Chapter 9
The Forms of Resource Descriptions

Ryan Shaw
Murray Maloney

9.1. Introduction ............................................................... 357
9.2. Structuring Descriptions ................................................ 359
9.3. Writing Descriptions ..................................................... 380
9.4. Worlds of Description ................................................... 388
9.5. Key Points in Chapter Nine .............................................. 393

9.1 Introduction

Throughout this book, we have emphasized the importance of separately considering fundamental organizing principles, application-specific concepts, and details of implementation. The three-tier architecture we introduced in §1.6 is one way to conceptualize this separation. In §6.7, we contrasted the implementation-focused perspective for analyzing relationships with other perspectives that focus on the meaning and abstract structure of relationships. In this chapter, we present this contrast between conceptualization and implementation in terms of separating the content and form of resource descriptions.

In the previous chapters, we have considered principles and concepts of organizing in many different contexts, ranging from personal organizing systems to cultural and institutional ones. We have noted that some organizing systems have limited scope and expected lifetime, such as a task-oriented personal organizing system like a shopping list. Other organizing systems support broad uses that rely on standard categories developed through rigorous processes, like a product catalog.

By this point you should have a good sense of the various conceptual issues you need to consider when deciding how to describe a resource in order to meet the
goals of your organizing system. Considering those issues will give you some sense of what the content of your descriptions should be. In order to focus on the conceptual issues, we have deferred discussion of specific implementation issues. Implementation involves choosing the specific form of your descriptions, and that is the topic of this chapter.

We can approach the problem of how to form resource descriptions from two perspectives: structuring and writing. From one perspective, resource descriptions are things that are used by both people and computational agents. From this perspective, choosing the form of resource descriptions is a kind of design. This is easy to see for certain kinds of resource descriptions, notably signs and maps found in physical environments like airport terminals, public libraries, and malls. In these spaces, resource descriptions are quite literally designed to help people orient themselves and find their way. But any kind of resource description, not just those embedded in the built environment, can be viewed as a designed object. Designing an object involves making decisions about how it should be structured so that it can best be used for its intended purpose. From a design perspective, choosing the form of a resource description means making decisions about its structure.

In §6.5, we took a structural perspective on resources and the relationships among them. In this chapter, we will take a structural perspective on resource descriptions. The difference is subtle but important. A structural perspective on resource relationships focuses on how people or computational processes associate, arrange, and connect those resources. A structural perspective on resource descriptions focuses on how those associations, arrangements, and connections are explicitly represented or implemented in the descriptions we create. Mismatches between the structure imposed on the resources being organized and the structure of the descriptions used to implement that organization could result in an organizing system that is complex, inefficient, and difficult to maintain, as you will see in our first example (Example 9.1, Description structured as a dictionary).

The structures of resource descriptions enable or inhibit particular ways of interacting with those descriptions, just as the descriptions themselves enable or inhibit particular ways of interacting with the described resources. (See §3.4 Designing Resource-based Interactions (page 109), and Chapter 10, Interactions with Resources) Keep in mind that resource descriptions are themselves information resources, so much of what we will say in this chapter is applicable to the structures and forms of information resources in general. Put another way, the structure and form of information resources informs the design of resource descriptions.

From another perspective, creating resource descriptions is a kind of writing. I may describe something to you orally, but such a description might not be very
useful to an organizing system unless it were transcribed. Organizing systems need persistent descriptions, and that means they need to be written. In that sense, choosing the form of a resource description means making decisions about notation and syntax.

Modern Western culture tends to make a sharp distinction between designing and writing, but there are areas where this distinction breaks down, and the creation of resource descriptions in organizing systems is one of them. In the following sections, we will use designing and writing as two lenses for looking at the problem of how to choose the form of resource descriptions. Specifically, we will examine the spectrum of options we have for structuring descriptions, and the kinds of syntaxes we have for writing those descriptions.

9.2 Structuring Descriptions

Choosing how to structure resource descriptions is a matter of making principled and purposeful design decisions in order to solve specific problems, serve specific purposes, or bring about some desirable property in the descriptions. Most of these decisions are specific to a domain: the particular context of application for the organizing system being designed and the kinds of interactions with resources it will enable. Making these kinds of context-specific decisions results in a model of that domain. (See §5.3.1.2 Abstraction in Resource Description (page 192).)

Over time, many people have built similar kinds of descriptions. They have had similar purposes, desired similar properties, and faced similar problems. Unsurprisingly, they have converged on some of the same decisions. When common sets of design decisions can be identified that are not specific to any one domain, they often become systematized in textbooks and in design practices, and may eventually be designed into standard formats and architectures for creating organizing systems. These formally recognized sets of design decisions are known as abstract models or metamodels. Metamodels describe structures commonly found in resource descriptions and other information resources, regardless of the specific domain. While any designer of an organizing system will usually create a model of her specific domain, she usually will not create an entirely new metamodel but will instead make choices from among the metamodels that have been formally recognized and incorporated into existing standards. The resulting model is sometimes called a “domain-specific language.” Reusing standard metamodels can bring great economical advantages, as developers can reuse tools designed for and knowledge about these metamodels, rather than having to start from scratch.

In the following sections, we examine some common kinds of structures used as the basis for metamodels. But first, we consider a concrete example of how the structure of resource descriptions supports or inhibits particular uses. As we
explained in Chapter 1, the concept of a resource de-emphasizes the differences between physical and digital things in favor of focusing on how things, in general, are used to support goal-oriented activity. Different kinds of books can be treated as information resources regardless of the particular mix of tangible and intangible properties they may have. Since resource descriptions are also information resources, we can similarly consider how their structures support particular uses, independent of whether they are physical, digital, or a mix of both.

During World War II, a British chemist named W. E. Batten developed a system for organizing patents. The system consisted of a language for describing the product, process, use, and apparatus of a patent, and a way of using punched cards to record these descriptions. Batten used cards printed with matrices of 800 positions (see Figure 9.1). Each card represented a specific value from the vocabulary of the description language, and each position corresponded to a particular patent. To describe patent #256 as covering extrusion of polythene to produce cable coverings, one would first select the cards for the values poly-
Punchcards were an important information input and storage medium for decades, even before the invention of computers. The Hollerith keyboard punch was used to transcribe the information collected in the 1890 US census. The template being used in this photo is for recording information about a farm. The punch cards were tabulated by electromechanical machines. A merger of four tabulating machine companies in 1911 created a company whose current name is IBM.

This keyboard punch machine is in the collection of the Computer History Museum in Mountain View, California. (Photo by R. Glushko.)

thene, extrusion, and cable coverings, and then punch each card at the 256th position. The description of patent #256 would thus extend over these three cards.

The advantage of this structure is that to find patents covering extrusion of polythene (for any purpose), one needs only to select the two cards corresponding to those values, lay one on top of the other, and hold them up to a light. Light will shine through wherever there is a position corresponding to a patent described using those values. Patents meeting a certain description are easily found due to the structure of the cards designed to describe the patents.

Of course, this system has clear disadvantages as well. Finding the concepts associated with a particular patent is tedious, because every card must be inspected. Adding a new patent is relatively easy as long as there is an index that allows the cards for specific concepts to be located quickly. However, once the cards run out of space for punching holes, the whole set of cards must be duplicated to accommodate more patents: a very expensive operation. Adding new concepts is potentially easy: simply add a new card. But if we want to be able to find existing patents using the new concept, all the existing patents would have to be re-examined to determine whether their positions on the new card should be punched: also an expensive operation.

The structure of Batten’s cards supported rapid selection of resources given a partial description. The kinds of structures we will examine in the following sections are not quite so elaborate as Batten’s cards. But like the cards, each kind of structure supports more efficient mechanical execution of certain operations, at the cost of less efficient execution of others.
9.2.1 Kinds of Structures

Sets, lists, dictionaries, trees, and graphs are kinds of structures that can be used to form resource descriptions. As we shall see, each of these kinds is actually a family of related structures. These structures are *abstractions*: they describe formal structural properties in a general way, rather than specifying an exact physical or textual form. Abstractions are useful because they help us to see common properties shared by different specific ways of organizing information. By focusing on these common properties, we can more easily reason about the operations that different forms support and the affordances that they provide, without being distracted by less relevant details.

