9 Frames and Inheritance

In this chapter, you learn about *frames*, *slots*, and *slot values*, and you learn about *inheritance*, a powerful problem-solving method that makes it possible to know a great deal about the slot values in *instances* by virtue of knowing about the slot values in the *classes* to which the instances belong.

With basic frame-representation ideas in hand, you learn that frames can capture a great deal of commonsense knowledge, informing you not only about what assumptions to make, but also about for what information to look and how to look for that information. You learn that much of this knowledge is often embedded in *when-constructed procedures*, *when-requested procedures*, *when-read procedures*, *when-written procedures*, and *with-respect-to procedures*.

By way of illustration, you see how to use frames to capture the general properties of various kinds of dwarfs, and you see how to use frames to capture the properties of various kinds of newspaper stories.

Once you have finished this chapter, you will understand that frames can capture a great deal of commonsense knowledge, including knowledge about various sorts of objects ranging from individuals to events. You will also know how the CLOS inheritance procedure determines a precedence ordering among multiple classes.

FRAMES, INDIVIDUALS, AND INHERITANCE

In this section, you learn about frames and their relation to semantic nets. In particular, you learn how to capture general knowledge that holds for

most of the individuals in a class. This capability enables you to make use of the following general knowledge about fairy-tale dwarfs:

- Fairy-tale competitors and gourmands are fairy-tale dwarfs.
- Most fairy-tale dwarfs are fat.
- Most fairy-tale dwarfs' appetites are small.
- Most fairy-tale gourmands' appetites are huge.
- Most fairy-tale competitors are thin.

You also learn the details of one particularly good mechanism for deciding which general knowledge about classes to transfer to individuals.

Frames Contain Slots and Slot Values

At this point, it is convenient to introduce a few terms that make it easier to think about semantic nets at a level slightly higher than the lowest level, where there are just nodes and links.

As shown in figure 9.1, each node and the links that emanate from it can be collected together and called a **frame**. Graphically, frames may be shown in an alternate, rectangle-and-slot notation. Each frame's name is the same as the name of the node on which the frame is based. The names attached to the slots are the names of the links emanating from that frame's node. Accordingly, you can talk about a **slot**, rather than about a link that emanates from a node. Similarly, you can talk about **slot values** rather than about the destinations of links emanating from a node. Thus, the language of frames, slots, and slot values is sometimes more concise, and hence clearer, than is the language of nodes and links, although both describe the same concepts.

Frames may Describe Instances or Classes

Many frames describe individual things, such as Grumpy, an individual dwarf. These frames are called **instance frames** or **instances**. Other frames describe entire classes, such as the dwarf class. These frames are called **class frames** or **classes**.

As soon as you know the class to which an instance belongs, you generally assume a lot. Unless you know you are dealing with an exception, you assume, for example, that dwarfs are fat.

A special slot, the Is-a slot, short for is-a-member-of-the-class, ties instances to the classes that they are members of. In figure 9.2, for example, a dwarf named Blimpy is identified as a member of the Managers class.

Another special slot, the Ako slot, short for a-kind-of, ties classes together. The Managers class is a subclass of the Competitors class, for example. The Managers class is also a **direct subclass** of the Competitors class because there is an Ako slot in the Managers class that is filled with the Competitors class. The Managers class is just a subclass of the Dwarfs class, however, because you have to traverse more than one Ako slot to get from the Managers class to the Dwarfs class. Symmetrically,

