular interest to us:

```
owl:sameAs
rdfs:seeAlso
rdfs:isDefinedBy
```

since we can follow these properties to find more about their subjects. In fact, this is the idea behind the Follow-Your-Nose algorithm, which we will see more in the next chapter. Note that other properties can also be used in Follow-Your-Nose search, so the above list can grow, but for now, these three are the most obvious ones.

Based on the above discussion, our last query is to find all the resources that have the above properties. List 13.18 uses `owl:sameAs` as an example to show how the search is done.

List 13.18 Use `owl:sameAs` to find links

```
1: package test;
2:
3: import java.util.Iterator;
4: import java.util.Map;
5:
6: import com.hp.hpl.jena.rdf.model.Model;
7: import com.hp.hpl.jena.rdf.model.ModelFactory;
8: import com.hp.hpl.jena.rdf.model.NodeIterator;
9: import com.hp.hpl.jena.rdf.model.RDFNode;
10: import com.hp.hpl.jena.rdf.model.ResIterator;
11: import com.hp.hpl.jena.rdf.model.Resource;
12: import com.hp.hpl.jena.rdf.model.Statement;
13: import com.hp.hpl.jena.rdf.model.StmtIterator;
14: import com.hp.hpl.jena.vocabulary.OWL;
15: import com.hp.hpl.jena.vocabulary.RDF;
16:
17: public class ReadRDFModel {
18:
19:     public static final String RDF_FILE =
19a:         "http://dbpedia.org/data/Roger_Federer.rdf";
20:
21:     public static void main( String[] args ) {
22:
23:         Model model = ModelFactory.createDefaultModel();
24:         model.read(RDF_FILE);
25:         // model.write(System.out, "N3");
26:
```

```

27:      // show all the namespaces in the model
...
34:
35:      // show all the classes and their instances
...
51:
52:      // show all instances that have a owl:sameAs property
53:      System.out.println("\nfollowing instances have
53a:                          owl:sameAs property:");
54:      StmtIterator statements = model.listStatements
54a:                          ((Resource)null,OWL.sameAs,(RDFNode)null);
55:      while ( statements.hasNext() ) {
56:          Statement statement = statements.nextStatement();
57:          Resource subject = statement.getSubject();
58:          if ( subject.isAnon() ) {
59:              System.out.print(" (" + subject.getId() + ")");
60:          } else {
61:              System.out.print(" (" + subject.getURI() + ")");
62:          }
63:          System.out.print(" OWL.sameAs ");
64:          Resource object = (Resource)(statement.getObject());
65:          if ( object.isAnon() ) {
66:              System.out.print("(" + object + ")");
67:          } else if ( object.isLiteral() ) {
68:              System.out.print("(" + object.toString() + ")");
69:          } else if ( object.isResource() ) {
70:              System.out.print("(" + object.getURI() + ")");
71:          }
72:          System.out.println();
73:      }
74:
75:  }
76: }

```

And the related code is from lines 52–72 (the rest are the same as in List 13.15). The key method to call is `listStatements()` as in line 54. Since we pass in null value for both the subject and the object, the method tries to find all the statements that use `owl:sameAs` as the property, and the subjects and objects of these statements can be anything. This way, we will be able to find all the subjects and objects that are linked together by `owl:sameAs`. Lines 57–72 print the result in a more readable way.

Now run the code against Roger Federer's RDF document, we should see the result as shown in List 13.19.

List 13.19 Part of the output from List 13.18

```

the following instance(s) has/have owl:sameAs property:
  (http://dbpedia.org/resource/Roger_Federer) OWL.sameAs
  (http://rdf.freebase.com/ns/guid.9202a8c04000641f80000000019f52)
  (http://mpii.de/yago/resource/Roger_Federer) OWL.sameAs
  (http://dbpedia.org/resource/Roger_Federer)

```


We can get more understanding about a given RDF model by inspecting other properties that we are interested in, and with what you have learned here, it should be a fairly straightforward process.

Meanwhile, understand that accessing and querying a given RDF Model as what we have done here is considered to be a fairly low-level view of the RDF graph. Other query techniques, such as SPARQL query language, can provide more compact and powerful results, as we will show you in the next chapter. However, what you have learned here will help you to understand Jena more and will also give you some light-weighted querying methods without using SPARQL.

13.3 Handling Persistent RDF Models

13.3.1 *From In-memory Model to Persistent Model*

So far to this point, we have been working with in-memory models. These models are either created from scratch or populated from existing files. The files can be located in a local file system or can be downloaded from given URLs.

Although in-memory models are quite useful, they do have some disadvantages. To name a few,

- The RDF model has to be repopulated from scratch each time the application launches, thus requiring a longer start-up time.
- Any change made to the in-memory model will be lost when the application is shut down.
- Applications based upon in-memory models will not scale as we start to work with larger models.

A better solution is to store ontology and instance models in a database-backed RDF store, and then operate on the models just as we have discussed so far. This solution is called the *persistent model* solution, where the models are continually and transparently persisted with the backing store.

Jena is shipped with support for a collection of standard database systems, including MySQL, PostgreSQL, SQL Server, Oracle and Derby. In addition, Jena's database adapters use standard JDBC drivers to manage these database engines as triple stores, and Jena will create and manage its own table layout in the database systems; the details are hidden from the applications.

For developers like us, this means we can select one of the above database systems as the data store, and Jena hides the variations of SQL syntax in different databases by offering the related APIs that we can use. For example, when we call `listStatements()` on a database-backed model, Jena will construct the appropriate SQL query, execute it through the underlying database engine, and translate the query results into a `ResultSet` object that our application can manipulate to retrieve each `Statement` object.

With this solution, our RDF models will stay in the database regardless of whether the application is on or not; the application accesses these models via Jena APIs without loading the models into the memory. In addition, whatever the application has changed will stay in the models, and the next time that application starts, we will see the changes we have made from last time.

We will see the details of this solution in the next section. We also need to choose a database management system. In our case, we will go with MySQL, mainly due to the fact that it is freely available for different platforms.

13.3.2 *Setting Up MySQL*

In this section, we will setup MySQL on your machine. If you have MySQL already, you can skip this section and move on to the next section.

To setup MySQL, the first step is to download MySQL software and the JDBC driver. At this point, you should be able to find the downloadable files from the following URL:

<http://www.mysql.com/downloads/>

If this URL does not exist at the time you are reading the book, go to MySQL's homepage

<http://www.mysql.com/>

and try to find the download link from this official home page.

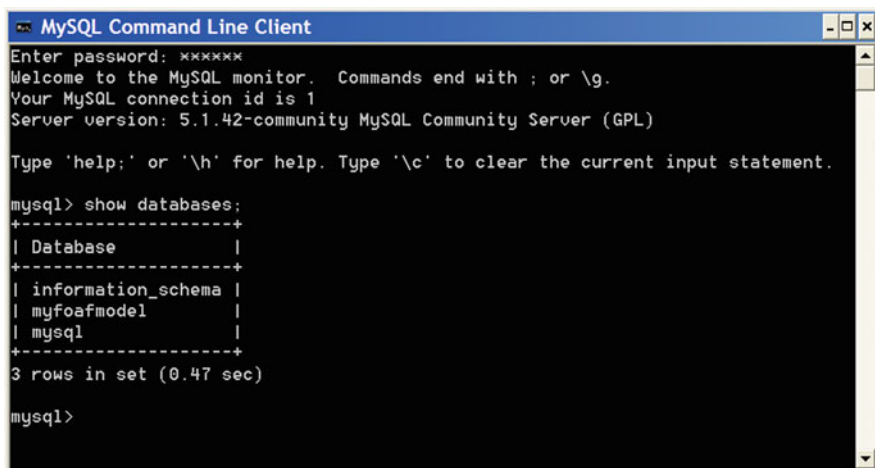
At this point, the freely downloadable version is called MySQL Community Server, and that is the one you should download. After the download is completed, you can double click the downloaded package to start installation on your machine.

Once the installation is finished, you will be asked to configure MySQL Server. For example, here you have a chance to configure the port number, which is defaulted to be 3306. You can either keep the default number or use another number, but make sure the number you are using is not in use by other server software on your machine.

Another configuration task is to specify the user ID and user password. For me, I used the given `root` user as my user name, and I entered `passwd` as my password. You can choose your combination, but you need to remember them, since they will be needed in your Java code.

Once you are done with the configuration step, you need to continue downloading the JDBC driver that goes together with MySQL. To do so, click `Connectors` link on the same download page; you will be presented with a page that includes a list of connectors. Within these connectors, you should download `Connector/J`, the driver for Java platform. This connector is simply a jar file that you need to use in your Java code, and you can simply add it to your user library as we have discussed early in this chapter.

Now, MySQL has been set up on your machine. To make sure MySQL database engine is working fine, you can fire up *MySQL Command Line Client*, and after entering the password, try to issue some SQL commands, as shown in Fig. 13.8. If you can do all these successfully, MySQL is correctly set up and running, and we are ready to move on.



```

MySQL Command Line Client
Enter password: *****
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 1
Server version: 5.1.42-community MySQL Community Server (GPL)

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> show databases;
+-----+
| Database |
+-----+
| information_schema |
| myfoafmodel |
| mysql |
+-----+
3 rows in set (0.47 sec)

mysql>

```

Fig. 13.8 Make sure MySQL is correctly set up

13.3.3 Database-Backed RDF Models

13.3.3.1 Single Persistent RDF Model

By far, we know `Model` interface is a key abstraction in Jena. It represents an RDF model, which has a collection of statements. There are several implementations of `Model` interface; each one of them is for a different type of model, such as an in-memory model, a file based model, an inferencing model, and a database-backed model.

For our purpose, we need the implementation for the database-backed model. This is the class called `ModelRDB`, and `createModelRDBMaker()` method call on `ModelFactory` class can answer a `ModelMaker` object that understands how to handle `ModelRDB` object.

List 13.20 Shows the code we can use to make my own FOAF document into a persistent RDF model in MySQL.

List 13.20 Load my FOAF document and make it a persistent RDF model in MySQL

```

1: import com.hp.hpl.jena.db.DBConnection;
2: import com.hp.hpl.jena.db.IDBConnection;
3: import com.hp.hpl.jena.rdf.model.Model;

```

```

4: import com.hp.hpl.jena.rdf.model.ModelFactory;
5: import com.hp.hpl.jena.rdf.model.ModelMaker;
6: import com.hp.hpl.jena.rdf.model.Property;
7: import com.hp.hpl.jena.rdf.model.Resource;
8: import com.hp.hpl.jena.rdf.model.Statement;
9: import com.hp.hpl.jena.rdf.model.StmtIterator;
10: import com.hp.hpl.jena.util.FileManager;
11: import com.hp.hpl.jena.util.PrintUtil;
12:
13: public class DBModelTester {
14:
15:     public static final String RDF_FILE =
15a:         "http://www.liyangyu.com/foaf.rdf";
16:     public static final String ONTOLOGY_FILE =
16a:         "http://xmlns.com/foaf/0.1/";
17:
18:     private static String className = "com.mysql.jdbc.Driver";
19:     private static String DB_URL =
19a:         "jdbc:mysql://localhost:3306/myFoafModel";
20:     private static String DB_USER = "root";
21:     private static String DB_PASSWD = "passwd";
22:     private static String DB_TYPE = "MySQL";
23:     private static String DOCUMENT_NAME = "myFoafRDF";
24:     private static String ONTOLOGY_NAME = "foaf.owl";
25:
26:     public static void main( String[] args ) {
27:
28:         IDBConnection conn = null;
29:         ModelMaker maker = null;
30:
31:         try {
32:             Class.forName(className);
33:             conn =
33a:                 new DBConnection(DB_URL,DB_USER,DB_PASSWD,DB_TYPE);
34:             } catch (Exception e) { e.printStackTrace(); }
35:
36:             maker = ModelFactory.createModelRDBMaker(conn);
37:             Model m = null;
38:
39:             if ( !maker.hasModel(DOCUMENT_NAME) == true ) {
40:                 System.out.println( "Loading instance
40a:                     document - one time only" );
41:                 m = maker.createModel(DOCUMENT_NAME);
42:                 FileManager.get().readModel(m,RDF_FILE);
43:             } else {
44:                 m = maker.getModel(DOCUMENT_NAME);
45:             }
46:             printStatements(m, null, null, null);
47:
48:             // close the connection
49:             try {
50:                 conn.close();

```

```

51:         } catch(Exception e) { e.printStackTrace(); }
52:
53:     }
54:
55:     private static void printStatements(Model m, Resource s,
55a:         Property p, Resource o) {
56:         for (StmtIterator I = m.listStatements(s,p,o);
56a:             I.hasNext(); ) {
57:             Statement stmt = I.nextStatement();
58:             System.out.println(" - " + PrintUtil.print(stmt));
59:         }
60:     }
61: }

```

The interesting lines in List 13.20 start from line 18, where the driver for MySQL is specified. If you have the experience of connecting to a backend database using Java platform, this line and the next couple of lines will look familiar to you. Line 19 specifies the database URL, which uses 3306 as the default port number. If you have specified other port number during the configuration process, you should use that port number instead of 3306. Also, `myFoafModel` is the name of the database we are going to create, and my FOAF document will stay in this database. You can certainly choose a name you like for your database.

Lines 20–22 specify the user name, the password, and the database type in order to create a connection to MySQL database engine. The user name, password we have selected at the setup time will be used here. Note that you have to use `MySQL` for `DB_TYPE`, since we have MySQL database as our backend database systems.

Lines 23 and 24 define the names of two RDF models we are going to load into our database: `myFoafRDF` is the name of the model that represents my own FOAF document, and `foaf.owl` is the name of the model that represents the FOAF ontology. List 13.20 will load only my own FOAF document into database, and the FOAF ontology will be handled later in the next section.

Lines 31–34 create the connection to MySQL backend database, and it is a fairly standard code you should use.

Line 36 is the key line. `createModelRDBMaker()` method call on `ModelFactory` class answers a `ModelMaker` object that understands how to handle RDF models and further change them into persistent graphs in the database. Note that we need to pass the database connection we have created in lines 31–34 to `createModelRDBMaker()` method so that the created `ModelMaker` object can operate in the database we have specified.

Once a `ModelMaker` object that connects to the backend database has been created, we can use it to load my FOAF document and make it a persistent RDF graph. To do so, line 39 checks if my FOAF document is already loaded into the database; if not, `createModel()` method on `ModelMaker` object is called to create a model that has the name specified by `DOCUMENT_NAME` string (line 41). This model is then populated by reading my FOAF file (line 42); in this case, this file can be obtained from the path given by line 15.

Note that this model creation and population process is executed only once (line 40). The second time you run the same code, my FOAF document is already in the database, and it will be mapped to a model directly (line 44). Clearly, this is the reason why we say my FOAF document now becomes a persistent model.

Once we reach line 46, we have an RDF model on hand, and we can do anything with it, as if it were an in-memory RDF model that we are familiar with from the previous sections. In our case, we simply print out all the statements contained in this model.

Lines 49–51 are some routine housekeeping work, and it is necessary to ensure a clean database shutdown which also helps to release system resources.

Now, what exactly is inside the backend database? To understand more about how this works, before you run the code in List 13.20, fire up the MySQL Command Line Client as shown in Fig. 13.8, and list all the databases that are currently in MySQL. For me, here is what I have:

```
mysql> show databases;
+-----+
| Database          |
+-----+
| information_schema |
| mysql             |
+-----+
```

and now, run the code in List 13.20. And if you are running it for the first time, you should see some output such as the following:

```
Loading instance document - one time only
- (http://www.liyangyu.com/foaf.rdf#liyang
  http://xmlns.com/foaf/0.1/name 'liyang yu')
- (http://www.liyangyu.com/foaf.rdf#liyang
  http://xmlns.com/foaf/0.1/title 'Dr')
- (http://www.liyangyu.com/foaf.rdf#liyang
  http://xmlns.com/foaf/0.1/givenname 'liyang')
...
```

Once the run is successfully finished, we can go back to MySQL Command Line Client window and list all the databases again:

```
mysql> show databases;
+-----+
| Database          |
+-----+
| information_schema |
| myfoafmodel      |
| mysql             |
+-----+
```

As you can see, `myfoafmodel` is now included in the database list. To see more about it, let us check out all the tables in this database:

```
mysql> show tables from myfoafmodel;
```

```
+-----+
| Tables_in_myfoafmodel |
+-----+
| jena_g1t0_reif        |
| jena_g1t1_stmt       |
| jena_graph           |
| jena_long_lit        |
| jena_long_uri        |
| jena_prefix          |
| jena_sys_stmt        |
+-----+
```

Clearly, this is all created by Jena framework, and there is no need for us to do anything with them. However, out of curiosity, we can always check out each one of them. If we do so, the following will be some of our discoveries:

- Table `jena_g1t1_stmt` holds all the statements contained in my FOAF document.
- Table `jena_graph` holds all the models in the database.

For example, to inspect what is inside `jena_graph` table, we can do the following:

```
mysql> select * from myfoafmodel.jena_graph;
```

```
+-----+-----+
| ID | Name          |
+-----+-----+
| 1  | myFoafRDF    |
+-----+-----+
```

As you see, my FOAF document is currently the only model in the database, and we will see how to add another model into the database in the next section.

Before we move on, let us do some exercise to get more understanding about persistent model. First, add the following lines into the code shown in List 13.20 and make sure you add these lines after line 46 in List 13.20:

```
// update the model
m.getResource("http://www.liyangyu.com/foaf.rdf#liyang").
addProperty(FOAF.nick, "laoyu");
```

run the code again with the above lines added, and then shut down your application. Now, the above update you have done to the model is saved in the database. To see this, use MySQL Command Line Client to inspect `jena_g1t1_stmt` table; you will see that the update you made to model does stay in the database.

13.3.3.2 Multiple Persistent RDF Models

In the previous section, we have learned how to load a single RDF document into a backend database so as to make it a persistent RDF model. However, a real application often involves a number of RDF documents, including both instance and ontology files. To handle this situation, a Jena database can store multiple models, and typically, each model is represented by its own set of tables in the database.

Let us again use my FOAF document as our example. We have loaded it into the database in the previous section, and in this section, we will load the FOAF ontology into the same database. Therefore, a single database will hold two models. List 13.21 shows how this is done.

List 13.21 Load my FOAF document and FOAF ontology to make them persistent models

```

1: import com.hp.hpl.jena.db.DBConnection;
2: import com.hp.hpl.jena.db.IDBConnection;
3: import com.hp.hpl.jena.rdf.model.Model;
4: import com.hp.hpl.jena.rdf.model.ModelFactory;
5: import com.hp.hpl.jena.rdf.model.ModelMaker;
6: import com.hp.hpl.jena.rdf.model.Property;
7: import com.hp.hpl.jena.rdf.model.Resource;
8: import com.hp.hpl.jena.rdf.model.Statement;
9: import com.hp.hpl.jena.rdf.model.StmtIterator;
10: import com.hp.hpl.jena.util.FileManager;
11: import com.hp.hpl.jena.util.PrintUtil;
12:
13: public class DBModelTester {
14:
15:     public static final String RDF_FILE =
15a:         "http://www.liyangyu.com/foaf.rdf";
16:     public static final String ONTOLOGY_FILE =
16a:         "http://xmlns.com/foaf/0.1/";
17:
18:     private static String className = "com.mysql.jdbc.Driver";
19:     private static String DB_URL =
19a:         "jdbc:mysql://localhost:3306/myFoafModel";
20:     private static String DB_USER = "root";
21:     private static String DB_PASSWD = "passwd";
22:     private static String DB_TYPE = "MySQL";
23:     private static String DOCUMENT_NAME = "myFoafRDF";
24:     private static String ONTOLOGY_NAME = "foaf.owl";
25:
26:     public static void main( String[] args ) {
27:
28:         IDBConnection conn = null;
29:         ModelMaker maker = null;
30:
31:         try {

```



```

32:         Class.forName(className);
33:         conn = new
33a:             DBConnection(DB_URL,DB_USER,DB_PASSWD,DB_TYPE);
34:     } catch (Exception e) { e.printStackTrace(); }
35:
36:     maker = ModelFactory.createModelRDBMaker(conn);
37:     Model m = null;
38:
39:     if ( !maker.hasModel(DOCUMENT_NAME) == true ) {
40:         System.out.println( "Loading instance document -
40a:             one time only" );
41:         m = maker.createModel(DOCUMENT_NAME);
42:         FileManager.get().readModel(m,RDF_FILE);
43:     } else {
44:         m = maker.getModel(DOCUMENT_NAME);
45:     }
46:     printStatements(m, null, null, null);
47:
48:     if ( !maker.hasModel(ONTOLOGY_NAME) ) {
49:         System.out.println( "Loading ontology document -
49a:             one time only" );
50:         m = maker.createModel(ONTOLOGY_NAME);
51:         FileManager.get().readModel(m,ONTOLOGY_FILE);
52:     } else {
53:         m = maker.getModel(ONTOLOGY_NAME);
54:     }
55:     printStatements(m, null, null, null);
56:
57:     try {
58:         conn.close();
59:     } catch(Exception e) { e.printStackTrace(); }
60:
61: }
62:
63: private static void printStatements(Model m, Resource s,
63a:     Property p, Resource o) {
64:     for (StmtIterator i = m.listStatements(s,p,o);
64a:         i.hasNext(); ) {
65:         Statement stmt = i.nextStatement();
66:         System.out.println(" - " + PrintUtil.print(stmt));
67:     }
68: }
69: }

```

Based on what we have learned from List 13.20, List 13.21 does not require too much explanation. Lines 48–55 are the new lines added: they first check whether the FOAF ontology already exists in the database; if yes, load it into the model, otherwise, create a persistent model representing this ontology in the database.

After you have run the code in List 13.21, we can again use MySQL Command Line Client to check the result.

First, you will see there are more tables now in the database:

```
mysql> show tables from myfoafmodel;
+-----+
| Tables_in_myfoafmodel |
+-----+
| jena_g1t0_reif         |
| jena_g1t1_stmt        |
| jena_g2t0_reif         |
| jena_g2t1_stmt        |
| jena_graph            |
| jena_long_lit         |
| jena_long_uri         |
| jena_prefix           |
| jena_sys_stmt         |
+-----+
```

And since table `jena_graph` holds all the models in the database, we can take a look at its content:

```
mysql> select * from myfoafmodel.jena_graph;
+----+-----+
| ID | Name          |
+----+-----+
|  1 | myFoafRDF    |
|  2 | foaf.owl     |
+----+-----+
```

Clearly, a new model, `foaf.owl`, is now added into the database.

You can continue to check other tables one by one to confirm that we have successfully created two persistent models in our database system, which we will not cover in detail. In general, if your application operates on large RDF documents, persistent model should always be considered, and quite often, they should be the best solution to your application as well.

13.4 Inferencing Using Jena

13.4.1 Jena Inferencing Model

In Jena's world, inferencing or reasoning refers to the process of deriving additional facts that are not explicitly expressed by both the instance documents and the ontology documents. The term reasoner is used to refer to a specific code object that can actually perform the derivation. Sometimes, a reasoner is also called a inference engine.