9.2.1.1 Blobs

The simplest kind of structure is no structure at all. Consider the following description of a book: *Sebald’s novel uses a walking tour in East Anglia to meditate on links between past and present, East and West.* This description is an unstructured text expression with no clearly defined internal parts, and we can consider it to be a *blob*. Or, more precisely, it has structure, but that structure is the underlying grammatical structure of the English language, and none of that grammatical structure is explicitly represented in a surface structure when the sentence is expressed. As readers of English we can interpret the sentence as a description of the subject of the book, but to do this mechanically is difficult. On the other hand, such a written description is relatively easy to create, as the describer can simply use natural language.

A blob need not be a blob of text. It could be a photograph of a resource, or a recording of a spoken description of a resource. Like blobs of text, blobs of pixels or sound have underlying structure that any person with normal vision or hearing can understand easily. But we can treat these blobs as unstructured, because none of the underlying structure in the visual or auditory input is explicit, and we are concerned with the ways that the structures of resource descriptions support or inhibit mechanical or computational operations.

9.2.1.2 Sets

The simplest way to structure a description is to give it parts and treat them as a *set*. For example, the description of Sebald’s novel might be reformulated as a set of terms: *Sebald, novel, East Anglia, walking, history*. Doing this has lost much of the meaning, but something has been gained: we now can easily distinguish *Sebald* and *walking* as separate items in the description. This makes it easier to find, for example, all the descriptions that include the term *walking*. (Note that this is different from simply searching through blob-of-text descriptions for the word *walking*. When treated as a set, the description *Fiji, fire walk-
ing, memoir does not include the term walking, though it does include the term fire walking.)

Sets make it easy to find intersections among descriptions. Sets are also easy to create. In §8.1.2 we looked at “folksonomies,” organizing systems in which non-professional users create resource descriptions. In these systems, descriptions are structured as sets of “tags.” To find resources, users can specify a set of tags to obtain resources having descriptions that intersect at those tags. This is more valuable if the tags come from a controlled vocabulary, making intersections more likely. But enforcing vocabulary control adds complexity to the description process, so a balance must be struck between maximizing potential intersections and making description as simple as practical.

A set is a type or class of structure. We can refine the definition of different kinds of sets by introducing constraints. For example, we might introduce the constraint that a given set has a maximum number of items. Or we might constrain a set to always have the same number of items, giving us a fixed-size set. We can also remove constraints. Sets do not contain duplicate items (think of a tagging system in which it does not make sense to assign the same tag more than once to the same resource). If we remove this uniqueness constraint, we have a different structure known as a “bag” or “multiset.”

9.2.1.3 Lists

Constraints are what distinguish lists from sets. A list, like a set, is a collection of items with an additional constraint: their items are ordered. If we were designing a tagging system in which it was important that the order of the tags be maintained, we would want to use lists, not sets. Unlike sets, lists may contain duplicate items. In a list, two items that are otherwise the same can be distinguished by their position in the ordering, but in a set this is not possible. For example, we might want to organize the tags assigned to a resource, listing the most used tag first, the least frequently used last, and the rest according to their frequency of use.

Again, we can introduce constraints to refine the definition of different kinds of lists, such as fixed-length lists. If we constrain a list to contain only items that are themselves lists, and further specify that these contained lists do not themselves contains lists, then we have a table (a list of lists of items). A spreadsheet is a list of lists.
9.2.1.4 Dictionaries

One major limitation of *lists* and *sets* is that, although items can be individually addressed, there is no way to distinguish the items except by comparing their values (or, in a list, their positions in the ordering). In a set of terms like *Sebald, novel, East Anglia, walking, history*, for example, one cannot easily tell that *Sebald* refers to the author of the book while *East Anglia* and *walking* refer to what it is about. One way of addressing this problem is to break each item in a set into two parts: a *property* and a *value*. So, for example, our simple set of tags might become *author: Sebald, type: novel, subject: East Anglia, subject: walking, subject: history*. Now we can say that *author*, *type*, and *subject* are the properties, and the original items in the set are the values.

```
author
    Sebald

type
    novel

subject1
    East Anglia

subject2
    walking

subject3
    history
```

This kind of structure is called a *dictionary*, a *map* or an *associative array*. A *dictionary* is a set of property-value pairs or entries. It is a set of entries, not a list of entries, because the pairs are not ordered and because each entry must have a unique key. Note that this specialized meaning of *dictionary* is different from the more common meaning of “dictionary” as an alphabetized list of terms accompanied by sentences that define them. The two meanings are related, however. Like a “real” dictionary, a *dictionary* structure allows us to easily find the value (such as a definition) associated with a particular property or key (such as a word). But unlike a real dictionary, which orders its keys alphabetically, a *dictionary* structure does not specify an order for its keys.

Dictionaries are ubiquitous in resource descriptions. Structured descriptions entered using a form are easily represented as dictionaries, where the form items’ labels are the properties and the data entered are the values. Tabular data with a “header row” can be thought of as a set of dictionaries, where column headers are the properties for each dictionary, and each row is a set of corresponding values. Dictionaries are also a basic type of data structure found in nearly all programming languages (referred to as associative arrays).
Again, we can introduce or remove constraints to define specialized types of dictionaries. A sorted dictionary adds an ordering over entries; in other words, it is a list of entries rather than a set. A *multimap* is a dictionary in which multiple entries may have the same key.

### 9.2.1.5 Trees

In dictionaries as they are commonly understood, properties are terms and values are their corresponding definitions. The terms and values are usually words, phrases, or other expressions that can be ordered alphabetically. But if generalize the notion of a dictionary as abstract sets of property-value pairs, the values can be anything at all. In particular, the values can themselves be dictionaries. When a dictionary structure has values that are themselves dictionaries, we say that the dictionaries are *nested*. Nesting is very useful for resource descriptions that need more structure than what a (non-nested) dictionary can provide.

*Figure 9.2. Four Nested Dictionaries.*

When a dictionary contains other dictionaries, they are said to be nested.

*Figure 9.2, Four Nested Dictionaries.* presents an example of nested dictionaries. At the top level there is one dictionary with a single entry having the property *a*. The value associated with *a* is a dictionary consisting of two entries, the first having property *b* and the second having property *c*. The values associated with *b* and with *c* are also dictionaries.

If we nest dictionaries like this, and our “top” dictionary (the one that contains all the others) has only one entry, then we have a kind of *tree* structure. *Figure 9.3, A Tree of Properties and Values.* shows the same properties and values
as Figure 9.2, this time arranged to make the tree structure more visible. *Trees* consist of *nodes* (the letters and numbers in Figure 9.3) joined by *edges* (the arrows). Each node in the tree with a circle around it is a property, and the value of each property consists of the nodes below (to the right of) it in the tree. A node is referred to as the *parent* of the nodes below it, which in turn are referred to as the *children* of that node. The edges show these “parent of” relationships between the nodes. The node with no parent is called the *root* of the tree. Nodes with no children are called *leaf* nodes.

**Figure 9.3. A Tree of Properties and Values.**

![Diagram of a tree structure](image)

An alternative representation of nested dictionaries is as a tree. The lowest level or leaf nodes of the tree contain property values.

The *XML Information Set* defines a specific kind of tree structure by adding very specific constraints, including ordering of child nodes, to the basic definition of a tree. The addition of an ordering constraint distinguishes XML trees from nested dictionaries, in which child nodes do not have any order (because dictionary entries do not have an ordering). Ordering is an important constraint for resource descriptions, since without ordering it is impossible to, for example, list multiple authors while guaranteeing that the order of authors will be maintained. Figure 9.3 depicts a kind of tree with a different set of constraints: all non-leaf nodes are properties, and all leaves are values. We could also define a
tree in which every node has both a property and a value. Trees exist in a large variety of flavors, but they all share a common topology: the edges between nodes are directed (one node is the parent and the other is the child), and every node except the root has exactly one parent.

Trees provide a way to group statements describing different but related resources. For example, consider the description structured as a dictionary here:

**Example 9.1. Description structured as a dictionary**

- author given names → Winfried Georg
- author surname → Sebald
- title → Die Ringe des Saturn
- pages → 371

The dictionary groups together four property-value pairs describing a particular book. (The arrows are simply a schematic way to indicate property-value relations. Later in the chapter we look at ways to “write” these relations using some specific syntax.)

But really the first two entries are not describing the book; they are describing the book’s author. So, it would be better to group those two statements somehow. We can do this by nesting the entries describing the author within the book description, creating a tree structure:

**Example 9.2. Nesting an author description within a book description**

- author →
  - given names → Winfried Georg
  - surname → Sebald
- title → Die Ringe des Saturn
- pages → 371

Using a tree works well in this case because we can treat the book as the primary resource being described, making it the root of our tree, and adding on the author description as a “branch.”
We also could have chosen to make the author the primary resource, giving us a

tree like the one in Example 9.3.