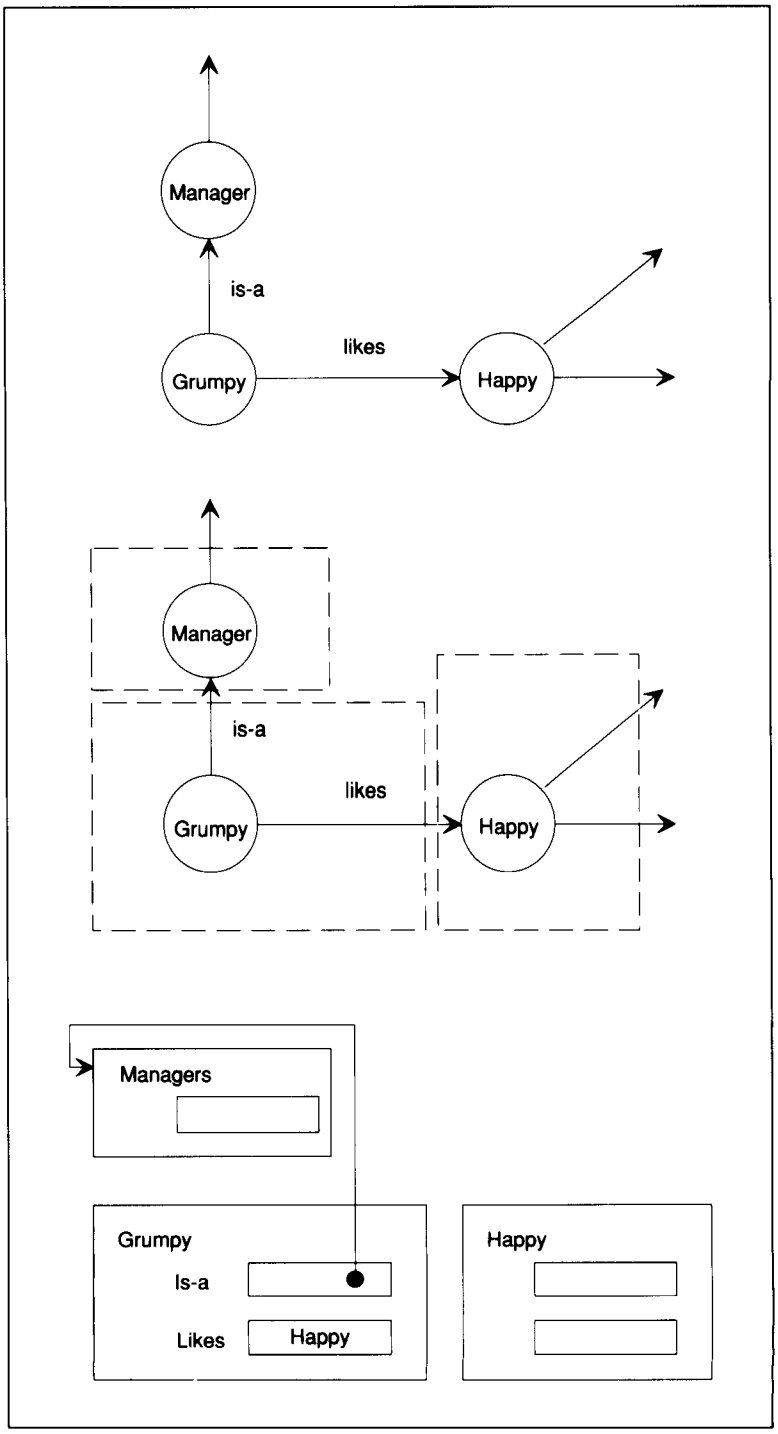

Figure 9.1 A semantic net can be viewed either as a collection of nodes and links or as a collection of frames. At the top, a semantic net is viewed as a collection of nodes and links. In the middle, the same semantic net is divided into chunks, each of which consists of a node and the links that emanate from it. Next, at the bottom, each chunk is shown as a frame with slots and slot values. As the Grumpy frame illustrates, slot values may be shown as frame names or as links connected to frames.