Jena provides a number of reasoners for us to use. The following is a list of frequently used reasoners:

- *RDFS rule reasoner*: an inference engine that supports almost all of the RDFS entailments.
- *OWL 1 reasoner*: the default reasoner that supports most of the frequently used OWL 1 constructs. In practice, this reasoner is considered to be a “full” one, and details can be found at Jena’s official Web site.
- *OWL 1 Mini reasoner*: a slightly cut down version of the “full” OWL 1 reasoner.
- *OWL 1 Micro reasoner*: a smaller but faster one.

To find the supported RDFS and OWL 1 constructs for each reasoner, refer to Jena’s official Web site, which will provide the most up-to-date information. The goal of this section is to show you the basics of using a reasoner.

The steps needed to use a reasoner for inference are quite standard, as summarized here:

1. Choose a reasoner: we use this step to notify Jena system what reasoner we wish to use. One way to do this is to use `ReasonerRegistry` class.
2. Load the ontology document and bind it with the reasoner we have chosen so that the reasoner knows all the facts expressed in the ontology document. One way to do this is to use `bindSchema()` method provided by the reasoner object.
3. Load the instance RDF document, and together with the reasoner, we can use `ModelFactory` class to create an inference model, which contains not only the loaded instance data but also the inferred statements.

Note that the inference model created at step 3 not only contains all the original facts from the instance document and ontology document but also has all the derived statements which represent the additional facts found by the reasoner. In other words, except for the steps listed above, we never have to explicitly invoke any reasoner, and once we have the inference model built, we have it all.

13.4.2 Jena Inferencing Examples

For the code examples in this section, we will continue using my FOAF document, together with the FOAF ontology. However, in order to make things a little more interesting, some change has been made to my FOAF document, as shown in List 13.22.

List 13.22 My FOAF document with some change for testing inference in Jena

```

1: <?xml version="1.0" encoding="UTF-8"?>
2: <rdf:RDF
3:     xml:lang="en"
4:     xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"

```

```

5:      xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
6:      xmlns:foaf="http://xmlns.com/foaf/0.1/">
7:
8:    <rdf:Description
9:      rdf:about="http://www.liyangyu.com/foaf.rdf#liyang">
10:     <foaf:name>liyang yu</foaf:name>
11:     <foaf:title>Dr</foaf:title>
12:     <foaf:givenname>liyang</foaf:givenname>
13:     <foaf:family_name>yu</foaf:family_name>
13b:    <foaf:mbox_sha1sum>
13a:      1613a9c3ec8b18271a8fe1f79537a7b08803d896
13b:    </foaf:mbox_sha1sum>
14:     <foaf:homepage rdf:resource="http://www.liyangyu.com/">
15:     <foaf:workplaceHomepage
15a:       rdf:resource="http://www.delta.com/">
16:     <rdf:type
16a:       rdf:resource="http://xmlns.com/foaf/0.1/Person"/>
17:
18:     <foaf:knows>
19:       <!-- the following is for testing purpose -->
20:       <foaf:Person>
21:         <foaf:mbox
21a:           rdf:resource="mailto:libby.miller@bristol.ac.uk"/>
22:         <foaf:homepage
22a:           rdf:resource="http://www.ilrt.bris.ac.uk/~ecemm/">
23:         </foaf:Person>
24:       </foaf:knows>
25:
26:     <foaf:topic_interest
26a:       rdf:resource="http://dbpedia.org/resource/Semantic_Web"/>
27:
28:   </rdf:Description>
29:
30:
31: <rdf:Description
31a:   rdf:about="http://www.liyangyu.com/foaf.rdf#yiding">
32:   <foaf:mbox_sha1sum>
32a:     1613a9c3ec8b18271a8fe1f79537a7b08803d896
32b:   </foaf:mbox_sha1sum>
33: </rdf:Description>
34:
35: </rdf:RDF>

```

Note that only change is in lines 31–33: a new resource is added, with resource URI given by the following:

<http://www.liyangyu.com/foaf.rdf#yiding>

and note that we have not made any statement about its type and its properties except that we have specified its `foaf:mbox_sha1sum` property (line 32), which assumes the same value as in line 13.

In fact, line 32 is the key about this new resource. Based on FOAF ontology, `foaf:mbox_sha1sum` is an inverse functional property; in other words, if two resources hold the same value on this property, these two resources, although identified by two different URIs, are actually the same thing.

As a result, we expect to see the following facts be added into the inference model created by Jena:

- These two URIs, <http://www.liyangyu.com/foaf.rdf#yiding> and <http://www.liyangyu.com/foaf.rdf#liyang>, represent the same resource in the world.
- Therefore, <http://www.liyangyu.com/foaf.rdf#yiding> is also a `foaf:Person`, and all the properties (and their values) owned by <http://www.liyangyu.com/foaf.rdf#liyang> should also be true for <http://www.liyangyu.com/foaf.rdf#yiding>. For example, it also has the `foaf:title` property with the same value, the `foaf:name` property with the same value.

List 13.23 shows how the inference model is created.

List 13.23 Example to show Jena inferencing capability

```

1: package test;
2:
3: import java.util.Iterator;
4: import com.hp.hpl.jena.rdf.model.InfModel;
5: import com.hp.hpl.jena.rdf.model.Model;
6: import com.hp.hpl.jena.rdf.model.ModelFactory;
7: import com.hp.hpl.jena.rdf.model.Property;
8: import com.hp.hpl.jena.rdf.model.Resource;
9: import com.hp.hpl.jena.rdf.model.Statement;
10: import com.hp.hpl.jena.rdf.model.StmtIterator;
11: import com.hp.hpl.jena.reasoner.Reasoner;
12: import com.hp.hpl.jena.reasoner.ReasonerRegistry;
13: import com.hp.hpl.jena.reasoner.ValidityReport;
14: import com.hp.hpl.jena.util.FileManager;
15: import com.hp.hpl.jena.util.PrintUtil;
16:
17: public class InfModelTester {
18:
19:     public static final String RDF_FILE =
19a:         "c:/liyang/myWebsite/currentPage/foaf.rdf";
20:     public static final String OWL_FILE =
20a:         "http://xmlns.com/foaf/0.1/";
21:
22:     public static void main( String[] args ) {
23:
24:         // load instance data
25:         Model data = ModelFactory.createDefaultModel();
26:         FileManager.get().readModel(data, RDF_FILE);
27:         // use data.read() if reading from Web URL

```

```

28:
29: // load the ontology document
30: Model ontology = ModelFactory.createDefaultModel();
31: ontology.read(OWL_FILE);
32:
33: // get the reasoner
34: Reasoner owlReasoner = ReasonerRegistry.getOWLReasoner();
35: owlReasoner = owlReasoner.bindSchema(ontology);
36:
37: // use the reasoner and instance data to create
37a: // an inference model
38: InfModel infModel =
38a: ModelFactory.createInfModel(owlReasoner, data);
39:
40: // some validation to make us happy
41: ValidityReport vr = infModel.validate();
42: if ( vr.isValid() == false ) {
43:     System.out.print("ontology model validation failed.");
44:     for (Iterator i = vr.getReports(); i.hasNext(); ) {
45:         System.out.println(" - " + i.next());
46:     }
47:     return;
48: }
49:
50: Resource yu = infModel.getResource
50a:     ("http://www.liyangyu.com/foaf.rdf#yiding");
51: System.out.println("yu *:");
52: printStatements(infModel, yu, null, null);
53:
54: }
55:
56: private static void printStatements(Model m, Resource
56a:     s, Property p, Resource o) {
57:     for (StmtIterator i = m.listStatements(s,p,o);
57a:         i.hasNext(); ) {
58:         Statement stmt = i.nextStatement();
59:         System.out.println(" - " + PrintUtil.print(stmt));
60:     }
61: }
62: }

```

List 13.23 should be fairly easy to follow if we map it to the steps that are needed for creating inference model using Jena (see Sect. 13.4.1). More specifically, line 34 maps to step 1, lines 30–31 and line 35 implement step 2, and lines 25–26 and line 38 are the last step. Again, once line 38 is executed, inference model `infModel` holds both the original facts and newly derived facts.

To see the derived facts, we can ask all the facts about the resource identified by <http://www.liyangyu.com/foaf.rdf#yiding>, we expect to see lots of new facts added about this resource. This is done in lines 50–52, with the help from a

simple private helper method call `printStatements()` (lines 56–61), which does not require too much explanation.

Now, run the code in List 13.23 and List 13.24 shows some of the newly derived facts (line numbers are added for explanation purpose).

List 13.24 Derived facts about resource **yiding**

```

1: - (http://www.liyangyu.com/foaf.rdf#yiding owl:sameAs
    http://www.liyangyu.com/foaf.rdf#liyang)
2: - (http://www.liyangyu.com/foaf.rdf#yiding rdf:type
    http://xmlns.com/foaf/0.1/Person)
3: - (http://www.liyangyu.com/foaf.rdf#yiding rdfs:label
    'liyang yu')
4: - (http://www.liyangyu.com/foaf.rdf#yiding foaf:name
    'liyang yu')
5: - (http://www.liyangyu.com/foaf.rdf#yiding
    foaf:knows 3550803e:1245b5759eb:-8000)
6: - (http://www.liyangyu.com/foaf.rdf#yiding
    foaf:family_name 'yu')
7: - (http://www.liyangyu.com/foaf.rdf#yiding
    foaf:givenname 'liyang')
8: - (http://www.liyangyu.com/foaf.rdf#yiding foaf:title 'Dr')
9: - (http://www.liyangyu.com/foaf.rdf#yiding
    foaf:topic_interest
    http://dbpedia.org/resource/Semantic_Web)
10: - (http://www.liyangyu.com/foaf.rdf#yiding
    foaf:homepage http://www.liyangyu.com)
11: - (http://www.liyangyu.com/foaf.rdf#yiding
    foaf:workplaceHomepage http://www.delta.com)

```

Clearly, the inference engine has successfully recognized the fact that these two resources are the same object in the real world (line 1), and it has also assigned all the properties owned by <http://www.liyangyu.com/foaf.rdf#liyang> to the resource identified by <http://www.liyangyu.com/foaf.rdf#yiding>, as shown by line 2–11.

Understand that besides these basic steps of using the inference engine, we can in fact make a choice among different reasoner configurations. For example, line 34 of List 13.23 asks an OWL 1 reasoner from Jena, and since no parameter is passed in to the call, the default OWL 1 reasoner (“full” version) is returned back.

To require an OWL 1 reasoner rather than the default one, you need the help from another important class called `OntModelSpec`, which hides the complexities of configuring the inference model. More specifically, a number of common objects that represent different configurations have been pre-declared as constants in `OntModelSpec`, and Table 13.1 shows some of these configurations (for the complete list, refer to official Jena document).

Table 13.1 Reasoner configuration using `OntModelSpec`

<code>OntModelSpec</code>	Language profile	Storage model	Reasoner
<code>OWL_MEM</code>	OWL 1 full	In-memory	None
<code>OWL_MEM_RULE_INF</code>	OWL 1 full	In-memory	Rule-based reasoner with OWL rules
<code>OWL_DL_MEM_RULE_INF</code>	OWL 1 DL	In-memory	Rule-based reasoner with OWL rules
<code>RDFS_MEM_RDFS_INF</code>	RDFS	In-memory	Rule reasoner with RDFS-level rules

To create an inference model with a desired specification, class `ModelFactory` should be used, with the pre-declared configuration and the instance document, as shown in List 13.25. Again, List 13.25 accomplishes exactly the same thing as List 13.23 does, but with the flexibility of making your own choice of reasoner configuration.

List 13.25 Using `ontModelSpec` to create ontology model

```

1: package test;
2:
3: import java.util.Iterator;
4: import com.hp.hpl.jena.ontology.OntModel;
5: import com.hp.hpl.jena.ontology.OntModelSpec;
6: import com.hp.hpl.jena.rdf.model.Model;
7: import com.hp.hpl.jena.rdf.model.ModelFactory;
8: import com.hp.hpl.jena.rdf.model.Property;
9: import com.hp.hpl.jena.rdf.model.Resource;
10: import com.hp.hpl.jena.rdf.model.Statement;
11: import com.hp.hpl.jena.rdf.model.StmtIterator;
12: import com.hp.hpl.jena.reasoner.ValidityReport;
13: import com.hp.hpl.jena.util.FileManager;
14: import com.hp.hpl.jena.util.PrintUtil;
15:
16: public class InfModelTester {
17:
18:     public static final String RDF_FILE =
18a:         "c:/liyang/myWebsite/currentPage/foaf.rdf";
19:     public static final String OWL_FILE =
19a:         "http://xmlns.com/foaf/0.1/";
20:
21:     public static void main( String[] args ) {
22:
23:         // load instance data
24:         Model data = ModelFactory.createDefaultModel();
25:         FileManager.get().readModel(data, RDF_FILE);
26:         // use data.read() if reading from Web URL
27:
28:         // create my ontology model

```



```

29:      OntModel ontModel = ModelFactory.createOntologyModel
29a:          (OntModelSpec.OWL_MEM_RULE_INF,data);
30:      ontModel.read(OWL_FILE);
31:
32:      // some validation to make us happy
33:      ValidityReport vr = ontModel.validate();
34:      if ( vr.isValid() == false ) {
...
40:      }
41:
42:      Resource yu = ontModel.getResource
42a:          ("http://www.liyangyu.com/foaf.rdf#yiding");
43:      System.out.println("yu *:");
44:      printStatements(ontModel, yu, null, null);
45:
46:  }
47:
48:  private static void printStatements(Model m,Resource s,
48a:          Property p,Resource o) {
...
53:  }
54:  }

```

As you can tell, this is a slightly different approach compared to the one shown in List 13.23. Instead of using `ModelFactory.createDefaultModel()`, line 29 creates an ontology model, which uses the `OntModelSpec.OWL_MEM_RULE_INF` configuration together with the instance document. After line 30 is executed, a complete ontology model that consists of the desired reasoner, the ontology file itself, and the instance data for the reasoner to work on is created. Compared to the basic model created by `ModelFactory.createDefaultModel()` method, this ontology model has all the derived statements.

As a little experiment, we can try the `RDFS_MEM_RDFS_INF` configuration; change line 29 in List 13.25 to the following:

```

OntModel ontModel = ModelFactory.createOntologyModel
                    (OntModelSpec.RDFS_MEM_RDFS_INF,data);

```

and run the code, you will see the difference. Since RDFS is not able to handle inverse functional property, most of the inferred facts are gone. In fact, none of the statements in List 13.24 will show up.

13.5 Summary

In this chapter, we have presented Jena as an example framework for application development on the Semantic Web. After finishing this chapter, you should be able to work effectively with Jena library. More specifically,

- understand how to setup Jena as your development framework;
- understand how to use Jena library to conduct basic RDF model operations, such as reading an RDF model, creating an RDF model, and interrogating an RDF model;
- understand why persistent RDF models are important, how to use Jena library to create persistent models, and how to inspect these persistent models in the selected database system;
- understand the concept of inference model and its different configurations, how to use Jena library to create inference model with a specific configuration, also understand how to retrieve the derived statements from the inference model.

You will see more features offered by Jena library in the next two chapters, and with what you have learned here, you should be able to explore Jena library on your own as well.

Chapter 14

Follow Your Nose: A Basic Semantic Web Agent

Developing applications on the Semantic Web requires a set of complex skills, yet this skill set does land itself on some basic techniques. In the previous chapter, we have learned some basics, and in this chapter, we will continue to learned some more.

Follow-Your-Nose method is one such basic technique you want to master. It could be quite useful in many cases. This chapter will focus on this method and its related issues.

14.1 The Principle of Follow-Your-Nose Method

14.1.1 What Is Follow-Your-Nose Method?

Follow Your Nose is not something new to us at all; when you are surfing on the Web, quite often, your strategy is to follow your nose. For example, the current page you are reading will normally contain links to other pages that might be interesting to you, and clicking one of the links will take you to the next page. Similarly, the next page will again contain links to other pages, which you can click to go further, so on and so forth.

Follow-Your-Nose policy is not only used by human readers but also used by soft agents. Perhaps the most obvious example is the crawler used by search engines. We have discussed the work flow of search engines in [Sect. 8.1.1.2](#). Based on what we have learned there, we understand Follow-Your-Nose method is the main strategy used by a crawler.

Let us go back to the Web of Linked Data, a practical version of the Semantic Web. Obviously, the most distinguished feature of the Web of Linked Data is the fact that there exist a fairly large amount of links among different datasets. This feature does provide us with a chance to apply the Follow-Your-Nose method. In other words, one can make data discovery by following these links and navigating from one dataset to another, so on and so forth, until the stop criteria are met. This is the concept of Follow Your Nose in the world of the Semantic Web.

For example, let us say we want to know more about the tennis player Roger Federer. We can dereference the URI

`http://dbpedia.org/resource/Roger_Federer`

and part of the returned RDF document is shown in List 14.1.

List 14.1 owl:sameAs links defined in Federer’s RDF document

```

1: <rdf:Description
1a:   rdf:about="http://dbpedia.org/resource/Roger_Federer">
2:   <owl:sameAs rdf:resource= "http://rdf.freebase.com/ns/
2a:     guid.9202a8c04000641f80000000019f525" />
3: </rdf:Description>
4: <rdf:Description
4a:   rdf:about="http://mpii.de/yago/resource/Roger_Federer">
5:   <owl:sameAs
5a:     rdf:resource="http://dbpedia.org/resource/Roger_Federer" />
6: </rdf:Description>

```

Now, we can dereference the links in line 2, which will take us to another RDF document that contains more facts about Roger Federer. Furthermore, once you have landed on this new document, you can repeat this Follow-Your-Nose procedure and discover new data about him, so on and so forth.

Note that we don’t have to concentrate only on owl:sameAs links; we can follow any link that might be interesting to us. For example, we can follow the link to know more about Switzerland, the country where he was born.

What about the second link shown in line 4 of List 14.1? We can certainly follow that as well. In general, if we have more than one links to follow, we will face a situation that actually asks for a decision from us: the so-called depth-first search vs. the breadth-first search.

More specifically, by using the depth-first search, we will first ignore the second link in line 4 and only follow the link in line 2 to reach a new document. In addition, we will follow any new links found on this new document to reach another new document, so on and so forth. We will continue to reach deeper until there is nothing to follow or the stopping criteria have been met. At that point, we go back one level to follow one link on that level. Only when there is no more link to follow at all, we will go all the way back to follow the link shown in line 4. As you can tell, we go deeper first before we explore any link on the same level, and that is the reason why we call this the depth-first Follow-Your-Nose method.

The breadth-first search works in the opposite way. After following the first link in line 2, even we have found new links on the new document, we will go back to follow the link in line 4. In other words, we always try to cover the same level links before we go any deeper.

In practice, whether to use depth-first search or breadth-first search is a decision that depends on the specific application and its domain. We will come back to this decision later. For now, understanding the idea of Follow Your Nose is the goal.

14.1.2 URI Declarations, Open Linked Data, and Follow-Your-Nose Method

Before we move on to build a Follow-Your-Nose agent, there is one important issue to understand; on the Web of Linked Data, what makes this Follow-Your-Nose policy possible? Is it safe to believe dereferencing any given link will lead us to a new data file?

The answer is yes; Follow-Your-Nose method on the Web of Linked Data should be able to accomplish its goal. The reason is due to the following two facts:

- The links are typed links, i.e., they have clearly defined semantics which enables us to know which links to follow.
- The basic principles of Linked Data.

The first reason is obvious. For example, the following two properties

```
rdfs:seeAlso  
owl:sameAs
```

have clearly defined semantics, and they are considered as links between different datasets. Any application understands that by following these links; it can gather more facts about the subject resource.

Depending on different application scenarios, there may be more properties that can be considered as links, but the above two are the most obvious ones. Furthermore, with the help from ontology documents, a soft agent can decide on the fly which links should be followed and which links should be ignored based on the main goal of the application. This is possible since the links can be typed links and their types (classes) are clearly defined in the ontology file.

A good example is shown in List 11.4 (Sect. 11.2.3.1). The two `rdfs:seeAlso` links in List 11.4 are all typed links. If our agent is trying to use Follow-Your-Nose method to gather information about my publications (not CVs), it can easily decide which one of those two links should be followed.

In fact, this is also one of the reasons why Follow-Your-Nose method can work well on the Semantic Web. On our traditional Web, all the links (identified by `<a href>` tag) are untyped links; there is simply no scalable and reliable way for a soft agent to tell which one to follow and which one to avoid.

The second reason is less obvious but it is equally important. More specifically, the action of Follow Your Nose by a soft agent is to dereference a given URI. If

the given URI cannot be dereferenced, there will be no way to carry out the policy successfully.

Fortunately, based on the Linked Data principle that we have discussed in [Chap. 11](#), the following are true:

- If the URI contains a fragment identifier (Hash URI), the part of the URI before the "#" sign should lead to a URI declaration page that is served with an RDF document.
- If the URI does not contain a fragment identifier, an attempt to dereference the URI should be answered with a 303-redirect which leads to a URI declaration page served with an RDF document as well.

In other words, the use of URIs as names, and in particular URIs can be dereferenced using the HTTP protocol, is a critical enabling factor for the Follow-Your-Nose approach.

Finally, understand that Follow-Your-Nose method is not an optional extra; on the contrary, it is fundamentally necessary in order to support the highly devolved, loosely coupled nature of Linked Data Web.

In the next section, let us take a look at a real example showing you how it can be implemented with the help from the Jena package.

14.2 A Follow-Your-Nose Agent in Java

14.2.1 *Building the Agent*

In this section we will code a simple agent that implements the idea of Follow Your Nose, and we are going to use it to collect everything that has been said about tennis player Roger Federer on the Web of Linked Data.

To do so, we will start from DBpedia. And again, here is the URI that represents Roger Federer in DBpedia:

```
http://dbpedia.org/resource/Roger_Federer
```

Starting from this seed URI, we will carry out the following three steps:

1. Dereference the seed URI to get an RDF document that describes Roger Federer.
2. To collect the facts about Roger Federer, harvest all the statements that satisfy the following pattern:

```
<http://dbpedia.org/resource/Roger_Federer>
<someProperty> <someValue> .
```

3. To gather the links to follow, find all the statements that satisfy the following two patterns, and the collection of *subjectResource* and *objectResource* is the links to follow:

```
<http://dbpedia.org/resource/Roger_Federer>
  owl:sameAs <objectResource> .
<subjectResource>
  owl:sameAs <http://dbpedia.org/resource/Roger_Federer>.
```

The first step does not need much explanation. Based on what we have learned from [Chap. 13](#), we also know how to dereference a given URI by using Jena library.

The second step is data discovery. In this example, the agent collects all the facts about the resource identified by the given URI. Obviously, the given statement pattern in step 2 will accomplish the goal.

The third step is the step where the idea of Follow Your Nose is implemented. Specifically, property `owl:sameAs` is used to find the links. As a result, these links are all URI alias which all represent Roger Federer. Whichever link we follow, we are only collecting facts about Roger Federer. Note that Roger Federer's URI can be either the subject or the object; if it appears as the object, the subject URI is the link to follow, and if it assumes the role of subject, the object URI is the link to follow. This is why we have two statement patterns to consider in step 3.

Once these steps are done on a given dataset, the agent not only has collected some facts about Federer but also has discovered a set of links to follow. The next action is to repeat these steps by selecting a link from the collected link set and dereferencing it to collect more data and more links.

This process is repeatedly executed until there are no more links to follow, and at which point, we declare success. On the Web of Linked Data, anyone from anywhere can say something about Roger Federer, and whatever you have said, we have them all.