**Example 9.3. Nesting book descriptions within an author description**

given names → Winfried Georg
surname → Sebald
books authored →
  1. title → Die Ringe des Saturn
      pages → 371
  2. title → Austerlitz
      pages → 416

9.2.1.6 Graphs

Suppose we were describing two books, where the author of one book is the
subject of the other, as in Example 9.4, Two related descriptions:

**Example 9.4. Two related descriptions**

  1. author → Mark Richard McCulloch
      title → Understanding W. G. Sebald
      subject → Winfried Georg Sebald
  2. author → Winfried Georg Sebald
      title → Die Ringe des Saturn

By looking at these descriptions, we can guess the relationship between the two
books, but that relationship is not explicitly represented in the structure: we
just have two separate dictionaries and have inferred the relationship by match­
ing property values. It is possible that this inference could be wrong: there
might be two people named *Winfried Georg Sebald*. How can we structure these
descriptions to explicitly represent the fact that the *Winfried Georg Sebald*
that is the subject of the first book is the same *Winfried Georg Sebald* who authored
the second?

One possibility would be to make *Winfried Georg Sebald* the root of a tree, simi­
lar to the approach taken in Example 9.3, Nesting book descriptions within an
author description, adding a *book about* property alongside the *books authored*
one. This solution would work fine if people were our primary resources, and it
thus made sense to structure our descriptions around them. But suppose that
we had decided that our descriptions should be structured around books, and
that we were using a vocabulary that took this perspective (with properties such
as *author* and *subject* rather than *books authored* and *books about*). We should
not let a particular structure limit the organizational perspective we can take,
as Batten’s cards did. Instead, we should consciously choose structures to suit
our organizational perspective. How can we do this?
If we treat our two book descriptions as trees, we can join the two branches (subject and author) that share a value. When we do this, we no longer have a tree, because we now have a node with more than one parent (Figure 9.4, Descriptions Linked into a Graph.). The structure in Figure 9.4, Descriptions Linked into a Graph, is a graph. Like a tree, a graph consists of a set of nodes connected by edges. These edges may or may not have a direction (§6.6.3 Directionality (page 257)). If they do, the graph is referred to as a “directed graph.” If a graph is directed, it may be possible to start at a node and follow edges in a path that leads back to the starting node. Such a path is called a “cycle.” If a directed graph has no cycles, it is referred to as an “acyclic graph.”

A tree is just a more constrained kind of graph. Trees are directed graphs because the “parent of” relationship between nodes is asymmetric: the edges are arrows that point in a certain direction. (See §6.3.2.1 Symmetry (page 236).) Furthermore, trees are acyclic graphs, because if you follow the directed edges from one node to another, you can never encounter the same node twice. Finally, trees have the constraint that every node (except the root) must have exactly one parent.

In Figure 9.4, Descriptions Linked into a Graph, we have violated this constraint by joining our two book trees. The graph that results is still directed and acyclic, but because the Winfried George Sebald node now has two parents, it is no longer a tree.

Graphs are very general and flexible structures. Many kinds of systems can be conceived of as nodes connected by edges: stations connected by subway lines, people connected by friendships, decisions connected by dependencies, and so on. Relationships can be modeled in different ways using different kinds of graphs. For example, if we assume that friendship is symmetric (see §6.3.2.1 Symmetry (page 236)), we would use an undirected graph to model the relationship. However, in web-based social networks friendship is often asymmetric (you might “friend” someone who does not reciprocate), so a directed graph is more appropriate.

Often it is useful to treat a graph as a set of pairs of nodes, where each pair may or may not be directly connected by an edge. Many approaches to characterizing structural relationships among resources (see §6.5.3 Structural Relationships between Resources (page 250)) are based on modeling the related resources as a set of pairs of nodes, and then analyzing patterns of connectedness among

Stop and Think: Social Network Properties

Compare the concept of “friend” in Facebook with that of “follower” in Twitter, in terms of the semantic properties discussed in §6.3.2 Properties of Semantic Relationships (page 236) and the graph properties discussed in this section.
Figure 9.4. Descriptions Linked into a Graph.

Descriptions can be linked to form a graph when the value assigned to two different properties is the same.

them. As we will see, being able to break down a graph into pairs is also useful when we structure resource descriptions as graphs.

9.2.2 Comparing Metamodels: JSON, XML and RDF

Now that we are familiar with the various kinds of metamodels used to structure resource descriptions, we can take a closer look at some specific metamodels. A detailed comparison of the affordances of different metamodels is beyond the scope of this chapter. Here we will simply take a brief look at three popular metamodels—JSON, XML, and RDF—in order to see how they further specify and constrain the more general kinds of metamodels introduced above.

9.2.2.1 JSON

JavaScript Object Notation (JSON)

JavaScript Object Notation (JSON) is a textual format for exchanging data that borrows its metamodel from the JavaScript programming language. Specifically, the JSON metamodel consists of two kinds of structures found in JavaScript: lists (called “arrays” in JavaScript) and dictionaries (called
“objects” in JavaScript). Lists and dictionaries contain values, which may be strings of text, numbers, Booleans (true or false), or the null (empty) value. Again, these types of values are taken directly from JavaScript. Lists and dictionaries can be values too, meaning lists and dictionaries can be nested within one another to produce more complex structures such as tables and trees.

Lists, dictionaries, and a basic set of value types constitute the JSON metamodel. Because this metamodel is a subset of JavaScript, the JSON metamodel is very easy to work with in JavaScript. Since JavaScript is the only programming language that is available in all web browsers, JSON has become a popular choice for developers who need to work with data and resource descriptions on the web. (See §9.3.2 Writing Systems (page 382) later in this chapter.) Furthermore, many modern programming languages provide data structures and value types equivalent to those provided by JavaScript. So, data represented as JSON is easy to work with in many programming languages, not just JavaScript.

### 9.2.2.2 XML Information Set

The XML Information Set metamodel is derived from data structures used for document markup. (See §5.2.2.2.) These markup structures—elements and attributes—are well suited for programmatically manipulating the structure of documents and data together.

**XML Infoset**

The XML Infoset is a tree structure, where each node of the tree is defined to be an “information item” of a particular type. Each information item has a set of type-specific properties associated with it. At the root of the tree is a “document item,” which has exactly one “element item” as its child. An element item has a set of attribute items, and a list of child nodes. These child nodes may include other element items, or they may be character items. (See §9.2.1 Kinds of Structures (page 362) below for more on characters.) Attribute items may contain character items, or they may contain typed data, such as name tokens, identifiers and references. Element identifiers and references (ID/IDREF) may be used to connect nodes, transforming a tree into a graph.

**Figure 9.5, A Description Structure.** is a graphical representation of how an XML document might be used to structure part of a description of an author and his works. This example demonstrates how we might use element items to model the domain of the description, by giving them names such as author and title. The character items that are the children of these elements hold the content of the description: author names, book titles, and so on. Attribute items are used to hold auxiliary information about this content, such as its language.
This example also demonstrates how the XML Infoset supports mixed content by allowing element items and character items to be “siblings” of the same parent element. In this case, the Infoset structure allows us to specify that the book description can be displayed as a line of text consisting of the original title and the translated title in parentheses. The elements and attributes are used to indicate that this line of text consists of two titles written in different languages, not a single title containing parentheses.

If not for mixed content, we could not write narrative text with hypertext links embedded in the middle of a sentence. It gives us the ability to identify the subcomponents of a sentence, so that we could distinguish the terms “Sebald,” “walking” and “East Anglia” as an author and two subjects.

Using schemas to define data representation formats is a good practice that facilitates shared understanding and contributes to long-term maintainability in institutional or business contexts. An XML schema represents a contract among the parties subscribing to its definitions, whereas JSON depends on out-of-band communication among programmers. The notion that “the code is the docu-
mentalization” may be fashionable among programmers, but modelers prefer to de-
sign at a higher level of abstraction and then implement.

The XML Infoset presents a strong contrast to JSON and does not always map in
a straightforward way to the data structures used in popular web scripting lan-
guages. Whereas JSON’s structures make it easier for object-oriented program-
mers to readily exchange data, they lack any formal schema language and can-
not easily handle mixed content.