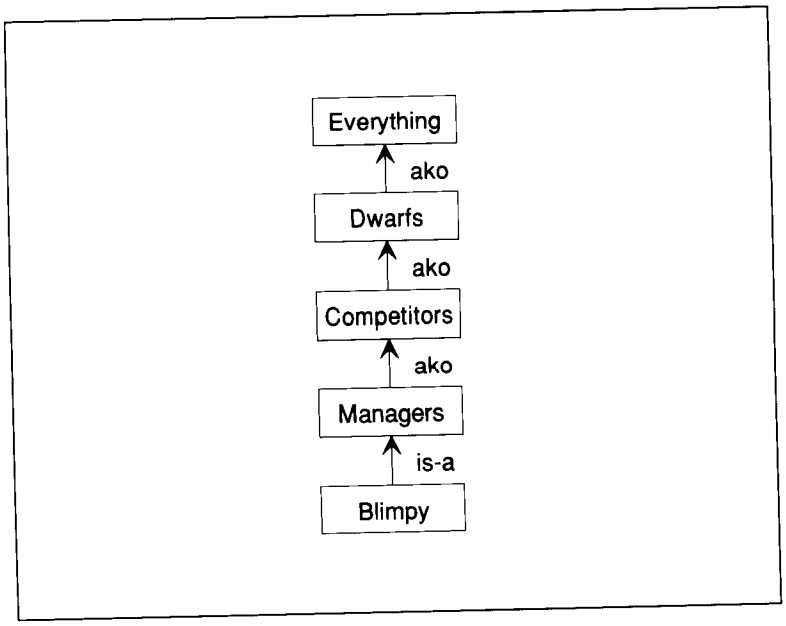

Figure 9.2 A simple class hierarchy. Blimpy is a member of the Managers class, which is a direct subclass of the Competitors class and a subclass of the Dwarfs class. Every class is considered to be, ultimately, a subclass of the Everything class.

the Competitors class is said to be a **direct superclass** of the Managers class, and the Dwarfs class is said to be a superclass of the Managers class.

Note that it is convenient to draw class hierarchies with Is-a and Ako links connecting frames that are actually connected via values in Is-a and Ako slots. Thus, the vocabulary of nodes and links is often mixed with the vocabulary of frames and slots.

Frames Have Access Procedures

To make and manipulate instances and classes, you need access procedures, just as you do for any representation. In figure 9.3, the **class constructor** makes a Manager frame that has one direct superclass, the Competitors class, which appears in the Ako slot. In general, the class constructor can make class frames that contain other slots and more than one direct superclass.

An **instance constructor** makes instance frames. Its input consists of the name of the class to which the instance belongs; its output is an instance that belongs to those classes. The new instance is connected automatically to the class frames via an Is-a slot in the new instance.

A **slot writer** installs slot values. Its input is a frame, the name of a slot, and a value to be installed. Finally, a **slot reader** retrieves slot values. Its input is a frame and the name of a slot; its output is the corresponding slot value.

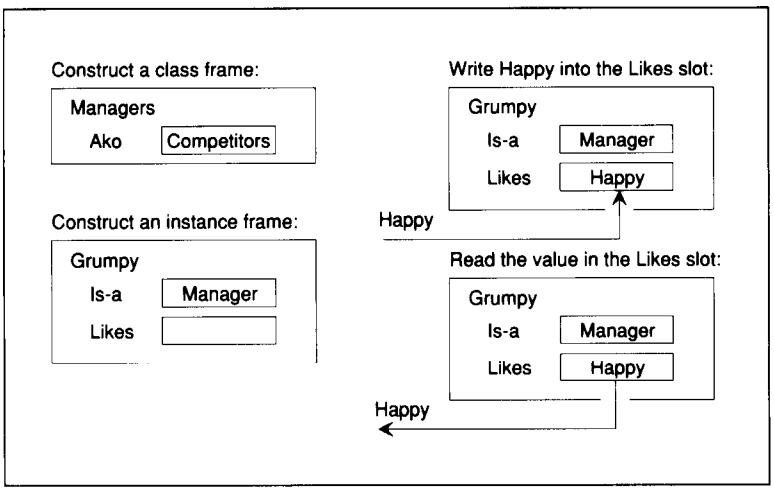

Figure 9.3 Instance frames and class frames are data types that are made and accessed with various constructors, writers, and readers.

Inheritance Enables When-Constructed Procedures to Move Default Slot Values from Classes to Instances

The slots in an instance are determined by that instance's superclasses. If a superclass has a slot, then the instance **inherits** that slot.

