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,

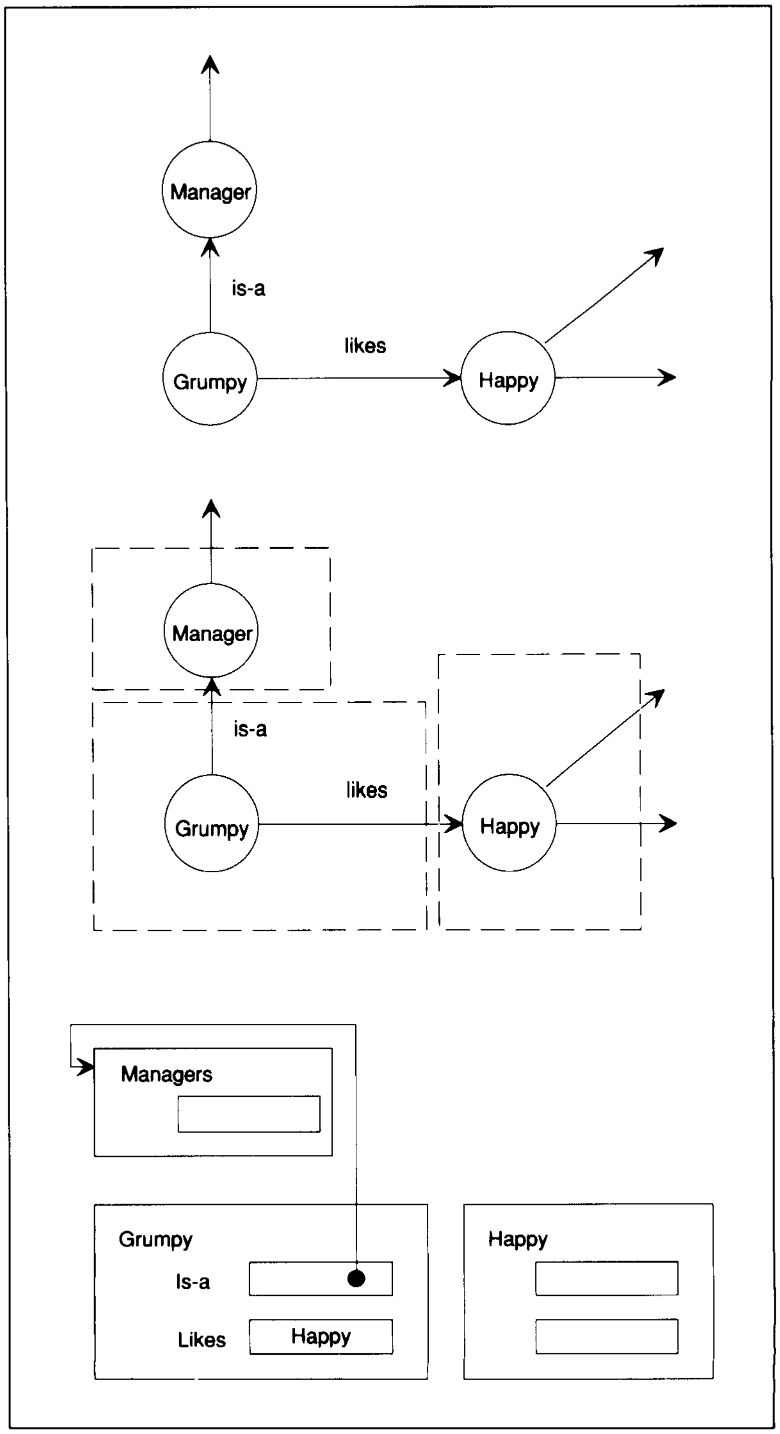

Figure 9.1 A semantic net can be viewed either as a collection of nodes and links or as a collection of frames. At the top, a semantic net is viewed as a collection of nodes and links. In the middle, the same semantic net is divided into chunks, each of which consists of a node and the links that emanate 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.

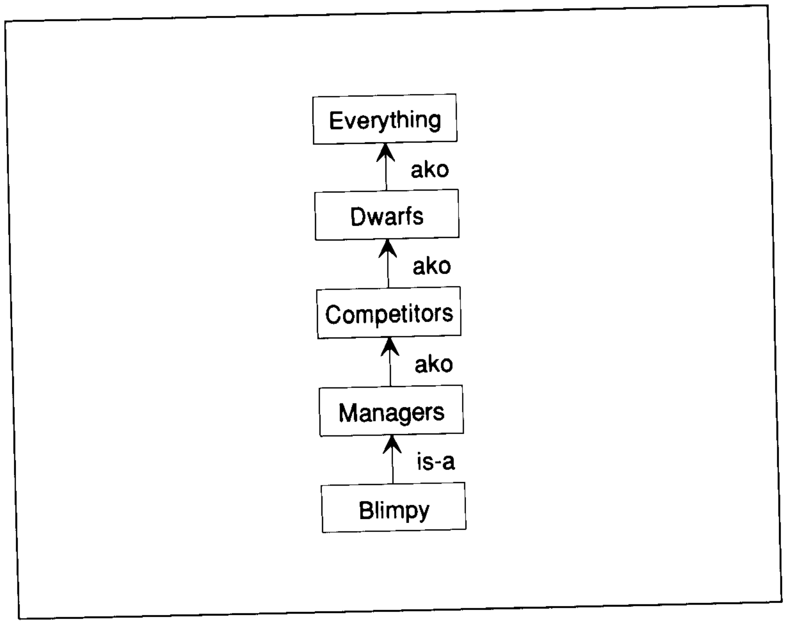

Figure 9.2 A simple class hierarchy. Blimpy is a member of the Managers class, which is a direct subclass of the Competitors class and a subclass of the Dwarfs class. Every class is considered to be, ultimately, a subclass of the Everything class.

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.