9.2.2.3 RDF

In Figure 9.4, Descriptions Linked into a Graph., we structured our resource de-
scription as a graph by treating resources, properties, and values as nodes, with
edges reflecting their combination into descriptive statements. However, a more
common approach is to treat resources and values as nodes, and properties as
the edges that connect them. Figure 9.6, Treating Properties as Edges Rather
Than Nodes. shows the same description as Figure 9.4, Descriptions Linked into a
Graph., this time with properties treated as edges. This roughly corresponds to
the particular kind of graph metamodel defined by RDF. (§5.2.2.4 Resource De-
scription Framework (RDF) (page 184))

We have noted that we can treat a graph as a set of pairs of nodes, where each
pair may be connected by an edge. Similarly, we can treat each component of
the description in Figure 9.6, Treating Properties as Edges Rather Than Nodes. as
a pair of nodes (a resource and a value) with an edge (the property) linking
them. In the RDF metamodel, a pair of nodes and its edge is called a triple, be-
cause it consists of three parts (two nodes and one edge). The RDF metamodel
is a directed graph, so it identifies one node (the one from which the edge is
pointing) as the subject of the triple, and the other node (the one to which the
edge is pointing) as its object. The edge is referred to as the predicate or (as we
have been saying) property of the triple.

Figure 9.7, Listing Triples Individually. lists separately all the triples in Figure 9.6
However, there is something missing in Figure 9.7. Figure 9.6 clearly indicates
that the Winfried George Sebald who is the subject of book 1 is the same Win-
fried George Sebald who is the author of book 2. In Figure 9.7, Listing Triples In-
dividually, this relationship is not clear. How can we tell if the Winfried George
Sebald of the third triple is the same as the Winfried George Sebald of the triple
statement? For that matter, how can we tell if the first three triples all involve
the same book 1? This is easy to show in a diagram of the entire description
description graph, where we can have multiple edges attached to a node. But when we dis-
eggregate that graph into triples, we need some way of uniquely referring to no-
des. We need identifiers (§4.4.3 Choosing Good Names and Identifiers (page
163)). When two triples have nodes with the same identifier, we can know that it
We can treat each component of a description as a pair of nodes (a resource and a value) with an edge (the property) linking them. Here, we have two book resources that are related to four values through five properties. The single value node, “Winfried George Sebald” is the subject of one book while being the author of the second book. The books are depicted as boxes, the edges as labeled arrows and the values as text strings.

is the same node. RDF achieves this by associating URIs with nodes. (See §5.2.2.4 Resource Description Framework (RDF) (page 184))

The need to identify nodes when we break down an RDF graph into triples becomes important when we want to “write” RDF graphs—create textual representations of them instead of depicting them—so that they can be exchanged as data. Tree structures do not necessarily have this problem, because it is possible to textually represent a tree structure without having to mention any node.
Lists each of the triples individually. Here, each statement relates one resource to one value through an edge. Thus, we have two distinct “Winfried George Sebald” value nodes. The books are depicted as boxes, the edges as labeled arrows and the values as text strings.

more than once. Thus, one price paid for the generality and flexibility of graph structures is the added complexity of recording, representing or writing those structures.

9.2.2.4 Choosing Your Constraints

This tradeoff between flexibility and complexity illustrates a more general point about constraints. In the context of managing and interacting with resource descriptions, constraints are a good thing. As discussed above, a tree is a graph with very specific constraints. These constraints allow you to do things with trees that are not possible with graphs in general, such as representing them
textually without repeating yourself, or uniquely identifying nodes by the path from the root of the tree to that node. This can make managing descriptions and the resources they describe easier and more efficient—if a tree structure is a good fit to the requirements of the organizing system. For example, an ordered tree structure is a good fit for the hierarchical structure of the content of a book or book-like document, such as an aircraft service manual or an SEC filing. On the other hand, the network of relationships among the people and organizations that collaborated to produce a book might be better represented using a graph structure. XML is most often used to represent hierarchies, but is also capable of representing network structures.

9.2.3 Modeling within Constraints

A metamodel imposes certain constraints on the structure of our resource descriptions. But in organizing systems, we usually need to further specify the content and composition of descriptions of the specific types of resources being organized. For example, when designing a system for organizing books, it is not sufficient to say that a book’s description is structured using XML, because the XML metamodel constrains structure and not the content of descriptions. We need also to specify that a book description includes a list of contributors, each entry of which provides a name and indicates the role of that contributor. This kind of specification is a model to which our descriptions of books are expected to conform. (See §5.3.1.2 Abstraction in Resource Description (page 192).)

When designing an organizing system we may choose to reuse a standard model. For example, ONIX for Books is a standard model (conforming to the XML metamodel) developed by the publishing industry for describing books.

If no such standard exists, or existing standards do not suit our needs, we may create a new model for our specific domain. But we will not usually create a new metamodel: instead we will make choices from among the metamodels, such as JSON, XML, or RDF, that have been formally recognized and incorporated into existing standards. Once we have selected a metamodel, we know the constraints we have to work with when modeling the resources and collections in our specific domain.

9.2.3.1 Specifying Vocabularies and Schemas

Creating a model for descriptions of resources in a particular domain involves specifying the common elements of those descriptions, and giving those elements standard names. (See §5.3 The Process of Describing Resources (page 188)) The model may also specify how these elements are arranged into larger structures, for example, how they are ordered into lists nested into trees. Metamodels vary in the tools they provide for specifying the structure and composition of domain-specific models, and in the maturity and robustness of the meth-
ods for designing them. RDF and XML each provide different, metamodel-specific tools to define a model for a specific domain. But not every metamodel provides such tools.

In XML, models are defined in separate documents known as schemas. An XML schema defining a domain model provides a vocabulary of terms that can be used as element and attribute names in XML documents that adhere to that model. For example, Onix for Books schema specifies that an author of a book should be called a Contributor, and that the page count should be called anExtent. An XML schema also defines rules for how those elements, attributes, and their content can be arranged into higher-level structures. For example, the Onix for Books specifies that the description of a book must include a list of Contributor elements, that this list must have at least one element in it, and that each Contributor element must have a ContributorRole child element.

If two descriptions share the same XML schema and use only that schema, then combining them is straightforward. If not, it can be problematic, unless someone has figured out exactly how the two schemas should “map” to one another. Finding such a mapping is not a trivial problem, as XML schemas may differ semantically, lexically, structurally, or architecturally despite sharing a common implementation form. (See Chapter 6, Describing Relationships and Structures.)

Tree structures can vary considerably while still conforming to the XML Infoset metamodel. Users of XML often specify rules for checking whether certain patterns appear in an XML document (document-level validation). This is less often done with RDF, because graphs that conform to the RDF metamodel all have the same structure: they are all sets of triples. This shared structure makes it simple to combine different RDF descriptions without worrying about checking structure at the document level. However, sometimes it is desirable to check descriptions at the document level, as when part of a description is required. As with XML, if consumers of those descriptions want to assert that they expect those descriptions to have a certain structure (such as a required property), they must check them at the document level.

Because the RDF metamodel already defines structure, defining a domain-specific model in RDF mainly involves specifying URIs and names for predicates. A set of RDF predicate names and URIs is known as an RDF vocabulary. Publication of vocabularies on the web and the use of URIs to identify and refer to predicate definitions are key principles of Linked Data and the Semantic Web. (Also see §6.8.1, as well as later in this chapter.)

For example, the Resource Description and Access (RDA) standard for cataloging library resources includes a set of RDF vocabularies defining predicates usable in cataloging descriptions. One such predicate is:

<http://rdvocab.info/Elements/extentOfText>
which is defined as “the number and type of units and/or subunits making up a resource consisting of text, with or without accompanying illustrations.” The vocabulary further specifies that this predicate is a refinement of a more general predicate:

<http://rdvocab.info/Elements/extent>

which can be used to indicate, “the number and type of units and/or subunits making up a resource” regardless of whether it is textual or not.

JSON lacks any standardized way to define which terms can be used. That does not mean one cannot use a standard vocabulary when creating descriptions using JSON, only that there is no agreed-upon way to use JSON to communicate which vocabulary is being used, and no way to automatically check that it is being used correctly.

9.2.3.2 Controlling Values

So far, we have focused on how models specify vocabularies of terms and how those terms can be used in descriptions. But models may also constrain the values or content of descriptions. Sometimes, a single model will define both the terms that can be used for property names and the terms that can be used for property values. For example, an XML schema may enumerate a list of valid terms for an attribute value.

