CHAPTER 27

Object-Relational Databases

Abstract

This chapter introduces readers to the object-oriented paradigm and how the relational data model has been extended to support a hybrid object–relational data model. The chapter covers the basic principles of object-orientation and the nature of pure object-oriented databases. It then discusses the object–relational data model in depth, followed by how SQL provides support for that model. Topics include SQL data types for object-oriented features, user-defined data types, typed tables, and methods that are a part of user-defined data types.

Keywords

object-oriented paradigm
object-oriented databases
object-oriented DBMS
SQL
object-relational data model
SQL object-oriented features
user-defined data types
SQL user-defined data types
UDT
object-oriented inheritance
SQL typed tables

The relational data model has been a mainstay of business data processing for more than 30 years. Nothing has superseded it in the way the relational data model superseded the simple network data model. However, a newer data model—the object-oriented data model—has come into use as an alternative for some types of navigational data processing.

Note: To be completely accurate, the relational data model is the only data model that has a formal specification. The hierarchical data model and the OO data model do not. The closest thing the simple network data model has is the CODASYL specifications.

This chapter begins with an overview of some object-oriented concepts for readers who aren't familiar with the object-oriented paradigm. If you have object-oriented programming experience, then you can skip over the first parts of this chapter and begin reading with the section *The Object-Relational Data Model*.

The object-oriented paradigm was the brainchild of Dr Kristen Nygaard, a Norwegian who was attempting to write a computer program to model the behavior of ships, tides, and fjords. He found that the interactions were extremely complex and realized that it would be easier to write the program if he separated the three types of program elements and let each one model its own behavior against each of the others.

The object-oriented programming languages in use today (most notably C + +, Java, and JavaScript) are a direct outgrowth of Nygaard's early work. The way in which objects are used in databases today is an extension of object-oriented programming.

Note: This is in direct contrast to the relational data model, which was designed specifically to model data relationships, although much of its theoretical foundations are found in mathematical set theory.

Getting Started: Object-Orientation without Computing

To understand the role of objects in relational databases, you must first understand the object-oriented paradigm as it is used in object-oriented programming and pure object-oriented databases. The easiest way to do so is to begin with an example that has absolutely nothing to with programming at all.

Assume that you have a teenage daughter (or sister, whichever is more appropriate) named Jane and that your family is going to take a long car trip. Like many teens, Jane is less than thrilled about a trip with the family and, in particular, with spending so much time with her 12-year-old brother, especially since her parents have declared the trip a holiday from hand-held electronics. In self-defense, Jane needs something to keep her brother busy so he won't bother her as she reads while her parents are driving. She therefore decides to write up some instructions for playing solitaire games for him.

The first set of instructions is for the most common solitaire game, Klondike. As you can see in Figure 27.1, the deal involves seven piles of cards of increasing depth, with the top card turned over. The rest of the deck remains in the draw pile. Jane decides to break the written instructions into two main parts: information about the game and questions her brother might ask. She therefore produces instructions that look something like Figure 27.2. She also attached the illustration of the game's deal.