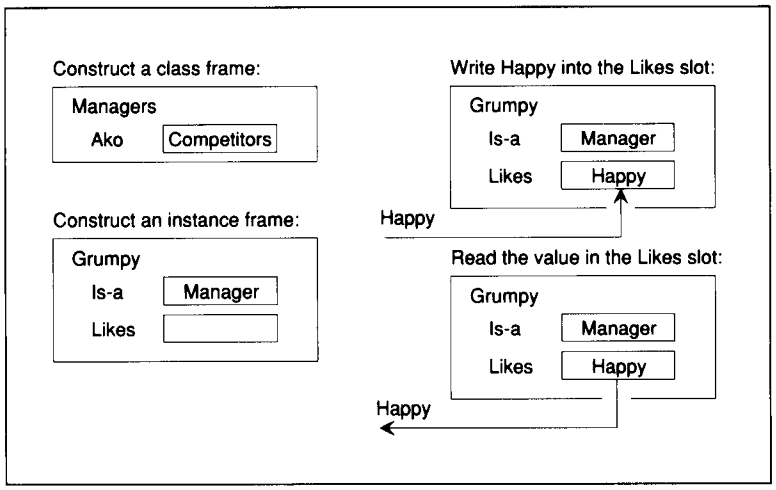

Figure 9.3 Instance frames and class frames are data types that are made and accessed with various constructors, writers, and readers.

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.

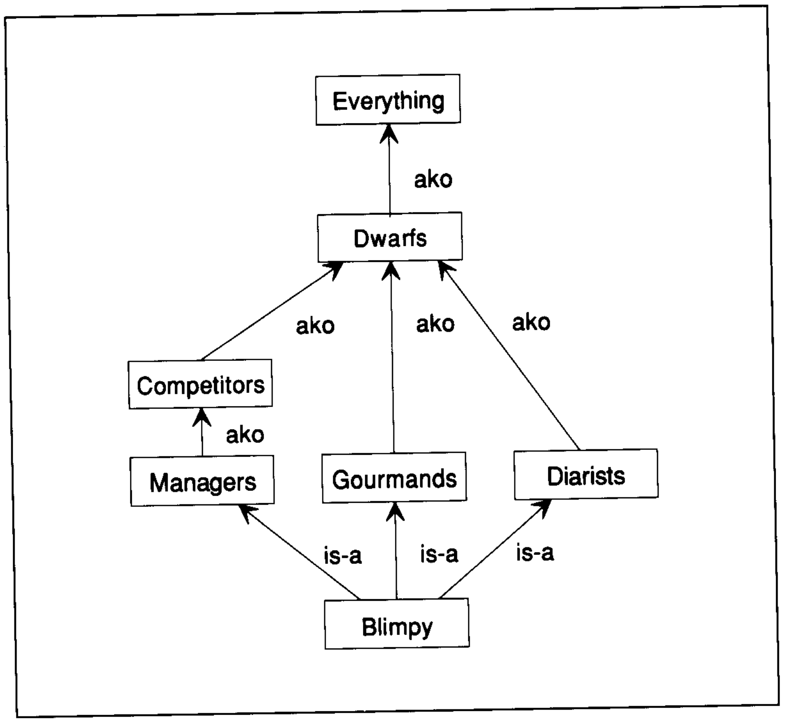

Figure 9.4 Another class hierarchy. Because Blimpy belongs to both the Gourmands class and to the Diarists class, as well as the Managers class, the class hierarchy branches upward. Because the Dwarfs class has three subclasses—Competitors, Gourmands and Diarists—the class hierarchy branches downward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