Often, however, there are separate, specialized vocabularies of terms intended for use as property values in resource descriptions. Typically these vocabularies provide values for use within statements that describe what a resource is about. Examples of such subject vocabularies include the Library of Congress Subject Headings (LOC-SH) and the *Medical Subject Headings (MeSH)*. Other vocabularies may provide authoritative names for people, corporations, or places. Classification schemes are yet another kind of vocabulary, providing the category names for use as the values in descriptive statements that classify resources.

Because different metamodels take different approaches to specifying vocabularies, there will usually be different versions of these vocabularies for use with different metamodels. For example the LCSH are available both as XML conforming to the *Metadata Authority Description Standard (MADS)* schema, and as RDF using the *Simple Knowledge Organization System (SKOS)* vocabulary.

Specifying a vocabulary is just one way models can control what values can be assigned to properties. Another strategy is to specify what types of values can be assigned. For example, a model for book descriptions may specify that the value of a *pages* property must be a positive integer. Or it could be more specific; a course catalog might give each course an identifier that contains a two-letter department code followed by a 1-3 digit course number. Specifying a data type like this with a *regular expression* narrows down the set of possible values...
for the property without having to enumerate every possible value. (See the sidebar.)

In addition to or in lieu of specifying a type, a model may specify an encoding scheme for values. An encoding scheme is a specialized writing system or syntax for particular types of values. For example, a model like Atom for describing syndicated web content requires a publication date. But there are many different ways to write dates: 9/2/76, 2 Sept. 1976, September 2nd 1976, etc. Atom also specifies an encoding scheme for date values. The encoding scheme is RFC3339, a standard for writing dates. When using RFC3339, one always writes a date using the same form: 1976-09-02.

Encoding schemes are often defined in conjunction with standardized identifiers. (See §4.4.3.1 Make Names Informative (page 164).) For example, International Standard Book Numbers (ISBN) are not just sequences of Arabic numerals: they are values written using the ISBN encoding scheme. This scheme specifies how to separate the sequence of numerals into parts, and how each of these parts should be interpreted. The ISBN 978-3-8218-4448-0 has five parts, the first three of which indicate that the resource with this identifier is 1) a product of the book publishing industry, 2) published in a German-speaking country, and 3) published by the publishing house Eichborn.

Encoding schemes can be viewed as very specialized models of particular kinds of information, such as dates or book identifiers. But because they specify not only the structure of this information, but also how it should be written, we can also view them as specialized writing systems. That is, encoding schemes specify how to textually represent information.

In the second half of this chapter, we will focus on the issues involved in textually representing resource descriptions—writing them. Graphs, trees, dictionaries, lists, and sets are general types of structures found in different metamodels. Thinking about these broad types and how they fit or do not fit the ways we want to model our resource descriptions can help us select a specific metamodel. Specific metamodels such as the XML Infoset or RDF are formalized and standardized definitions of the more general types of structures discussed above. Once we have selected a metamodel, we know the constraints we have to work with when modeling the resources and collections in our specific domain. But because metamodels are abstract and exist only on a conceptual level, they can only take us so far. If we want to create, store, and exchange individual resource descriptions, we need to make the structures defined by our abstract metamodels concrete. We need to write them.
9.3 Writing Descriptions

Suppose that I am organizing books, and I have decided that it is important for the purposes of this organizing to know the title of each book and how many pages it has. Before me I have a book, which I examine to determine that its title is *Die Ringe des Saturn* and it has 371 pages. **Example 9.5, Basic ways of writing part of a book description.** lists a few of the ways to write this description. Let us examine these various forms of writing to see what they have in common and where they differ.

**Example 9.5. Basic ways of writing part of a book description.**

<table>
<thead>
<tr>
<th>The title is <em>Die Ringe des Saturn</em> and it has 371 pages.</th>
</tr>
</thead>
<tbody>
<tr>
<td>{ book: {&quot;title&quot;:&quot;Die Ringe des Saturn&quot;,&quot;pages&quot;:371} }</td>
</tr>
<tr>
<td>&lt;div class=&quot;book&quot;&gt; The title is &lt;span class=&quot;title&quot;&gt;Die Ringe des Saturn&lt;/span&gt; and it has &lt;span class=&quot;pages&quot;&gt;371 pages.&lt;/span&gt; &lt;/div&gt;</td>
</tr>
</tbody>
</table>

We examine the notations, writing systems and syntax of each of these description forms, and others, in the following sections.

9.3.1 Notations

First, let us look at the actual marks on the page. To write you must make marks or—more likely—select from a menu of marks using a keyboard. In either case, you are using a **notation**: a set of characters with distinct forms. The Latin alphabet is a notation, as are Arabic numerals. Some more exotic notations include the symbols used for editorial markup and alchemical symbols. The characters in a notation usually have an ordering. Arabic numerals are ordered 1 2 3 and so on. English-speaking children usually learn the ordering of the Latin alphabet in the form of an alphabet song.

A character may belong to more than one notation. The examples in **Example 9.5, Basic ways of writing part of a book description.** use characters from a few different notations: the letters of the Latin alphabet, Arabic numerals, and a handful of auxiliary marks: . { } " :< > / $ Collectively, all of these characters—alphabet, numerals, and auxiliary marks—also belong to a notation called the American Standard Code for Information Interchange (ASCII).
ASCII is an example of a notation that has been codified and standardized for use in a digital environment. A traditional notation like the Latin alphabet can withstand a certain degree of variation in the form of a particular mark. Two people might write the letter A rather differently, but as long as they can mutually recognize each other’s marks as an “A,” they can successfully share a notation. Computers, however, cannot easily accommodate such variation. Each character must be strictly defined. In the case of ASCII, each character is given a number from 0 to 127, so that there are 128 ASCII characters. When using a computer to type ASCII characters, each key you press selects a character from this “menu” of 128 characters. A notation that has had numbers assigned to its characters is called a character encoding.

Table 9.1. ASCII

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NUL</td>
<td>DLE</td>
<td>space</td>
<td>0</td>
<td>@</td>
<td>P</td>
<td>`</td>
<td>p</td>
</tr>
<tr>
<td>1</td>
<td>SOH</td>
<td>DC1</td>
<td>!</td>
<td>1</td>
<td>A</td>
<td>Q</td>
<td>a</td>
<td>q</td>
</tr>
<tr>
<td>2</td>
<td>STX</td>
<td>DC2</td>
<td>&quot;</td>
<td>2</td>
<td>B</td>
<td>R</td>
<td>b</td>
<td>r</td>
</tr>
<tr>
<td>3</td>
<td>ETX</td>
<td>DC3</td>
<td>#</td>
<td>3</td>
<td>C</td>
<td>S</td>
<td>c</td>
<td>s</td>
</tr>
<tr>
<td>4</td>
<td>EOT</td>
<td>DC4</td>
<td>$</td>
<td>4</td>
<td>D</td>
<td>T</td>
<td>d</td>
<td>t</td>
</tr>
<tr>
<td>5</td>
<td>ENQ</td>
<td>NAK</td>
<td>%</td>
<td>5</td>
<td>E</td>
<td>U</td>
<td>e</td>
<td>u</td>
</tr>
<tr>
<td>6</td>
<td>ACK</td>
<td>SYN</td>
<td>&amp;</td>
<td>6</td>
<td>F</td>
<td>V</td>
<td>f</td>
<td>v</td>
</tr>
<tr>
<td>7</td>
<td>BEL</td>
<td>ETB</td>
<td>‘</td>
<td>7</td>
<td>G</td>
<td>W</td>
<td>g</td>
<td>w</td>
</tr>
<tr>
<td>8</td>
<td>BS</td>
<td>CAN</td>
<td>(</td>
<td>8</td>
<td>H</td>
<td>X</td>
<td>h</td>
<td>x</td>
</tr>
<tr>
<td>9</td>
<td>HT</td>
<td>EM</td>
<td>)</td>
<td>9</td>
<td>I</td>
<td>Y</td>
<td>i</td>
<td>y</td>
</tr>
<tr>
<td>A</td>
<td>LF</td>
<td>SUB</td>
<td>*</td>
<td>:</td>
<td>J</td>
<td>Z</td>
<td>j</td>
<td>z</td>
</tr>
<tr>
<td>B</td>
<td>VT</td>
<td>ESC</td>
<td>+</td>
<td>;</td>
<td>K</td>
<td>]</td>
<td>k</td>
<td>{</td>
</tr>
<tr>
<td>C</td>
<td>FF</td>
<td>FS</td>
<td>,</td>
<td>&lt;</td>
<td>L</td>
<td>\</td>
<td>l</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>CR</td>
<td>GS</td>
<td>-</td>
<td>=</td>
<td>M</td>
<td>}</td>
<td>m</td>
<td>}</td>
</tr>
<tr>
<td>E</td>
<td>SO</td>
<td>RS</td>
<td>.</td>
<td>&gt;</td>
<td>N</td>
<td>^</td>
<td>n</td>
<td>~</td>
</tr>
<tr>
<td>F</td>
<td>SI</td>
<td>US</td>
<td>/</td>
<td>?</td>
<td>O</td>
<td>_</td>
<td>o</td>
<td>DEL</td>
</tr>
</tbody>
</table>