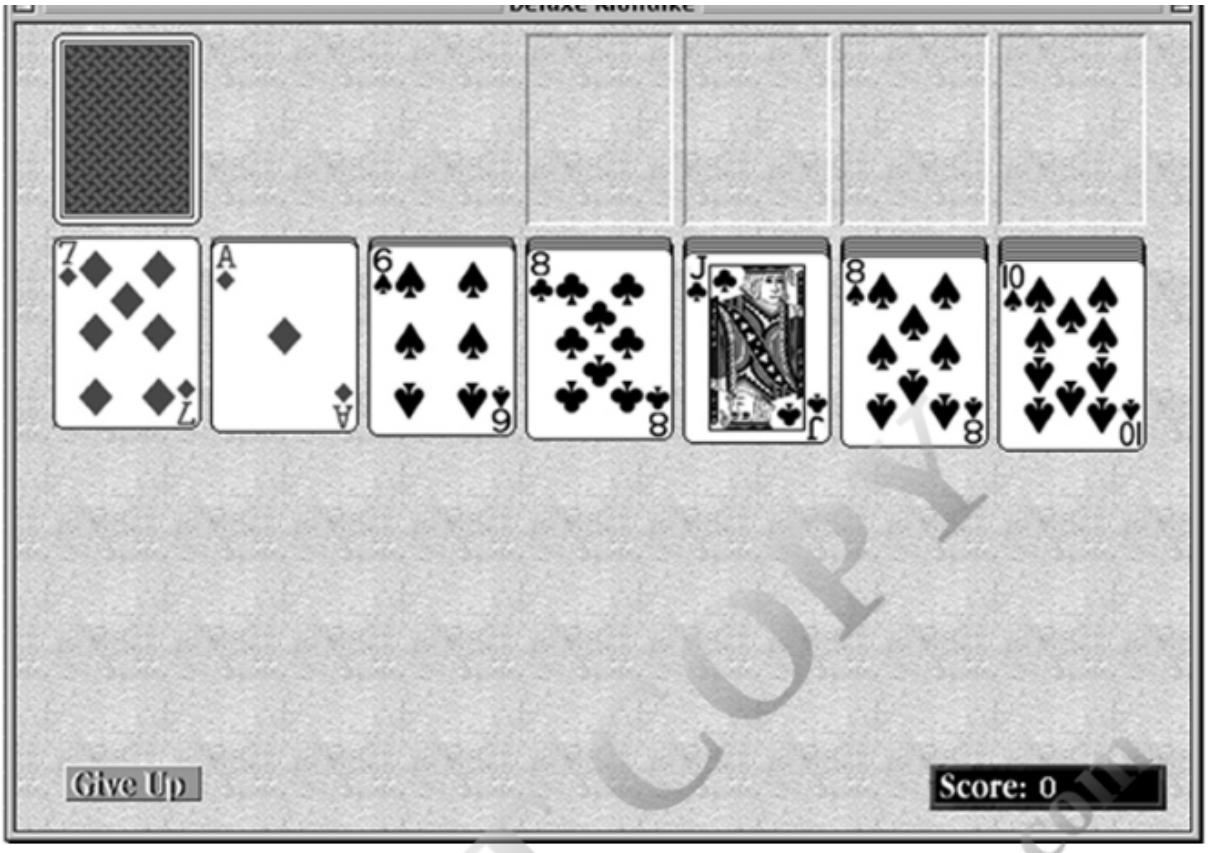


FIGURE 27.1 The starting layout for Klondike.

```
Information about the game
      Name: Klondike
      Illustration: See next page
      Decks: One
      Dealing: Deal from left to right
            First pass: First card face up six cards down.
            Second pass: First card face up on top of pile #2, five
                  cards down on remaining piles.
            Third pass: First card face up on top of pile #3; four
                  cards down on remaining piles.
            …repeat pattern for total of seven passes.
            Place remaining cards in draw pile, face down.
      Playing: One or two cards can be turned from the draw pile
            at a time. As encountered, put aces above layout. Build up
            from aces in suits. Build down on the deal, opposite suit
            colors. Can move from the middle of a stack moving card
            and all cards built below it.
            Move only kings into empty spots on the layout.
            If turning one card, make only one pass through the draw
                  Pile.
            If turning three cards, make as many passes as you like
                  through the draw pile.
      Winning: All cards built on top of their aces.
Questions to Ask
      What is the name of the game?
            Read Name section.
      How many decks do I need?
```

```
How many decks do I need?
      Read Decks section.
What does the layout look like?
      Read Illustration section.
How do I deal the game?
      Read Dealing section.
How do I play the game?
      Read Playing section.
How do I know when I've won?
      Read Winning section.
```

FIGURE 27.2 **Instructions for playing Klondike.**

The next game she tackles is Canfield. Like Klondike, it is played with one deck, but the deal and play are slightly different (see Figure 27.3). Jane uses the same pattern for the instructions as she did for Klondike because it cuts down the amount of writing she has to do (see Figure 27.4).

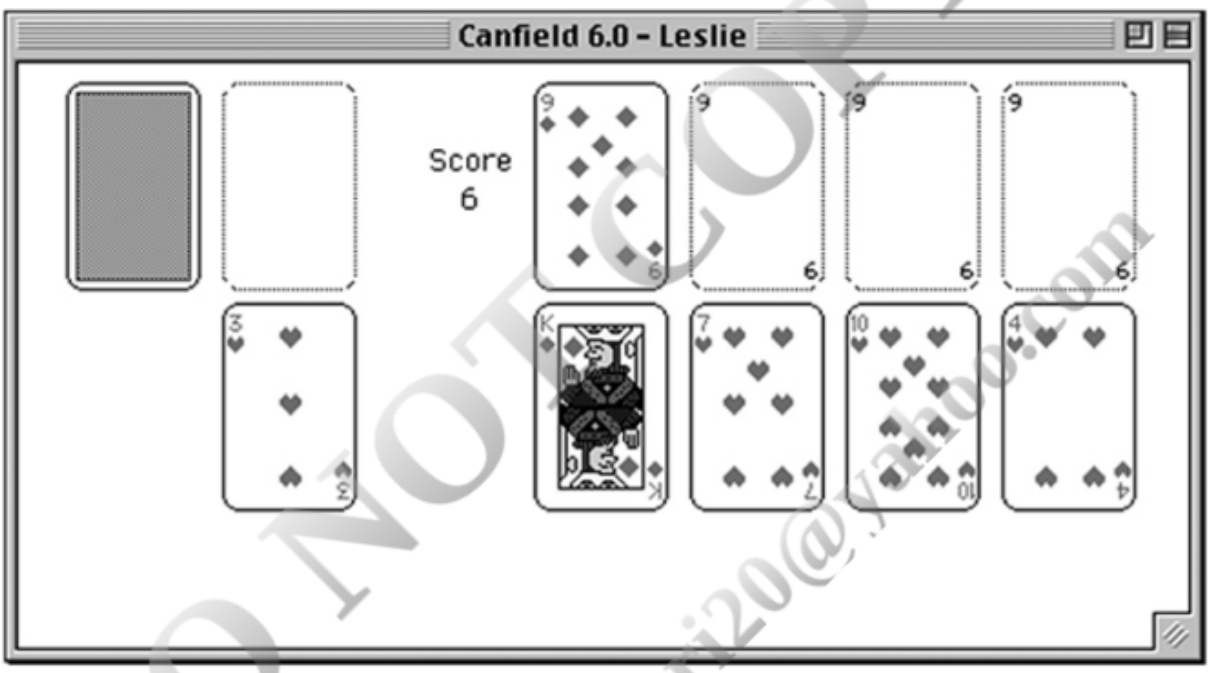


FIGURE 27.3 **The starting deal for Canfield.**

```
Information about the game
      Name: Canfield
      Illustration: See next page
      Decks: One
      Dealing: Deal four cards face up.
            Place one additional card above the first four as the
.                 starting card for building suits.
            The remaining cards stay in the draw pile.
      Playing: Turn one card at a time, going through the deck as many
                  times as desired.
            Build down from deal, opposite suit colors.
            Can move cards from the middle of stack, moving card and
                  all cards built below it.
            Place cards of the same value as the initial foundation
                  card above the deal as encountered.
            Build up in suits from the foundation cards.
            Any card can be placed in any empty space in the deal.
```

```
      Winning: All cards built on top of the foundation cards.
Questions to Ask
      What is the name of the game?
            Read Name section.
      How many decks do I need?
            Read Decks section.
      What does the layout look like.
            Read Illustration section.
      How do I deal the game?
            Read Dealing section.
      How do I play the game?
            Read Playing section.
      How do I know when I've won?
            Read Winning section.
```

FIGURE 27.4 **Instructions for playing Canfield.**

And finally, just to make sure her brother doesn't get too bored, Jane prepares instructions for Forty Thieves (see Figure 27.5). This game uses two decks of cards and plays in a very different way from the other two games (see Figure 27.6). Nonetheless, preparing the instructions for the third game is fairly easy because she has the template for the instructions down pat.