Lists 14.2–14.4 show the classes which implement this agent.

List 14.2 FollowYourNose.java class definition

```
1: package test;
2:
3: import com.hp.hpl.jena.rdf.model.Model;
4: import com.hp.hpl.jena.rdf.model.ModelFactory;
5: import com.hp.hpl.jena.rdf.model.NodeIterator;
6: import com.hp.hpl.jena.rdf.model.Property;
7: import com.hp.hpl.jena.rdf.model.RDFNode;
8: import com.hp.hpl.jena.rdf.model.ResIterator;
9: import com.hp.hpl.jena.rdf.model.Resource;
10: import com.hp.hpl.jena.rdf.model.Statement;
11: import com.hp.hpl.jena.rdf.model.StmtIterator;
12: import com.hp.hpl.jena.vocabulary.OWL;
```

```

13:
14: public class FollowYourNose {
15:
16:     private URICollection sameAsURIs = null;
17:
18:     public FollowYourNose(String uri) {
19:         sameAsURIs = new URICollection();
20:         sameAsURIs.addNewURI(uri);
21:     }
22:
23:     public void work() {
24:
25:         // get the next link to follow
26:         String currentURI = sameAsURIs.getNextURI();
27:         if ( currentURI == null ) {
28:             return;
29:         }
30:
31:         try {
32:
33:             // de-reference this link
34:             Model instanceDocument =
34a:                 ModelFactory.createDefaultModel();
35:             instanceDocument.read(currentURI);
36:
37:             // do the data collection
38:             collectData(instanceDocument, currentURI);
39:
40:             // find the next links to follow
41:             updateURICollection(sameAsURIs, currentURI,
41a:                               instanceDocument, OWL.sameAs);
42:
43:         } catch (Exception e) {
44:             System.out.println("*** errors when handling (" +
44a:                               currentURI + ") ***");
45:         }
46:
47:         System.out.println("\n---- these links are yet
47a:                               to follow ---- ");
48:         sameAsURIs.showAll();
49:         System.out.println("----- ");
50:
51:         // following our nose
52:         work();
53:
54:     }
55:
56:     private void collectData(Model model, String uri) {

```



```

57:     if ( uri == null ) {
58:         return;
59:     }
60:     int factCounter = 0;
61:     System.out.println("Facts about <" + uri + ">:");
62:     for (StmtIterator si = model.listStatements();
62a:         si.hasNext(); ) {
63:         Statement statement = si.nextStatement();
64:         if ( uri.equalsIgnoreCase(
64a:             statement.getSubject().getURI() == true ) {
65:             factCounter ++;
66:             System.out.print(" - <" +
66a:                 statement.getPredicate().toString() + ">:<");
67:             System.out.println(statement.getObject().toString()
67a:                 + ">");
68:             if ( factCounter >= 10 ) {
69:                 return;
70:             }
71:         }
72:     }
73: }
74:
75: private void updateURICollection(
75a:     URICollection uriCollection, String uri,
76:     Model model, Property property) {
77:     if ( uri == null ) {
78:         return;
79:     }
80:     // check object
81:     Resource resource = model.getResource(uri);
82:     NodeIterator objects =
82a:         model.listObjectsOfProperty(resource, property);
83:     while ( objects.hasNext() ) {
84:         RDFNode object = objects.next();
85:         if ( object.isResource() ) {
86:             Resource tmpResource = (Resource)object;
87:             uriCollection.addNewURI(tmpResource.getURI());
88:         }
89:     }
90:     // check the subject
91:     ResIterator subjects =
91a:         model.listSubjectsWithProperty(property, resource);
92:     while ( subjects.hasNext() ) {
93:         Resource subject = subjects.nextResource();
94:         uriCollection.addNewURI(subject.getURI());
95:     }
96: }
97:
98: }

```

List 14.3 URICollection.java definition

```
1: package test;
2:
3: import java.net.URI;
4: import java.util.HashSet;
5: import java.util.Iterator;
6: import java.util.Stack;
7:
8: public class URICollection {
9:
10:     private Stack URIs = null;
11:     private HashSet domainCollection = null;
12:
13:     public URICollection() {
14:         URIs = new Stack();
15:         domainCollection = new HashSet();
16:     }
17:
18:     public void addNewURI(String uri) {
19:         if ( uri == null ) {
20:             return;
21:         }
22:         try {
23:             URI thisURI = new URI(uri);
24:             if ( domainCollection.contains(thisURI.getHost())
24a:                 == false ) {
25:                 domainCollection.add(thisURI.getHost());
26:                 URIs.push(uri);
27:             }
28:         } catch(Exception e) {};
29:     }
30:
31:     public String getNextURI() {
32:         if ( URIs.empty() == true ) {
33:             return null;
34:         }
35:         return (String)(URIs.pop());
36:     }
37:
38:     public void showAll() {
39:         for ( int i = 0; i < URIs.size(); i ++ ) {
40:             System.out.println(URIs.elementAt(i).toString());
```

```
41:     }
42: }
43:
44: }
```

List 14.4 FollowYourNoseTester.java definition

```
1: package test;
2:
3: public class FollowYourNoseTester {
4:
5:     public static final String startURI =
6:         "http://dbpedia.org/resource/Roger_Federer";
7:
8:     public static void main(String[] args) {
9:         FollowYourNose fyn = new FollowYourNose(startURI);
10:        fyn.work();
11:
12: }
13:
14: }
```

The key class is `FollowYourNose.java`, defined in List 14.2. Given our earlier explanation, understanding this class is straightforward. It has one private member variable, called `sameAsURIs`, which holds all the links yet to follow (we will get to the management of these links shortly). Lines 18–21 show its constructor and a starting URI is passed in and added into `sameAsURIs`; therefore initially, there is only one link to explore.

The key member function is `work()`, defined in lines 23–54. It first checks `sameAsURIs` link set to get the next link to follow. If there is none left, the whole process is done (lines 26–29). If there is indeed one link to follow, it then implements the three steps we have discussed earlier: lines 34–35 implement step 1, line 38 implements step 2, and line 41 implements step 3.

Note that to finish steps 2 and 3, some helper functions are created. Lines 56–73 is a private member function called `collectData()`, which defines the details of step 2. Similarly, lines 75–96 create another private member function called `updateURICollection()`, which defines the details of step 3. With our earlier description about these steps, understanding these two functions should be fairly easy. We will discuss them more after we finish discussing `work()` method.

Once all these steps are finished, `work()` shows the links yet to follow (lines 47–49) and then calls itself recursively (line 52) to follow these links. By doing so, the process of discovering data, collecting links, and following links will continue

and will finally come to stop when there is no more links to follow. Note that this recursive style is quite often seen in crawler-like agent like this; using this recursive calling will make your coding easier and cleaner.

Let us now go back to its two private methods. As far as this example agent is concerned, collecting data simply means to print them out, and this is the main work flow implemented by `collectData()` method. More specifically, a `for` loop is used to iterate on each statement in the current RDF model (line 62); for the current statement, if the subject happens to be the URI that represents Federer (line 64), we will print out its property name and the value of the property (lines 66–67).

Note that `model.listStatements()` is the key Jena API call used by `collectData()` method, which lists all the statements contained in the model. Since we are examining RDF documents in DBpedia or other datasets contained in the Web of Linked Data, if we print out everything we have collected, in most cases it will be a long list of facts. Therefore, we print only the first 10 facts (lines 60, 65, 68–70) from each RDF document.

Method `updateURICollection()` also makes use of several useful Jena API calls. First, `model.getResource()` method (line 81) takes Federer's URI as its input parameter and returns the corresponding resource object that represents Federer. This resource object is then used in two other API calls, namely, `model.listObjectsOfProperty()` and `model.listSubjectsWithProperty()` (lines 82 and 91), which examine the two statement patterns as discussed in step 3. Once the patterns are matched, the corresponding URIs are added to our link collection (lines 87 and 94). Note that method `updateURICollection()` is called with `owl:sameAs` as one of the input parameters (line 41); therefore, only `owl:sameAs` links are considered by the agent.

At this point, we have a good understand about this key class. You can modify its two private methods, `collectData()` and `updateURICollection()`, accordingly to make the agent fit your own need. For example, instead of printing out all the facts, `collectData()` can create a persistent RDF model using a backend database so that later on, we can query the facts using SPARQL.

Finally, before we move on, let us discuss the other two helper classes briefly.

Class `URICollection.java` (List 14.3) is created to hold the links yet to follow. When the agent first starts, this collection only holds the initial URI that represents Federer. More links are added to it during the work course of the agent, and finally, there should be none left and therefore the agent stops.

Two things to note about this collection class. First, its underlying data structure is a stack (line 10, 14), so it implements a depth-first Follow-Your-Nose policy, rather than breadth-first. Second, on the Web of Linked Data, it is often true that the links are two-way ones, i.e., dataset A has a link to dataset B, and B has a link back to A. As a result, if we collect every URI on the link, we may get into a infinite loop. To take this into account, we want to avoid adding the same URI back to the collection. Lines 24–26 of List 14.3 implement this idea; for each incoming new URI, we check its domain; if this domain has been added before, this incoming URI is considered as previously processed and is not added to the link collection.

Class `FollowYourNoseTester.java` (List 14.4) does not require any explanation; it is a simple driver class to start your agent. Note that the initial URI is specified at line 5; if you want to try some other URIs, this is the line you can change.

14.2.2 Running the Agent

List 14.5 shows part of the result when using Roger Federer's URI as the seed URI.

List 14.5 Result when using Federer's URI (line numbers are added for explanation purpose)

```

1: Facts about <http://dbpedia.org/resource/Roger_Federer>:
2: - <http://dbpedia.org/property/redirect> :
   <http://dbpedia.org/resource/Roger_Federer>
3: - <http://dbpedia.org/property/doublestitles> :
   <8^http://www.w3.org/2001/XMLSchema#integer>
4: - <http://dbpedia.org/property/careerprizemoney> :
   <US$45,790,270@en>
5: - <http://dbpedia.org/property/olympicsdoublesresult> :
   <http://dbpedia.org/resource/Roger_Federer/
     olympicsdoublesresult/OlympicEvent>
6: - <http://dbpedia.org/property/relatedInstance> :
   <http://dbpedia.org/resource/Roger_Federer/succession_box1>
7: - <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> :
   <http://dbpedia.org/ontology/TennisPlayer>
8: - <http://dbpedia.org/ontology/residence> :
   <http://dbpedia.org/resource/Switzerland>
9: - <http://dbpedia.org/property/abstract> :
   <Roger Federer ... >
10: - <http://www.w3.org/2000/01/rdf-schema#comment> :
   <Roger Federer ... >
11: - <http://dbpedia.org/property/country> :
   <http://dbpedia.org/resource/Switzerland>
12:
13: ---- these links are yet to follow ----
14: http://rdf.freebase.com/ns/guid.
14a:                               9202a8c04000641f80000000019f525
15: http://mpii.de/yago/resource/Roger_Federer
16: -----
17: Facts about <http://mpii.de/yago/resource/Roger_Federer>:
18:
19: ---- these links are yet to follow ----
20: http://rdf.freebase.com/ns/guid.
20a:                               9202a8c04000641f80000000019f525
21: -----

```

```

22: Facts about <http://rdf.freebase.com/ns/guid.
22a:           9202a8c04000641f800000000019f525>:
23: - <http://rdf.freebase.com/ns/tennis.tennis_player.
23a:           highest_singles_ranking> :
      <http://rdf.freebase.com/ns/guid.
23b:           9202a8c04000641f8000000004fba118>
24: - <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> :
      <http://rdf.freebase.com/ns/base.popstra.celebrity>
25: - <http://rdf.freebase.com/ns/type.object.key> :
25a: <234daeea:1246364ddc3:-7ff0>
26: - <http://rdf.freebase.com/ns/common.topic.article> :
      <http://rdf.freebase.com/ns/guid.
26a:           9202a8c04000641f800000000019f52f>
27: - <http://rdf.freebase.com/ns/type.object.key> :
      <234daeea:1246364ddc3:-8000>
28: - <http://rdf.freebase.com/ns/type.object.key> :
      <234daeea:1246364ddc3:-7ff8>
29: - <http://rdf.freebase.com/ns/type.object.key> :
      <234daeea:1246364ddc3:-7fed>
30: - <http://rdf.freebase.com/ns/type.object.name> :
      <??????? ??????@uk>
31: - <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> :
      <http://rdf.freebase.com/ns/base.rogerfederer.topic>
32: - <http://rdf.freebase.com/ns/base.popstra.
23a:           celebrity.friendship> :
      <http://rdf.freebase.com/ns/guid.
23b:           9202a8c04000641f800000000c5ab937>

```

Some explanation about this result will help you understand more about Follow-Your-Nose policy.

First off, lines 2–11 in List 14.5 shows the first 10 facts the agent has collected from dereferencing the seed URI. At the time of this writing, the RDF document retrieved from the seed URI contains the following two statements:

```

<rdf:Description
  rdf:about="http://mpii.de/yago/resource/Roger_Federer">
  <owl:sameAs
    rdf:resource="http://dbpedia.org/resource/Roger_Federer"/>
</rdf:Description>

```

```

<rdf:Description
  rdf:about="http://dbpedia.org/resource/Roger_Federer">
  <owl:sameAs rdf:resource="http://rdf.freebase.com/ns/guid.
    9202a8c04000641f800000000019f525"/>
</rdf:Description>

```

Our agent has detected this, and following `owl:sameAs` link means dereferencing the following two URIs:

```
http://mpii.de/yago/resource/Roger_Federer
http://rdf.freebase.com/ns/guid.9202a8c04000641f800000000019f525
```

And these are the URI alias you see in lines 14–15 in List 14.5.

URI http://mpii.de/yago/resour-ce/Roger_Federer is dereferenced next by the agent. However, no facts have been collected there (lines 17, 18).

The agent then moves on to dereference the next URI (line 20). It is able to collect some more facts about Roger Federer from the retrieved RDF document, as shown by lines 22–32 of List 14.5. Similarly, the agent then looks for property `owl:sameAs` in this RDF document to continue its journey in the Linked Data Web, so on and so forth.

To get more familiar with this example, you can use some of your favorite URIs as the seeds to see how Follow-Your-Nose policy works in real world. It is a simple idea; however it does tell us a lot about the Linked Data Web.

The following list contains some suggested URIs you can try:

```
http://dbpedia.org/resource/Tim_Berners-Lee
http://dbpedia.org/resource/Semantic_Web
http://dbpedia.org/resource/Beijing
http://dbpedia.org/resource/Nikon_D300
```

14.2.3 More Clues for Follow Your Nose

So far at this point, we have been using `owl:sameAs` as the only link when implementing our Follow-Your-Nose agent. In fact, there are more clues we can use.

Another obvious one is `rdfs:seeAlso` property. However, we need to be careful when following `rdfs:seeAlso` link. More specifically, based on the Open Linked Data principle, we will be able to retrieve representations of object from the Web; however, no constraints are placed on the format of those representations. This is especially true with `rdfs:seeAlso` property; the response could be a JPEG file instead of an RDF document.

In general, when coding Fellow-Your-Nose agent, we may have to use content negotiation process and may also have to implement more protections around this to make the agent more robust.

Another similar property is `rdfs:isDefinedBy`, which is a sub-property of `rdfs:seeAlso`. Our discussion about `rdfs:seeAlso` is applicable to this property as well, and it is another link that a Follow-Your-Nose agent should consider.

Coding a Follow-Your-Nose agent is sometimes more of an art than a technique; it does require creative heuristics to make the collected facts more complete. For

a given resource, different Follow-Your-Nose agents can very possibly deliver different fact sets. A key factor, again, is about how to find clues to discover more facts.

Besides using more properties as links as we have discussed above, sometimes, taking into the consideration of your specific resource and related properties from popular ontologies may be a good idea.

For example, given the fact that Roger Federer is an instance of `foaf:Person`, another FOAF property, `foaf:knows`, could be something we want to consider. More specifically, if Roger Federer `foaf:knows` another resource `R`, it is then likely resource `R` also `foaf:knows` Roger Federer, and it is also possible that resource `R` has said something about Federer too. Therefore, dereferencing the URI of resource `R` and scanning the retrieved RDF document to find facts about Federer is another action the agent can take.

Another FOAF property is `foaf:primaryTopicOf`. A statement such as

```
<http://dbpedia.org/resource/Roger_Federer>
foaf:isPrimaryTopicOf object .
```

means the resource identified by the `object` URI is mainly about Roger Federer. Therefore, dereferencing the URI of the `object` and trying to find more facts there about Federer is a good direction to pursue.

By now, you should have got the point. The key is to discover and examine every potential link when implementing a Follow-Your-Nose agent. What you have learned here should be a good starting point, and now it is up to you to make it smarter.

14.2.4 Can You Follow Your Nose on Traditional Web?

Now that we have finished a Follow-Your-Nose agent on the Semantic Web, we can continue on with an interesting comparison: can we do the same on the traditional document Web, i.e., to find everything that people have said about Federer on the Web?

To get started, we first have to design a heuristic that is similar to the algorithm used by our Follow-Your-Nose agent.

A possible solution is to start from his own home page. Once we download the page, we can collect everything about him from that page, including address, phone numbers, and e-mail addresses.

It is safe to assume that his home page has links (`` tags) that point to other Web sites. These Web sites may describe different tennis events, including the four grand slams, and in addition, they may include Web sites that are related to different tennis organizations. We can also make the assumption that if Roger Federer is talking about these resources on his home page, these resources are also likely to talk about him. Therefore, we can follow these links to check out each Web site individually, with the goal of finding information about him from these pages.

At the first glance, this sounds like a plan. However, once you begin to put it into action, you will see the problems right away.

The first major hurdle comes from the fact that we probably have to do screen scraping in order to obtain the information we want. Given the fact that information on each page is not structured data, but simple text for human eyes, you can easily imagine how difficult this screen-scraping process can be, if not totally impossible.

In fact, even screen scraping each page is possible; the maintenance of our agent could be very costly. Each Web site is always under active change; a successful parsing today does not mean another successful one a certain amount of time later. For example, the “indicators/flags” that have been used to locate particular information block might not exist anymore. Our agent has to be constantly modified in order to process the updated Web documents.

The second major hurdle is related to the fact that there is no unique identifier our agent can use to identify Roger Federer on each Web page. For example, one page might refer him by his full name, another page might call him the *Swiss Maestro*, and a Web page created by his fans may address him as *Fed Express*. As a result, our agent, once finishes downloading a given page, will find it difficult to determine if the page has any description about him at all. In the worst case, even a matching on the name does not necessarily mean the page is about him.

Now let us assume that all the above steps are feasible and doable, and we can indeed gather something about Roger Federer. The question then is what to do with the gathered information?

Obviously, the facts are collected from different source pages, which do not offer any structured data and also do not share any common vocabularies or any common data model (such as RDF). As a result, the collected information will be most likely unusable when it comes to supporting the following capabilities:

- Reasoning on the collected information to discover new facts
This is one of the major motivations for Follow-Your-Nose data aggregation. With the help from ontologies, based on the aggregated information, we can discover facts that are not presented in any of the source document. And clearly, the page content we have harvested from the traditional Web cannot be used for this purpose at all.
- Structured query language to answer questions
Another important operation on the collected dataset is to execute SPARQL queries so as to get direct answers to many questions. We can certainly make queries against one single RDF source file; however, only after Follow-Your-Nose data aggregation is implemented, will we be confident that the answer is complete and correct. Furthermore, since we can run our Follow-Your-Nose agent at any time, the query result will therefore be able to include the newly discovered facts. Clearly, data collected from traditional Web will not be able to be used for any query easily, and including new results normally means new development and costly maintenance.

The conclusion is that a simple task like this can be prohibitively difficult under the current Web environment, and even if it is feasible, the result will be quite difficult to use.

At the mean time, this task can be easily accomplished under the Semantic Web, as we have seen already. In addition, the collected facts can be used for further reasoning and can be queried by language such as SPARQL.

Besides the above, our Follow-Your-Nose agent also enjoys the following benefits provided directly by the structured data on the Semantic Web:

- Easy to maintain

In fact, there is no need to maintain anything. Any source RDF document can be modified to add new facts at any time; the same agent can again harvest all these related descriptions with the same ease, no code change is ever needed.

- Dynamic and up to date

This is a natural extension of the maintainability of the agent; you can run it at any time to update the collected dataset. The owners of data sources do not have to tell you anything or notify you the fact that one of them has published some new facts. Again, dynamic and distributed data aggregation is one of the many benefits provided by the vision of the Semantic Web.

14.3 A Better Implementation of Follow-Your-Nose Agent: Using SPARQL Queries

So far in this chapter, we have done quite a few queries against RDF models. Some of the key methods we have been using include the following:

```
model.listStatements();  
model.listSubjectsWithProperty();  
model.listObjectsOfProperty();
```

And we were able to locate the specific information we needed from the model.

However, as we have discussed, this type of model interrogation is often considered as a method that provides a low-level view of the model. Query language such as SPARQL, on the other hand, can offer a more compact and powerful query notation for the same result. Therefore, when it comes to building agents such as our Follow-Your-Nose agent, it is sometimes more compact and more efficient to use SPARQL queries.

To help using SPARQL in applications, Jena provides a query engine called ARQ which supports SPARQL query language, including a set of powerful APIs that we can use in our applications.

In this section, we will re-write our Follow-Your-Nose agent so that SPARQL queries will be used instead of simple model interrogation. By doing so, not only we can have a better agent but also we can show you how to submit and execute SPARQL queries by using Jena APIs.

14.3.1 In-memory SPARQL Operation

Before we start, understand that the Follow-Your-Nose agent developed in the previous section had all the data models in memory, and all the model interrogation was also executed against these in-memory models.

In this section, this will continue to be the case. The SPARQL queries we are going to use are therefore executed locally in memory. It is certainly possible to execute a SPARQL query remotely, with the query result returned back to the client. We will cover this in the next section.

Let us start with our rewrite. A review of our Follow-Your-Nose agent indicates that the only Java class we need to change is the `FollowYourNose.java` in List 14.2. More specifically, we need to replace the query part in these two methods:

```
collectData()
updateURICollection()
```

The query used in `collectData()` method is fairly simple; all it does is try to find everything that has been said about Roger Federer. The SPARQL query shown in List 14.6 will accomplish exactly the same thing.

List 14.6 SPARQL query to find everything that has been said about Roger Federer

```
SELECT ?propertyValue ?propertyName
WHERE {
  <http://dbpedia.org/resource/Roger_Federer>
    ?propertyName ?propertyValue.
}
```

The query used in `updateURICollection()` is not difficult either. Again, it tries to find all the statements that have the following pattern:

```
<http://dbpedia.org/resource/Roger_Federer>
owl:sameAs <objectResource>
<subjustResource> owl:sameAs
<http://dbpedia.org/resource/Roger_Federer>
```

And the SPARQL query in List 14.7 exactly accomplishes the same goal.