The most ambitious character coding in existence is Unicode, which as of version 6.0 assigns numbers to 109,449 characters. Unicode makes the important distinction between characters and glyphs. A character is the smallest meaningful unit of a written language. In alphabet-based languages like English, characters are letters; in languages like Chinese, characters are ideographs. Unicode treats all of these characters as abstract ideas (Latin capital A) rather than specific marks (A A A A). A specific mark that can be used to depict a character is a glyph. A font is a collection of glyphs used to depict some set of characters. A Unicode font explicitly associates each glyph with a particular number in the
Unicode character encoding. The inability of computers to use contextual understanding to bridge the gap between various glyphs and the abstract character depicted by those glyphs turns out to have important consequences for organizing systems.

Different notations may include very similar marks. For example, modern music notation includes marks for indicating the pitch of note, known as accidentals. One of these music notation marks is ♯ (“sharp”). The sharp sign looks very much like the symbol used in English as an abbreviation for the word number, as in We’re #1! If you were to write a sharp sign and a number sign by hand, they would probably look identical. In a non-digital environment, we would rely on context to understand whether the written mark was being used as part of music notation, or mathematical notation, or as an English abbreviation.

Computers, however, have no such intuitive understanding of context. Unicode encodes the number sign and the sharp sign as two different characters. As far as a computer using Unicode is concerned, # and ♯ are completely different, and the fact that they have similar-looking glyphs is irrelevant. That is a problem if, for example, a cataloger has carefully described a piece of music by correctly using the sharp sign, but a person looking for that piece of music searches for descriptions using the number sign (since that is what you get when you press the keyboard button with the symbol that most closely resembles a sharp sign).

9.3.2 Writing Systems

A writing system employs one or more notations, and adds a set of rules for using them. Most writing systems assume knowledge of a particular human language. These writing systems are known as glottic writing systems. But there are many writing systems, such as mathematical and musical ones, that are not tied to human languages in this way. Many of the writing systems used for describing resources belong to this latter group, meaning that (at least in principle) they can be used with equal facility by speakers of any language.

Some writing systems are closely identified with specific metamodels. For example, XML and JSON are both 1) metamodels for structuring information and 2) writing systems for textually representing information. In other words, they specify both the abstract structure of a description and how to write it down. It is possible to conceive of other ways to textually represent the structure of these metamodels, but for each of these metamodels just one writing system has been standardized.

RDF, on the other hand, is only a metamodel, not a writing system. RDF only defines an abstract structure, not how to write that structure. So how do we write information that is structured as RDF? It turns out that we have many choices. Unlike XML and JSON, several different writing systems for the RDF
metamodel have been standardized, including N-Triples, Turtle, RDFa, and RDF/XML. Each of these is a writing system that is abstractly described by the RDF metamodel.

Writing systems provide rules for arranging characters from a notation into meaningful structures. A character in a notation has no inherent meaning. Characters in a notation only take on meaning in the context of a writing system that uses that notation. For example: what does the letter I from the Latin alphabet mean? That question can only be answered by looking at how it is being used in a particular writing system. If the writing system is American English, then whether I has a meaning depends on whether it is grouped with other letters or whether it stands alone. Only in the latter case does it have an assignable meaning. However in the arithmetic writing system of ancient Rome, which also uses as a notation the letters of the Latin alphabet, I has a different meaning: one.

This example also serves to illustrate how the ordering of a notation can differ from the ordering of a writing system that uses that notation. According to the ordering of the Latin alphabet, the twelfth letter L comes before the twenty-second letter V. But in the Roman numeric writing system, V (the number 5) comes before L (the number 50). Unless we know which ordering we are using, we cannot arrange L and V “in order.”

**Table 9.2. Roman Numerals**

<table>
<thead>
<tr>
<th>Roman Number</th>
<th>Arabic Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>1</td>
</tr>
<tr>
<td>V</td>
<td>5</td>
</tr>
<tr>
<td>X</td>
<td>10</td>
</tr>
<tr>
<td>L</td>
<td>50</td>
</tr>
<tr>
<td>C</td>
<td>100</td>
</tr>
<tr>
<td>D</td>
<td>500</td>
</tr>
<tr>
<td>M</td>
<td>1000</td>
</tr>
</tbody>
</table>

This kind of difference in ordering can arise in more subtle ways as well. When we alphabetically order names, we first compare the first character of each name, and arrange them according to the ordering of the writing system. The first known use of alphabetical ordering was in the Library of Alexandria about two thousand years ago, when Zenodotus arranged the collection according to the first letter of resource names. If the first characters of two names are the same, we compare the second character, and so on. We can also apply this same kind of ordering procedure to sequences of numerals. If we do, then 334 will come before 67, because 3 (the first character of the first sequence) comes before 6 (the first character of the second sequence) according to the ordering of our notation (Arabic numerals). However, it is more common when ordering se-
quences of numerals to treat them as decimal numbers, and thus to use the ordering imposed by the decimal system. In the decimal writing system, 67 precedes 334, since the latter is a greater number.

This difference is important for organizing systems. Computers will sort values differently depending on whether they are treating sequences of numerals as numbers or just as sequences. Some organizing systems mix multiple ways of ordering the same characters. For example, Library of Congress call numbers have four parts, and sequences of Arabic numerals can appear in three of them. In the second part, indicating a narrow subject area, and fourth part, indicating year of publication, sequences of numerals are treated as numbers and ordered according to the decimal system. In the third part, however, sequences of numerals are treated as sequences and ordered “notationally” as in the example above (334 before 67).

Differences in ordering demonstrate just one way that multiple writing systems may use the same notation differently. For example, the American English and British English writing systems both use the same Latin alphabet, but impose slightly different spelling rules. The Japanese writing system employs a number of notations, including traditional Chinese characters (kanji) as well as the Latin alphabet (rōmaji). Often, writing systems do not share the same exact notation but have mostly overlapping notations. Many European languages, for example, extend the Latin alphabet with characters such as Å and Ü that add additional marks, known as diacritics, to the basic characters.

In organizing systems it is often necessary to represent values from one writing system in another writing system that uses a different notation, a process known as transliteration. For example, early computer systems only supported the ASCII notation, so text from writing systems that extend the Latin alphabet had to be converted to ASCII, usually by removing (or sometimes transliterating) diacritics. This made the non-ASCII text usable in an ASCII-based computerized organizing system, at the expense of information loss.

Even in modern computer systems that support Unicode, however, transliteration is often needed to support organizing activities by users who cannot read text written using its original system. The Library of Congress and the American Library Association provide standard procedures for transliterating text from over sixty different writing systems into the (extended) Latin alphabet.
9.3.3 Syntax

The examples in Example 9.5, Basic ways of writing part of a book description, express the same information using different writing systems. The examples use the same notation (ASCII) but differ in their syntax: the rules that define how characters can be combined into words and how words can be combined into higher-level structures.

• Consider the first entry: The title is Die Ringe des Saturn and it has 371 pages. The leading capital letter and the period ending this sequence of characters indicate to us that this is a sentence. This sentence is one way we might use the English writing system to express two statements about the book we are describing. A statement is one distinct fact or piece of information. In glottic writing systems like English, there is usually more than one sentence we could write to express the same statement. For example, instead of it has 371 pages we might have written the number of pages is 371. English writing also enables us to construct complex sentences that express more than one statement.

In contrast, when we create descriptions of resources in an organizing system, we generally use non-glottic writing systems in which each sentence only expresses a single statement, and there is just one way to write a sentence that expresses a given statement. These restrictions make these writing systems less expressive, but simplify their use. In particular, since there is a one-to-one correspondence between sentences and statements, we can drop the distinction and just talk about the statements of a description.