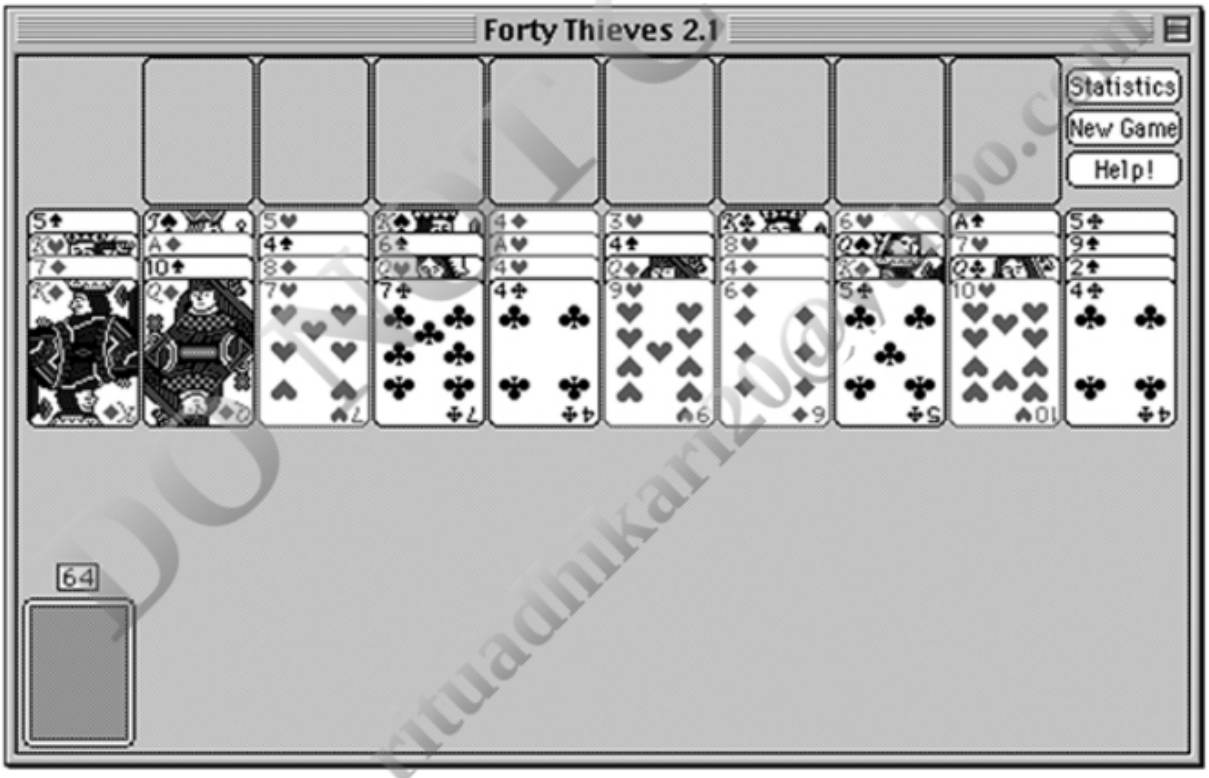


FIGURE 27.5 **The starting layout for Forty Thieves.**

```
Information about the game
      Name: Forty Thieves
      Illustration: See next page
      Decks: Two
      Dealing: Make 10 piles of four cards, all face up.
            Jog cards so that the values of all cards can be seen.
            Remaining cards stay in the draw pile.
      Playing: Turn one card at a time.
```

```
            Make only one pass through the draw pile.
            Build down in suits.
            Only the top card of a stack can be moved.
            As aces are encountered, place at top of deal and build up
                  in suits from the aces.
            Any card can be moved into any open space in the layout.
      Winning: All cards built on top of their aces.
Questions to Ask
      What is the name of the game?
            Read Name section.
      How many decks do I need?
            Read Decks section.
      What does the layout look like.
            Read Illustration section.
      How do I deal the game?
            Read Dealing section.
      How do I play the game?
            Read Playing section.
      How do I know when I've won?
            Read Winning section.
```

FIGURE 27.6 **Instructions for playing Forty Thieves.**

After completing three sets of instructions, it becomes clear to Jane that having the template for the instructions makes the process extremely easy. Jane can use the template to organize any number of sets of instructions for playing solitaire. All she has to do is make a copy of the template and fill in the values for the information about the game.

Basic OO Concepts

Objects

If someone were writing an object-oriented computer program to manage the instructions for playing solitaire, each game would be known as an *object*. It is a self-contained element used by the program, conceptually similar to an instance of an entity. It has things that it knows about itself: its name, the illustration of the layout, the number of decks needed to play, how to deal, how to play, and how to determine when the game is won. In object-oriented terms, the values that an object stores about itself are known as *attributes* or *variables* or, occasionally, *properties*.