List 14.7 SPARQL query to find those resources that `owl:sameAs` Roger Federer

```
SELECT ?aliasURI WHERE {
  { <http://dbpedia.org/resource/Roger_Federer>
    <http://www.w3.org/2002/07/owl#sameAs> ?aliasURI.
  }
}
```

```

union
{ ?aliasURI <http://www.w3.org/2002/07/owl#sameAs>
  <http://dbpedia.org/resource/Roger_Federer>.
}
}

```

With all the correct SPARQL queries established, let us now take a look at the steps that are needed to execute SPARQL queries in our application. Note that the steps we are going to discuss are applicable for SELECT queries. For CONSTRUCT queries, DESCRIBE queries, and ASK queries, the steps are slightly different. If you are not familiar with these queries, reviewing [Chap. 6](#) will get you back to speed.

1. Prepare the query string, which represents the SPARQL query that you want to use to get information from RDF model or RDF dataset. Lists 14.6 and 14.7 are examples of query strings.
2. Create a `Query` object by using `QueryFactory.create()` method, with the query string as the input parameter.
3. Create a `QueryExecution` object by calling method `QueryExecutionFactory.create()`; the `Query` object just created and the RDF model are passed in as parameters.
4. Call `execSelect()` method on the `QueryExecution` object to execute the query, which returns the query results.
5. Handle the query results in a loop to get the needed information.
6. Call `close()` method on `QueryExecution` object to release system resource.

List 14.8 shows the new version of `collectData()` method, which implements the above steps.

List 14.8 `collectData()` method is now implemented by using SPARQL query

```

1: private void collectData(Model model, String uri) {
2:
3:     if ( uri == null ) {
4:         return;
5:     }
6:
7:     int factCounter = 0;
8:
9:     String queryString =
10:         "SELECT ?propertyName ?propertyValue " +
11:         "WHERE {" +
12:         " <" + uri + "> ?propertyName ?propertyValue." +
13:         "}";

```

```

14:
15: Query query = QueryFactory.create(queryString);
16: QueryExecution qe =
16a:     QueryExecutionFactory.create(query,model);
17:
18: try {
19:     ResultSet results = qe.execSelect();
20:     while ( results.hasNext() ) {
21:         QuerySolution soln = results.nextSolution() ;
22:         factCounter ++;
23:         Resource res = (Resource)(soln.get("propertyName"));
24:         System.out.print(" - <" + res.getURI() + "> : ");
25:         RDFNode node = soln.get("propertyValue");
26:         if ( node.isLiteral() ) {
27:             System.out.println(((Literal)node).getLexicalForm());
28:         } else if ( node.isResource() ) {
29:             res = (Resource)node;
30:             if ( res.isAnon() == true ) {
31:                 System.out.println("<" + res.getLocalName() + ">");
32:             } else {
33:                 System.out.println( "<" + res.getURI() + ">");
34:             }
35:         }
36:         if ( factCounter >= 10 ) {
37:             break;
38:         }
39:     }
40: }
41: catch(Exception e) {
42:     // doing nothing for now
43: }
44: finally {
45:     qe.close();
46: }
47:
48: }

```

With the discussion of the general steps, List 14.8 is fairly straightforward. Line 9 prepares the query string as shown in List 11.6, which also implements step 1 as discussed above. Line 15 maps to step 2 and line 16 maps to step 3. Line 19 is the execution of the query, and results are also returned (step 4).

The code segment that needs some explanation is in lines 20–39, the loop that handles the query result (step 5). Recall what we have learned in [Chap. 6](#): a given query returns a set of statements as result, and each one of these statements is called a solution. In Jena SPARQL API, one such solution is represented by an instance of

QuerySolution class (line 21), and to get what we are looking for from each solution, we need to use the same variable name as we have used in the query string.

For example, line 23 tries to get the property name. Therefore, `propertyName` as the variable name has to be used, since that variable name is also used in the query string of line 9. Similarly, line 25 tries to get the property value by using the `propertyValue` variable.

Once we get the property value back, a little more work is needed. In general, since a property value is always on the object position, it can be either a literal or a resource. If the property value is a simple literal value, we just print it out (lines 26–27). If it is a resource, it can further be a blank node or a named resource, and we have to handle them differently (lines 28–35).

Finally, line 45 implements step 6, where the query is closed so that all system-related resources are released.

This is the general process of executing a SPARQL query by using Jena SPARQL APIs. You will find yourself using these APIs quite often, and your code should also follow the same pattern as shown here.

List 14.9 shows the new version of `updateURICollection()` method. With what we have learned so far, you should be able to understand it easily.

List 14.9 `updateURICollection()` method is now implemented by using SPARQL query

```

1: private void updateURICollection(URICollection uriCollection,
1:a         String uri,
2:         Model model,Property property) {
3:     if ( uri == null ) {
4:         return;
5:     }
6:
7:     String queryString =
8:         "SELECT ?aliasURI " +
9:         "WHERE {" +
10:        " { <" + uri + "> <" + OWL.sameAs + "> ?aliasURI. } " +
11:        " union " +
12:        " { ?aliasURI <" + OWL.sameAs + "> <" + uri + ">. } " +
13:        " }";
14:
15:     Query query = QueryFactory.create(queryString);
16:     QueryExecution qe =
16a:         QueryExecutionFactory.create(query,model);
17:
18:     try {
19:         ResultSet results = qe.execSelect();
20:         while ( results.hasNext() ) {
21:             QuerySolution soln = results.nextSolution() ;

```

```
22:         RDFNode node = soln.get("aliasURI");
23:         if ( node.isResource() ) {
24:             Resource res = (Resource)node;
25:             if ( res.isAnon() == false ) {
26:                 uriCollection.addNewURI(res.getURI());
27:             }
28:         }
29:     }
30: }
31: catch(Exception e) {
32:     // doing nothing for now
33: }
34: finally {
35:     qe.close();
36: }
37: }
```

Now we have a Follow-Your-Nose agent that is completely written by using SPARQL queries. Run it, you should see exactly the same result.

14.3.2 Using SPARQL Endpoints Remotely

Note that so far in this chapter, we have been downloading the RDF models or datasets into our local memory and then processing them in our memory. The problem associated with this approach is quite obvious. downloading a large data file is always time consuming, and sometimes, the file could be large enough for our limited memory to handle, as we have also discussed in the previous chapter.

A more practical way, especially when you are developing large-scale real applications, is to submit the query across the Internet and post it to the SPARQL endpoint offered by the underlying dataset, the query is then executed on that remote site, and the final result is returned back to the application for processing. By doing so, there is no downloading needed; the network bandwidth is free for other use, and only the request and the response are being interchanged.

This kind of remote SPARQL query can also be implemented programmatically. For example, Jena's SPARQL API provides remote query request/response processing, as we will see shortly. Also, note that remote dataset access over SPARQL protocol does require that the underlying dataset provides a SPARQL endpoint that supports remote data access.

At the time of this writing, some datasets on the Linked Data Web support SPARQL endpoints, some don't. For this reason, re-writing our Follow-Your-Nose agent to make it use remote data access is not quite possible, simply because we don't know which dataset offers remote SPARQL endpoint.

In this section, we will therefore simply use one example dataset to show how remote dataset access is implemented. Hope by the time you are reading this book, most datasets on the Linked Data Web will be supporting SPARQL endpoints.

The example dataset we are going to use is the DBpedia dataset, and here is the SPARQL endpoint it supports:

`http://dbpedia.org/sparql`

and List 14.10 shows the code that remotely accesses this dataset. Again, the submitted query is the query shown in List 14.6.

List 14.10 Example of accessing DBpedia remotely

```

1: package test;
2:
3: import com.hp.hpl.jena.query.Query;
4: import com.hp.hpl.jena.query.QueryExecution;
5: import com.hp.hpl.jena.query.QueryExecutionFactory;
6: import com.hp.hpl.jena.query.QueryFactory;
7: import com.hp.hpl.jena.query.QuerySolution;
8: import com.hp.hpl.jena.query.ResultSet;
9: import com.hp.hpl.jena.rdf.model.Literal;
10: import com.hp.hpl.jena.rdf.model.RDFNode;
11: import com.hp.hpl.jena.rdf.model.Resource;
12:
13: public class RemoteSPARQLAccess {
14:
15:     final static String resourceURI =
15a:         "http://dbpedia.org/resource/Roger_Federer";
16:     final static String DBpediaSPARQLEndpoint =
16a:         "http://dbpedia.org/sparql";
17:
18:     public static void main(String[] args) {
19:
20:         String queryString =
21:         "SELECT ?propertyName ?propertyValue " +
22:         "WHERE {" +
23:         " <" + resourceURI + "> ?propertyName ?propertyValue." +
24:         "}";
25:
26:         Query query = QueryFactory.create(queryString);
27:         QueryExecution qe = QueryExecutionFactory.sparqlService
27a:             (DBpediaSPARQLEndpoint, query);
28:
29:         try {
30:             ResultSet results = qe.execSelect();
31:             while ( results.hasNext() ) {
32:                 QuerySolution soln = results.nextSolution() ;
33:                 Resource res = (Resource) (soln.get("propertyName"));
34:                 System.out.print(" - <" + res.getURI() + "> : ");
35:                 RDFNode node = soln.get("propertyValue");
36:                 if ( node.isLiteral() ) {

```



```

37:         System.out.println(((Literal)node).
37a:             getLexicalForm());
38:     } else if ( node.isResource() ) {
39:         res = (Resource)node;
40:         if ( res.isAnon() == true ) {
41:             System.out.println( "<" +
41a:                 res.getLocalName() + ">");
42:         } else {
43:             System.out.println( "<" + res.getURI() + ">");
44:         }
45:     }
46: }
47: }
48: catch(Exception e) {
49:     // doing nothing for now
50: }
51: finally {
52:     qe.close();
53: }
54: }
55:
56: }

```

Compared to List 14.8, the only difference is in line 27; the `Query Execution` instance is created by calling `QueryExecutionFactory.sparqlService()` method, not `QueryExecutionFactory.create()` method anymore. Also, instead of passing in the model to the method (line 16 of List 14.8), the SPARQL endpoint provided by DBpedia is passed.

The above is all that we need to do when accessing a remote SPARQL endpoint; the rest is taken care of for us by Jena. As you can tell, we then receive and process the results as if we were querying an in-memory dataset.

Finally, List 14.11 shows part of the query result.

List 14.11 Part of the result generated by List 14.10

```

- <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> :
  <http://dbpedia.org/ontology/Athlete>
- <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> :
  <http://dbpedia.org/ontology/Person>
- <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> :
  <http://dbpedia.org/ontology/Resource>
- <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> :
  <http://dbpedia.org/ontology/TennisPlayer>
- <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> :
  <http://dbpedia.org/class/yago/AustralianOpenChampions>
- <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> :
  <http://dbpedia.org/class/yago/USOpenChampions>
- <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> :

```

```

<http://dbpedia.org/class/yago/
    TennisPlayersAtThe2000SummerOlympics>
- <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> :
  <http://dbpedia.org/class/yago/
    OlympicTennisPlayersOfSwitzerland>
- <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> :
  <http://xmlns.com/foaf/0.1/Person>
- <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> :
  <http://dbpedia.org/class/yago/
    TennisPlayersAtThe2004SummerOlympics>
- <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> :
  <http://dbpedia.org/class/yago/LivingPeople>
- <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> :
  <http://dbpedia.org/class/yago/WimbledonChampions>
- <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> :
  <http://dbpedia.org/class/yago/TennisPlayer110701180>
- <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> :
  <http://dbpedia.org/class/yago/PeopleFromBasel(city)>
- <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> :
  <http://dbpedia.org/class/yago/Person100007846>
- <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> :
  <http://dbpedia.org/class/yago/SwissTennisPlayers>
- <http://dbpedia.org/property/doublesrecord> : 112-72
- <http://dbpedia.org/property/doublestitles> : 8
- <http://dbpedia.org/ontology/plays> :
  Right-handed; one-handed back-hand

```

14.4 Summary

In this chapter, we have created a Follow-Your-Nose agent in Java. It is our first project in this book; not only it further shows the benefits of the Semantic Web but also it provides us with some frequently used programming techniques for our future development work.

Make sure you understand the following about the Follow-Your-Nose method and its Java implementation:

- its basic concept and steps;
- some technical details about Follow-Your-Nose agent, such as finding the properties that can be used as links, depth-first search vs. breadth-first search, error protection when it comes to dereferencing a given URI;
- possible changes you can make to its current implementation to make it smarter and more efficient.

Besides the technical aspects of a Follow-Your-Nose agent, you should be able to appreciate more about the vision of the Semantic Web. For example,

- using URI names for resources, and following the open Linked Data principles, these two together make it possible for Follow-Your-Nose method to work on the Web of Linked Data;
- it will be extremely difficult to implement a Follow-Your-Nose agent with good scalability on the traditional document Web, and you should understand the reason;
- not only it is easy to build a scalable Follow-Your-Nose agent on the Semantic Web but also it is almost effortless to maintain such an agent;
- the facts collected by a Follow-Your-Nose agent can also be easily queried by using a query language such as SPARQL.

Finally, this chapter also shows some useful details about issuing SPARQL queries programmatically. More specifically,

- understand the basic steps of issuing SPARQL queries and retrieving the query result in your Java project;
- understand the benefit of remotely querying datasets using SPARQL endpoint, and finally;
- understand the language constructs to make the remote access.

Chapter 15

More Application Examples on the Semantic Web

The goal of this chapter is to continue showing you the *How-To* part on the Semantic Web. We will build two application examples, which are more complex than the ones we have created in the last two chapters. To some extent, none of these two applications here is final yet; it is up to you to make them more intelligent and powerful.

As the final chapter of this book, we hope to convince you that on the Semantic Web, the possibility of developing different applications is only limited by our imagination. The two application examples presented here will serve as a hint, so you can discover and plan your own, which I hope will show more values of the Semantic Web.

15.1 Building Your Circle of Trust: A FOAF Agent You Can Use

This is the first part of this chapter, and we will build a FOAF agent so that you will be receiving only secured incoming e-mails. The rest of this section will present the details of this agent.

15.1.1 Who Is on Your E-mail List?

It is safe to say that most of us use at least one e-mail system of some kind. As much as we love it as a communication tool, we have also realized that there are several things that can threaten the usefulness of a given e-mail system. For example, e-mail bombardment, spamming, phishing (to acquire sensitive information fraudulently, such as your user name, password, credit card number) and certainly e-mail worms, just to name a few.

To lessen these threats, different e-mail systems have implemented different security measurements. One solution is to simply block any e-mail address that is never used in the system before. For example, if a given incoming e-mail is sent from an e-mail address that you have never exchanged e-mail with, the system will directly route this e-mail into the Spam box.

Obviously, this strategy will result in the lost of some e-mail messages that are actually important. On the other hand, if a given e-mail address is stored in the address book before hand, the system will assume this e-mail address is trustable and will not trash any e-mail message from this address even if there has never been any communication with this address yet.

Therefore, to take advantage of this security measurement and also make sure that you are getting all the e-mails, a good solution is to add all the trustworthy e-mail addresses into your contact book ahead of time, a process we call “building your circle of trust.”

The difficult part of this solution is the fact that you will not be able to foresee who would send you e-mail, especially those whom you don’t know yet. In addition, manually editing this circle of trust is quite tedious. An automatic way of building the e-mail address directory and also dynamically updating it would be a much better approach.

With the help from the Semantic Web technology, it is possible to create some automatic tools like this. One solution is based on the linked FOAF documents on the Web, and we will discuss it in this section. The reason of using linked FOAF data is obvious; it is Linked Data which machine can process, and it is about human networking with e-mail address as one of its common data elements.

15.1.2 The Basic Idea

Let us think about a given FOAF document. Obviously, if its author has indicated that she/he has a friend by using `foaf:knows` property, it is then safe to collect that person’s e-mail address into the circle of trust of the author.

Based on this approach, when we scan the author’s FOAF file, we should at least collect all the e-mail addresses of the friends that have been mentioned in this document. This step can be easily done by following the `foaf:knows` property.

To extend the author’s e-mail list, the next step is to explore his/her social network. To do so, the following assumption is made; if any of the author’s friends is trustable, this friend’s friend should also be trustable, and this friend’s friend’s friend is also trustable, so on and so forth, and all their e-mail addresses can all be included in the circle of trust. This is a reasonable and plausible assumption – after all, FOAF project itself is about “Friend of a Friend”!

Therefore, to extend the e-mail list for the author, we will take one of the author’s friends, and check her/his FOAF document to collect the e-mail addresses of all her/his friends. We will then repeat this collection process until all the documents from all the friends of friends have been explored.

To put this idea into action, let us start with Dan Brickley, the creator of FOAF project. We will use his FOAF file as the starting point and build a trusted e-mail list for him. To see how we are going to proceed, take a look at the List 15.1, which is taken from his current FOAF document.

List 15.1 Part of the statements from Dan Brickley's FOAF document

```

1: <knows>
2:   <Person>
3:     <mbox rdf:resource="mailto:libby.miller@bristol.ac.uk"/>
4:     <mbox rdf:resource="mailto:libby@asemantics.com"/>
5:   </Person>
6: </knows>
7:
8: <knows>
9:   <Person
10:     rdf:about="http://www.w3.org/People/Berners-Lee/card#i">
11:     <name>Tim Berners-Lee</name>
12:     <isPrimaryTopicOf rdf:resource=
13:       "http://en.wikipedia.org/wiki/Tim_Berners-Lee"/>
14:     <homepage rdf:resource=
15:       "http://www.w3.org/People/Berners-Lee/" />
16:     <mbox rdf:resource="mailto:timbl@w3.org"/>
17:     <rdfs:seeAlso rdf:resource=
18:       "http://www.w3.org/People/Berners-Lee/card"/>
19:   </Person>
20: </knows>
21:
22: <knows>
23:   <Person>
24:     <name>Dean Jackson</name>
25:     <rdfs:seeAlso rdf:resource=
26:       "http://www.grorg.org/dean/foaf.rdf"/>
27:     <mbox rdf:resource="mailto:dean@w3.org"/>
28:     <mbox rdf:resource="mailto:dino@grorg.org"/>
29:     <mbox_sha1sum>
30:       6de4ff27ef927b9ba21ccc88257e41a2d7e7d293</mbox_sha1sum>
31:     <homepage rdf:resource="http://www.grorg.org/dean/" />
32:   </Person>
33: </knows>

```

Note that List 15.1 shows three different ways of adding a friend into a FOAF document:

- Lines 1–6 is the simplest way to add one friend: only e-mail address is given, no URI is included, and no `rdfs:seeAlso` is used when describing the friend.
- Lines 18–27 is the second way of describing a friend: e-mail address is provided, `rdfs:seeAlso` is used to provide more information about the friend (line 21); however, no URI of the friend is given.
- Lines 8–16 is the third way of adding a friend: the URI of the friend is provided (line 9), e-mail address is given, and `rdfs:seeAlso` is also used (line 14).

These three different ways of describing a friend in a FOAF document have no effect on how we collect the e-mail addresses from these friends, but they do mean

different methods of exploring the social network. More specially, the third one gives us the most information we can use when it comes to exploring the network:

- We can dereference the given URI of the friend to get more e-mail addresses.

On the Web of Linked Data, we can assume data publishers do follow the principles of Linked Data. As a result, if we dereference the URI of a friend, we should be getting an RDF document that describes this friend, and we can expect to find e-mails addresses of her/his friends.

For example, in List 15.1, one URI is given by the following (line 9) URL:

```
http://www.w3.org/People/Berners-Lee/card#i
```

and we can get e-mail addresses of Tim Berners-Lee's friends when we dereference this URI.

- We can follow the `rdfs:seeAlso` link to extend the network even more.

`rdfs:seeAlso` does not have formal semantics defined, i.e., its `rdfs:domain` property and `rdfs:range` property are all general `rdfs:Resource` class. However, it does specify another resource that might provide additional information about the subject resource. As a result, we can follow this link, i.e., dereference the object resource, which should provide information about this friend, and it is also possible to locate some more e-mail addresses of this friend's friends.

For example, in List 15.1, line 14 provides a URI that can be dereferenced as discussed above.

With this understanding of the third method, the second method (lines 18–27 in List 15.1) is simpler: all we can do is to follow the `rdfs:seeAlso` link since there is no friend URI we can dereference. Therefore, the second way of adding friends gives us less chance for expanding the e-mail list.

By the same token, the first method (lines 1–6, List 15.1) does not allow us to do any further exploring directly; all we can do is to collect the e-mail address there and stop. Note that theoretically we can still find the FOAF file of this friend (Libby Miller); however we are not going to do that here in this example, we simply stop exploring the sub-network headed by this particular friend.

As a side note, how do we find a person's FOAF file if we have only his e-mail address from `foaf:mbox` property? Since `foaf:mbox` is an inverse functional property, the simplest solution is to visit the Web and check each and every single FOAF document you can encounter until you have located one document whose main subject also assumes the same value on `foaf:mbox` property. That main subject, a `foaf:Person` instance, should be the resource you are looking for. As you can tell, the reason why we are not implementing this solution is mainly due to the consideration of the efficiency of our agent. If you ever want to change this example into a real-world application, you might consider implementing this part.

15.1.3 Building the *EmailAddressCollector* Agent

15.1.3.1 *EmailAddressCollector*

Based on our previous discussion, the algorithm of the e-mail list builder agent is fairly straightforward:

0. Make Dan Brickley's URI our `currentURI`.
1. Dereference `currentURI`, which will give us an RDF document back, call it `currentRDFDocument`.
2. From `currentRDFDocument`, collect all the e-mail addresses of Dan's friends.
3. For each friend, if she/he has a representing URI, add this URI into `friendsToVisit` (an `URICollection` instance); if she/he also has a `rdfs:seeAlso` property defined, collect the object URI of this property into `friendsToVisit` as well.
4. Repeat step 3 until all friends are covered.
5. Retrieve a new URI from `friendsToVisit` collection, make it `currentURI`, go back to step 1. If no URI is left in `friendsToVisit` collection, stop.

And the code is given in List 15.2.