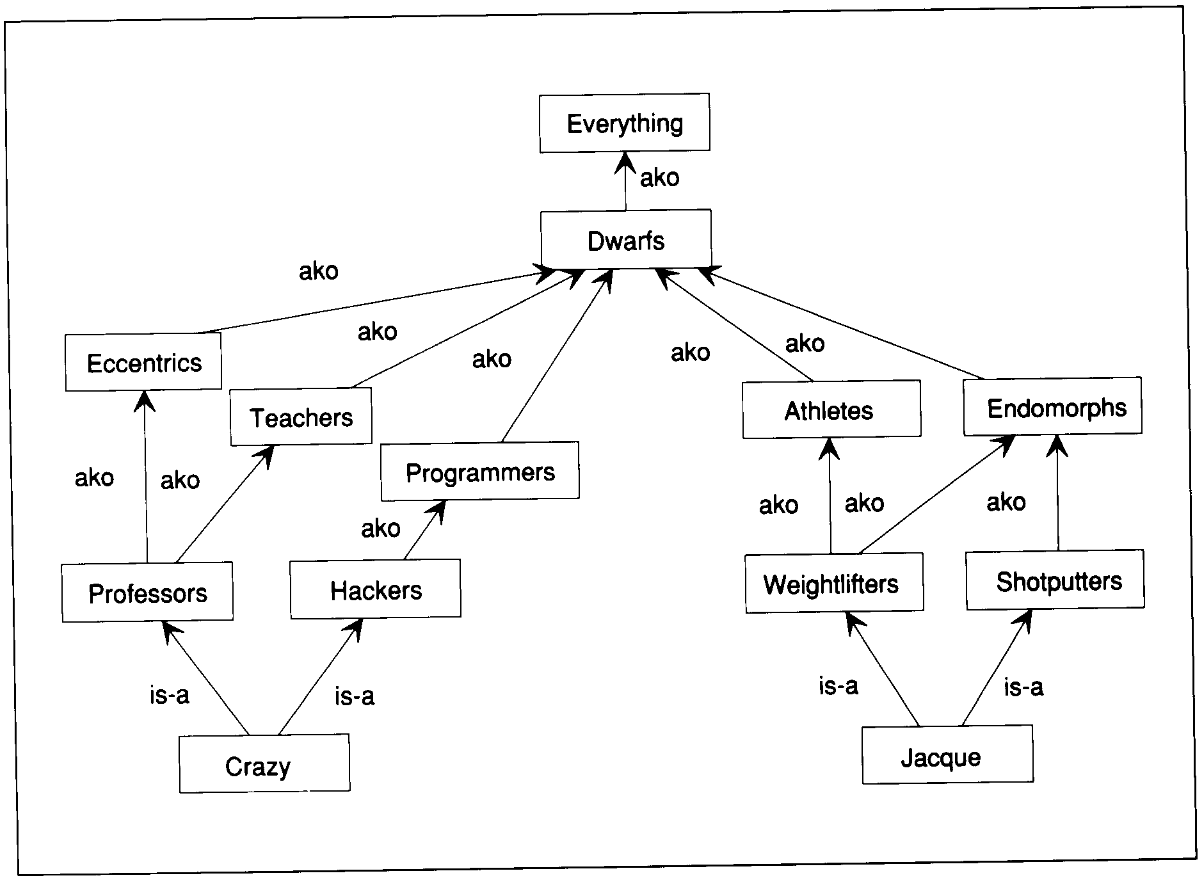

Figure 9.5 Another class hierarchy, with Is-a and Ako links shown. The depth-first, left-to-right, up-to-join approach produces appropriate class-precedence lists for both Crazy and Jacque.

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

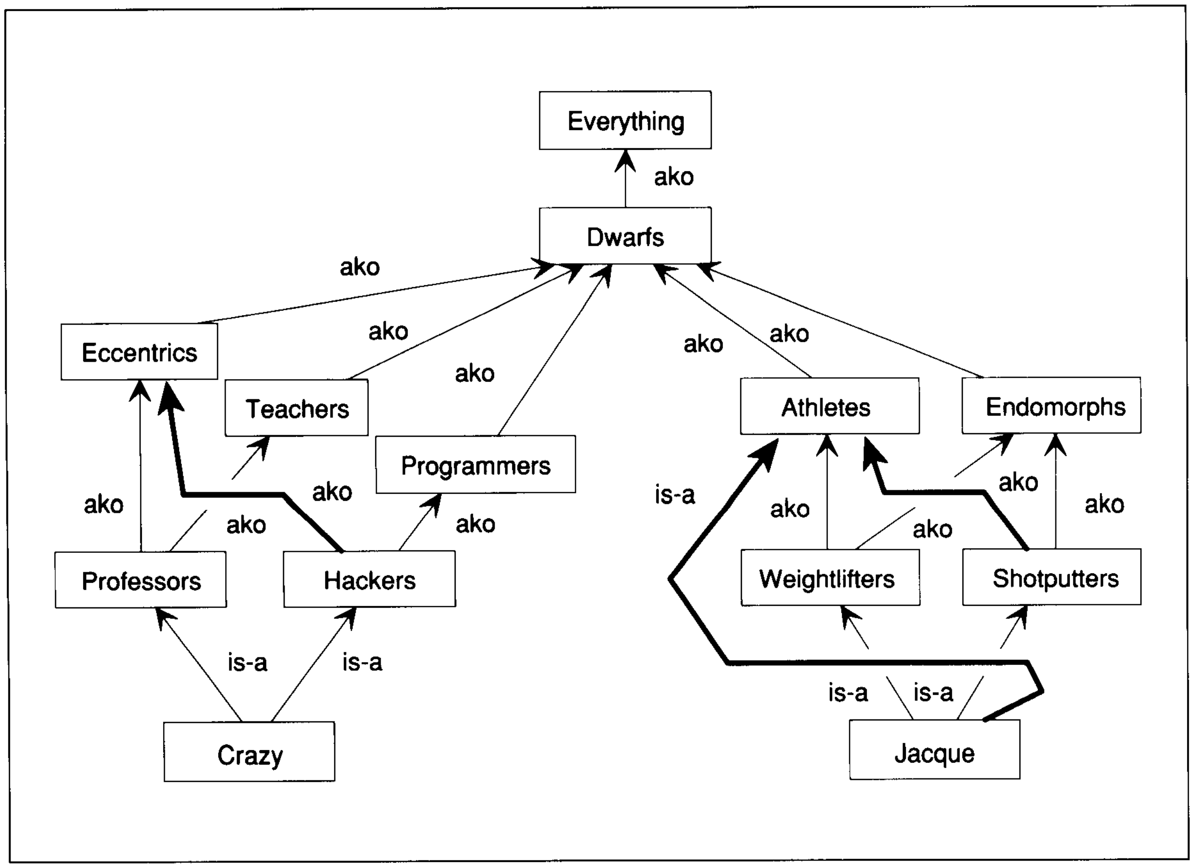

Figure 9.6 Still another class hierarchy, with one new Is-a link and two new Ako links shown by thick lines. This time, the depth-first, left-to-right, up-to-join approach does not produce appropriate class-precedence lists for either Crazy or Jacque.

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.

Thus, you can reconstruct either a nonbranching path up a class hierarchy or the left-to-right order across a set of direct superclasses from appropriately constructed lists of pairs. In one instance, you ensure that no class is listed before any of its superclasses; in the other instance, you ensure that the left-to-right order of direct superclasses is preserved.

Now you already understand the key idea behind the topological-sorting procedure; all that remains is to learn about a clever way of constructing a list of pairs such that both the upward and rightward constraints are expressed.

The first step in forming a class-precedence list for an instance using topological sorting is to form an exhaustive list consisting of the instance itself and all classes that can be reached via Is-a and Ako links from that instance. For Crazy, for example, this list contains Crazy, Professors, Eccentrics, Dwarfs, Everything, Teachers, Hackers, and Programmers. Note that this list constitutes raw material for building the class precedence list; it is not the class-precedence list itself.

The next step is to form a list of pairs for the one instance and the many classes on the raw-materials list. To make discussion easier, let us refer to both the instance and the classes on the raw-materials list as *items*.

To form a list of pairs for an item on the raw-materials list, think of passing a fish hook through the item and that item's direct superclasses, as shown in figure 9.7. Next, walk along the fish hook from barb to eyelet while making a list of pairs of adjacent items encountered on the hook. Following the fish hook for Crazy produces Crazy–Professors and Professors–Hackers. Following the fish hook for Professors produce Professors–Eccentrics and Eccentrics–Teachers; following the fish hook for Hackers produce Hackers–Eccentrics and Eccentrics–Programmers.