Each solitaire game object also has some things it knows how to do: explain how to deal, explain how to play, explain how to identify a win, and so on. In object-oriented programming terminology, actions that objects know how to perform are called *methods*, *services*, *functions*, *procedures*, or *operations*.

Note: It is unfortunate, but there is no single accepted terminology for the object-oriented paradigm. Each programming language or DBMS chooses which terms it will use. You therefore need to recognize all of the terms that might be used to describe the same thing.

An object is very security-minded. It typically keeps the things it knows about itself private and releases that information only through a method whose purpose is to share data values (an *accessor method*). For example, a user or program using one of the game objects cannot access the contents of the Dealing variable directly. Instead, the user or program must execute the How Do I Deal the Game? method to see that data.

Objects also keep private the details of the procedures for the things they know how to do, but they make it easy for someone to ask them to perform those actions. Users or programs cannot see what is inside the methods. They see only the result of the method. This characteristic of objects is known as *information hiding* or *encapsulation*.

An object presents a public interface to other objects that might use it. This provides other objects with a way to ask for data values or for actions to be performed. In the example of the solitaire games, the questions that Jane's little brother can ask are a game's public interface. The instructions below each question represent the procedure to be used to answer the question. A major benefit of data encapsulation is that as long as the object's public interface remains the same, you can change the details of the object's methods without needing to inform any other objects that might be using those methods. For example, the card game objects currently tell the user to "read" the contents of an attribute. However, there is no reason that the methods couldn't be changed to tell the user to "print" the contents of an attribute. The user would still access the method in the same way, but the way in which the method operates would be slightly different.

An object requests data or an action by sending a *message* to another object. For example, if you were writing a computer program to manage the instructions for solitaire games, the program (an object in its own right) could send a message to the game object asking the game object to display the instructions for dealing the game. Because the actual procedures of the method are hidden, your program would ask for the instruction display and then you would see the instructions on the screen. However, you would not need to worry about the details of how the screen display was produced. That is the job of the game object rather than the object that is asking the game to do something.

An object-oriented program is made up a collection of objects, each of which has attributes and methods. The objects interact by sending messages to one another. The trick, of course, is figuring out which objects a program needs and the attributes and methods those objects should have.

Classes

The template on which the solitaire game instructions are based is the same for each game. Without data, it might be represented as in Figure 27.7.

The template on which the solitaire game instructions are based is the same for each game. Without data, it might be represented as in Figure 27.7. The nice thing about this template is that it provides a consistent way of organizing all the characteristics of a game. When you want to create the instructions for another game, you make a copy of the template and "fill in the blanks." You write the data values for the attributes. The procedures that make up the answers to the questions someone might ask about the game have already been completed.

```
Information about the game
      Name:
      Illustration:
      Decks:
      Dealing:
      Playing:
      Winning:
Questions to Ask
      What is the name of the game?
            Read Name section.
      How many decks do I need?
            Read Decks section.
      What does the layout look like.
            Read Illustration section.
      How do I deal the game?
            Read Dealing section.
      How do I play the game?
            Read Playing section.
      How do I know when I've won?
            Read Winning section.
```

FIGURE 27.7 **The solitaire game instruction template.**

In object-oriented terminology, the template on which similar objects like the solitaire game objects are based is known as a *class*. When a program creates an object from a class, it provides data for the object's variables. The object can then use the methods that have been written for its class. All of the objects created from the same class share the same procedures for their methods. They also have the same types of data, but the values for the data may differ, for example, just as the names of the solitaire games are different.

A class is also a data type. In fact, a class is an implementation of what is known as an *abstract data type*, which is just another term for a user-defined data type. The implication of a class being a data type is that you can use a class as the data type of an attribute in a relation.

Suppose, for example, you were developing a class to handle data about the employees in your organization. The attributes of the class might include the employee ID, the first name, the last name, and the address. The address itself is made up of a street, city, state, and zip. Therefore, you would probably create an address class with those attributes and then, rather than duplicating those attributes in the employee class, simply indicate that an object of the employee class will include an object created from the address class to contain the employee's address.

Types of Classes

There are three major types of classes used in an object-oriented program:

- *Control classes*: Control classes neither manage data nor have visible output. Instead, they control the operational flow of a program. For example, *application classes* represent the programs themselves. In most cases, each program creates only one object from an application class. The application class's job includes starting the execution of the program, detecting menu selections (or other user interface events), and executing the correct program code to satisfy the user's requests.
- *Entity classes*: Entity classes are used to create objects that manage data. The solitaire game class, for example, is an entity class. Classes for people, tangible objects, and events (for example, business meetings) are entity classes. Most object-oriented programs have at least one entity class from which many objects are created. In fact, in its simplest sense, the object-oriented data model is built from the representation of relationships among objects created from entity classes.
- *Interface classes*: Interface classes handle the input and output of information. For example, if you are working with a graphic user interface, then each window and menu used by the program is an object created from an interface class.

In an object-oriented program, entity classes do not do their own input and output (I/O). Keyboard input is handled by interface objects that collect data and send it to entity objects for storage and processing. Screen and printed output is formatted by interface objects that get data for display from entity objects. When an entity object becomes part of a database, the DBMS takes care of the file I/O; the rest of the I/O is handled by application programs or DBMS utilities.

Why is it so important to keep data manipulation separate from I/O? Wouldn't it be simpler to let the entity object manage its own I/O? It might be simpler in the short run, but if you decided to change a screen layout, you would need to modify the entity class. If you keep them separate, then data manipulation procedures are independent of data display. You can change one without affecting the other. In a large program, this can not only save you a lot of time, but also help you avoid programming errors. In a database environment, the separation of I/O and data storage becomes especially critical because you do not want to modify data storage each time you decide to modify the look and feel of a program.