List 15.2 *EmailAddressCollector.java* definition

```

1: package test;
2:
3: import java.util.HashSet;
4: import java.util.Iterator;
5:
6: import com.hp.hpl.jena.query.Query;
7: import com.hp.hpl.jena.query.QueryExecution;
8: import com.hp.hpl.jena.query.QueryExecutionFactory;
9: import com.hp.hpl.jena.query.QueryFactory;
10: import com.hp.hpl.jena.query.QuerySolution;
11: import com.hp.hpl.jena.query.ResultSet;
12: import com.hp.hpl.jena.rdf.model.Literal;
13: import com.hp.hpl.jena.rdf.model.Model;
14: import com.hp.hpl.jena.rdf.model.ModelFactory;
15: import com.hp.hpl.jena.rdf.model.RDFNode;
16: import com.hp.hpl.jena.rdf.model.Resource;
17: import com.hp.hpl.jena.sparql.vocabulary.FOAF;
18: import com.hp.hpl.jena.vocabulary.RDFS;
19:
20: public class EmailAddressCollector {
21:
22:     private URICollection friendsToVisit = null;
23:     private HashSet emailAddresses = null;
24:
25:     public EmailAddressCollector(String uri) {
26:         emailAddresses = new HashSet();

```



```

27:     friendsToVisit = new URICollection();
28:     friendsToVisit.addNewURI(uri);
29: }
30:
31: public void work() {
32:
33:     // get the next URI to work on (step 1 in algorithm)
34:     String currentURI = friendsToVisit.getNextURI();
35:     if ( currentURI == null ) {
36:         return;
37:     }
38:
39:     try {
40:         System.out.println("\n...visiting <" +
40a:             currentURI + ">");
41:
42:         // dereference currentURI (step 1 in algorithm)
43:         Model currentRDFDocument =
43a:             ModelFactory.createDefaultModel();
44:         currentRDFDocument.read(currentURI);
45:
46:         // collect everything about currentURI
46a:         // (step 2-4 in algorithm)
47:         int currentSize = friendsToVisit.getSize();
48:         collectData(currentRDFDocument, currentURI);
49:         System.out.println("..." + emailAddresses.size() +
49a:             " email addresses collected.");
50:         System.out.println("..." + (friendsToVisit.getSize() -
50a:             currentSize) + " new friends URI added.");
51:         System.out.println("...all together " +
51a:             friendsToVisit.getSize() + " more to visit.");
52:
53:     } catch (Exception e) {
54:         System.out.println("*** errors when handling (" +
54a:             currentURI + ") ***");
55:     }
56:
57:     // extend the social network by following
57a:     // friends of friends' (step 5 in algorithm)
58:     work();
59:
60: }
61:
62: private void collectData(Model model, String uri) {
63:
64:     if ( uri == null ) {
65:         return;
66:     }
67:
68:     String queryString =
69:         "SELECT ?myself ?who ?email ?seeAlso " +

```

```

70:     "WHERE {" +
71:     "   ?myself <" + FOAF.knows + "> ?who. " +
72:     "   optional { ?who <" + FOAF.mbox + "> ?email. }" +
73:     "   optional { ?who <" + RDFS.seeAlso + "> ?seeAlso. }" +
74:     "   }";
75:
76:
77: Query query = QueryFactory.create(queryString);
78: QueryExecution qe =
79a:     QueryExecutionFactory.create(query,model);
80:
81: try {
82:     ResultSet results = qe.execSelect();
83:     while ( results.hasNext() ) {
84:         QuerySolution soln = results.nextSolution() ;
85:         Resource who = (Resource) soln.get("who");
86:
87:         // step 2 in algorithm
88:         Resource email = (Resource) soln.get("email");
89:         if ( email != null ) {
90:             if ( email.isLiteral() ) {
91:                 emailAddresses.add(((Literal)email).
91a:                    getLexicalForm());
92:             } else if ( email.isResource() ) {
93:                 emailAddresses.add(email.getURI());
94:             }
95:         } else {
96:             // there is no foaf:mbox property value
96a:            // for this friend
97:         }
98:
99:         // step 3 in algorithm
100:        if ( who.isAnon() == false ) {
101:            friendsToVisit.addNewURI(who.getURI());
102:        } else {
103:            // there is no URI specified for this friend
104:        }
105:
106:        // step 3 in algorithm
107:        Resource seeAlso = (Resource) soln.get("seeAlso");
108:        if ( seeAlso != null ) {
109:            if ( seeAlso.isLiteral() ) {
110:                friendsToVisit.addNewURI(((Literal)seeAlso).
110a:                    getLexicalForm());
111:            } else if ( seeAlso.isResource() ) {
112:                friendsToVisit.addNewURI(seeAlso.getURI());
113:            }
114:        } else {
115:            // there is no rdfs:seeAlso property specified
115a:           // for this friend

```

```

116:         }
117:
118:     }
119: }
120: catch(Exception e) {
121:     // doing nothing for now
122: }
123: finally {
124:     qe.close();
125: }
126:
127: }
128:
129: public void showemailAddresses() {
130:     if ( emailAddresses != null ) {
131:         Iterator it = emailAddresses.iterator();
132:         int counter = 1;
133:         while ( it.hasNext() ) {
134:             System.out.println(counter + ": " +
134a:                             it.next().toString());
135:             counter ++;
136:         }
137:     }
138: }
139:
140: }

```

Lines 25–29 is the constructor of the `EmailAddressCollector` agent. The URI that is passed in to this constructor represents the person for whom we would like to create an e-mail list. In our example, this will be Dan Brickley’s URI given by the following:

```
http://danbri.org/foaf.rdf#danbri
```

Line 26 creates a `HashSet` object, `emailAddresses`, which holds all the collected e-mail addresses. The reason of using a `HashSet` is to make sure there is no repeated e-mail address in this collection.

Line 27 creates another important collection, `friendsToVisit`, which is an instance of class `URICollection`. As we have discussed earlier, this class uses a stack as its underlying data structure to implement depth-first search. In addition, the implementation of this class makes sure no URI is repeatedly visited. For our application, `friendsToVisit` holds the URIs that needed to be dereferenced next. A given URI in this collection represents either a friend or an object value of `rdfs:seeAlso` property. Also note that at the time the agent gets started, Dan Brickley’s URI is the only URI stored in `friendsToVisit`; it is used as the seed URI (line 28) for the whole collecting process.

Lines 31–60 is the key method, `work()`, which implements the algorithm presented at the beginning of this section. Line 34 gets the next URI that needs to be

dereferenced from `friendsToVisit` collection; lines 43–44 dereference this URI and create a default RDF model in memory. These steps map to step 1 as described in the algorithm.

Line 48 collects the necessary information from the created model, and the details are implemented in `collectData()` method. Once `collectData()` is executed, steps 2–4 as described in our algorithm are completed and we are ready to move on to the next URI contained in the `friendsToVisit` collection.

Obviously, the handling of the next URI is an exact repeat of the above steps. As a result, we process the next friend’s document by recursively calling `work()` method (line 58). This recursive calling step maps to step 5 as described in our algorithm.

`collectData()` (lines 62–127) is where the data collection work is done. Let us understand it by starting with the SPARQL query string (lines 68–74). List 15.3 shows this query in a more readable format.

List 15.3 SPARQL query used in line 68 of List 15.2

```

1: SELECT ?myself ?who ?email ?seeAlso
2: WHERE {
3:     ?myself <http://xmlns.com/foaf/0.1/knows> ?who.
4:     optional {
5:         ?who <http://xmlns.com/foaf/0.1/mbox> ?email.
6:     }
7:     optional {
8:         ?who <http://www.w3.org/2000/01/rdf-schema#seeAlso>
8a:         ?seeAlso.
9:     }
10: }
```

This query finds all the friends (identified by `foaf:knows` property) from the current RDF model and also gets their `foaf:mbox` values and `rdfs:seeAlso` values if available.

To let you understand it better, Table 15.1 shows part of the result if we had written a small separate Java class just to execute the query. Note that in order to fit into the page, `?myself` field is not included in Table 15.1.

As shown in Table 15.1, some friends have both `foaf:mbox` and `rdfs:seeAlso` defined (b0, b2 and ***), some only have `foaf:mbox` or `rdfs:seeAlso` defined (b4, b5), and some have no `foaf:mbox` nor `rdfs:seeAlso` defined (b10). In addition, for all the friends he knows, Dan Brickley has not used any URI to represent them, except for Tim Berners-Lee (see the *** line, we put *** there in order to fit into the page). Therefore, from Dan’s FOAF file, that is the only URI that identifies a person; all the other collected URIs are coming from values of `rdfs:seeAlso` property.

With the understanding of the query string, the rest of the code is easy to understand. The above query is created and executed (lines 77–78), and the result set is examined line by line to get the required information (lines 82–118). More

Table 15.1 Part of the result when running the query shown in List 15.3

who	email	seeAlso
_:b0	<mailto:em@w3.org>	< http://purl.org/net/eric/webwho.xrdf >
_:b2	<mailto:barstow@w3.org>	< http://www.w3.org/People/Barstow/webwho.rdf >
...		
_:b4	<mailto:libby.miller@bristol.ac.uk>	
_:b5		< http://people.w3.org/amy/foaf.rdf >
...		
_:b10		
***	<mailto:timbl@w3.org>	< http://www.w3.org/People/Berners-Lee/card >

***: <http://www.w3.org/People/Berners-Lee/card#i>

specifically, lines 88–97 implement step 2 as described in our algorithm, where the e-mail addresses are collected. Note that although most FOAF files use a resource as the value of `foaf:mbox` property (lines 92–94), some FOAF files use value string as its value (lines 90–92); therefore we need to collect them by using different methods. Either way, the collected e-mail addresses are stored in the `emailAddresses` collection (lines 91 and 93).

Lines 100–116 implement step 3 of our algorithm. Since it is perfectly fine not to use any URI to represent a given person, we first have to check if a friend is identified by an URI or a blank node (line 100). If a friend is identified by a URI, this URI will be visited later to expand the e-mail network, and for now, we simply save it in our `friendsToVisit` collection (line 101). If a friend is represented by a blank node, there is not much we can do (line 103) except to continue checking whether there is a `rdfs:seeAlso` property value we can use to explore the sub-network that is headed by this friend.

Lines 107–116 are used to check `rdfs:seeAlso` property. Similarly, since `rdfs:seeAlso` property can use either datatype or object type as its value, we need to check both (lines 108–114). Either way, if a value of this property is found, we save it into `friendsToVisit` collection so we can visit it later (lines 110 and 112).

The above examination is repeated for all the records in the result set. Once this process is done, we may have gathered a number of e-mail addresses and identified quite a few new URIs yet to visit. This is precisely where the circle of trust is being built and expanded.

At this point, we have discussed the details of our implementation. It is obvious that this way of expanding the e-mail network could result in a fairly deep and wide search tree that needs to be covered. For example, when I ran this collector on my own PC, after 1 hour or so, it ran out of memory, with about 150 e-mail addresses collected and still 1,000 more friends of friends yet to explore.

Unless you have access to large machines which provide much stronger computing power than a home PC, you do want to change the code a little bit, in order to

make a complete run. List 15.4 shows the modified definition of `URICollection` class. Since we have seen this class already, only the method with the change is included here.

List 15.4 `URICollection.java` definition

```
1: package test;
2:
3: import java.net.URI;
4: import java.net.URISyntaxException;
5: import java.util.HashSet;
6: import java.util.Iterator;
7: import java.util.Stack;
8:
9: public class URICollection {
10:
11:     private Stack URIs = null;
12:     private HashSet visitedURIs = null;
13:
14:     public URICollection() {
15:         URIs = new Stack();
16:         visitedURIs = new HashSet();
17:     }
18:
19:     public void addNewURI(String uri) {
20:         if ( uri == null ) {
21:             return;
22:         }
23:         // testing purpose: we don't want to go into
23a: // these social sites
24:         URI myURI = null;
25:         try {
26:             myURI = new URI(uri);
27:             if ( myURI.getHost().contains("my.opera.com") ) {
28:                 return;
29:             }
30:             if ( myURI.getHost().contains("identi.ca") ) {
31:                 return;
32:             }
33:             if ( myURI.getHost().contains("advogato.org") ) {
34:                 return;
35:             }
36:             if ( myURI.getHost().contains("foaf.qdos.com") ) {
37:                 return;
38:             }
39:             if ( myURI.getHost().contains("livejournal.com") ) {
40:                 return;
41:             }
42:             if ( myURI.getHost().contains("openlinksw.com") ) {
43:                 return;
44:             }

```

```

45:     if ( myURI.getHost().contains("rdf.opiumfield.com") ) {
46:         return;
47:     }
48:     if ( myURI.getHost().contains("ecademy.com") ) {
49:         return;
50:     }
51:     if ( myURI.getHost().contains("revyu.com") ) {
52:         return;
53:     }
54:     if ( myURI.getHost().contains("xircles.codehaus.org") ) {
55:         return;
56:     }
57: } catch (URISyntaxException e1) {
58:     // e1.printStackTrace();
59: }
60: // end of testing purpose: you can clean this part out to
60a: // do more crawling
61:
62: try {
63:     if ( visitedURIs.contains(uri) == false ) {
64:         visitedURIs.add(uri);
65:         URIs.push(uri);
66:     }
67: } catch(Exception e) {};
68: }
69:
70: public String getNextURI() {
...
75: }
76:
... // other methods you have seen already
103: }
104:
105: }

```

As you can tell, lines 23–60 are added to make the crawling process more focus. More specifically, we take a look at the URI that is passed in for collection; if it is from a social Web site, we simply reject it and move on. In general, these social sites can link quite a large number of people into their networks, and we would not want them to be included in our e-mail list.

With the above change in place, I could finish the run in a little over an hour, with 216 e-mail addresses collected for Dan Brickley.

Finally, List 15.5 is the driver I use to run the collector, and not much explanation is needed here.

List 15.5 Test driver for our e-mail collector

```

1: package test;
2:
3: public class EmailListBuilderTester {

```

```

4:
5:     // http://www.w3.org/People/Berners-Lee/card#i
6:     public static final String startURI =
6a:         "http://danbri.org/foaf.rdf#danbri";
7:
8:     public static void main(String[] args) {
9:
10:         EmailAddressCollector eac =
10a:             new EmailAddressCollector(startURI);
11:         eac.work();
12:
13:         // here are all the collected email addresses
14:         eac.showemailAddresses();
15:
16:     }
17:
18: }

```

15.1.3.2 Running the `EmailAddressCollector` Agent

Now, find the following class definitions from the package you have downloaded:

```

EmailAddressCollector.java
URICollection.java
EmailAddressCollectorTester.java

```

Build the application and fire it up, you can see the search is in action. For example, the following is part of the output when I was running the agent using Dan's FOAF file as the starting point:

```

...visiting <http://danbri.org/foaf.rdf#danbri>
...21 email addresses collected.
...17 new friends URI added.
...all together 17 more to visit.

...visiting <http://heddley.com/edd/foaf.rdf>
...43 email addresses collected.
...10 new friends URI added.
...all together 26 more to visit.

...visiting <http://clark.dallas.tx.us/kendall/foaf.rdf>

```

And you can add more output lines into the source code so that you can see more clearly about what is happening during the search. The following is part of the collected e-mail addresses:

```

1: mailto:nova@radarnetworks.com
2: mailto:aditkal@yahoo.com

```


3: mailto:lac@ecs.soton.ac.uk
4: mailto:rafa@sidar.org
5: mailto:et@progos.hu
6: mailto:alrerer@mit.edu
7: mailto:kidehen@openlinksw.com
8: mailto:nmg@ecs.soton.ac.uk
9: mailto:mauro.buratti@nontorno.com
10: mailto:stefan.decker@deri.org
11: mailto:wvasconc@csd.abdn.ac.uk
12: mailto:joe.brickley@btopenworld.com
13: mailto:simonstl@simonstl.com
14: mailto:em@zepheira.com
15: mailto:simon.price@bristol.ac.uk
16: mailto:hendler@cs.umd.edu
17: mailto:giles@gilest.org
18: mailto:ian.sealy@bristol.ac.uk
19: mailto:b.j.norton@open.ac.uk
20: mailto:danny666@virgilio.it
21: mailto:ben@benhammersley.com
22: mailto:swh@ecs.soton.ac.uk
23: mailto:elias@torrez.us
24: mailto:dino@w3.org
25: mailto:phil@chimpen.com
...

To add these e-mail addresses into your e-mail system's address book, all you need to do is cut-and-paste. And now, you have created your circle of trust based on your FOAF network, and this is made possible by the open Linked Data principles and the Semantic Web technologies.

It is also important to realize the dynamic nature of the Web: your friend's circle is growing and so is yours: you will get to know new friends, and your friends' friends will get to expand their circles, so on and so forth. As a result, it is important to run this agent from time to time, you will see a quick grow of the harvested e-mail addresses.

15.1.4 Can You Do the Same for Traditional Web?

As usual, try to figure out a plan which helps you to implement exactly the same agent on the traditional Web. The truth is that it is going to be quite difficult, if not completely impossible. And even if you can do it now, wait until you run it again a while later: you will need to make significant changes to make it work again. I will leave this to you to consider, which will for sure make you appreciate the value of the Semantic Web more.

15.2 A ShopBot on the Semantic Web

So far in this chapter, we have built a simple FOAF agent, which works in the environment of the Semantic Web and exhibits the traits that we have always been looking for from Web agents: a much smarter way of accomplishing the given task, with the required scalability and maintainability.

In the second half of this chapter, we are going to create another interesting Web application: a smart ShopBot. We will first discuss ShopBot in general and will then build a ShopBot step by step in a simulated Semantic Web world. This will not only show you some necessary techniques when it comes to development on the Semantic Web but will also give you some hints for other possible applications.

15.2.1 A ShopBot We Can Have

A ShopBot is a software agent that works on the Web. For a particular product specified by the consumer, it searches different online stores and retailers so as to provide the consumer with the prices offered by these stores and retailers.

There are quite a few Web sites powered by ShopBot. For example,

www.pricescan.com
www.pricewatch.com
www.dealtime.com

You can visit these sites to get a feeling about ShopBot.

It is important to realize that ShopBots normally operate by a form of screen scraping (there have been some changes in recent years, as discussed below), meaning that they download the pages and get the necessary information by parsing the content. In general, screen scraping is a method used to extract data from a set of Web pages with the goal of consolidating the data on a single Web page, which can then be conveniently viewed by a user.

The fact that ShopBots operate by screen scraping also dictates their basic assumption during the course of their work. Once a page is downloaded, a ShopBot will have to search for the name of the specified product. If it can locate the item successfully, it will then search for the nearest set of characters that has a dollar sign, assuming this set of characters is the price of the item.

Obviously, this solution may not be as robust as we expect. For instance, imagine that on a given Web page, after the name of the product, the first set of characters that has a dollar sign is actually the suggested retail price, and somewhere down the page, another set of characters that also has a dollar sign is the current sale price. A ShopBot may completely miss the target in this case.

To handle the situations like above, a ShopBot just has to be “smarter”, either by applying some heuristics that works for most of the time or by processing the pages on a case-by-case basis. In fact, some Web sites powered by ShopBots have established agreements with big retailers to make sure the price information can be correctly obtained.

In recent years, some ShopBots are created based on the Mashup concept we have introduced in [Chap. 11](#). Instead of screen scraping, these new breeds of ShopBots obtain the price and related information of a product by consuming the Web services provided by a given retailer. This is certainly a much better solution compared to screen scraping, since the results from the Web service calls are structured data where no assumption is needed. However, two issues still exist. First, not all the retailers offer Web services, and second, Web services offered by different retailers have to be consumed by using different APIs; the development team of a ShopBot therefore has to learn a new set of APIs every time a new retailer is considered. As a result, this solution is still not a scalable one.

In fact, even with better Mashup support and smarter heuristics, the ShopBots we can build today can only offer limited usage to us. More specifically, we have to tell the ShopBot exactly what is the product we are looking for; we cannot simply *describe* what we want to buy.

For example, we cannot tell the ShopBot to search for a camera that is manufactured by Nikon and has at least 6 megapixels as its picture quality, and can support a lens which has an 18–200mm zoom. Instead, we have to specifically tell it to find a product that has a model number given by Nikon D200.

This clearly presents some difficulty for probably most of us. Quite often, we can only describe what we want and we do not have a particular product in mind. Ideally, a particular product that satisfies our needs should be part of the information we get by using a ShopBot.

To solve this issue, some Web sites powered by ShopBots allow you to search for a product first by using a search engine-like interface on their sites. For example, you can type in the word camera to start your search. However, it is then up to you to read all the returned items to figure out exactly which product you would like to have. Again, this is a tedious manual process that can take up lots of time before you settle down with one specific product.

The Semantic Web seems to be the right solution for all these issues. By adding semantics to a retailer's Web page, the content of the Web site becomes understandable to a ShopBot. For example, an added RDF file describing the products is for the ShopBot to read, while the traditional HTML content is still there for human eyes.

In the next few sections, we will construct a ShopBot we really want: a ShopBot that can accept our description of a product instead of a specific model number, a ShopBot that is easy to maintain and also has excellent scalability.

15.2.2 A ShopBot We Really Want

15.2.2.1 How Does It Understand Our Needs?

To make sure a ShopBot can understand our description about what we want, using RDF model to express the description is a good choice. For a casual user, a HTML form can be presented to him, and he/she can express his/her needs by filling out the

form, which is then mapped to an RDF model. In other words, there is no need for a user to learn RDF beforehand.

Now, let us assume we would like to buy a camera, and the following RDF document shown in List 15.6 describes what exactly we want for our ideal camera.

List 15.6 An RDF document describing what camera we are looking for

```

1: <?xml version="1.0" encoding="UTF-8"?>
2: <rdf:RDF
3:     xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
4:     xml:base="http://www.liyangyu.com/shopbot/request">
5:     <!-- here is what I am looking for -->
6:     <Digital rdf:ID="myDigitalCamera"
7:         xmlns="http://www.liyangyu.com/camera#">
8:         <effectivePixel rdf:datatype=
9:             "http://www.liyangyu.com/camera#MegaPixel">
10:             6.0</effectivePixel>
11:         <body>
12:             <Body>
13:                 <shutterSpeed>
14:                     <ValueRange>
15:                         <minValue rdf:datatype=
16:                             "http://www.w3.org/2001/XMLSchema#float">
17:                             0.0005</minValue>
18:                         </ValueRange>
19:                     </shutterSpeed>
20:                 </Body>
21:             </body>
22:         <lens>
23:             <Lens>
24:                 <focalLength rdf:datatype=
25:                     "http://www.w3.org/2001/XMLSchema#string">
26:                     18-200mm</focalLength>
27:                 <aperture>
28:                     <ValueRange>
29:                         <minValue rdf:datatype=
30:                             "http://www.w3.org/2001/XMLSchema#float">
31:                             1.8</minValue>
32:                         <maxValue rdf:datatype=
33:                             "http://www.w3.org/2001/XMLSchema#float">
34:                             22</maxValue>
35:                         </ValueRange>
36:                     </aperture>
37:                 </Lens>
38:             </lens>

```

```

32:
33:   </Digital>
34:
35: </rdf:RDF>

```

Figure 15.1 represents List 15.6 in an RDF graph format.

Note that that in Fig. 15.1, an oval represents a class and a box represents an instance or resource; the URI of the resource is included inside the box. If the box is representing a blank node, there is no URI given inside the box. Also, if a property value takes a simple string or a float number, the number or string is simply used without having a box.

Since the only purpose of List 15.6 is to describe our target camera, the URI we use to identify this camera will not be reused by anyone, and the properties we have said about this ideal camera are not important to the outside world either. Therefore, we have used an `rdf:ID` (line 7) together with `xml:base` attribute (line 3) to identify the camera, which has the following URI:

`http://www.liyangyu.com/shopbot/request#myDigitalCamera`

and all the property values, such as `myCamera:Lens`, `myCamera:Body`, and `myCamera:ValueRange`, are represented by blank nodes. Notice that we do use our camera ontology in List 15.6 (the default namespace in line 7), and we will come back to this point later.