Following fish hooks for all the items on the raw materials list for Crazy yields the following pairs:

Node	Fish-hook pairs
Crazy	Crazy–Professors, Professors–Hackers
Professors	Professors–Eccentrics, Eccentrics–Teachers
Eccentrics	Eccentrics–Dwarfs
Dwarfs	Dwarfs–Everything
Teachers	Teachers–Dwarfs
Hackers	Hackers–Eccentrics, Eccentrics–Programmers
Programmers	Programmers–Dwarfs
Everything	Everything

The next step is to look for an item that occupies the left side of one or more pairs, but does not occupy the right side of any pair. To make it easier to refer to such an item, let us say that it is **exposed**. In our example, Crazy is exposed by virtue of the pair Crazy–Professors and the absence of any pair with Crazy on the right side.

Whenever you find an exposed item, you add it to the end of the class-precedence list and strike out all pairs in which it occurs. For the example, this means starting the class-precedence list with Crazy and striking out Crazy–Professors:

Node	Fish-hook pairs
Crazy	~~Crazy–Professors~~, Professors–Hackers
Professors	Professors–Eccentrics, Eccentrics–Teachers
Eccentrics	Eccentrics–Dwarfs
Dwarfs	Dwarfs–Everything
Teachers	Teachers–Dwarfs
Hackers	Hackers–Eccentrics, Eccentrics–Programmers
Programmers	Programmers–Dwarfs
Everything	Everything

Class-precedence list: Crazy

Now, with the pair Crazy–Professors struck out, the Professors class is exposed, leading to the next addition to the class-precedence list and to the accompanying strike-out action:

Node	Fish-hook pairs
Crazy	~~Crazy–Professors~~, ~~Professors–Hackers~~
Professors	~~Professors–Eccentrics~~, Eccentrics–Teachers
Eccentrics	Eccentrics–Dwarfs
Dwarfs	Dwarfs–Everything
Teachers	Teachers–Dwarfs
Hackers	Hackers–Eccentrics, Eccentrics–Programmers
Programmers	Programmers–Dwarfs
Everything	Everything

Class-precedence list: Crazy, Professors

Now the Hackers class is exposed, so you add Hackers and strike Hackers–Eccentrics:

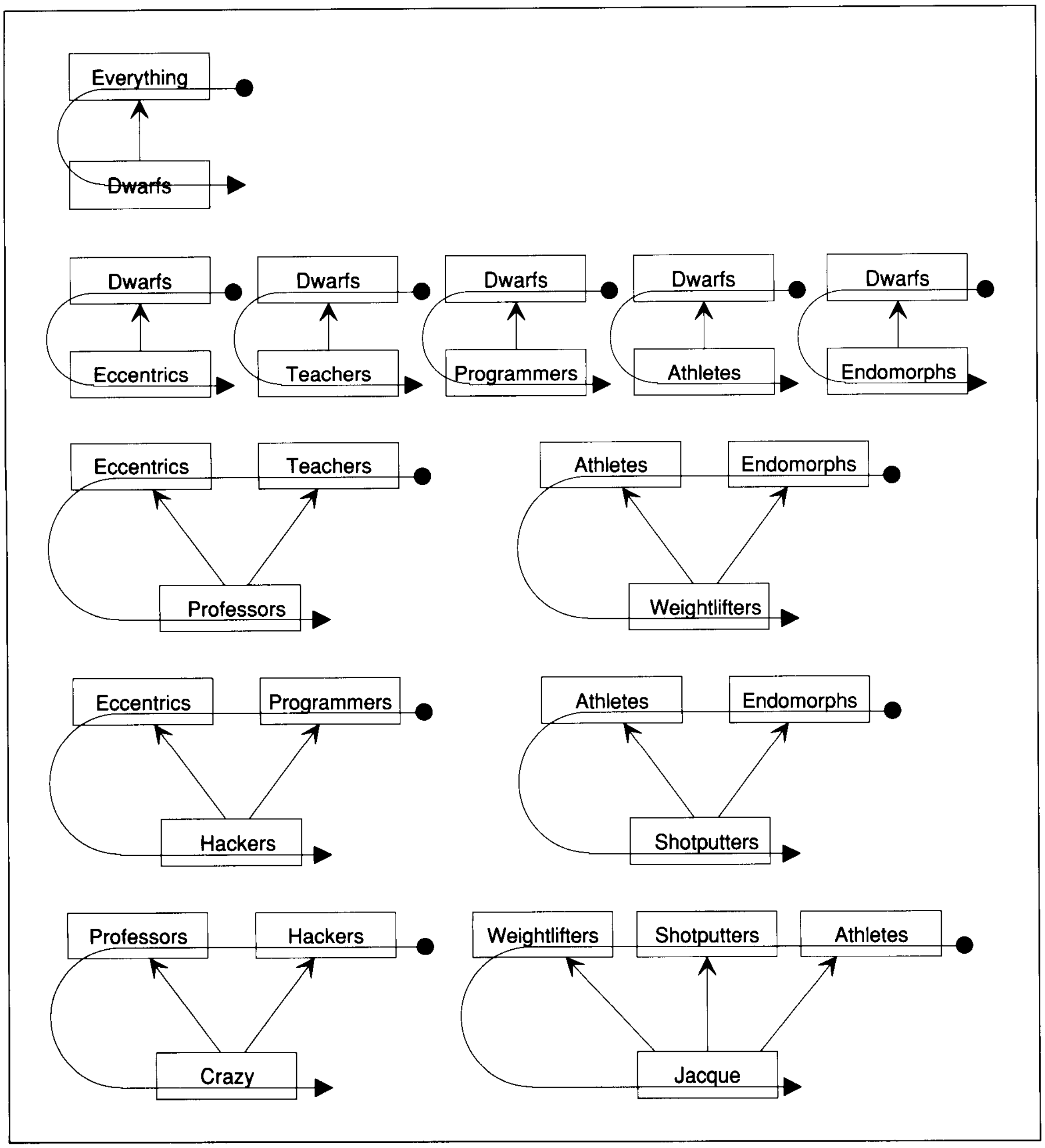

Figure 9.7 Fish hooks for Crazy, Jacque, and the classes reachable from them. These fish hooks yield lists of class pairs that enable the computation of a class-precedence list via the topological-sorting procedure.