Many object-oriented programs also use a fourth type of class: a *container* class. Container classes exist to "contain," or manage, multiple objects created from the same class. Because they gather objects together, they are also known as *aggregations*. For example, if you had a program that handled the instructions for playing solitaire, then that program would probably have a container class that organized all the individual card game objects. The container class would keep the objects in some order, list them for you, and probably search through them as

individual card game objects. The container class would keep the objects in some order, list them for you, and probably search through them as well. Many pure object-oriented DBMSs require container classes, known as *extents*, to provide access to all objects created from the same class. However, as you will see, container classes are not used when objects are integrated into a relational database.

Types of Methods

Several types of methods are common to most classes, including the following:

- *Constructors*: A constructor is a method that has the same name as the class. It is executed whenever an object is created from the class. A constructor, therefore, usually contains instructions to initialize an object's variables in some way.
- *Destructors*: A destructor is a method that is executed when an object is destroyed. Not all object-oriented programming languages support destructors, which are usually used to release system resources (for example, main memory allocated by the object). Java, in particular, does not use destructors because it cleans up memory by itself.
- *Accessors*: An accessor, also known as a *get method*, returns the value of a private attribute to another object. This is the typical way in which external objects gain access to encapsulated data.
- *Mutators*: A mutator, or *set method*, stores a new value in an attribute. This is the typical way in which external objects can modify encapsulated data.

The remaining methods defined for a class depend on the specific type of class, and the specific behaviors it needs to perform.

Method Overloading

One of the characteristics of a class is its ability to contain *overloaded* methods, methods that have the same name but require different data to operate. Because the data are different, the public interfaces of the methods are distinct.

As an example, assume that a human relations program has a container class named AllEmployees that aggregates all objects created from the Employee class. Programs that use the AllEmployees class create one object from the class and then relate all employee objects to the container using some form of program data structure.

To make the container class useful, there must be some way to locate specific employee objects. You might want to search by the employee ID number, by first and last name, or by telephone number. The AllEmployees class, therefore, contains three methods named "find." One of the three requires an integer (the employee number) as input, the second requires two strings (the first and last name), and the third requires a single string (the phone number). Although the methods have the same name, their public interfaces are different because the combination of the name and the required input data is distinct.

Many classes have overloaded constructors. One might accept interactive input, another might read input from a file, and a third might get its data by

copying data from another object (a *copy constructor*). For example, most object-oriented environments have a Date class that supports initializing a date object with three integers (day, month, year), the current system date, another Date object, and so on.

The benefit of method overloading is that the methods present a consistent interface to the programmer. In the case of our example of the AllEmployees container class, whenever a programmer wants to locate an employee, he or she knows to use a method named "find." Then the programmer just uses whichever of the three types of data he or she happens to have. The object-oriented program locates the correct method by using the entire public interface (its *signature*), made up of the name and the required input data.

Class Relationships

The classes in an object-oriented environment aren't always independent. The basic object-oriented paradigm has two major ways to relate objects, distinct from any logical data relationships that might be included in a pure object-oriented database: inheritance and composition.

Inheritance

As a developer or database designer is working on an object-oriented project, he or she may run into situations where there is a need for similar—but not identical—classes. If these classes are related in a general to specific manner, then the developer can take advantage of one of the major features of the object-oriented paradigm, known as *inheritance*.

Inheriting Attributes

To see how inheritance works, assume that you are writing a program (or developing a database) to manage a pet shop. One of the entity classes you will use is Animal, which will describe the living creatures sold by the shop. The data that describe objects created from the Animal class include the English and Latin names of the animal, the animal's age, and the animal's gender. However, the rest of the data depend on what type of animal is being represented. For example, for reptiles, you want to know the length of the animal, but for mammals, you want to know the weight. And for fish, you don't care about the weight or length, but you do want to know the color. All the animals sold by the pet shop share some data, yet have pieces of data that are specific to certain subgroups.

You could diagram the relationship as in Figure 27.8. The Animal class provides the data common to all types of animals. The subgroups—Mammals, Reptiles, and Fish—*add* the data specific to themselves. They don't need to repeat the common data because they *inherit* them from Animal. In other words, Mammals, Reptiles, and Fish all include the four pieces of data that are part of Animal.