Sometimes, slot values are specified after an instance is constructed. After Blimpy is constructed, for example, you can indicate that Blimpy is smart by inserting the value Smart in Blimpy's Intelligence slot.

Alternatively, the slot values of an instance may be specified, somehow, by the classes of which the instance is a member. It might be, for example, that Dwarfs are fat in the absence of contrary information; also it might be that Competitors are thin, again in the absence of contrary information.

By writing down, in one place, the knowledge that generally holds for individuals of that class, you benefit from the following characteristics of shared, centrally located knowledge:

Shared knowledge, located centrally, is

- ▷ Easier to construct when you write it down
- ▷ Easier to correct when you make a mistake
- ▷ Easier to keep up to date as times change
- ▷ Easier to distribute because it can be distributed automatically

---

One way to accomplish knowledge sharing is to use **when-constructed procedures** associated with the classes of which the instance is a member. Here is an example that supplies a value for the physique slot of individual dwarfs:

To fill the Physique slot when a new Dwarf is constructed,

▷ Write Fat in the slot.

The expectations established by when-constructed procedures are called **defaults**.

In the simplest class hierarchies, no more than one when-constructed procedure supplies a default for any particular slot. Often, however, several when-constructed procedures, each specialized to a different class, supply default values for the same slot. Here, for example, a second when-constructed procedure provides a default value for the Physique slot of individual Competitors:

To fill the Physique slot when a new Competitor is constructed,

▷ Write Thin in the slot.

Whenever an individual is both a Competitor and Dwarf, both procedures compete to supply the default value. Of course, you could specify an inheritance procedure that allows multiple procedures to supply defaults, but the usual practice is to allow just one procedure.

How can you decide which when-constructed procedure is the winner? First, you learn about the special case in which no individual has more than one Is-a link and no class has more than one Ako link. Once this foundation is in place, you learn about more complicated hierarchies in which individuals and class have multiple inheritance links.

One way to decide which when-constructed procedure to use, albeit a way limited to single-link class hierarchies, is to think of classes themselves as places where procedures can be attached. One of the sample procedures, because it deals with new Dwarfs, is attached to the Dwarf class; the other is attached to the Competitors class. That way, you can find both by a search up from the new instance through Is-a links and Ako links.

Because each class in the class hierarchy in the example has only one exiting Ako link, it is easy to form an ordered list consisting of Blimpy and the classes to which Blimpy belongs. This ordered list is called the **class-precedence list**:

Blimpy
Managers class
Competitors class ← procedure stored here
Dwarfs class ← procedure stored here
Everything class

A procedure that is specialized to one of the classes on the class-precedence list is said to be **applicable**.

Suppose, for example, that you have just constructed Blimpy. You have Blimpy's class-precedence list, which supplies two procedures for computing values for the Physique slot. The procedure attached to the Competitor's class says that Blimpy is Thin and the procedure attached to the Dwarf class says that Blimpy is Fat. This kind of ambiguity is always resolved in favor of the most specific applicable procedure—the one that is encountered first on the class-precedence list. In the example, as shown by the class-precedence list, the first of the procedure-supplying classes encountered is the Competitors class, so the procedure attached there is the one that determines Blimpy's physique when Blimpy is constructed. Evidently, Blimpy is Thin.

A Class Should Appear Before All Its Superclasses

When there is more than one Is-a link above an instance or more than one Ako link above a class, then the class hierarchy is said to **branch**.† Because branching class hierarchies are more difficult to handle, yet are ubiquitous in intelligent systems, the rest of this section is devoted to explaining the issues involved, and to presenting a procedure that deals with those issues.

As an illustration, consider the class hierarchy shown in figure 9.4. Suppose that there are two procedures for computing Appetite:

To fill the Appetite slot when a new Dwarf is constructed,

▷ Write Small in the slot.

To fill the Appetite slot when a new Gourmand is constructed,

▷ Write Huge in the slot.

Because the class hierarchy branches, you must decide how to flatten the class hierarchy into an ordered class-precedence list.