Node	Fish-hook pairs
Crazy	~~Crazy–Professors~~, ~~Professors–Hackers~~
Professors	~~Professors–Eccentrics~~, Eccentrics–Teachers
Eccentrics	Eccentrics–Dwarfs
Dwarfs	Dwarfs–Everything
Teachers	Teachers–Dwarfs
Hackers	~~Hackers–Eccentrics~~, Eccentrics–Programmers
Programmers	Programmers–Dwarfs
Everything	Everything

Class-precedence list: Crazy, Professors, Hackers

Now the Eccentrics class is exposed, so you add Eccentrics and strike Eccentrics–Teachers, Eccentrics–Dwarfs, and Eccentrics–Programmers:

Node	Fish-hook pairs
Crazy	~~Crazy–Professors~~, ~~Professors–Hackers~~
Professors	~~Professors–Eccentrics~~, ~~Eccentrics–Teachers~~
Eccentrics	~~Eccentrics–Dwarfs~~
Dwarfs	Dwarfs–Everything
Teachers	Teachers–Dwarfs
Hackers	~~Hackers–Eccentrics~~, ~~Eccentrics–Programmers~~
Programmers	Programmers–Dwarfs
Everything	Everything

Class-precedence list	Crazy
	Professors
	Hackers
	Eccentrics

At this point, there are two exposed classes, Teachers and Programmers. Accordingly, you need a way to break ties. One possible tie breaker—one that tends to prevent erratic movement through the tree—is to select the class that is a direct superclass of the lowest-precedence class on the emerging class-precedence list.

In the example, however, neither the Teachers class nor the Programmers class is a direct superclass of the lowest-precedence class on the class-precedence list, the Eccentrics class.

Generalizing a bit, you move from lowest precedence to highest precedence on the emerging class-precedence list, encountering the Hackers class. Because the Programmers class is a direct superclass of the Hackers class, but the Teachers class is not, the tie is broken and you can proceed:

Node	Fish-hook pairs
Crazy	~~Crazy–Professors~~, ~~Professors–Hackers~~
Professors	~~Professors–Eccentrics~~, ~~Eccentrics–Teachers~~
Eccentrics	~~Eccentrics–Dwarfs~~
Dwarfs	Dwarfs–Everything
Teachers	Teachers–Dwarfs
Hackers	~~Hackers–Eccentrics~~, ~~Eccentrics–Programmers~~
Programmers	~~Programmers–Dwarfs~~
Everything	Everything

Class-precedence list	Crazy
	Professors
	Hackers
	Eccentrics
	Programmers

From here on, progress is uneventful, with the following result:

Node	Fish-hook pairs
Crazy	~~Crazy–Professors~~, ~~Professors–Hackers~~
Professors	~~Professors–Eccentrics~~, ~~Eccentrics–Teachers~~
Eccentrics	~~Eccentrics–Dwarfs~~
Dwarfs	~~Dwarfs–Everything~~
Teachers	~~Teachers–Dwarfs~~
Hackers	~~Hackers–Eccentrics~~, ~~Eccentrics–Programmers~~
Programmers	~~Programmers–Dwarfs~~
Everything	~~Everything~~

Class-precedence list	Crazy	
	Professors	
	Hackers	← procedure to be stored here
	Eccentrics	← procedure to be stored here
	Programmers	
	Teachers	
	Dwarfs	
	Everything	

Now, suppose you create two personality-determining when-constructed procedures, one for Hackers and one for Eccentrics. The when-constructed procedure that is specialized to the Hacker class indicates that hackers are shy:

```
To fill the Personality slot when a new Hacker is constructed,
▷ Write Shy in the slot.
```

On the other hand, the when-constructed procedure that is specialized to the Eccentrics class indicates that eccentrics are weird:

```
To fill the Personality slot when a new Eccentric is constructed,
▷ Write Weird in the slot.
```

Now suppose that the Crazy instance is constructed after these when-constructed procedures are defined. Is Crazy Shy or Weird? Evidently, Crazy is Shy, because Hackers appears before Eccentrics on Crazy's class-precedence list.

In summary, when new individuals are created, when-constructed procedures supply default slot values. The class-precedence list determines which when-constructed procedures are appropriate:

```
To fill the slots in a new instance,
▷ Compute the class-precedence list for the new instance
  using the topological-sorting procedure.
▷ For each slot,
  ▷ Collect all when-constructed procedures for that slot.
  ▷ Move along the class-precedence list, from the most
    specific end. Stop when you encounter a class that is
    referred to by one of the slot-specific when-constructed
    procedures. Call this when-constructed procedure
    the most specific when-constructed procedure for the
    slot.
  ▷ Use that most specific when-constructed procedure.
```

To compute the class-precedence list, you can use the topological sorting procedure, which honors the subclass–superclass principle and the left-to-right principle:

```
To compute an instance's class-precedence list,
▷ Create fish-hook pairs
▷ Until all the fish-hook pairs are eliminated
    ▷ Find the exposed classes.
    ▷ Select the exposed class that is a direct superclass
      of the lowest-precedence class on the emerging class-
      precedence list.
    ▷ Add the selected class to the emerging class-precedence
      list.
    ▷ Strike all fish-hook pairs that contain the newly added
      class.
```

DEMON PROCEDURES

So far, you have seen that slot values can be established by inheritance when instances are constructed, or by the direct use of a writer for slots. In this section, you see that reading or writing can activate **when-requested procedures**, **when-read procedures**, or **when-written procedures**. Sometimes these procedures are called **demons** because they lurk about doing nothing unless they see the request, read, or write operations they were designed to look for. In contrast to ordinary demons, these when-requested, when-read, and when-written demons are entirely friendly.