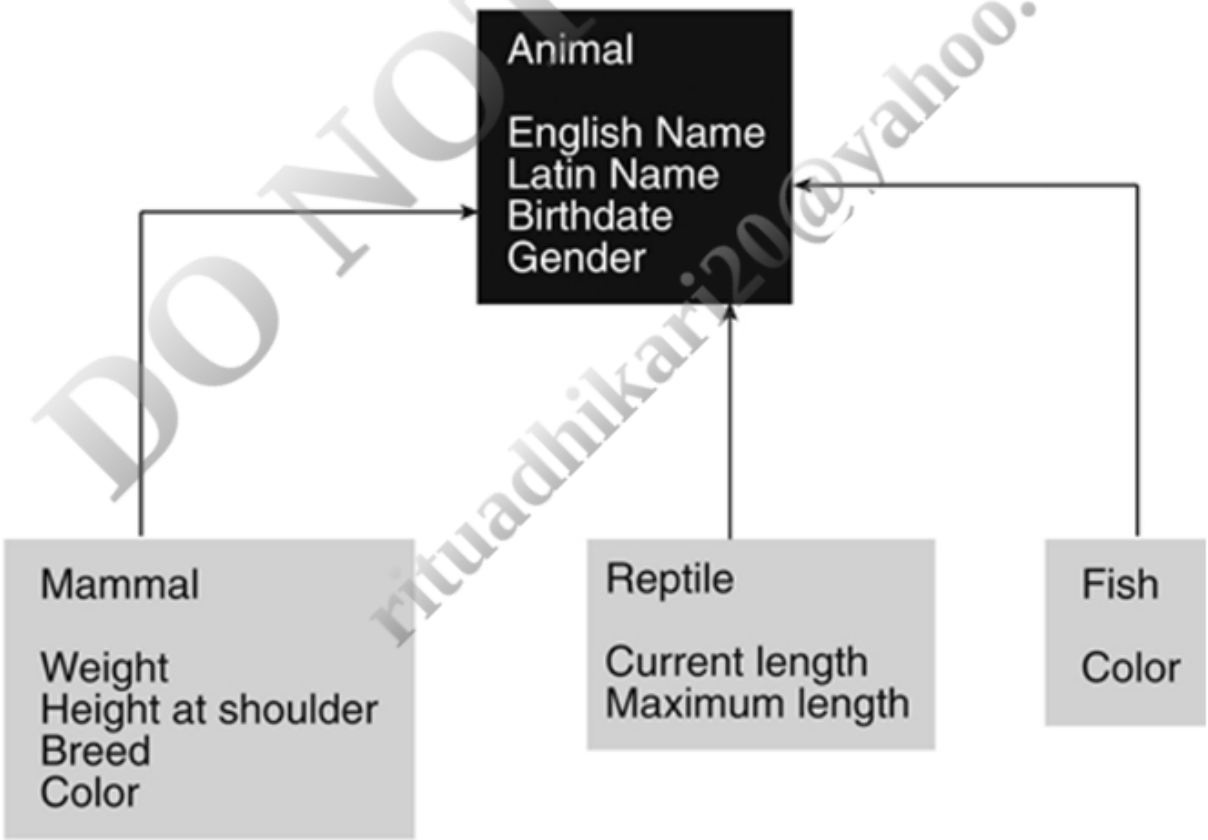


FIGURE 27.8 **The relationship of classes for an object-oriented environment for a pet shop.**

If you look closely at Figure 27.8, you'll notice that the lines on the arrows go from the subgroups to Animal. This is actually contrary to what is happening: The data from Animal are flowing down the lines into the subgroups. Unfortunately, the direction of the arrows is dictated by convention, even though it may seem counterintuitive.

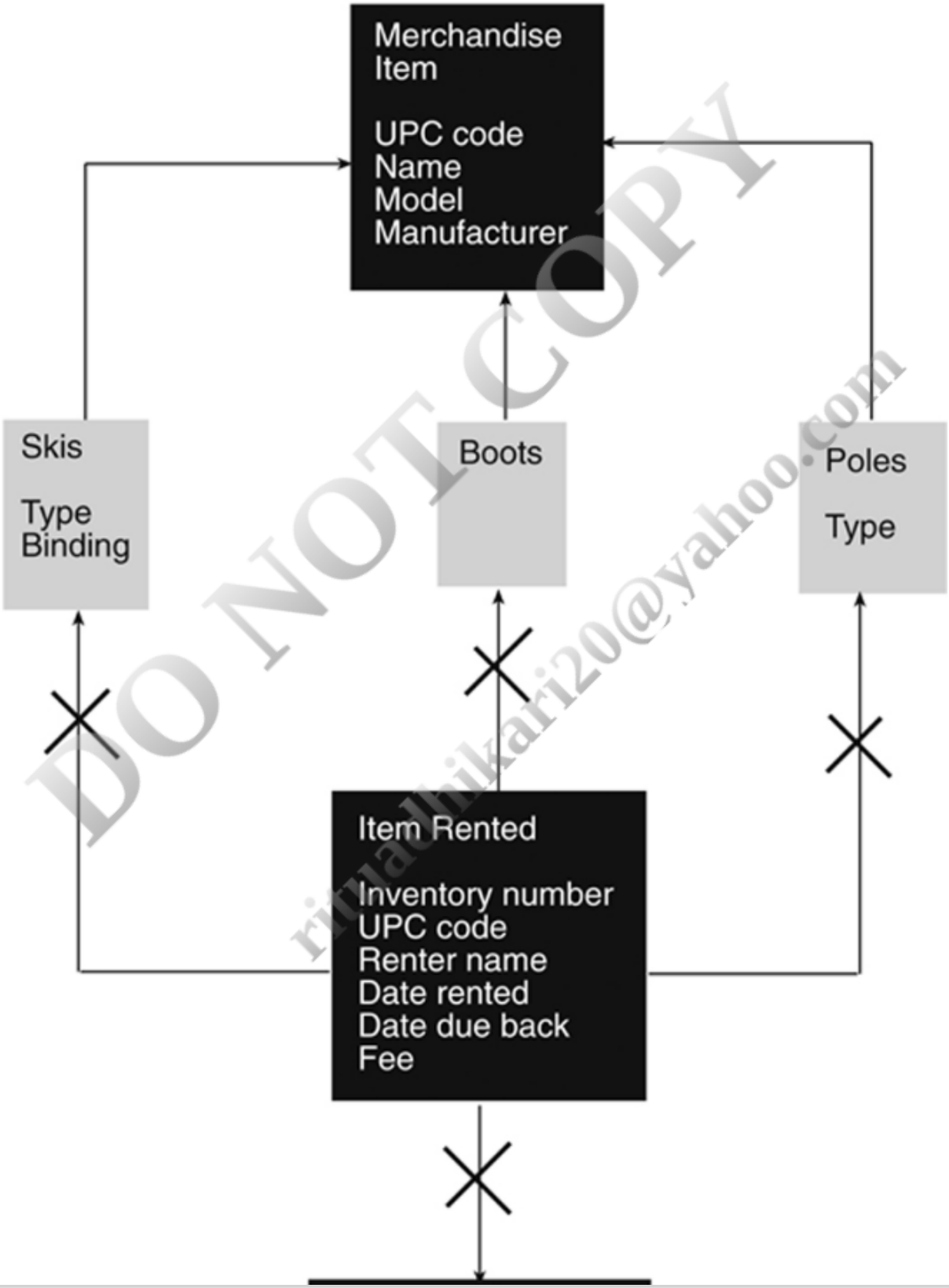
In object-oriented terminology, the subgroups are known as *subclasses* or *derived classes*. The Animal class is a *superclass* or *base class*.