Now, based on List 15.6, here is what we are looking for:

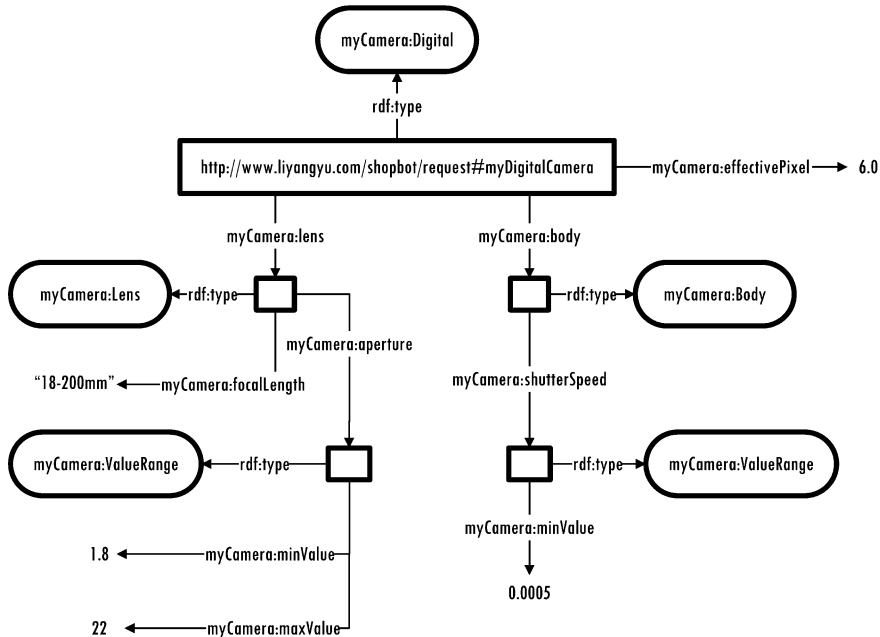


Figure 15.1 List 15.6 in a graph format

- We want a digital camera (line 7).
- The camera should have at least a 6.0-megapixel resolution (line 9).
- The shutter speed of the camera should be able to reach as fast as 1/2,000 s (lines 11–19).
- The camera should have a lens that has a zoom range of 18–200 mm, and minimum aperture is given by 1.8, with the maximum aperture having a value of 22 (lines 21–30).

As you can see, to be able to describe our needs like the above is a great enhancement to the user experience when using the ShopBot: all we have described is a camera that satisfies our needs, and we don't have to specify a product model at all.

Note that our camera ontology is specified in line 7 by using the following statement:

```
xmlns="http://www.liyangyu.com/camera#"
```

and List 15.6 is equivalent to List 15.7, which probably looks more familiar to you.

List 15.7 An equivalent form of List 15.6

```
1: <?xml version="1.0" encoding="UTF-8"?>
2: <rdf:RDF
2a:     xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
3:     xmlns:myCamera="http://www.liyangyu.com/camera#"
4:     xml:base="http://www.liyangyu.com/shopbot/request">
5:
6:     <!-- here is what I am looking for -->
7:
8:     <myCamera:Digital rdf:ID="myDigitalCamera">
9:
10:     <myCamera:effectivePixel rdf:datatype=
10a:         "http://www.liyangyu.com/camera#MegaPixel">
10b:         6.0</myCamera:effectivePixel>
11:
12:     <myCamera:body>
13:         <myCamera:Body>
14:             <myCamera:shutterSpeed>
15:                 <myCamera:ValueRange>
16:                     <myCamera:minValue rdf:datatype=
16a:                         "http://www.w3.org/2001/XMLSchema#float">
16b:                         0.0005</myCamera:minValue>
17:                 </myCamera:ValueRange>
18:             </myCamera:shutterSpeed>
19:         </myCamera:Body>
20:     </myCamera:body>
21:
22:     <myCamera:lens>
23:         <myCamera:Lens>
24:             <myCamera:focalLength rdf:datatype=
24a:                 "http://www.w3.org/2001/XMLSchema#string">
```

```

24b:         18-200mm</myCamera:focalLength>
25:         <myCamera:aperture>
26:           <myCamera:ValueRange>
27:             <myCamera:minValue rdf:datatype=
27a:               "http://www.w3.org/2001/XMLSchema#float">
27b:               1.8</myCamera:minValue>
28:             <myCamera:maxValue rdf:datatype=
28a:               "http://www.w3.org/2001/XMLSchema#float">
28b:               22</myCamera:maxValue>
29:           </myCamera:ValueRange>
30:         </myCamera:aperture>
31:       </myCamera:Lens>
32:     </myCamera:lens>
33:
34: </myCamera:Digital>
35:
36: </rdf:RDF>

```

15.2.2.2 How Does It Find the Next Candidate?

Making the ShopBot understand our need is only the first step; the next step is to make it work as we expected. Before we dive into the details, let us summarize our assumptions for our ShopBot:

1. There is a list of retailers that the ShopBot will visit.
2. Each one of these retailers publishes on the Web its own product catalog documents by using RDF model.
3. When it comes to describing camera in their RDF catalog documents, all of these retailers have agreed to use our camera ontology.

First off, in its most general form, to make sure it will not miss any potential retailer and product, a ShopBot will have to visit the Web just as a crawler does. In this example, we will not ask it to crawl every Web site it has randomly encountered, rather, it will crawl some Web sites from a pre-defined list that has been given to it as part of the input. For instance, this list might include retailers such as BestBuy, RitzCamera, Sam's Club, just to name a few. Obviously, doing so can greatly improve the performance of the ShopBot and can also ensure that our ShopBot will visit only the sites that we trust, which is another important issue on the Web.

The second assumption is vital for an agent that works on the Semantic Web. For instance, BestBuy could have one catalog file for all the PCs it sells, another catalog for all the TVs, another catalog for all the cameras. Furthermore, these catalog files are created by using terms defined in some ontologies, and they have to be published by the retailer on its Web site so that our ShopBot can have access to these catalog files. Clearly, this is one big extra step that has to be taken by the retailers.

Once our ShopBot reaches a retailer's Web site, it will only inspect its published catalog files and skip all the traditional Web documents that are constructed using

HTML on the same Web site. Obviously, this is one important difference between this new ShopBot and the one we currently have in our traditional Web.

The third assumption is to make sure all the retailers are sharing a common vocabulary so that it is easier for the ShopBot to work. This is also the reason why ontology reuse is vital in the world of the Semantic Web. However, in reality, it may as well be true that some retailers who sell cameras have instead used some other camera ontology to describe his items. We will come back to this point in a later section.

With the above assumptions in place, our ShopBot can continue to locate the next retailer and further decide whether any camera products are offered by this retailer.

In general, a given retailer could have a large number of RDF documents published on its Web site, and only a few of these RDF files are real product catalogs. Therefore, when visiting a retailer's Web site, the first step is to decide, for a given RDF file, whether it contains some description of camera products. If the conclusion is yes, it will then be considered as a candidate that can be potentially collected by our ShopBot.

To make this decision, the following observation is the key: if a given catalog RDF file has made use of our camera ontology, this catalog file will be considered as one candidate for further verification. If not, our ShopBot will skip this file.

For example, List 15.6 specifies our request, and the terms used in List 15.6 have been taken from the following ontology namespaces:

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

Now, our ShopBot encounters an RDF document as shown in List 15.8 from a retailer's Web site.

List 15.8 An example catalog document in RDF format

```
1: <?xml version="1.0" encoding="UTF-8"?>
2: <rdf:RDF
3:     xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
4:     xml:base="http://www.retailerExample1.com/onlineCatalog">
5:   <Retailer rdf:ID="Retailer-1"
6:     xmlns="http://www.eBusiness.org/retailer#">
7:     <location>Norcross, GA</location>
8:     <address>6066 Cameron Pointe</address>
9:     <webSite>www.bestStuff.com</webSite>
10:
11:   <catalog rdf:parseType="Collection">
12:
13:     <Notebook rdf:ID="PC_TOP_08"
13a:     xmlns="http://www.ontologyExample.org/ontology/pcspec#">
14:
15:       <memory>
16:       <Memory>
```



```

17:         <mem_capacity>1.0GB</mem_capacity>
18:     </Memory>
19: </memory>
20:
21: <processor>
22:     <Processor>
23:         <cpu_name>Intel Pentium M processor</cpu_name>
24:         <cpu_speed>2.0GHz</cpu_speed>
25:     </Processor>
26: </processor>
27:
28: <harddisk>
29:     <HardDrive>
30:         <hd_capacity>120GB</hd_capacity>
31:     </HardDrive>
32: </harddisk>
33:
34: </Notebook>
35:
36: <Processor rdf:ID="intel_m"
36a: xmlns="http://www.ontologyExample.org/ontology/pcspect#">
37:     <cpu_name>Intel Pentium M processor</cpu_name>
38:     <cpu_speed>2.0GHz</cpu_speed>
39:     <cost>
40:         <Money>
41:             <price>$199.99</price>
42:         </Money>
43:     </cost>
44: </Processor>
45:
46: <Mointor rdf:ID="Monitor-Hanns-G" xmlns=
46a: "http://www.ontologyExample.org/ontology/monitorspec#">
47:     <dimension>
48:         <Dimension>
49:             <size>17</size>
50:             <resolution>1440x900</resolution>
51:         </Dimension>
52:     </dimension>
53: </Mointor>
54:
55: <DSLR rdf:ID="Nikon_D70"
55a: xmlns="http://www.liyangyu.com/camera#">
56:     <body>
57:         <Body>
58:             <shutter>
59:                 <ValueRange>
60:                     <minValue rdf:datatype=
60a: "http://www.w3.org/2001/XMLSchema#float">
60b:                         0.00002</minValue>
61:                 </ValueRange>
62:             </shutter>

```

```

63:         </Body>
64:     </body>
65: <lens>
66:     <Lens>
67:         <zoomRange rdf:datatype=
67a:             "http://www.w3.org/2001/XMLSchema#string">
67b:             18-200mm</zoomRange>
68:     </Lens>
69: </lens>
70: </DSLR>
71:
72: </catalog>
73:
74: </Retailer>
75:
76: </rdf:RDF>

```

Obviously, this retailer is selling more than just cameras; it is also selling PCs, processors, and monitors. Our ShopBot is able to discover that terms used in this catalog are from the following ontology namespaces:

```

http://www.w3.org/1999/02/22-rdf-syntax-ns#
http://www.liyangyu.com/camera#
http://www.eBusiness.org/retailer#
http://www.ontologyExample.org/ontology/monitorspec#
http://www.ontologyExample.org/ontology/pcspec#

```

Clearly, our camera ontology namespace

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

has been used by both the request file (List 15.6) and the catalog file (List 15.8). The ShopBot will then treat this catalog file as one candidate and will further investigate whether there is a matched camera contained in this catalog file. We will discuss the matching process in the next section.

15.2.2.3 How Does It Decide Whether There Is a Match or Not?

First off, you might have noticed that in List 15.8, the following property is used (line 67) to describe the Nikon D70 camera:

```
http://www.liyangyu.com/camera#zoomRange
```

and it is not a term from our camera ontology. In fact, in this chapter, in order to make our ShopBot more interesting, we will change our camera ontology a little bit. More specifically, our camera ontology is given in List 5.30, and we will make some changes to several property definitions, as shown in List 15.9.

List 15.9 Property definition changes in our camera ontology (see List 5.30)

```

1: <owl:ObjectProperty
1a:   rdf:about="http://www.liyangyu.com/camera#effectivePixel">
2:   <owl:equivalentProperty rdf:resource="#resolution"/>
3:   <rdfs:domain rdf:resource="#Digital"/>
4:   <rdfs:range
4a:     rdf:resource="http://www.liyangyu.com/camera#MegaPixel"/>
5: </owl:ObjectProperty>
6: <rdfs:Datatype
6a:   rdf:about="http://www.liyangyu.com/camera#MegaPixel">
7:   <rdfs:subClassOf
7a:     rdf:resource="http://www.w3.org/2001/XMLSchema#decimal"/>
8: </rdfs:Datatype>
9:
10: <owl:ObjectProperty
10a:   rdf:about="http://www.liyangyu.com/camera#shutterSpeed">
11:   <owl:equivalentProperty rdf:resource="#shutter"/>
12:   <rdfs:domain rdf:resource="#Body"/>
13:   <rdfs:range rdf:resource="#ValueRange"/>
14: </owl:ObjectProperty>
15:
16: <owl:DatatypeProperty
16a:   rdf:about="http://www.liyangyu.com/camera#focalLength">
17:   <owl:equivalentProperty rdf:resource="#zoomRange"/>
18:   <rdfs:domain rdf:resource="#Lens"/>
19:   <rdfs:range
19a:     rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
20: </owl:DatatypeProperty>
21: <rdfs:Datatype
21a:   rdf:about="http://www.w3.org/2001/XMLSchema#string"/>

```

As shown in List 15.9, the only change we have made to the current camera ontology (List 5.30) is in lines 2, 11, and 17. And now, `myCamera:effectivePixel` is equivalent to `myCamera:resolution`, `myCamera:shutterSpeed` is equivalent to `myCamera:shutter`, and finally, `myCamera:focalLength` is equivalent to `myCamera:zoomRange`. With these changes, line 67 in List 15.8 simply uses a different term which has the same meaning as `myCamera:focalLength` does.

Now, to further decide whether there is a real match or not, our ShopBot first has to decide which item described in this given catalog could potentially satisfy our needs. In our example, the ShopBot will figure out that Nikon D70 described in the catalog has a type of DSLR, which is a sub-class of `Digital`. As a result, it can be considered as a candidate product. Also note that that this item will be the only product from List 15.8 that will be further considered by the ShopBot.

With this potential candidate, the ShopBot will have to dive into more details:

- We are looking for a digital camera whose lens has a specific value for its `focalLength` property. In the catalog, Nikon D70's lens has a property called `zoomRange`. With the inferencing power provided by the camera ontology, our

ShopBot will be able to understand that `focalLength` and `zoomRange` are equivalent properties; therefore, the description of what we are looking for does match the description of the item on sale.

- The same process has to be repeated for other terms. For example, our ShopBot will also understand that `shutter` and `shutterSpeed` are also equivalent. If all the terms match, the given product can be considered as a match to our need.

With all these said, let us move on to the construction of our ShopBot, and you will have another chance to see the inferencing power at work.

15.2.3 Building Our ShopBot

Given the previous discussions, the building of our ShopBot becomes fairly easy. Let us start with some basic utilities first.

15.2.3.1 Utility Methods and Class

The first utility method we would like to mention is the method shown in List 15.10, which is used to understand our search request.

List 15.10 Method to understand our search need

```

1: private boolean getItemToSearch(Model m) {
2:
3:     String queryString =
4:         "SELECT ?subject ?predicate ?object " +
5:         "WHERE {" +
6:             " ?subject <" + RDF.type + "> ?object. " +
7:             " }";
8:
9:     Query q = QueryFactory.create(queryString);
10:    QueryExecution qe = QueryExecutionFactory.create(q,m);
11:    ResultSet rs = qe.execSelect();
12:
13:    // collect the data type property names
14:    while ( rs.hasNext() ) {
15:        ResultBinding binding = (ResultBinding)rs.next();
16:        RDFNode rn = (RDFNode)binding.get("subject");
17:        if ( rn != null ) {
18:            targetItem = rn.toString();
19:        }
20:        rn = (RDFNode)binding.get("object");
21:        if ( rn != null ) {
22:            targetType = rn.toString();
23:        }

```

```

24:  }
25:  qe.close();
26:
27:  if ( targetItem == null || targetItem.length() == 0 ){
28:    return false;
29:  }
30:  if ( targetType == null || targetType.length() == 0 ){
31:    return false;
32:  }
33:  return true;
34: }

```

More specifically, we express our need in an RDF document (as shown in Lists 15.6 and 15.7), and we create an RDF model based on this document. This model is then passed to this method so that we can have the following two pieces of information:

- the URI that represents the product we are looking for and
- the type information of the product.

To accomplish this, we run the following SPARQL query against the model that represents our need:

```

SELECT ?subject ?predicate ?object
WHERE {
    ?subject <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
    ?object.
}

```

and this query is coded in lines 3–7. Once the query is run against the model, we can scan the result for the URI and type information as shown in lines 14–24. Note that it is important to understand the following two assumptions here:

- One request document describes only one item.
- We will always give a URI to represent this item; in other words, it will not be represented by a blank node.

With these assumptions, the query in lines 3–7 and the scan in lines 14–24 will be able to find the item and its type. Note that URI that represents the item is stored in variable called `targetItem`, and its type is stored in variable `targetType`.

Once the above is done, both `targetItem` and `targetType` should have their value. If either one of them is not populated, a `false` value is returned (lines 27–32), indicating the RDF document that represents the request is not properly created.

If you run the method against the request document shown in List 15.6, these two variables will hold the following values when this method finishes:

```

targetItem:
    http://www.liyangyu.com/shopbot/request#myDigitalCamera

```

```
targetType:
    http://www.liyangyu.com/camera#Digital
```

The second utility method is used to solve the problem discussed in Sect. 15.2.2.2. This method itself uses another method to accomplish the goal, and these two methods are shown in List 15.11.

List 15.11 Methods used to decide whether a given catalog document should be further investigated

```
1: private boolean isCandidate(Model m) {
2:
3:     if ( m == null ) {
4:         return false;
5:     }
6:
7:     HashSet ns = new HashSet();
8:     this.collectNamespaces(m,ns);
9:     return ns.contains(ontologyNS);
10:
11: }
12:
13: private void collectNamespaces(Model m,HashSet hs) {
14:     if ( hs == null || m == null ) {
15:         return;
16:     }
17:     NsIterator nsi = m.listNameSpaces();
18:     while ( nsi.hasNext() ) {
19:         hs.add(nsi.next().toString());
20:     }
21: }
```

These two methods should be fairly straightforward to follow. Method `collectionNamespaces()` uses Jena API `listNameSpaces()` method to collect all the namespaces used on a given RDF model (line 17), and these namespaces are saved in a Hash set as shown in lines 18–20.

Method `isCandidate()` is called when our ShopBot encounters a new RDF document. Based on this RDF document, a model is created and passed on to method `isCandidate()`. Inside the method, a collection of namespaces used by the given model is created by calling `collectNamespaces()` method (line 8), and if the namespace of our camera ontology is one of the namespaces used (line 9), method `isCandidate()` will return `true`, indicating the given RDF document should be further investigated. Note that variable `ontologyNS` holds the namespace of our camera ontology.

Another important utility is `findCandidateItem()` method. Once the ShopBot has decided that a given RDF document possibly contains the product we are

interested in, it then needs to decide exactly what are these products, and this is done by `findCandidateItem()` method. List 15.12 shows the definition of the method.

List 15.12 Method used to find all the candidate products in a given RDF catalog

```

1: private Vector findCandidateItem(Model m) {
2:
3:     Vector candidates = new Vector();
4:     String queryString =
5:         "SELECT ?candidate " +
6:         "WHERE {" +
7:         "   ?candidate <" + RDF.type + "> <" + targetType + ">. " +
8:         "   }";
9:
10:    Query q = QueryFactory.create(queryString);
11:    QueryExecution qe = QueryExecutionFactory.create(q,m);
12:    ResultSet rs = qe.execSelect();
13:
14:    while ( rs.hasNext() ) {
15:        ResultBinding binding = (ResultBinding)rs.next();
16:        RDFNode rn = (RDFNode)binding.get("candidate");
17:        if ( rn != null && rn.isAnon() == false ) {
18:            candidates.add(rn.toString());
19:        }
20:    }
21:    qe.close();
22:    return candidates;
23: }
```

As you can tell, the idea is very simple. Since we know we are looking for something that has <http://www.liyangyu.com/camera#Digital> as its type, all we need to do is to run the following query against the RDF model that represents the given catalog document:

```

SELECT ?candidate
WHERE {
    ?candidate <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
    <http://www.liyangyu.com/camera#Digital>.
}
```

Note that this query is implemented by lines 4–8, and variable `targetType` has <http://www.liyangyu.com/camera#Digital> as its value.

The rest of the method is quite straightforward: a SPARQL query is created and submitted to run against the catalog (line 10–11) and the results are analyzed in lines 14–20, where the URIs of the candidate products are collected.

Now, the interesting part comes from the fact that if you simply take the RDF catalog shown in List 15.8 and create a simple RDF model from it and use method `findCandidateItem()` on this model, you will not find any candidate product at all.

This is not surprising. After all, the Nikon D70 camera described in the catalog has a type of `DSLR`, and it is not the exact type as we have specified in the query. However, as we have discussed earlier, `DSLR` is a sub-class of `Digital`; therefore, Nikon D70 is also an instance of `Digital` camera and it should be returned in the query result.

Now, in order for our ShopBot to see this, we cannot use a simple RDF model to represent the catalog file. Instead, we need to create an ontology model that has the inferencing power. More specifically, List 15.13 shows the steps needed.

List 15.13 To find the candidate products from a given catalog document, we need to create an inferencing model

```
1: Model catalogModel = getModel(catalog);
2:
3: // create ontology model for inferencing
4: OntModel ontModel = ModelFactory.createOntologyModel
4a:         (OntModelSpec.OWL_MEM_RULE_INF, catalogModel);
5: FileManager.get().readModel(ontModel, ontologyURL);
6:
7: candidateItems = findCandidateItem(ontModel);
```

First off, `catalog` represents the path of the given RDF catalog file as shown in List 15.8 and `ontologyURL` is the path of our camera ontology. Line 1 calls another utility method, `getModel()`, to create a simple RDF model based on the catalog document, and this model is stored in variable `catalogModel`. Lines 4 and 5 use `catalogModel` and `ontologyURL` to create an ontology model, which has the derived statements added by the OWL inferencing engine provided by Jena.

One way to make this more understandable is to print out all the statements about the type of Nikon D70 camera contained in the two models at both lines 2 and 6 in List 15.13. More specifically, line 1 calls `getModel()` to create a simple RDF model based on the catalog document. If we were to collect all the type statements about Nikon D70 at line 2, we would collect only one such statement.

```
<http://www.retailerExample1.com/onlineCatalog#Nikon_D70>
<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
<http://www.liyangyu.com/camera#DSLR>.
```

And this is exactly the reason why the query (lines 4–8 in List 15.12) fails to identify Nikon D70 as a potential product for us. Now, if we were to collect the same type statements in line 6 against the ontology model, we would see the following statements:

```
<http://www.retailerExample1.com/onlineCatalog#Nikon_D70>
<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
<http://www.liyangyu.com/camera#DSLR>.
```

```
<http://www.retailerExample1.com/onlineCatalog#Nikon_D70>
```



```

<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
<http://www.w3.org/2002/07/owl#Thing>.

<http://www.retailerExample1.com/onlineCatalog#Nikon_D70>
<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
<http://www.liyangyu.com/camera#Camera>.

<http://www.retailerExample1.com/onlineCatalog#Nikon_D70>
<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
<http://www.liyangyu.com/camera#Digital>.

<http://www.retailerExample1.com/onlineCatalog#Nikon_D70>
<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
<http://www.w3.org/2000/01/rdf-schema#Resource>.

```

As you can tell, all these statements except the first one are added by Jena's inferencing engine, and they are all contained in the ontology model. Clearly, this is why the query in lines 4–8 (List 15.12) can now successfully identify all the potential products from the given catalog file.

We will see more reasoning power along the same line in the next section. Before we move on, we need to discuss a utility class named `CameraDescription`, and you will see more about its usage in the later sections.

This class has the following private variables for describing a camera:

```

private float pixel;
private float minFocalLength;
private float maxFocalLength;
private float minAperture;
private float maxAperture;
private float minShutterSpeed;
private float maxShutterSpeed;

```

And obviously, each one of these variables represents a property that is defined in our camera ontology. In fact, this class is quite easy to understand once you see its connection to the camera ontology.