When-Requested Procedures Override Slot Values

After an instance has been constructed, you can replace slot values installed at creation time. If you like, you can go one step further, overriding any existing slot values altogether, using when-requested procedures. One such when-requested procedure indicates that athletes generally exercise as a hobby:

```
When a value for the Hobby slot of an Athlete is requested,
▷ Return Exercise.
```

Thus, Exercise becomes a sort of virtual slot value. No slot actually has Exercise in it, but it seems as though the Hobby slots of all Athletes have Exercise in them nevertheless.

When-requested procedures do not need be as simple as the when-requested, hobby-determining procedure for Athletes. They can, for example, take advantage of slot values already established by when-constructed procedures:

When a value for the Hobby slot of a Dwarf is requested,

- ▷ If the dwarf's Personality slot is filled with Shy, return Reading.
- ▷ Otherwise, return Dancing.

Now that there are two hobby-determining procedures, you need a way to choose between them. Naturally, it makes sense to use the same precedence-determining topological-sorting procedure that you have seen already in the context of choosing among when-constructed procedures.

For Crazy's Hobby slot, the governing procedure is the when-requested, hobby-determining procedure for Dwarfs that examines the dwarf's Personality slot. Inasmuch as Crazy's Personality slot was filled with Shy when the Crazy instance was constructed, Crazy's hobby must be reading. On the other hand, for Jacque's Hobby slot, the governing procedure is the when-requested procedure for Athletes that straightaway indicates the Athlete's Hobby is Exercise.

When-Read and When-Written Procedures Can Maintain Constraints

When-read and when-written procedures are activated when slot values are, respectively, read and written. The following when-written procedure is activated whenever a value is written into the Physique slot of an Athlete after the Athlete is constructed:

When a value is written in the Physique slot of an Athlete,

- ▷ If the new value is Muscular, write Large in the Athlete's Appetite slot.

Evidently, this when-written procedure captures a constraint relating an Athlete's Physique to the Athlete's Appetite: If the new slot value is Muscular, then Large is written into the Appetite slot. Thus, Muscular Athletes have Large Appetites, in contrast to Gourmands, who have Huge Appetites, and ordinary Dwarfs, who have Small Appetites.

As the example illustrates, when-read and when-written procedures can be used to ensure that a change in one slot's value is reflected in an appropriate, automatic change to another slot's value. In this role, they perform as constraint-enforcing bookkeepers.

In contrast to when-constructed and when-requested procedures, all applicable when-read and when-written procedures always are activated—rather than only the one with the highest precedence as determined by the topological sorting procedure. Given that all applicable when-read and when-written procedures are activated, however, there is a question of order. Sophisticated frame systems provide you with a variety of options.

With-Respect-to Procedures Deal with Perspectives and Contexts

Sometimes, the proper way to think about an instance is determined by a particular perspective. A particular dwarf, Blimpy, may be considered big for a dwarf, but small when viewed from the perspective of, say, **Snow White**. At other times, the proper way to think about an instance is conditioned by the context in which instance lies. A particular person, for example, may be happy when hiking in the mountains, yet grumpy when traveling on an airplane.

To deal with these dependencies, you use **with-respect-to procedures**, which are when-requested procedures that are specialized to more than one class. The following, for example, are two with-respect-to size-determining procedures, each of which is specialized to two classes, the first being the class to which an instance belongs, and the second being the reference class:

When a value for the Size slot of Blimpy, from the perspective of a typical dwarf, is requested,

▷ Return Big.

When a value for the Size slot of Blimpy, from the perspective of a typical person, is requested,

▷ Return Small.

Similarly, you can define with-respect-to procedures that involve context:

When a value for the Mood slot of Patrick, in the context of Mountain Hiking, is requested,

▷ Return Happy.

When a value for the Mood slot of Patrick, in the context of Airplane Travel, is requested,

▷ Return Grumpy.

Inheritance and Demons Introduce Procedural Semantics

When no demons are used, frame systems can be viewed as semantic nets. When demons are used, however, a great deal of procedural knowledge can be incorporated into a particular frame system. Accordingly, the mechanisms that enable the incorporation of procedural knowledge are prominent in the specification for frame systems:

A **frame system** is a representation

That is a semantic net

In which

▷ The language of nodes and links is replaced by the language of frames and slots.

▷ Ako slots define a hierarchy of class frames.

▷ Is-a slots determine to which classes an instance frame belongs.

▷ Various when-constructed, when-requested, when-read, when-written, and with-respect-to procedures supply default values, override slot values, and maintain constraints.

▷ A precedence procedure selects appropriate when-constructed, when-requested, when-read, when-written, and with-respect-to procedures by reference to the class hierarchy.

With constructors that

▷ Construct a class frame, given a list of superclasses, and a list of slots

▷ Construct an instance frame, given a list of direct superclasses

▷ Construct a when-requested, when-read, when-written, or with-respect-to procedure

With writers that

▷ Establish a slot's value, given an instance, a slot, and a value

With readers that

▷ Produce a slot's value, given an instance and a slot

Recall that you have an example of procedural semantics when meaning is defined by a set of procedures that operate on descriptions in a representation. Those procedures often lie outside the representation. In a frame system, however, powerful procedures are brought into the representation itself, becoming part of it.

One kind of incorporated procedural knowledge lies in the procedures for computing class precedence and using class-precedence to determine default slot values, thus contributing to the meaning of the Is-a and Ako slots. Another kind of incorporated procedural knowledge lies in demon procedures, many of which are permanent parts of whole classes of frame systems.

The idea of incorporating procedural knowledge into a representation is extremely powerful. In subsequent chapters, you see that you can build powerful problem solvers merely by adding some class definitions and constraint-enforcing demons to an off-the-shelf, generic frame system. Little is left to be done from scratch, other than the description of particular problems or situations.