The trick to understanding inheritance is to remember that subclasses represent a more specific occurrence of their superclass. The relationships between a base class and its derived classes therefore can be expressed using the phrase "is a":

between a base class and its derived classes therefore can be expressed using the phrase “is a”:

- A mammal is an animal.
- A reptile is an animal.
- A fish is an animal.

If the “is a” phrasing does not make sense in a given situation, then you are not looking at inheritance. As an example, assume that you are designing an object-oriented environment for the rental of equipment at a ski rental shop. You create a class for a generic merchandise item, and then subclasses for the specific types of items being rented, as in the top four rectangles in Figure 27.9. Inheritance works properly here because skis are a specific type of merchandise item, as well as boots and poles.



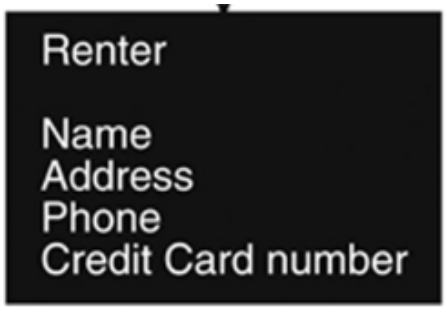
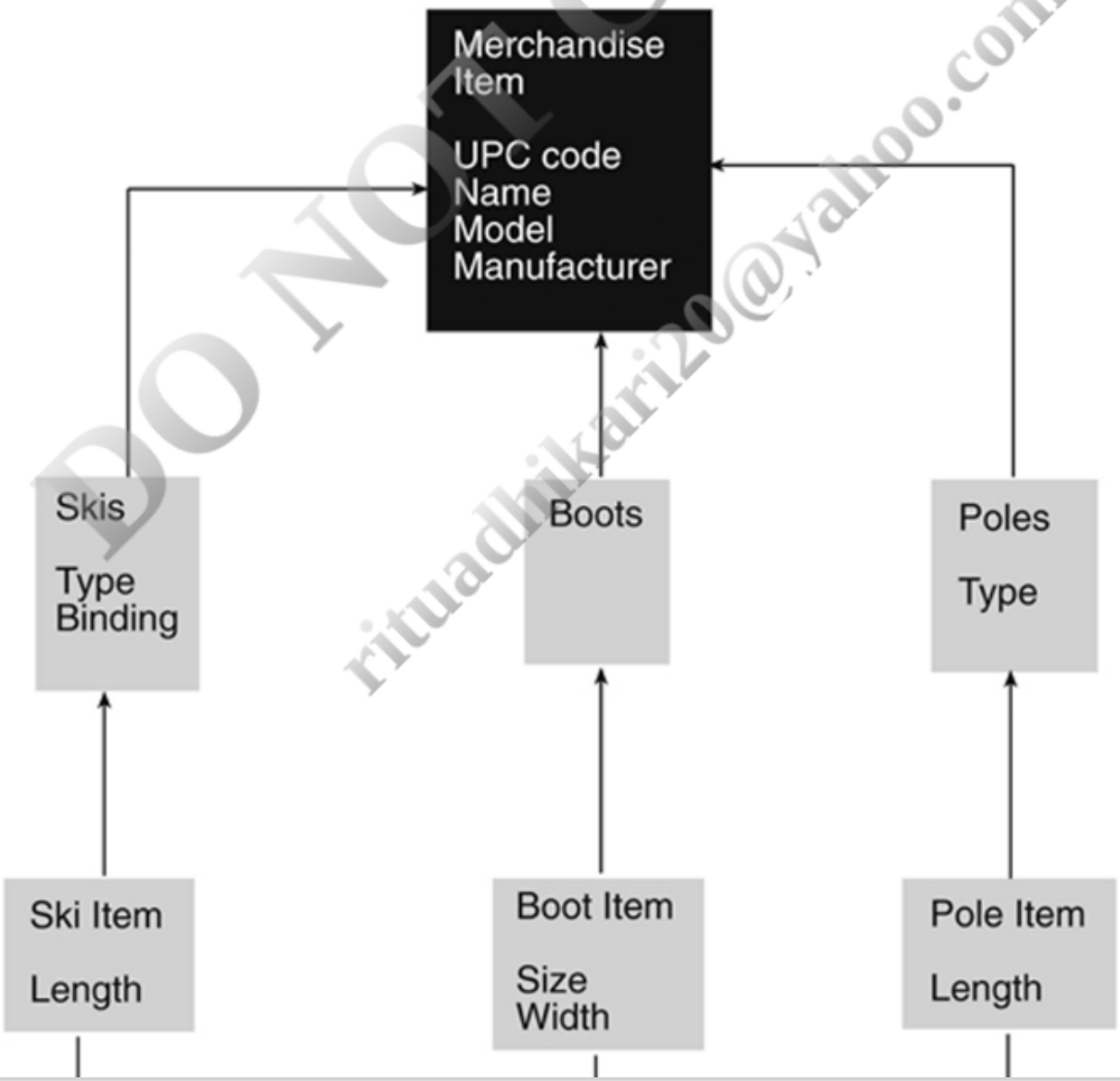


FIGURE 27.9 **Inheritance and no inheritance in an object-oriented environment for a ski equipment rental.**

However, you run into trouble when you begin to consider the specific items being rented and the customer doing the renting (the renter). Although there is a logical database-style relationship between a renter and an item being rented, inheritance does not work because the "is a" test fails. A rented item is not a renter!