A key method we would like to briefly mention is `sameAs()` method, which tries to decide whether a given camera can satisfy our needs or not. This method takes an instance of `CameraDescription` class and returns `true` if the calling instance is the same as the parameter instance, and returns `false` otherwise. You can check out the method to see how “same as” is defined in the method, but here is one example. The cameras

```

pixel value is: 6.0^^http://www.liyangyu.com/camera#MegaPixel
focalLength value is:
  18-200mm^^http://www.w3.org/2001/XMLSchema#string
min aperture value is:
  1.8^^http://www.w3.org/2001/XMLSchema#float
max aperture value is: 22^^http://www.w3.org/2001/XMLSchema#float

```

```
min shutterSpeed value is:
    0.0005^^http://www.w3.org/2001/XMLSchema#float
max shutterSpeed value is not specified.
```

and

```
focalLength value is:
    18-200mm^^http://www.w3.org/2001/XMLSchema#string
min aperture value is not specified.
max aperture value is not specified.
min shutterSpeed value is:
    0.00002^^http://www.w3.org/2001/XMLSchema#float
max shutterSpeed value is not specified.
```

are considered to be the same. Note that only those available properties are taken into account. In other words, if one camera has a pixel value defined and the other one does not, then at least these two cameras are not different on this aspect. In addition, for those available properties, such as shutter speed as an example, if the parameter camera instance is not worse than the calling instance, the method will return a `true` value. Therefore, a `true` value indicates the camera instance passed to the method can be selected as a candidate, since it satisfies our requirements as far as all the explicitly defined properties are concerned.

15.2.3.2 Processing the Catalog Document

Once our ShopBot has decided that the given RDF catalog document contains some products that can potentially be what we are looking for, it will start the process of handling the catalog document. In this section, we will discuss the main idea of processing the catalog document and we will also see the key method to implement the idea.

The main idea is again closely related to the camera ontology we are using. For a given candidate camera contained in the catalog document, since there is only a fixed number of properties defined in the ontology that can be used to describe a camera, we can query the value of each one of these properties for this candidate camera and store the value of that particular property into a `CameraDescription` instance. Once we are done with all the properties, this `CameraDescription` instance can be used in a comparison to decide whether the given camera is a match or not.

At this point, based on our camera ontology, the following methods defined by `CameraDescription` class can be used to query the property value of a candidate camera:

```
private String getPixel(Model m, String itemURI)
private String getFocalLength(Model m, String itemURI)
private String getAperture(Model m, String itemURI,
                           int minMaxFlag)
private String getShutterSpeed(Model m, String itemURI,
                               int minMaxFlag)
```

Let us take a look at one example. List 15.14 shows the definition of `getPixel()` method.

List 15.14 Definition of `getPixel()` method

```

1: private String getPixel(Model m, String itemURI) {
2:
3:     String queryString =
4:         "SELECT ?value " +
5:         "WHERE {" +
6:         "    <" + itemURI +
6a:         "> <http://www.liyangyu.com/camera#effectivePixel>
6b:         ?value. " +
7:         "    }";
8:
9:     Query q = QueryFactory.create(queryString);
10:    QueryExecution qe = QueryExecutionFactory.create(q,m);
11:    ResultSet rs = qe.execSelect();
12:
13:    while ( rs.hasNext() ) {
14:        ResultBinding binding = (ResultBinding)rs.next();
15:        RDFNode rn = (RDFNode)binding.get("value");
16:        if ( rn != null && rn.isAnon() == false ) {
17:            return rn.toString();
18:        }
19:    }
20:    qe.close();
21:    return null;
22:
23: }
```

First off, note that at this point, our ShopBot has decided that the catalog file (List 15.8) does contain a camera that could potentially satisfy our need, and this camera has the following URI:

```
http://www.retailerExample1.com/onlineCatalog#Nikon_D70
```

which is passed in as the `itemURI` parameter to the method (line 1).

Now, the following SPARQL query is used to get the pixel value of the given camera:

```

SELECT ?value
WHERE {
    <http://www.retailerExample1.com/onlineCatalog#Nikon_D70>
    <http://www.liyangyu.com/camera#effectivePixel> ?value.
}
```

and this query is coded in lines 3–7 and is submitted to the model in lines 9–11.

As you can tell, in the query, `myCamera:effectivePixel` is specified as the property name. What if some retailer has used other name for the same query? `myCamera:resolution`, for example, can be used (see List 15.9).

Again, to handle this issue, we need the inferencing power from the ontology model. More specifically, the catalog document is represented as a ontology model (see List 15.13), and it is passed in as the `Model m` parameter to the method (line 1). The query shown in lines 3–7 will be run against this ontology model that has all the inferred statements; the final result is that our ShopBot seems to be smart enough to realize that `myCamera:effectivePixel` and `myCamera:resolution` are equivalent properties, and the correct property value will be retrieved successfully.

Therefore, using an ontology model which has all the derived statements is the key to make sure a given query can return all the facts. This is also true for all the other methods when it comes to handling the catalog document. To see one more example, List 15.15 shows the method which queries the shutter speed of a given camera.

List 15.15 Definition of `getShutterSpeed()` method

```

1: private String getShutterSpeed(Model m, String itemURI,
1a:                               int minMaxFlag) {
2:
3:     String queryString = null;
4:     if ( minMaxFlag == MyShopBot.MIN) {
5:         queryString =
6:             "SELECT ?value " +
7:             "WHERE {" +
8:             "    <" + itemURI +
8a:             "> <http://www.liyangyu.com/camera#body> ?tmpValue0. " +
9:             "    ?tmpValue0
9a:             <http://www.liyangyu.com/camera#shutterSpeed>
9b:             ?tmpValue1." +
10:            "    ?tmpValue1 <http://www.liyangyu.com/camera#minValue>
10a:            ?value . " +
11:            "    }";
12:     } else {
13:         queryString =
14:             "SELECT ?value " +
15:             "WHERE {" +
16:             "    <" + itemURI +
16a:            "> <http://www.liyangyu.com/camera#body> ?tmpValue0. " +
17:             "    ?tmpValue0
17a:            <http://www.liyangyu.com/camera#shutterSpeed>
17b:            ?tmpValue1." +
18:            "    ?tmpValue1
18a:            <http://www.liyangyu.com/camera#maxValue> ?value . " +
19:            "    }";
20:     }
21:
22:     Query q = QueryFactory.create(queryString);

```

```

23:   QueryExecution qe = QueryExecutionFactory.create(q,m);
24:   ResultSet rs = qe.execSelect();
25:
26:   String resultStr = "";
27:   while ( rs.hasNext() ) {
28:       ResultBinding binding = (ResultBinding)rs.next();
29:       RDFNode rn = (RDFNode)binding.get("value");
30:       if ( rn != null && rn.isAnon() == false ) {
31:           return rn.toString();
32:       }
33:   }
34:   qe.close();
35:   return null;
36:
37: }

```

Obviously, querying the value of `myCamera:shutterSpeed` property is more complex than querying the value of `myCamera:effectivePixel` property. More specifically, we need the following SPARQL query to accomplish this:

```

SELECT ?value
WHERE {
  <http://www.retailerExample1.com/onlineCatalog#Nikon_D70>
  <http://www.liyangyu.com/camera#body> ?tmpValue0 .
  ?tmpValue0
  <http://www.liyangyu.com/camera#shutterSpeed> ?tmpValue1 .
  ?tmpValue1 <http://www.liyangyu.com/camera#minValue> ?value .
}

```

which in fact needs a reference chain to reach the required shutter speed value. This chain starts from `myCamera:body` property, which uses a resource as its value. This resource has a `myCamera:Body` type and is bonded to `tmpValue0` variable. Next, this `myCamera:Body` resource has a `myCamera:shutterSpeed` property, which uses another resource as its value, and this new resource is of type `myCamera:ValueRange` and is bonded to another variable called `tmpValue1`. Finally, resource `tmpValue1` has two properties, and for this query, we use `myCamera:minValue` property, which tells us the minimum shutter speed this camera can offer. Note that this query is coded from lines 5–11 in List 15.15.

Now, what about the maximum shutter speed this camera body can offer? A similar query is used (lines 13–19), with the only difference being that `myCamera:maxValue` property has replaced `myCamera:minValue` property. To let the method understand which value we need, we have to pass in a third parameter, `minMaxFlag`, as shown in line 1 of this method.

Similar to the query about `myCamera:effectivePixel` property, `myCamera:shutterSpeed` property also has an equivalent property. In order to recognize all the possible terms, we again need to represent our catalog document as an ontology model which has all the inferred statements, and pass it to the method using the `Model m` parameter as shown in line 1.

For our catalog RDF document (List 15.8), `myCamera:shutter` property is used instead of `myCamera:shutterSpeed` (lines 58–62). Since we have passed in the ontology model that represents this catalog, the above query is able to find the shutter speed value. This is another example that shows the reasoning power you have gained from an ontology model.

This also shows how much dependency we have on a specific ontology, since the above query is constructed based on the ontology structure. This gives a key reason why ontology reuse is important: applications are often developed to understand some specific ontology; by reusing this ontology, we can quickly develop a new application since a large portion of the existing applications can also be reused.

The other methods, namely `getFocalLength()` and `getAperture()`, are all quite similar to List 15.15. I will leave them for you to understand. At this point, we are ready to discuss the overall work flow of our ShopBot, and we will cover that in next section.

15.2.3.3 The Main Work Flow

With the above discussion, the overall work flow of our ShopBot is quite easy to understand. This work flow is defined in the following `work()` method, as shown in List 15.16.

List 15.16 Main work flow of our ShopBot agent

```

1:  private void work() {
2:
3:      // for the product we are looking for,
3a:     //get its URI and type info
4:     Model requestModel = getModel(requestRdf);
5:     if ( getItemToSearch(requestModel) == false ) {
6:         System.out.println("your request description
6a:             is not complete!");
7:     }
8:     System.out.println("this URI describes the resource
8a:         you are looking for:");
9:     System.out.println("<" + targetItem + ">");
10:    System.out.println("its type is given by
10a:        the following class:");
11:    System.out.println("<" + targetType + ">");
12:
13:    // find all the requested parameters
14:
15:    CameraDescription myCamera = new CameraDescription();
16:
17:    String targetPixel = getPixel(requestModel, targetItem);
18:    myCamera.setPixel(targetPixel);
19:    show("pixel(target)", targetPixel);
20:
21:    String targetFocalLength =
21a:        getFocalLength(requestModel, targetItem);

```

```

22: myCamera.setFocalLength(targetFocalLength);
23: show("focalLength(target)", targetFocalLength);
24:
25: String targetAperture =
25a:     getAperture(requestModel, targetItem, MyShopBot.MIN);
26: myCamera.setMinAperture(targetAperture);
27: show("min aperture(target)", targetAperture);
28:
29: targetAperture =
29a:     getAperture(requestModel, targetItem, MyShopBot.MAX);
30: myCamera.setMaxAperture(targetAperture);
31: show("max aperture(target)", targetAperture);
32:
33: String targetShutterSpeed =
33a:     getShutterSpeed(requestModel, targetItem, MyShopBot.MIN);
34: myCamera.setMinShutterSpeed(targetShutterSpeed);
35: show("min shutterSpeed(target)", targetShutterSpeed);
36:
37: targetShutterSpeed =
37a:     getShutterSpeed(requestModel, targetItem, MyShopBot.MAX);
38: myCamera.setMaxShutterSpeed(targetShutterSpeed);
39: show("max shutterSpeed(target)", targetShutterSpeed);
40:
41: CameraDescription currentCamera = new CameraDescription();
42: while ( true ) {
43:
44:     Model catalogModel = getModel(catalog);
45:
46:     // see if it has potential candidates
47:     if ( isCandidate(catalogModel) == false ) {
48:         continue;
49:     }
50:
51:     // create ontology model for inferencing
52:     OntModel ontModel = ModelFactory.createOntologyModel
52a:         (OntModelSpec.OWL_MEM_RULE_INF, catalogModel);
53:     FileManager.get().readModel(ontModel, ontologyURL);
54:
55:     // which item could be it?
56:     candidateItems = findCandidateItem(ontModel);
57:     if ( candidateItems.size() == 0 ) {
58:         continue;
59:     }
60:
61:     for ( int i = 0; i < candidateItems.size(); i ++ ) {
62:
63:         String candidateItem =
63a:             (String) candidateItems.elementAt(i);
64:         System.out.println("\nFound a candidate: " +
64a:             candidateItem);
65:         currentCamera.clearAll();
66:

```

```

67:         // find the pixel value
68:         String pixel = getPixel(ontModel, candidateItem);
69:         currentCamera.setPixel(pixel);
70:
71:         // find lens:focalLength value
72:         String focalLength =
72a:             getFocalLength(ontModel, candidateItem);
73:         currentCamera.setFocalLength(focalLength);
74:         show("focalLength", focalLength);
75:
76:         // find lens:aperture value
77:         String aperture =
77a:             getAperture(ontModel, candidateItem, MyShopBot.MIN);
78:         currentCamera.setMinAperture(aperture);
79:         show("min aperture", aperture);
80:         aperture =
80a:             getAperture(ontModel, candidateItem, MyShopBot.MAX);
81:         currentCamera.setMaxAperture(aperture);
82:         show("max aperture", aperture);
83:
84:         // find body:shutterSpeed value
85:         String shutterSpeed =
85a:             getShutterSpeed(ontModel, candidateItem, MyShopBot.MIN);
86:         currentCamera.setMinShutterSpeed(shutterSpeed);
87:         show("min shutterSpeed", shutterSpeed);
88:         shutterSpeed =
88a:             getShutterSpeed(ontModel, candidateItem, MyShopBot.MAX);
89:         currentCamera.setMaxShutterSpeed(shutterSpeed);
90:         show("max shutterSpeed", shutterSpeed);
91:
92:         if ( myCamera.sameAs(currentCamera) == true ) {
93:             System.out.println("found one match!");
93a:             // more interesting action?
94:         }
95:     }
96:
97:     break; // or next catalog file.
98: }
99:
100: }

```

The work flow starts from line 4, where an RDF model (`requestModel`) representing our search request is created. It is then passed into method `getItemToSearch()` so that our ShopBot can understand more about the requirement (line 5).

As we have discussed earlier, we describe our need by creating a resource. Method `getItemToSearch()` looks for the URI that represents this resource (saved in `targetItem` variable); it also collects the type information of this resource (saved in `targetType` variable). If it cannot find these two pieces of information, a false value is returned with an error message shown to the user (lines

5–7). In case of success, `targetItem` and `targetType` are echoed back to the user, as shown in lines 8–11.

Once the above is successfully done, our `ShopBot` creates a new instance of class `CameraDescription` named `myCamera`, which represents our detailed requirements about the camera we want to find (line 15).

To populate `myCamera`, `getPixel()` method is first called (line 17), with `requestModel` and `targetItem` as its parameters. This method returns the value of `myCamera:effectivePixel` property, which is then stored in `myCamera` instance, as shown in line 18.

Similarly, the `ShopBot` calls method `getFocalLength()` and stores the returned value into `myCamera` (lines 21–23). Then it calls `getAperture()` method to query both the minimum and maximum aperture values, and these values are again used to populate `myCamera` instance (lines 25–31). Same is true for shutter speed, as shown from lines 33–39. Once all these are done, our `ShopBot` has finished the task of understanding our needs: all our requirements about our camera is now stored in `myCamera` instance.

At this point, our `ShopBot` is ready to handle catalog files. To do so, the first thing it does is to create another `CameraDescription` instance called `currentCamera`, which is used as a place to hold the descriptions of a potential camera found in a given catalog document (line 41).

Obviously, there are multiple catalog documents our `ShopBot` needs to visit, and it will visit them one by one, using a loop of some kind. So far, since we have established only one example catalog document (List 15.8), and given the fact that the exact same processing flow will work for any catalog file, I will not create more catalog file examples. Therefore, instead of actually having a list of documents to visit, I will simply use a `while()` loop (lines 42–98) to represent the idea of having a collection of documents to visit. It is up to you to come up with more example catalog files and change the `while()` loop so that none of these document will be missed.

Now, for a given catalog file on hand, our `ShopBot` first creates a simple RDF model to represent it (line 44) and then calls `isCandidate()` method to see whether there is any need to study this catalog more (line 47). If the current catalog file does not have any product that can potentially satisfy our needs, our `ShopBot` moves on to the next catalog document (line 48).

When the `ShopBot` decides to investigate more on the current catalog file, it first creates an ontology model based on the catalog document as shown in lines 52–53. As we have discussed earlier, this is necessary since the inferencing engine can add derived statements into the model, and it is the reason why all the queries can work correctly.

The `ShopBot` then tries to find from the catalog document all the products that could be the potential matches by calling `findCandidateItem()` method (line 56). We have discussed this method already, and it returns a collection of URIs (a `Vector`), with each one of these URIs representing such a potential product. In our example, this collection has only one product that has the following URI:

```
http://www.retailerExample1.com/onlineCatalog#Nikon_D70
```

If the returned collection is empty, meaning the current catalog document does not have any possible matches, our ShopBot skips this catalog and moves onto the next one, as shown on lines 57–59.

It is certainly possible that a given catalog document has more than one cameras that can potentially satisfy our needs. To examine these products one by one, we need to have a loop as shown in lines 61–95.

To start the examination, our ShopBot takes one URI from this collection (line 63). Since we have only one `currentCamera` instance that holds the description of the current product, it may still contain the description of the previous product, so our ShopBot clears it first, as shown in line 65.

Next, our ShopBot starts to query against the current catalog document to retrieve the values of the applicable properties, namely pixel value, focal length value, aperture value, and shutter speed value. And once a value is retrieved, it is saved into `currentCamera` instance. This process is shown from lines 67–90, and it is the same process that our ShopBot has used to understand the request document (lines 17–39).

Once this process is done, `currentCamera` instance holds the description of the current product. It is then used to compare with `myCamera` instance, which holds our request. If this comparison returns a `true` value, our ShopBot declares the fact that a match has been found (lines 92–94). In our example, to declare a match simply means a system printout is used to alert the user (line 93). You can change this line to whatever you would like to do. For example, collecting the retailer’s name and the Web site that describes this product in more details, including its pricing and shipping information.

Once all the potential products from the current catalog are examined, our ShopBot should continue onto the next catalog file, which should be done in line 97. Again, for this example, currently only one catalog file is used, so line 97 simply breaks the loop. For you, if you want our ShopBot to visit more catalog documents, this is the place where you should change.

At this point, we have finished building our ShopBot. Note that the classes and methods that have been discussed here are the major ones; you can find the complete code from the package you have downloaded. With what we have learned here, you should not have any problem in understanding the complete code.

15.2.3.4 Running Our ShopBot

The best way to understand our ShopBot agent is to download the complete code, compile it using your own java project and favorite IDE, and of course, run it.

Also, to make the ShopBot run, I have used the following files:

- request file: `C:/liyang/myWritings/data/myCameraDescription.rdf`
- catalog file: `C:/liyang/myWritings/data/catalogExample1.rdf`
- ontology file: `C:/liyang/myWritings/data/camera.owl`

You should be able to find these files when you download the code for our ShopBot, and you should change the paths of these files so that your ShopBot agent can have access to these files.

List 15.17 shows the driver we use to start the ShopBot.

List 15.17 Test driver for our ShopBot agent

```
1: public static void main(String[] args) {
2:
3:     MyShopBot myShopBot = new MyShopBot();
4:     myShopBot.work();
5:
6: }
```

List 15.18 shows the output you should see when running the ShopBot.

List 15.18 Screen output generated by our ShopBot agent

```
this URI describes the resource you are looking for:
<http://www.liyangyu.com/shopbot/request#myDigitalCamera>
its type is given by the following class:
<http://www.liyangyu.com/camera#Digital>
```

```
pixel(target) value is:
6.0^^http://www.liyangyu.com/camera#MegaPixel
focalLength(target) value is:
18-200mm^^http://www.w3.org/2001/XMLSchema#string
min aperture(target) value is:
1.8^^http://www.w3.org/2001/XMLSchema#float
max aperture(target) value is:
22^^http://www.w3.org/2001/XMLSchema#float
min shutterSpeed(target) value is:
0.0005^^http://www.w3.org/2001/XMLSchema#float
max shutterSpeed(target) value is not specified.
```

```
Found a candidate:
http://www.retailerExample1.com/onlineCatalog#Nikon_D70
focalLength value is:
18-200mm^^http://www.w3.org/2001/XMLSchema#string
min aperture value is not specified.
max aperture value is not specified.
min shutterSpeed value is:
0.00002^^http://www.w3.org/2001/XMLSchema#float
max shutterSpeed value is not specified.
```

```
found one match!
```

After understanding our ShopBot and seeing its example output, you can compile more catalog files for it to understand. Try it out and have fun!

15.2.4 Discussion: From Prototype to Reality

The ShopBot presented in this chapter is a simple prototype; its goal is to show you another application example on the Semantic Web. You should be able to see the following main points from this example:

- ShopBots created in the Semantic Web environment can do much more than traditional ShopBots. For example, you don't have to only search a specific product; instead, you can describe what you want and let the machine find it for you.
- There is no need for screen scraping; the Semantic Web makes it possible to create and run ShopBots in a much more scalable and maintainable way.

There are obviously lots of things you can do to make it better. They are the following:

- A ShopBot should be able to visit the Web to find catalog files from different retailers. In our example, we simply feed it with catalog files.
- For now, the catalog files we are looking for are RDF files. Since the “scarcity” of these files, what should be the best way to find them quickly?
- Instead of editing separate RDF documents, some retailers may decide to use RDFa or microformats to add semantic markups directly into their online HTML documents. How should we change our ShopBots to handle this situation?
- If you run our current ShopBot, you will note the fact that it runs slower when we have to create ontology models instead of simple RDF models. This might hurt the scalability and efficiency of our ShopBot. How to make improvements along this direction?

All these questions are important when constructing a real ShopBot. However, from prototype to real application, there are actually more important issues that remain to be solved. For example,

- there has to be a set of ontologies that the whole community accepts.

For a given application domain, a set of ontologies are needed. However, who will be responsible for creating these ontologies? How to reach the final agreement on these ontology files? Sometimes, there might be already different ontologies existing for a given domain. Instead of reinventing the wheel, it might be a good idea to merge these ontologies together to form a standard one. Yet how should this merge be done?

- each retailer has to semantically markup his/her catalog, and the markup has to be done based on the standard ontology files.

For the general public on the Web, semantic markup is never an easy process. To mark up a Web document, either a separate RDF document can be created or RDFa/microformats can be used to embed the markups into the Web document. Whichever method is chosen, there is a learning curve for the users to conquer, and they have to agree to use the terms from the standard ontologies. To overcome the technical challenges and learning curve, user motivation plays an important role as well.

Besides the above, there are other issues such as security and trust, and whether sharing sensitive data on the public Web is acceptable to each retailer. Also, from a technical perspective, building intelligent agents still have the issue of scalability and efficiency, especially when reasoning on large ontologies is involved.

Clearly, there is still a long way to cover before our prototype can finally become a practical application system on a large scale. However, it is our hope that understanding these issues will make our road ahead clear and also give you the motivation to continue the exploration on the Semantic Web, and finally, to turn the idea into a beautiful reality.

15.3 Summary

In this chapter, we have created two more applications on the Semantic Web. The first one is an agent that works with FOAF documents to create a trustable e-mail list for you and the second one is a ShopBot that understands your needs and tries to find those products that can satisfy your needs. With these examples, you should be able to understand the following:

- the basic steps and patterns when it comes to developing applications on the Semantic Web;
- soft agents on the Semantic Web can be developed with the scalability and efficiency that cannot be matched if the same agents were to be developed on the traditional document Web;
- in addition, soft agents are extremely suitable for dynamic and distributed data environment, and it is almost effortless to maintain these agents.