Object-Oriented Programming Focuses on Shared Knowledge

You can benefit from the virtues of knowledge sharing, not only when creating, writing, and reading slot values, but also when performing actions in general.

Consider, for example, the problem you face when you have to decide how to eat various foods at a fancy dinner. You can capture the advice offered by a typical etiquette book in *when-applied procedures* such as the following:

To *eat* when Soup is to be eaten,

▷ Use a big spoon.

To *eat* when Salad is to be eaten,

▷ Use a small fork.

To *eat* when the Entree is to be eaten,

▷ Use a big fork and a big knife.

Thus, a **when-applied procedure** is a procedure that helps you to perform an action in a manner suited to the object acted on.

Note that when-applied procedures, like other demon procedures, are shared among subclasses automatically. Accordingly, you do not need to write and maintain separate procedures for every possible subclass of the soup, salad, and entree classes. If, however, some soup, salad, or entree subclass calls for an unusual or special tool, you can construct another when-applied procedure easily, specialize it to the appropriate subclass, an thereby ensure that your new, specific procedure will displace the old, general one:

To *eat* when the Entree is a Lobster,

▷ Use a tiny fork and a nutcracker.

Essentially, an **object-oriented programming language** enables knowledge sharing by providing mechanisms for defining object classes, creating individuals, and writing when-applied procedures. The virtues of knowledge sharing have made object-oriented programming languages increasingly popular.

FRAMES, EVENTS, AND INHERITANCE

In the previous section, you learned how you can capture general knowledge about individuals by using frames. In this section, you learn how frames can capture general knowledge about events of the sort described in newspapers.

Digesting News Seems to Involve Frame Retrieving and Slot Filling

Any news report of an earthquake probably will supply the place; the time; the number of people killed, injured, and homeless; the amount of property damage; the magnitude on the Richter scale; and possibly the name of the geological fault that has slipped. To represent this kind of knowledge in frames, you need the Earthquake, Disaster, and Event frames shown in figure 9.8.

Now suppose you have a news story freshly arrived from a wire service. You want to use that story to fill in the slots in an appropriate instance frame. Curiously, for many news stories—earthquake stories in particular—primitive when-constructed procedures can fill in the slots by looking in the story for various sorts of numbers:

To fill the Time slot when a new Event is constructed,

▷ Find a number with a colon in it and write it in the slot.

To fill the Fatalities slot when a new Disaster is constructed,

▷ Find an integer near a word with a root such as *kill* or *die*, and write it in the slot.

To fill the Damage slot when a new Disaster is constructed,

▷ Find a number next to a dollar sign, and write it in the slot.

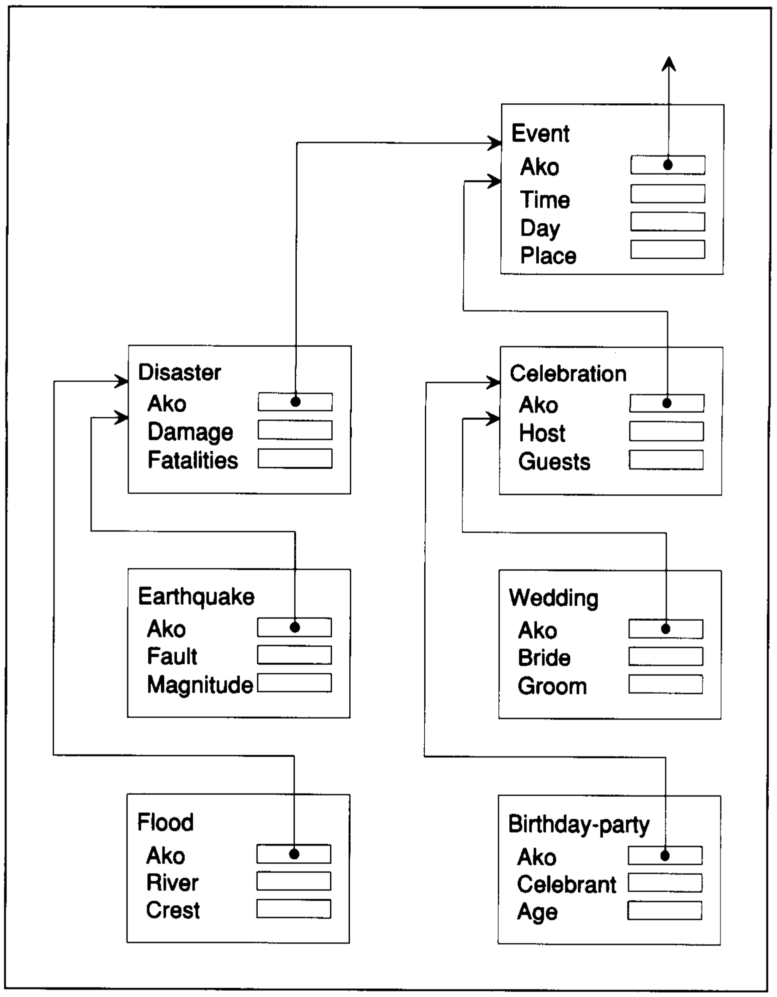

Figure 9.8 A net connecting frames for news stories. By inheritance on two levels, it is clear that earthquake stories typically have seven slots to be filled. All may have slot-filling procedures attached.

To fill the Magnitude slot when a new Earthquake is constructed,

▷ Find a decimal number between 1.0 and 10.0, and write it in the slot.

Other simple procedures can fill in nonnumeric slots:

To fill the Day slot when a new Event is constructed,

▷ Find a word such as *today*, *yesterday*, *tomorrow*, or the name of one of the days of the week, and write it in the slot.