Now we return to our example and look at the structure of the statement, The title is Die Ringe des Saturn and it has 371 pages. Spaces are used to separate the text into words, and English syntax defines the functions of those words. The verb is in this statement functions to link the word title to the phrase Die Ringe des Saturn. This is typical of the kind of statements found in a resource description. Each statement identifies and describes some aspect of the resource. In this case, the statement attributes the value Die Ringe des Saturn to the property title.

As we saw when we looked at description structures, we can analyze descriptions as involving properties of resources and their corresponding values or content. In a writing system like English, it is not always so straightforward to determine which words refer to properties and which refer to values. (This is why blobs are not ideal description structures.) Writing systems designed for expressing resource descriptions, on the other hand, usually define syntax that makes this determination easier. In our dictionary examples above, we used an arrow character → to indicate the relationship between properties and values.
This ease of distinguishing properties and values comes at a price, however. The syntax of English is forgiving: we can read a sentence with somewhat garbled syntax such as 371 pages it has and often still make out its meaning. This is usually not the case with writing systems intended for expressing resource descriptions. These systems strictly define their rules for how characters can be combined into higher-level structures. Structures that follow the rules are well formed according to that system.

- Take for example the second entry in Example 9.5, Basic ways of writing part of a book description.

  { book: {"title":"Die Ringe des Saturn","pages":371} }

This fragment is written in JSON. As explained earlier in this chapter, JSON is a metamodel for structuring information using lists and dictionaries. But JSON is also a writing system, which borrows its syntax from JavaScript. The JSON syntax uses brackets to textually represent lists [1,2,3] and braces to textually represent dictionaries {title:"Die Ringe des Saturn", "pages":371}. Within braces, the colon character : is used to link properties with their values, much as is was used in the previous example. So "pages":371 is a statement assigning the value 371 to the property pages.

- The third fragment is written in XML.


Like JSON, XML is a metamodel and also a writing system. Here we have XML elements and attributes. XML elements are textually represented as tags that are marked using the special characters <, > and /. So, this fragment of XML consists of a book element with a child element, title, and a pages attribute, each of which has some text content. In this case, pages="371" is a statement assigning the value 371 to the property pages. The difference is syntax is subtle; quotation marks surround the value and equal sign = is used to assign the property to its value.

- The fourth is a fragment of HTML.

  <div class="book">The title is <span class="title">Die Ringe des Saturn</span> and it has <span class="pages">371 pages</span>.</div>

The writing system that HTML employs is close enough to XML to ignore any differences in syntax. In this example, the CLASS attribute contains the property name and the property value is the element content.

- The fifth entry is a fragment of Turtle, one of the writing systems for RDF.

Turtle provides a syntax for writing down RDF **triples**. Each triple consists of a subject, predicate, and object separated by spaces. Recall that RDF uses URIs to identify subjects, predicates, and some objects; these URIs are written in Turtle by enclosing them in angle brackets `< >`. Triples are separated by period `. ` characters, but triples that share the same subject can be written more compactly by writing the subject only once, and then writing the predicate and object of each triple, separated by a semicolon `; ` character. This is what we see in Example 9.2, Nesting an author description within a book description: two triples that share a subject.

The two fragments in Example 9.6, Writing part of a book description in Semantic XML. demonstrate namespaces, terms from the Dublin Core and DocBook namespaces, and the facility with which XML embraces semantic encoding of description resources.

**Example 9.6. Writing part of a book description in Semantic XML.**

```
      dc:title="Die Ringe des Saturn"/>
...  
</book>

      <bookinfo>
      <title="Die Ringe des Saturn"></title>
      <pagenums>371 p.</pagenums>...
</bookinfo>
...  
</book>
```

- The first example extends the third fragment from Example 9.5, Basic ways of writing part of a book description.; the xmlns:dc="..." segment is a namespace declaration, which is associating dc with the quoted URI, which happens to be the Dublin Core Metadata Initiative (DCMI); the child `<dc:title>` element and the attached dc:extent="371" tell us that the corresponding values are attributable to the title and extent properties, respectively, from the Dublin Core namespace.

- The next fragment employs DocBook DTD namespace; we now have a `<pagenums>` element for which the meaning is contextually obvious; the title is still a title; an extra layer of markup reflects the fact that it could be metadata in the source file of a book that is being edited, is in production or is on your favorite tablet right now.

The two fragments in Example 9.7, Writing part of a book description in RDFa or microdata. demonstrate RDFa and microdata formats, which each rely upon specific attributes to establish the type of the property values contained by the
HTML elements. In each example, the book title is contained by a <span> element. Whereas RDFa relies upon the property attribute, the microdata example employs the itemprop attribute to specify that the contents of the element is, effectively, a “title” in exactly the same sense as we know that the contents of dc:title is a “title.”

**Example 9.7. Writing part of a book description in RDFa or microdata.**

```html

```

9.4 Worlds of Description

In the previous two sections we have considered descriptions as designed objects with particular structures and as written documents with particular syntaxes. As we have seen, there are many possible choices of structure and syntax. But these choices are never made in isolation. Just as an architect or designer must work within the constraints of the existing built environment, and just as any author must work with existing writing systems, descriptions are always created as part of a pre-existing “world” over which any one of us has little control.

In the final part of this chapter, we will consider how choices of structure and syntax have converged historically into broad patterns of usage. For lack of a better term, we call these broad patterns “worlds.” “World” is not a technical term and should not be taken too literally: the broad areas of application sketched here have considerable overlap, and there are many other ways one might identify patterns of description structure and syntax. That said, the three worlds described here do reflect real patterns of description form that influence tool and technology choices. In your own work creating and managing resource descriptions, it is likely that you will need to think about how your descriptions fit into one or more of these worlds.

9.4.1 The Document Processing World

The first world we will consider is concerned primarily with the creation, processing and management of hybrid narrative-transactional documents such as instruction manuals, textbooks, or annotated medieval manuscripts. (See The Document Type Spectrum (page 138)). These are quite different kinds of documents, but they all contain a mixture of narrative text and structured data, and
they all can be usefully modeled as tree structures. Because of these shared qualities, tools as different as publishing software, supply-chain management software, and scholarly editing software have all converged on common XML-based solutions. (“The XML world” would be another appropriate name for the document-processing world.)

This convergence was no accident, because XML was designed specifically to address the problem of how to add structure and data to documents by “marking them up.” XML is the descendant of Standard Generalized Markup Language (SGML), which in turn descended from International Business Machines (IBM)’s Generalized Markup Language, which was invented to enable the production and management of large-scale technical documentation. The explicitness of markup makes it well-suited for representing structure and content type distinctions in institutional contexts, where the scope, scale, and expected lifetime of organizing systems for information implies reuse by unknown people for unanticipated purposes.

As one might expect, tools and technologies in the document-processing world are optimized for manipulating and combining tree structures. A “toolchain” is set of tools intended to be used together to achieve some goal.

For programmers who do not use the XML toolchain, other programming languages also provide libraries for working with XML. This fact has led some to propose, and others to believe, that XML is a kind of universal format for exchanging data among systems. However, programmers have observed that a random XML Infoset does not map easily to the data structures commonly found in many programming languages. “Working with XML” frequently means translating from XML tree structures to data structures native to another language, usually meaning lists and dictionaries. This translation can be problematic and often means giving up many of the strengths of XML. By the same token, there are decades more practical experience working with markup languages and institutional publishing than there is with JSON and RDF.

XML is not a universal solution for every possible problem. That does not mean that it is not the best solution for a wide variety of problems, including yours. To gauge whether your resource descriptions are, or ought to be, part of the document-processing world, ask yourself the following questions:

- Do my resource descriptions contain mixtures of narrative text, hypertext, structured data and a variety of media formats?
- Can my descriptions easily be modeled using tree structures, hypertext links, and transclusion?
- Are the vocabularies I need or want to use made available using XML technologies?
• Do I need to work with a body of existing descriptions already encoded as XML?
• Do I need to interoperate with processes or partners that utilize the XML toolchain?
• Do I need to publish my resource descriptions in multiple formats from a single source?

If the answer to one or more of these questions is “yes,” then chances are good that you are working within the document processing world, and you will need to become familiar with conceptualizing your descriptions as trees and working with them using XML tools.