One choice is to use depth-first search. Depth-first search makes sense because the standard convention is to assume that information from specific classes should override information from more general classes. Left-to-right search makes sense too, but only because you need some way to specify the priority of each direct superclass, and the standard convention is to specify

†Generally, the treatment of frames in this chapter follows the conventions of the Common Lisp Object System, also known as CLOS. However, in contrast to the conventions of CLOS, multiple Is-a connections are allowed—CLOS forbids them for the sake of efficient implementation. There is no loss of generality in CLOS, however, because an instance can be attached to a class that is wholly dedicated to that instance and that has multiple Ako connections to the desired superclasses.

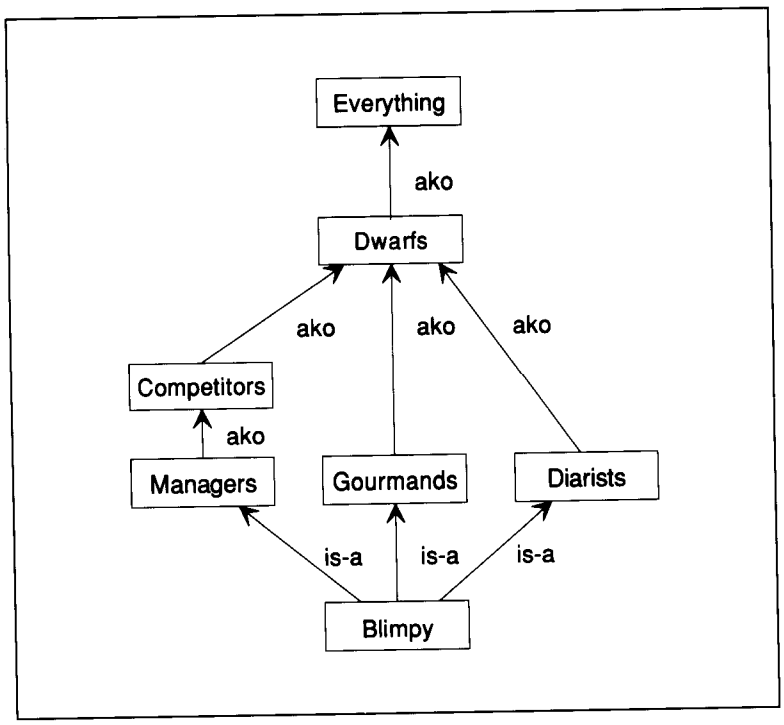

Figure 9.4 Another class hierarchy. Because Blimpy belongs to both the Gourmands class and to the Diarists class, as well as the Managers class, the class hierarchy branches upward. Because the Dwarfs class has three subclasses—Competitors, Gourmands and Diarists—the class hierarchy branches downward as well.

priority through the left-to-right superclass order provided to the class-constructor procedure.

Note, however, that you must modify depth-first search slightly, because you want to include all nodes exactly once on the class-precedence list. To perform **exhaustive depth-first search**, you explore all paths, depth first, until each path reaches either a leaf node or a previously-encountered node.

To search the class hierarchy shown in figure 9.4, using exhaustive depth-first search, you first follow the left branch at each node encountered; the resulting path includes Blimpy, Managers, Competitors, Dwarfs, and Everything. Then, you follow Blimpy's middle branch to Gourmands; the resulting path terminates at Gourmands, however, because you have already encountered the Dwarfs node. Finally, you follow Blimpy's right branch to Diarists, where you terminate the path.

Thus, exhaustive depth-first, left-to-right search produces the following class-precedence list for Blimpy:

```
Blimpy
Managers class
Competitors class
Dwarfs class          ← procedure stored here
Everything class
Gourmands class       ← procedure stored here
Diarists class
```

You can see that the first Appetite-computing when-constructed procedure encountered for Blimpy is the one attached to the Dwarfs class—the one that would indicate that Blimpy's appetite is small. This conclusion seems at odds with intuition, however, because the Gourmands class is a subclass of the Dwarfs class. Surely a class should supply more specific procedures than any of its superclasses. Rephrasing, you have a rule:

- Each class should appear on class-precedence lists before any of its superclasses.