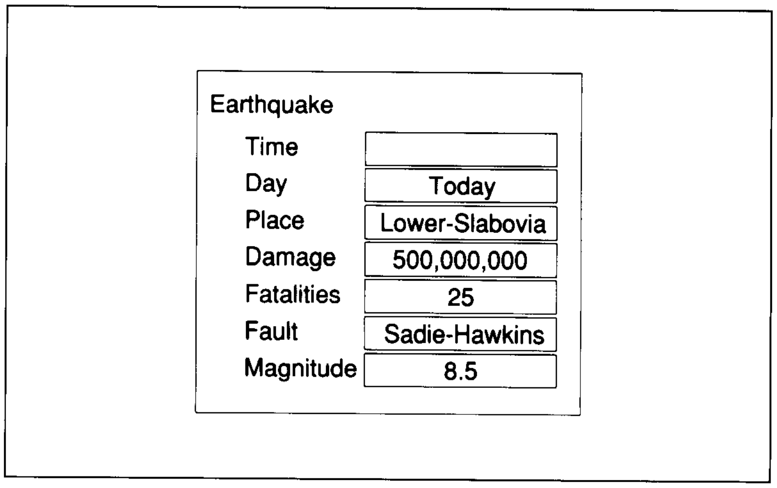

Figure 9.9 A frame produced by two news stories. One news story, correctly analyzed, is about a genuine earthquake. The other news story, muffed, is about earthquake research.

To fill the Place slot when a new Event is constructed,

▷ Find a name that appears in a dictionary of geographical places and write that name in the slot.

To fill the Fault slot when a new Earthquake is constructed,

▷ Find a proper name near the word *fault* and write it in the slot.

Consequently, analyzing stories such as the following can be easy, given that the title evokes the Earthquake frame:

Earthquake Hits Lower Slabovia
Today, an extremely serious earthquake of magnitude 8.5 hit Lower Slabovia, killing 25 people and causing $500 million in damage. The President of Lower Slabovia said that the hard-hit area near the Sadie Hawkins fault has been a danger zone for years.

In Chapter 7, you learned that, whenever a pattern is filled in with appropriate variable values, it is said to be *instantiated*. Figure 9.9 shows the instantiated frame constructed for the earthquake story. Once the frame is instantiated, the frame's slot values can be used to instantiate a summary pattern such as the following:

Earthquake Summary Pattern

An earthquake occurred in <*value in Location slot*> <*value in Day slot*>. There were <*value in Fatalities slot*> fatalities and $<*value in Damage slot*> in property damage. The magnitude was <*value in Magnitude slot*> on the Richter scale; the fault involved was the <*value in Fault slot*>.

Thus, you get the following summary by instantiating the earthquake summary pattern using data transferred in from an instantiated Earthquake frame:

Instantiated Earthquake Summary Pattern

An earthquake occurred in *Lower Slabovia today.* There were *25* fatalities and $500 *million* in property damage. The magnitude was *8.5* on the Richter scale; the fault involved was the *Sadie Hawkins.*

Evidently, the Earthquake frame stands between the story and its summary, helping to bridge the gap, so to speak.

Note, however, that slot filling using simple, special-purpose procedures can lead to silly results, given that the special-purpose procedures really do not understand stories. Consider this example:

Earthquake Study Stopped

Today, the President of Lower Slabovia killed 25 proposals totaling $500 million for research in earthquake prediction. Our Lower Slabovian correspondent calculates that 8.5 research proposals are rejected for every one approved. There are rumors that the President's science advisor, Sadie Hawkins, is at fault.

Shudder to think: This story could be summarized, naively, as though it were the story about an actual earthquake, producing the same frame shown before in figure 9.9 and the same instantiated earthquake summary pattern.

Of course, creating procedures for general news is much harder than creating procedures for specialized news. Interestingly, good news writers seem to use certain conventions that help:

- The title of a news story and perhaps the first sentence or two evoke a central frame.
- Subsequent material fills slots in the central frame. The slot-filling process evokes other frames introducing more open slots.
- Cause-effect relations are given explicitly. Readers do not need to deduce causes, because words such as *because* appear frequently.
- Few pronouns, if any, are used. In political news, for example, the nation's legislature may be referred to as "Congress," or "Capitol Hill," or "Washington's lawmakers," according to fancy.

- Few new frames, if any, need to be constructed. Creating new frames requires reflection, and reflection is discouraged.

Event-Describing Frames Make Stereotyped Information Explicit

You have seen that the information in event frames and when-constructed procedures make certain expectations and procedures explicit:

- The slots in event frames make explicit what you should expect to know about them.
- The when-constructed procedures associated with event frames make explicit how you can try to acquire what you expect to know.

By making explicit appropriate knowledge—what you expect to know and how to acquire what you expect to know—event frames and their associated procedures satisfy an important criterion for good representation.

SUMMARY

- A frame system can be viewed as a generalized semantic net. When you speak about frames, however, your language stresses instances or classes, rather than nodes, and stresses slots and slot values, rather than links and link destinations.
- Inheritance moves default slot values from classes to instances through the activation of the appropriate when-constructed procedure.
- To determine which when-constructed procedure dominates all other applicable when-constructed procedures, you have to convert a class hierarchy into a class-precedence list. Generally, the conversion should be such that each class appears before all that class's superclasses and each class's direct superclasses appear in order.
- When-requested procedures override slot values. When-read and when-written procedures maintain constraints. With-respect-to procedures deal with perspectives and contexts.
- Digesting news seems to involve inheritance. Your understanding of an earthquake news story, for example, benefits from your knowledge of the connection between earthquakes and disasters and your knowledge of the connection between disasters and events in general.
- 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, and easier to distribute because it can be distributed automatically.

BACKGROUND

Marvin Minsky is largely responsible for defining and popularizing many of the notions connected with frames [1975]. Other important contributions have been made via the many frame-oriented representation languages patterned after Minsky's ideas.

For a discussion of inheritance as embodied in a programming language, see the monumental reference work *Common Lisp, The Language*, by Guy L. Steele, Jr. [1990]

The discussion of news is based on the work of Gerald F. DeJong II [1979].