9.4.2 The Web World

The second “world” emerged in the early 1990s with the creation of the World Wide Web. The web was developed to address a need for simple and rapid sharing of scientific data. Of course, it has grown far beyond that initial use case, and is now a ubiquitous infrastructure for all varieties of information and communication services. (“The browser world” would be another appropriate name for what we are calling the Web World.)

Documents, data, and services on the web are conceptualized as resources, identified using Uniform Resource Identifiers (URI), and accessible through representations transferred via Hypertext Transfer Protocol (HTTP). Representations are sequences of bytes, and could be HTML pages, JPEG images, tabular data, or practically anything else transferable via HTTP. No matter what they are, representations transferred over the web include descriptions of themselves. These descriptions take the form of property-value pairs, known as “HTTP headers.” The HTTP headers of web representations are structured as dictionaries.

Dictionary structures appear many other places in web infrastructure. URIs may include a query component beginning with a ? character. This component is used for purposes such as providing query parameters to search services. The query component is commonly structured as a dictionary, consisting of a series of property-value pairs separated by the & character. For example, the following URI:

`https://www.google.com/search?q=sebald&tbs=qdr:m`

includes the query component q=sebald&tbs=qdr:m. This is a dictionary with the properties q and tbs, respectively specifying the search term and temporal constraints on the search.

Dictionary structures are easy to work with in any programming language, and they pervade various popular frameworks for programming the Web. In the pro-
gramming languages used to implement web services, HTTP headers and query parameters are easily mapped to dictionary data structures native to those languages. On the client side, there is only one programming language that runs within all web browsers: JavaScript. The dictionary is the fundamental data structure within JavaScript as well.

Thus it is unsurprising that JSON, a dictionary-structured, JavaScript-based syntax, has become the de facto standard for application-to-application interchange of data on the web in contexts that do not involve business transactions. Web services providing structured data intended for programmatic use can make that data available as JSON, which is well-suited for use either by JavaScript programs running within browsers, or by programs written in other languages running outside of browsers (e.g., smart phone applications).

It is now commonly accepted that there are useful differences of approach between the document-processing world and the Web World. This does not mean that the two worlds do not have significant overlaps. Some very important web representation types are XML-based, such as the Atom syndication format. Trees will continue to be the structure of choice for web representations that consist primarily of narrative rather than transactional data. But for structured descriptions that are intended to be accessed and manipulated on the Web, dictionary structures currently rule.

To gauge whether your resource descriptions are or ought to be part of the Web world, ask yourself the following questions:

- Is the web the primary platform upon which I will be making my descriptions available?
- Are my resource descriptions primarily structured, transaction-oriented data?
- Can my descriptions easily be modeled as lists of properties and values (dictionaries)?
- Are the vocabularies I need or want to use made available primarily using HTML technologies such as microdata or microformats?
- Do I need to make my descriptions easily usable for use within a wide array of programming languages?

If the answer to one or more of these questions is “yes,” then chances are good that you are working within the Web World, and you will need to become familiar with conceptualizing your descriptions as dictionaries and working with them using programming languages such as JavaScript.
9.4.3 The Semantic Web World

The last world we consider is still somewhat of a possible world, at least in comparison with the previous two. While the document processing world and the web world are well-established, the Semantic Web world is only starting to emerge, despite having been envisioned over a decade ago.

The vision of a Semantic Web world builds upon the web world, but adds some further prescriptions and constraints for how to structure descriptions. The Semantic Web world unifies the concept of a resource as it has been developed in this book, with the web notion of a resource as anything with a URI. On the Semantic Web, anything being described must have a URI. Furthermore, the descriptions must be structured as graphs, adhering to the RDF metamodel and relating resources to one another via their URIs. Advocates of Linked Data further prescribe that those descriptions must be made available as representations transferred over HTTP.

This is a departure from the web world. The web world is also structured around URIs, but it does not require that every resource being described have a URI. For example, in the web world a list of bibliographic descriptions of books by W.G. Sebald might be published at a specific URI, but the individual books themselves might not have URIs. In the Semantic Web world, in addition to the list having a URIs, each book would have a URI too, in addition to whatever other identifiers it might have.

Making an HTTP request to an individual book URI may return a graph-structured description of that book, if best practices for Linked Data are being followed. This, too, is a departure from the web world, which is agnostic about the form representations or descriptions of resources should take (although as we have seen, dictionary structures are often favored on the web when the clients consuming those descriptions are computer programs). On the Semantic Web, all descriptions are structured as RDF graphs. Each description graph links to other description graphs by referring to these related resources using their URIs. Thus, at least in theory, all description graphs on the Semantic Web are linked into a single massive graph structure. In practice, however, it is far from clear that this is an achievable, or even a desirable, goal.

Although the Semantic Web is in its infancy, a significant number of resource descriptions have already been made available in accordance with the principles outlined above. Descriptions published according to these principles are often referred to as “Linked Data.” Prominent examples include: DBpedia, a graph of descriptions of subjects of Wikipedia articles; the Virtual International Authority File (VIAF), a graph of descriptions of names collected from various national libraries’ name authority files; GeoNames, a graph of descriptions of places; and Data.gov.uk, a graph of descriptions of public data made available by the UK government.
Despite the growing amount of Linked Data, tools for working with graph-structured data are still immature in comparison to the XML toolchain and Web programming languages. Although there is an XML syntax for RDF, using the XML toolchain to work with graph-structured data is generally a bad idea. And just as most programming languages do not support natively working with tree structures, most do not support natively working with graph structures either. Storing and querying graph-structured data efficiently requires a graph database or triple store.

Still, the Semantic Web world has much to recommend it. Having a common way of identifying resources (the URI) and a single shared metamodel (RDF) for all resource descriptions makes it much easier to combine descriptions from different sources. To gauge whether your resource descriptions are or ought to be part of the Semantic Web world, ask yourself the following questions:

- Is the web the primary platform upon which I will be making my descriptions available?
- Is it important that I be able to easily and freely aggregate the elements of my descriptions in different ways and to combine them with descriptions created by others?
- Are my descriptions best modeled as graph structures?
- Have the vocabularies I need or want to use been created using RDF?
- Do I need to work with a body of existing descriptions that have been published as Linked Data?

If the answer to one or more of these questions is “yes,” then chances are good that you should be working within the Semantic Web world, and you ought to become familiar with conceptualizing your descriptions as graphs and working with them using Semantic Web tools.

### 9.5 Key Points in Chapter Nine

- We can approach the problem of how to form resource descriptions from two perspectives: structuring and writing.
  (See §9.1 Introduction (page 357))
- Blobs, sets, lists, dictionaries, trees, and graphs are all kinds of structures that can be used to form resource descriptions.
  (See §9.2.1 Kinds of Structures (page 362))
- A list, like a set, is a collection of items with an additional constraint: their items are ordered.
  (See §9.2.1.3 Lists (page 363))
A *dictionary*, also known as a *map* or an *associative array*, is a set of property-value pairs or *entries*.
(See §9.2.1.4 Dictionaries (page 364))

Nested dictionaries form a *tree*.
(See §9.2.1.4 Dictionaries (page 364))

Trees consist of *nodes* joined by *edges*.
(See §9.2.1.5 Trees (page 365))

The XML Infoset is a tree structure, where each node of the tree is defined to be an *information item* of a particular type.
(See §9.2.2.2 XML Information Set (page 371))

Using schemas to define data representation formats is a good practice that facilitates shared understanding and contributes to long-term maintainability.
(See §9.2.2.2 XML Information Set (page 371))

An “encoding scheme” is a specialized writing system or syntax for particular types of values. Encoding schemes specify how to *textually represent* information.
(See §9.3.1 Notations (page 380))

A *writing system* employs notations, and adds a set of rules for using them.
(See §9.3.2 Writing Systems (page 382))

Differences in ordering demonstrate just one way that multiple writing systems may use the same notation differently.
(See §9.3.2 Writing Systems (page 382))

Syntax is the rules that define how characters can be combined into words and how words can be combined into higher-level structures.
(See §9.3.3 Syntax (page 385))

The document processing world is concerned primarily with the creation, processing and management of hybrid narrative-transactional documents.
(See §9.4.1 The Document Processing World (page 388))

The Semantic Web world unifies the concept of a resource as it has been developed in this book, with the web notion of a resource as anything with a URI. Descriptions must be structured as graphs, adhering to the RDF metamodel and relating *resources* to one another via their URIs.
(See §9.4.3 The Semantic Web World (page 392))