To keep a class's superclasses from appearing before that class, you can modify depth-first, left-to-right search by adding the **up-to-join proviso**. The up-to-join proviso stipulates that any class that is encountered more than once during the depth-first, left-to-right search is ignored until that class is encountered for the last time.

Using this approach, the construction of Blimpy's class-precedence list proceeds as before until the Competitors class is added and the Dwarfs class is encountered. Because there are three paths from Blimpy to the Dwarfs class, the Dwarfs class is ignored the first and second times it is encountered. Consequently, the Gourmands class is the next one added to the class-precedence list, followed by the Diarists class. Then, the Dwarfs class is encountered for the third and final time, whereupon it is noted for the first time, enabling it and the Everything class to be added to the class-precedence list. Thus, the Gourmands class appears before the Dwarf class, as desired:

```
Blimpy
Managers class
Competitors class
Gourmands class       ← procedure stored here
Diarists class
Dwarfs class          ← procedure stored here
Everything class
```

Now the first appetite-computing procedure encountered is the one that says Blimpy's appetite is huge.

A Class's Direct Superclasses Should Appear in Order

The depth-first, left-to-right, up-to-join procedure for computing class-precedence lists still leaves something to be desired. Consider, for example,

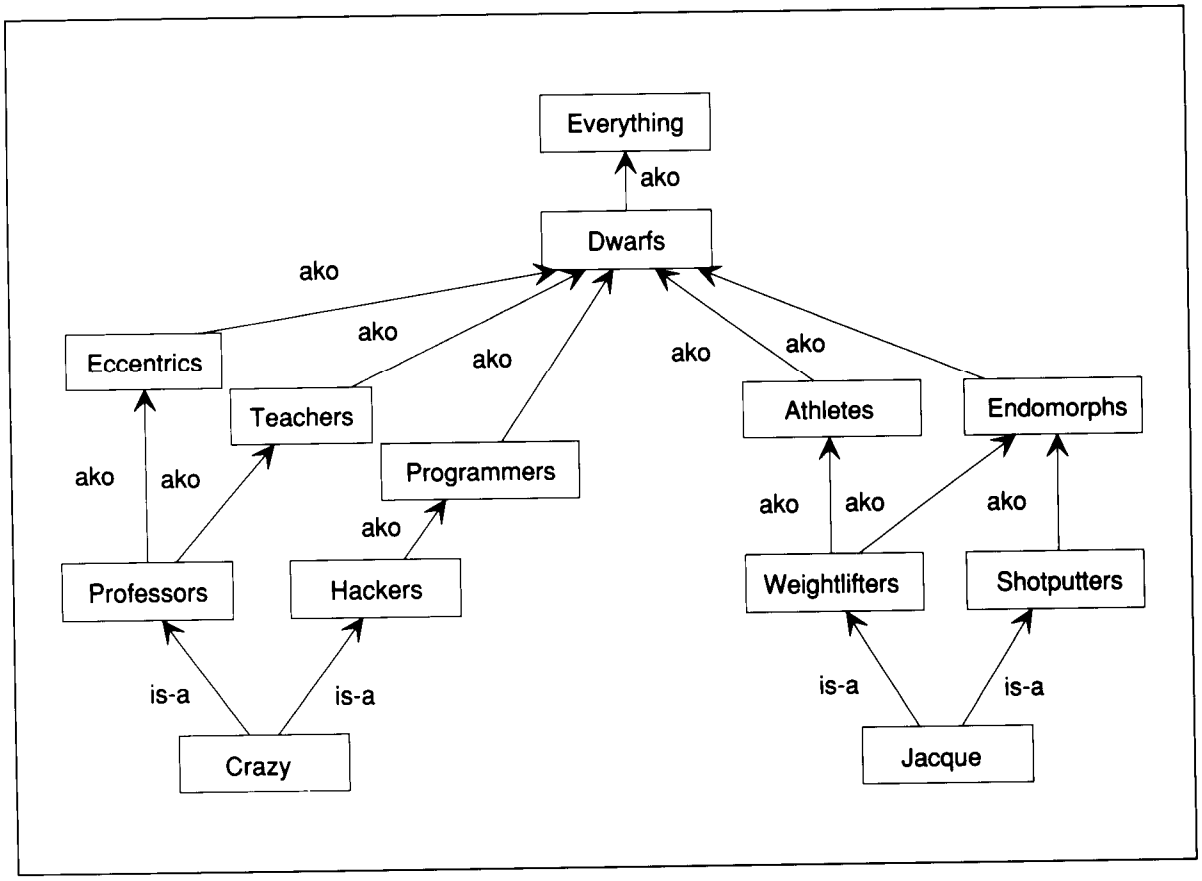

Figure 9.5 Another class hierarchy, with Is-a and Ako links shown. The depth-first, left-to-right, up-to-join approach produces appropriate class-precedence lists for both Crazy and Jacque.

the situation involving two other dwarfs, Crazy and Jacque, shown in figure 9.5.

The depth-first, left-to-right, up-to-join approach produces the following class-precedence lists for Crazy and Jacque:

Crazy	Jacque
Professors class	Weightlifters class
Eccentrics class	Athletes class
Teachers class	Shotputters class
Hackers class	Endomorphs class
Programmers class	Dwarfs class
Dwarfs class	Everything class
Everything class	

Nothing is amiss. No class appears after any of its own superclasses. Moreover, each class's direct superclasses appear in their given left-to-right order: The Professors class appears before the Hackers class; the Eccentrics class appears before the Teachers class; the Weightlifters class appears