Meanwhile, understand that to build production level soft agents in real world, quite a few issues have to be addressed first. Noticeably, the development of shared ontologies, the motivation for the general public to markup their Web documents and the related security/trust issues are the top items on the list.

Finally, congratulations on finishing the whole book. I hope you have learned something from this book, and more importantly, I hope you are now motivated to pursue more on the vision of the Semantic Web, and soon, you will be able to deliver applications on the Web that can make million users happy.

Index

A

Adding semantics to the current Web, *see* Semantic markup

Add machine-understandable meanings, 14

AI, 156

Amazon Web Services, 3

Ambiguity

ambiguity of XML document, 82

semantic ambiguity, 27

Anonymous node, *see* Blank node

Anonymous resource, *see* Anonymous node

Artificial Intelligence, *see* AI

Automatic

information extraction, 388

B

Base

in-scope base URI, 57

turtle base keyword, 66–69

xml base attribute, 57

Berners-Lee, Tim, 15

Binary relationship, 39

Blank node, 36, 38

local identifier, 39

n-ary relationship, 39

bnode, *see* Blank node

Breadth-first search, 534

Brickley, Dan, 292

C

Camera ontology in OWL 1, 192

Camera ontology in OWL 2, 233

Camera ontology written in RDFS, 132

Collections in RDF, 61–63

Common language in sharing information, 110

Containers in RDF, 59–61

Content negotiation, 417

Crawler, 316

Create a FOAF document, 303

Creator, in DC(Dublin Core) metadata schema, 80

cURL, 434

Cyganiak, Richard, 431

D

DAML+OIL, 156

DAML, 156

DARPA Agent Markup Language, *see* DAML

Data integration, 1

Data mining, 11

DBpedia, 381

Berlin page, 391

core datasets, 401

datasets, 381

DBpedia URI, 395

extended datasets, 405

extractor, 388–389, 394

Federer page, 382

generic infobox extraction method, 395

links to RDF Bookmashup, 406

links to Wikicompany dataset, 405

look and feel, 385

mapping-based extraction approach, 395

as part of Linked Data, 406

persondata dataset, 404

RDF dumps, 401

RDF icon, 391

SAPRQL endpoint, 397

SPARQL viewer, 397

titles dataset, 404

URI lookup service, 408

using SPARQL to access DBpedia, 398–401

DBpedia ontology, 390

access the ontology, 390

infobox attributes map to properties, 394–396

- infobox templates map to classes, 392–394
 - ontology dataset, 402
 - ontology infobox properties dataset, 403
 - ontology types dataset, 403
- DBpedia Project, 381
- Depth-first search, 534
- Dereferencable URIs, 29
- Dereferencing URIs, 29
- Description Logic, *see* DL
- Digital single lens reflex, *see* DSLR
- DL, 226–227
- Domain, 137
- Domain model, 479
- DSLR, 21
- Dublin Core, 79
- Dublin Core vocabulary, 79
- Dynamic object model pattern, 483

- E**
- EmailAddressCollector agent, 566

- F**
- Facets, 211
- Falcons, 441
 - concept search, 442
 - document search, 443
 - object search, 442
 - type pane, 442
- Federer, Roger, 381
- FOAF, 292
 - foaf:interest, 427
 - explorer, 302
 - foaf:, 293
 - foaf:Agent, 296
 - foaf:base_near, 428
 - foaf:depiction, 300
 - foaf:depicts, 300
 - foaf:Document, 296
 - foaf:firstName, 296
 - foaf:homepage, 296
 - foaf:Image, 296
 - foaf:knows, 298
 - foaf:mbox, 296
 - foaf:mbox_sha1sum, 298
 - foaf:name, 296
 - foaf:Organization, 296
 - foaf:Person, 295
 - foaf:Project, 296
 - in official language, 292–293
 - publish your FOAF document, 305–306
 - scutter, 302
 - vocabulary, 292
- FOAF Bulletin Board, 306
- FOAF-a-matic, 303

- Follow-Your-Nose
 - build a Follow-Your-Nose agent, 536–543
 - method, 533
 - run a Follow-Your-Nose agent, 543–545
- Framework, 467
- Friend of a Friend, *see* FOAF

- G**
- Gleaning Resource Descriptions from Dialects of Languages, *see* GRDDL
- GRDDL, 105
 - link element, 106
 - with microformats, 106–107
 - profile attribute, 105
 - with RDFa, 107

- H**
- Hash symbol, 419
- Hash URI, 28, 419
- Heath, Tom, 456
- Hello World example, 497–498

- I**
- Inference engine, 524
- Information integration, *see* Data integration
- Information resources, 414
- Internationalized Resource Identifiers, *see* IRI
- International Semantic Web Conference, *see* ISWC
- IRI, 159
- ISWC, 18

- J**
- Jena, 468, 473, 492
 - add(), 507
 - addProperty(), 504
 - bindSchema(), 525
 - createDefaultModel(), 503–504, 508
 - createModelRDBMaker(), 517, 519
 - create RDF model, 502–507
 - createResource(), 504
 - createStatement(), 507
 - download Jena package, 492–495
 - FileManager class, 509
 - getId(), 511
 - getNsPrefixMap(), 511
 - getURI(), 511
 - inference model, 528
 - inferencing examples, 525–531
 - isAnon(), 511
 - listObjectsOfProperty(), 511
 - listResourcesWithProperty(), 511
 - listStatements(), 514
 - Literal class, 504
 - in-memory RDF models, 501

- ModelFactory, 503
 - ModelMaker class, 517
 - ModelRDB class, 517
 - multiple persistent RDF models, 522–524
 - OntModelSpec class, 529
 - persistent RDF model, 515
 - Property class, 504
 - RDFNode interface, 504
 - RDF/XML-ABBR parameter, 506
 - read a RDF model, 507–509
 - ReasonerRegistry class, 525
 - Resource class, 504
 - single persistent RDF model, 517–521
 - understand a RDF model, 510–515
 - using Jena in Eclipse, 495–497
- Joseki, 244
- K**
- Keyword-matching, 13
 - Knowledge Organization Systems, *see* KOS
 - Knowledge representation, *see* KR
 - KOS, 138
 - KR, 156
- L**
- Linked Data, 16, 409
 - accessing the Web of Linked Data, 445
 - application example, 456–463
 - basic principles, 412
 - creating links, 427–433
 - creating links manually, 431
 - discover, 441
 - generating links automatically, 433
 - minimal requirements, 434
 - pattern-based algorithms, 433
 - publishing linked data on the Web, 436–438
 - size of, 411–412
 - use SPARQL to access the Web of Linked Data, 451
 - validator, 438
 - Linked Data browsers, 410, 445
 - Linked Data cloud, 451
 - Linked Open Data, *see* LOD
 - Linking Open Data Community Project, 412
 - LOD, 409
 - LOD cloud, 431
- M**
- Mashup, 411, 463
 - McBride, Brian, 468
 - MediaWiki, 334
 - Microformats, 88
 - hCard microformat, 89
 - and RDF, 94–95
 - syntax and examples, 89–94
 - Model-view-controller, *see* MVC architecture
 - Musicbrainz, 451
 - SPARQL endpoint, 451
 - Music Ontology, 424
 - MVC architecture, 479
 - MySQL, 516
 - Command Line Client, 517
 - Connector/J, 516
 - JDBC driver, 516
 - port number, 516
 - setup, 516
- N**
- Negation as failure, 282
 - NeOn, 476
 - OWL editor, 476
 - Toolkit, 476
 - Nikon D300, 22
 - Non-information resources, 415
- O**
- OIL, 156
 - Ontology, 137
 - Ontology development methodology, 484–489
 - basic steps, 487
 - basic tasks and fundamental rules, 485
 - bottom-up approach, 486
 - combination approach, 486
 - top-down approach, 486
 - Ontology driven architecture, *see* Ontology-driven software development method
 - Ontology-driven software development method, 482
 - Ontology engineering environment, 474
 - Ontology header, 219
 - Ontology Inference Layer, *see* OIL
 - OWL, 159
 - cardinality constraints, 165
 - Direct Model-Theoretic Semantics, 226
 - in official language, 156–158
 - from OWL 1 to OWL 2, 158–159
 - in plain English, 155–156
 - qualified cardinality constraints, 196
 - RDF-based Semantics, 226
 - value constraints, 165
 - OWL 1, 157
 - annotation property, 215
 - imports and versioning, 219
 - OWL 1 DL, 227–229
 - OWL 1 Full, 227–228
 - OWL 1 Lite, 227, 229

- owl:AllDifferent, 225
- owl:allValuesFrom, 165
- owl:AnnotationProperty, 216
- owl:cardinality, 170
- owl:Class, 161
- owl:complementOf, 174
- owl:DatatypeProperty, 180
- owl:differentFrom, 224
- owl:disjointWith, 117
- owl:distinctMembers, 225
- owl:equivalentClass, 176, 224
- owl:FunctionalProperty, 189
- owl:hasValue, 168
- owl:imports, 220
- owl:intersectionOf, 172
- owl:InverseFunctionalProperty, 192
- owl:inverseOf, 190
- owl:maxCardinality, 171
- owl:minCardinality, 171
- owl:ObjectProperty, 180
- owl:oneOf, 175
- owl:onProperty, 165
- owl:Ontology, 220
- owl:Restriction, 165
- owl:sameAs, 222
- owl:sameIndividualAs, 223
- owl:someValuesFrom, 167
- owl:SymmetricProperty, 185
- owl:Thing, 161
- owl:TransitiveProperty, 186
- owl:unionOf, 173
- owl:versionInfo, 221
- reasoning based on cardinality constraints, 171–172
- reasoning based on class enumeration, equivalent and disjoint, 177
- reasoning based on functionality property, 189
- reasoning based on inverse functional property, 191–192
- reasoning based on inverse property, 189–190
- reasoning based on owl:allValuesFrom, 166–167
- reasoning based on owl:hasValue, 170
- reasoning based on owl:someValuesFrom, 167–168
- reasoning based on set operators, 174
- reasoning based on symmetric property, 185
- reasoning based on transitive property, 186–187
- specifications, 156
- OWL 2, 157
 - axiom, 159
 - axiom annotation, 217
 - entities, 159
 - entity declaration, 218
 - expressions, 159
 - Functional-Style syntax, 160
 - imports and versioning, 221
 - keys, *see* OWL 2, owl:hasKey
 - Manchester syntax, 160
 - metamodeling, *see* OWL 2, punning
 - negative fact assertions, 199
 - OWL 2 DL, 230
 - OWL 2 EL, 230
 - OWL 2 Full, 230
 - OWL 2 QL, 230–231
 - OWL 2 RL, 230, 232
 - OWL 2 specifications, 157
 - owl:AllDisjointClasses, 197
 - owl:AllDisjointProperties, 206
 - owl:annotatedProperty, 217
 - owl:annotatedSource, 217
 - owl:annotatedTarget, 217
 - owl:assertionProperty, 200
 - owl:AsymmetricProperty, 205
 - owl:Axiom, 217
 - owl:datatypeComplementOf, 213
 - owl:disjointUnionOf, 198
 - owl:hasKey, 209
 - owl:hasSelf, 201
 - owl:intersectionOf, 213
 - owl:IrreflexiveProperty, 204
 - owl:maxQualifiedCardinality, 203
 - owl:minQualifiedCardinality, 202
 - owl:NegativeDataPropertyAssertion, 199
 - owl:NegativeObjectPropertyAssertion, 199
 - owl:onDatatype, 212
 - owl:propertyDisjointWith, 206
 - owl:qualifiedCardinality, 203
 - owl:ReflexiveProperty, 204
 - owl:sourceIndividual, 200
 - owl:targetIndividual, 200
 - owl:unionOf, 213
 - owl:versionIRI, 221
 - owl:withRestrictions, 212
 - OWL/XML, 161
 - property chain, 207
 - punning, 214
 - RDF/XML syntax, 160
 - reasoning based on cardinality restrictions, 203
 - reasoning based on disjoint property, 207
 - reasoning based on key, 210

- reasoning based on property chain, 209
 - reasoning based on reflexive, irreflexive and asymmetric property, 205
 - reasoning based on self restriction property, 201
 - supported datatypes, 211
 - syntactic sugar, 197
 - top and bottom properties, 219
- P**
- Page snippet, 319
 - Pellet, 472
 - Plug-in architecture, 477
 - Point And Shoot, 112
 - Property-value pair, 3
 - Protégé, 475
 - OWL API, 476
 - Programming Development Kit, 476
 - Protocol and RDF Query Language, *see* SPARQL
- Q**
- QName, 30
 - Qualified name, *see* QName
- R**
- RacerPro, 472
 - RDF
 - abstract model, 25–42
 - basic rule #1, 25
 - basic rule #2, 27
 - basic rule #3, 75
 - datatype URI, 37
 - definition, 20
 - graph, 26
 - graph structure of a statement, 26
 - implementation of the RDF abstract model, 26
 - language tag, 37
 - literals, 37
 - long form of RDF/XML syntax, 56
 - Notation-3, 65
 - N-triples, 65
 - object, 26
 - in official language, 19–21
 - in plain English, 21–25
 - predicate, 63
 - property, 35
 - property value, 36
 - rdf, 42
 - rdf:about, 44
 - rdf:Alt, 59
 - rdf:Bag, 59
 - rdf:datatype, 52
 - rdf:Description, 44
 - rdf:first, 61
 - rdf:ID, 56
 - rdf:li, 60
 - rdf:List, 61
 - rdf:nil, 61
 - rdf:nodeID, 55
 - rdf:object, 63
 - rdf:parseType, 51
 - rdf:predicate, 63
 - rdf:RDF, 42–43
 - rdf:resource, 44
 - rdf:rest, 61
 - rdf:Seq, 59
 - rdf:statement, 63
 - rdf:subject, 63
 - rdf:type, 45
 - rdf:value, 50
 - RDF/XML syntax, 42
 - reification of a statement, 63
 - reification vocabulary, 63
 - resource, 27
 - resource XML node, 44
 - serialization syntax, 42
 - short form of RDF/XML sterilization, 58
 - statement, 25
 - subject, 26
 - triple, 26
 - typed literal value, 37
 - typed node, 45
 - typed node element, *see* RDF, typed node
 - un-typed literal value, 37
 - validator, 84
 - vocabulary, 42
 - W3C specifications, 10
 - RDFa, 96
 - attributes and elements, 96–97
 - examples, 99–104
 - and RDF, 104
 - rules of markup, 97–99
 - RDF Bookmashup, 406
 - RDF data store, 243
 - RDFS, 111
 - in official language, 110–111
 - in plain English, 109–110
 - rdfs, 114
 - rdfs:Class, 114
 - rdfs:comment, 132
 - rdfs:Datatype, 129
 - rdfs:domain, 120
 - rdfs:isDefinedBy, 132
 - rdfs:label, 132
 - rdfs:Literal, 129

- rdfs:range, 120
- rdfs:Resource, 114
- rdfs:seeAlso, 131
- rdfs:subClassOf, 117
- rdfs:subPropertyOf, 126
- rdfs:XMLLiteral, 130
- reasoning based on RDFS ontology, 149–151
 - W3C recommendation, 110–111
- RDF Schema, *see* RDFS
- RDF/S, *see* RDFS
- RDF-S, *see* RDFS
- RDF triple store, 243
- Reasoner, 471, 524
 - inference process, 471
 - reasoning, 471
- Redland, 470
- Relationship between Linked Data and the Semantic Web, 17
- Remote SPARQL query, 553
- Resource Description Framework, *see* RDF
- Revyu, 456
- Rich Snippets, 319
 - aggregate review, 321
 - individual review, 321
 - microformats supported, 322
 - ontologies supported, 322
 - Testing Tool, 322
- S**
- SameAs, 422
- Sampras, Pete, 394
- Screen-scraping, 573
- Search engine, 315
 - anchors, 317
 - barrels, 317
 - basic flow, 315
 - crawling, 316
 - indexer, 317
 - indexing, 317
 - links, 317
 - PageRanking, 317
 - rank, 317
 - repository, 317
 - searching, 317
 - seed URLs, 316
 - sorter, 317
 - store server, 317
 - URL Resolver, 317
 - URL server, 316
- SearchMonkey, 323
 - badge, 328
 - creating presentation applications, 327
 - DataRSS, 325
 - Enhanced Result, 327
 - high level architecture, 325
 - Infobar, 327
 - microformats supported, 329
 - online development tool, 328
 - ontologies supported, 329
 - Page custom data service, 326
 - Search Gallery, 328
 - testing tool, 329
 - trigger URL, 328
 - Web Service custom data service, 326
 - XSLT Custom Data Service, 326
- Semantic annotation, *see* Semantic markup
- Semantic annotation in wiki, 335
 - link, 339
 - text, 343
- Semantic markup, 308
 - automatic markup, 313
 - manually markup, 313
 - procedure and example, 308–312
- Semantic mashups, 411
- Semantic MediaWiki, 334
 - Additional printouts, 350
 - built-in datatypes, 344
 - Factbox, 347
 - inferencing, 356
 - inferencing based on category hierarchy, 358
 - inferencing capability based on property hierarchy, 358
 - logical AND, 351
 - logical OR, 353
 - Page type, 345
 - Property, 342
 - query language, 350
 - RDF feed, 362
 - reuse existing ontologies, 372
 - semantic browsing interface, 348
 - Semantic wiki vocabulary and terminology, *see* SWiVT
 - Special:Ask, 350
 - sub-query, 354
 - SWiVT, 360
 - swikt:BuiltInType, 361
 - swikt:CustomType, 362
 - swikt:page, 361
 - swikt:Subject, 360
 - swikt:Type, 361
 - swikt:Wikipedia, 360
 - type, 344
- Semantics, 9, 14
- The Semantic Web, 15, 17

- Semantic Web development methodologies, 478–484
- Semantic Web search engines, 441, 478
- Semantic Web vs. Linked Data, 410
- Semantic wiki, 334
- Sesame, 469
- SHOE, 156
- ShopBot, 573
- ShopBot on the Semantic Web, 583
- Sig.ma, 446
- Simple HTML Ontology Extensions, *see* SHOE
- Simple Knowledge Organization Systems, *see* SKOS
- Sindice, 422, 443
- Sindice’s Data Web Services API, 443
- Single Lens Reflex, *see* SLR
- SKOS, 138
 - skos, 142
 - skos:altLabel, 143
 - skos:broader, 144
 - skos:broadMatch, 148
 - skos:closeMatch, 148
 - skos:Concept, 143
 - skos:ConceptScheme, 146
 - skos:definition, 144
 - skos:exactMatch, 144
 - skos:example, 144
 - skos:hasTopConcept, 147
 - skos:hiddenLabel, 143
 - skos:historyNote, 144
 - skos:inScheme, 146
 - skos:narrower, 144
 - skos:narrowMatch, 148
 - skos:note, 144
 - skos:prefLabel, 143
 - skos:related, 144
 - skos:relatedMatch, 148
 - skos:scopeNote, 144
 - specifications, 142
- Slash URI, 28
- SLR, 21
- Smart agent, 2
- SMORE, 312
- SPARQL 1, 277
- SPARQL 1.1 Query, 278
 - AS, 283
 - aggregate functions, 278
 - count() aggregate function, 279
 - expressions with SELECT, 283
 - MINUS operator, 282
 - negation, 281
 - NOT EXISTS operator, 282
 - projected expressions, 283
 - property paths, 285
 - sample() aggregate function, 280
 - Subquery, 280
 - sum() aggregate function, 279
- SPARQL 1.1 Update, 285
 - DELETE DATA operation, 287
 - DELETE operation, 288
 - graph creation, 289
 - graph management, 286
 - graph remove, 289
 - INSERT DATA operation, 286
 - INSERT operation, 287
 - LOAD and CLEAR operation, 289
 - SILNET keyword, 290
- SPARQL, 241–242
 - alternative match, 264
 - ask query, 249, 275
 - background graph, 267
 - BASE directive, 252
 - basic SELECT queries, 252–257
 - bind, 250
 - binding, *see* SPARQL, bind
 - FROM clause, 252
 - CONSTRUCT query, 249, 372
 - DESCRIBE query, 249, 275
 - distinct modifier, 260
 - endpoint, 244
 - filter keyword, 261
 - functions and operators, 263
 - generic endpoints, 244
 - graph pattern, 250
 - named graphs, 267
 - in official language, 241–242
 - offset/limit modifiers, 261
 - optional keyword, 257
 - order by modifier, 260
 - in plain English, 242–243
 - PREFIX definitions, 252
 - projection query, *see* SPARQL, SELECT query
 - query modifiers, 253
 - query solution, 259
 - SELECT clause, 252
 - SELECT query, 249, 252
 - solution, 259
 - specification, 241
 - specific endpoints, 244
 - triple pattern, 249
 - union keyword, 264
 - variable, 250
 - WHERE clause, 252
 - working with multiple graphs, 267–272

Spider, *see* Crawler
 Structured information, [17](#)
 Swoogle, [445](#)
 SWSE, [444](#)
 Synsets, [295](#)

T

Taxonomy, [139](#)
 Terse RDF Triple Language, *see* Turtle
 Thesaurus, [139](#)
 TopBraid, [477](#)
 Composer, [477](#)
 Turtle, [66](#)
 @base, [68](#)
 @prefix, [67](#)
 <>, [66](#)
 [], [71](#)
 commas (,), [70](#)
 semicolons, [70](#)
 Token a, [69](#)
 ttl, [66](#)
 Typed link, [411](#), [430](#)

U

Uniform Resource Identifier, *see* URI
 Uniform Resource Locator, *see* URL
 303 URI, [416](#)
 URI, [28](#)
 URI aliases, [421–423](#)
 URIfref, [28](#)
 URI reference, *see* URIfref
 303 URIs *vs.* hash URIs, [421](#)
 URL, [27](#)
 Use SPARQL to query in-memory RDF models, [549–553](#)
 Use SPARQL to query remote datasets, [553–556](#)

V

Vapour, [438](#)
 Virtuoso, [469](#), [473](#)
 Virtuoso Universal Server, *see* Virtuoso

W

W3C Semantic Web activity, [18](#)
 W3C Semantic Web activity news web site, [18](#)
 W3C Semantic Web community Wiki page, [18](#)
 W3C Semantic Web frequently asked questions, [18](#)
 W3C Semantic Web interest group, [18](#)
 Web of Data, [17](#), [410](#)
 Web data mining, [11](#)
 Web of Linked Data, [409–411](#)
 Web 2.0 mashup, [463](#)
 Web Ontology Language, *see* OWL
 Web services, [11](#)
 Well-known ontologies, [423](#)
 Wiki, [332](#)
 category system, [335](#)
 namespace, [336](#)
 wiki engine, [332](#)
 wikitext, [332](#)
 Wikicompany, [337](#)
 namespaces, [337](#)
 properties, [346](#)
 Wikipedia, [380](#)
 infobox, [385](#)
 template, [385](#)
 Wikipedia datasets, [401](#)
 WordNet, [295](#)

X

XML entity, [53](#)

Y

Yahoo! Slurp, [325](#)