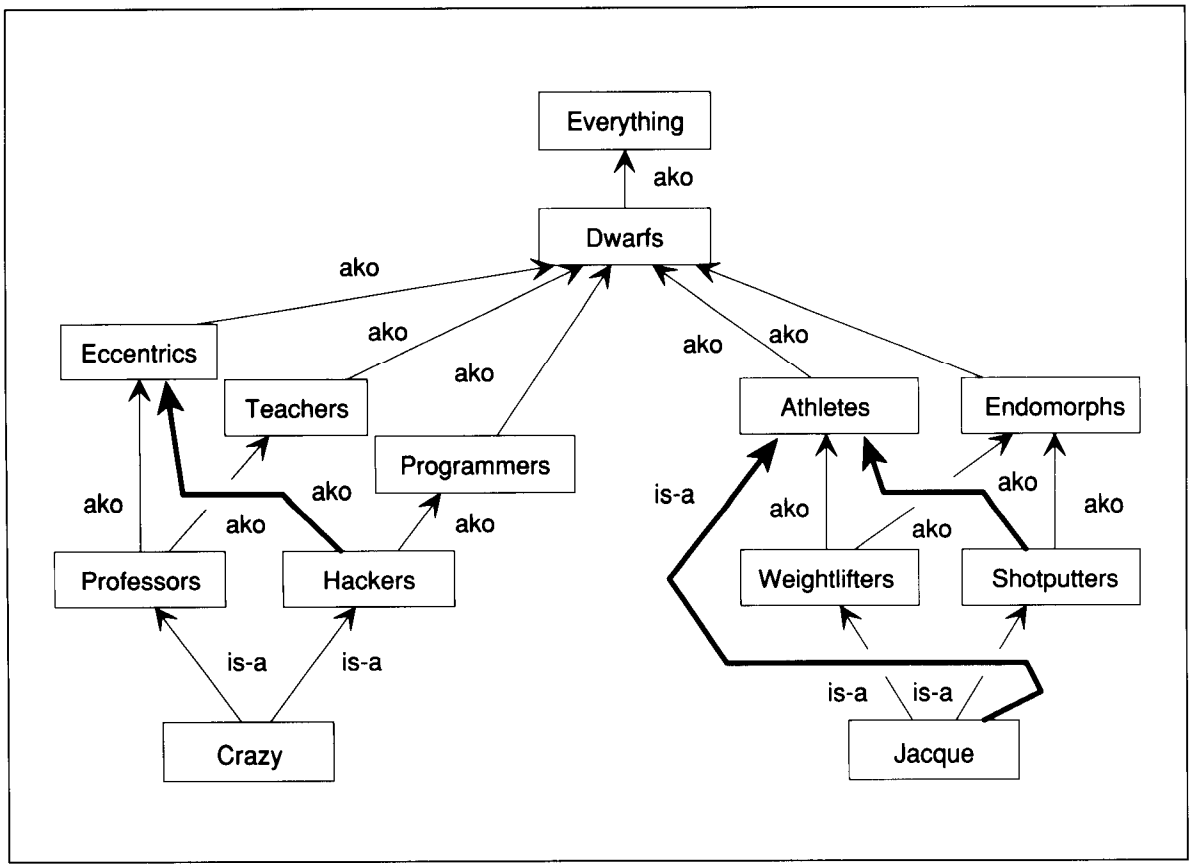

Figure 9.6 Still another class hierarchy, with one new Is-a link and two new Ako links shown by thick lines. This time, the depth-first, left-to-right, up-to-join approach does not produce appropriate class-precedence lists for either Crazy or Jacque.

before the Shotputters class; and the Athletes class appears before the Endomorphs class.

But suppose one Is-a link and two Ako links are added, as in figure 9.6. Now the class-precedence lists for Crazy and Jacque are different:

Crazy	Jacque
Professors class	Weightlifters class
Teachers class	Shotputters class
Hackers class	Endomorphs class
Eccentrics class	Athletes class
Programmers class	Dwarfs class
Dwarfs class	Everything class
Everything class	

Again, no class appears after any of its own superclasses, but the Eccentrics and Teachers classes—direct superclasses of the Professors class—are now out of the left-to-right order prescribed by the Ako links exiting from the Professors class. Similarly, the Athletes and Endomorphs classes—direct

superclasses of the Weightlifters class—are now out of the left-to-right order prescribed by the Ako links exiting from the Weightlifters class. In both instances, the order changes are caused by the addition of Ako links connected to other classes. These order changes are bad because left-to-right order, by convention, is supposed to indicate priority. You need a still better way to compute class-precedence lists that conforms to the following rule:

- Each direct superclass of a given class should appear on class-precedence lists before any other direct superclass that is to its right.

The Topological-Sorting Procedure Keeps Classes in Proper Order

The **topological-sorting procedure**, to be described in this section, is definitely more complicated than the depth-first, left-to-right, up-to-join procedure. The extra complexity is worthwhile, however, because the topological-sorting procedure keeps direct superclasses in order on class-precedence lists. Thus, you know the order of a class's direct superclasses on the class's class-precedence list as soon as you know how the direct superclasses are ordered: You do not need to know the entire structure of the class hierarchy.

Before you learn the details of the topological sorting procedure, however, you will find it helpful to see what happens when a path through a class hierarchy is expressed as a list of adjacent pairs. For example, the simple, nonbranching class hierarchy in figure 9.2 can be represented as three pairs of adjacent classes, Managers–Competitors, Competitors–Dwarfs, and Dwarfs–Everything.

Note that the order in which the pairs appear can be scrambled without hindering your ability to reconstruct the original path. First, you look for a class that occupies the left side of a pair but that does not occupy the right side of any other pair. There will always be such a class; once you find it, you need only to add it to the end of a list, to strike out the pair in which it appears, and to repeat.

Next consider the classes to which Blimpy belongs, as shown in figure 9.4. Blimpy is not just a Manager; he is also a Gourmand and a Diarist, in that left-to-right order. Now you can express that left-to-right order as a list of adjacent pairs, just as you previously expressed a path up a class hierarchy as a set of left-to-right pairs. This time, you get Managers–Gourmands and Gourmands–Diarists.

As before, you can scramble the order of the pairs without hindering your ability to reconstruct the original left-to-right order. Again, all you need to do is to look for a class that occupies the left side of a pair but that does not occupy the right side of a pair. Once you find it, you add it to the end of a list, strike out the pair in which it appears, and repeat.