The situation with merchandise items and rental inventory is more complex. The Merchandise Item, Skis, Boots, and Poles classes represent descriptions of types of merchandise, but not physical inventory. For example, the ski shop may have many pairs of one type of ski in inventory and many pairs of boots of the same type, size, and width. Therefore, what is being rented is individual inventory items, represented by the Item Rented class. A given inventory item is either skis, boots, or poles. It can only be *one*, not all three as shown in Figure 27.9. Therefore, an item rented is not a pair of skis, a pair of boots, and a set of poles. (You also have the problem of having no class that can store the size or length of an item.)

The solution to the problem is to create a separate "rented item" class for each type of merchandise, as in Figure 27.10. When you are looking at this diagram, be sure to pay attention to the direction of the arrows. The physical layout of the diagram does not correspond to the direction of the inheritance. Remember that, by convention, the arrows point from the derived class to the base class.



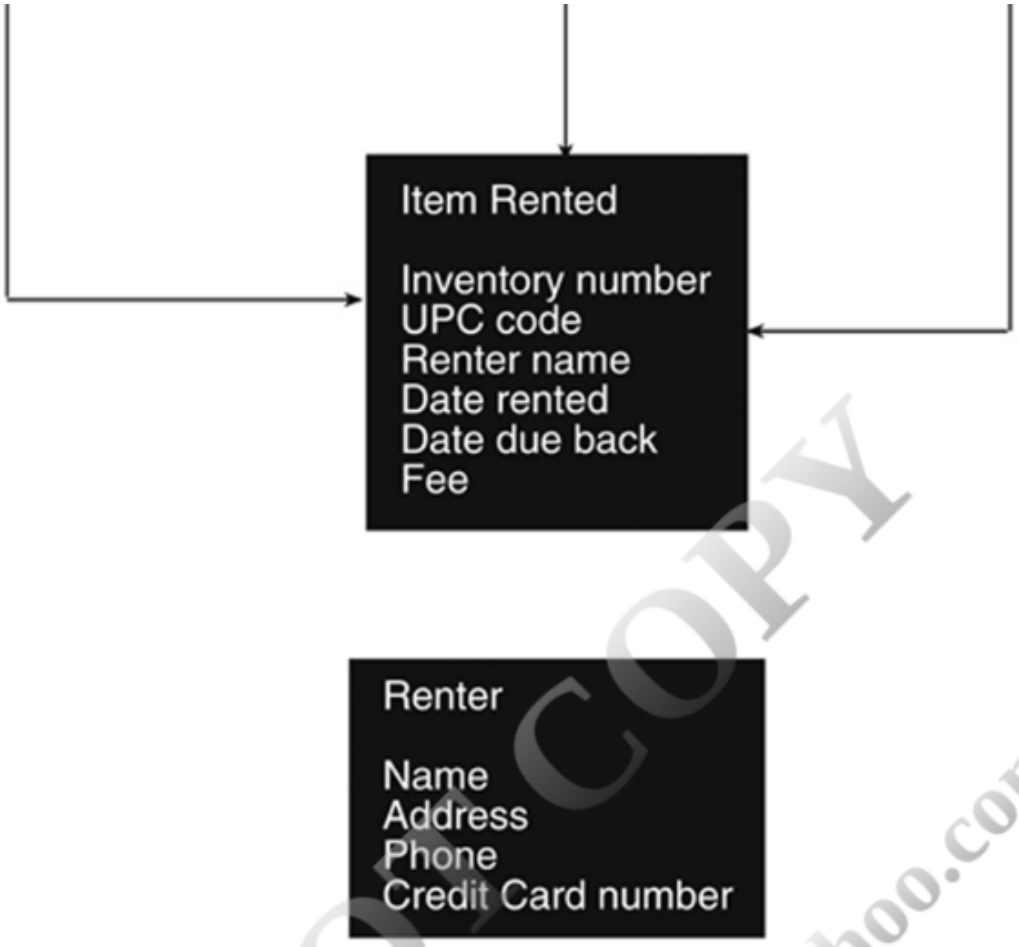


FIGURE 27.10 **Multiple inheritance in the data environment for a ski shop.**

The Ski Item class inherits information about the type of item it is from the Skis class. It also inherits information about an item being rented from the Item Rented class. A ski item "is a" pair of skis; a ski item "is a" rented item as well. Now the design of the classes passes the "is a" test for appropriate inheritance. (Note that it also gives you a class that can contain information such as the length and size of a specific inventory item.) The Renter class does not participate in the inheritance hierarchy at all.

Multiple Inheritance

When a class inherits from more than one base class, you have *multiple inheritance*. The extent to which multiple inheritance is supported in programming languages and DBMSs varies considerably from one product to another.

Abstract Classes

Not every class in an inheritance hierarchy is necessarily used to create objects. For example, in Figure 27.10, it is unlikely that any objects are ever created from the Merchandise Item or Item Rented classes. These classes are present simply to provide the common attributes and methods that their derived classes share.

Such classes are known as *abstract*, or *virtual*, classes. In contrast, classes from which objects are created are known as *concrete* classes.

Note: Many computer scientists use the verb "instantiate" to mean "creating an object from a class." For example, you could say that abstract classes are never instantiated. However, I find that term rather contrived (although not quite as bad as saying "we will now motivate the code" to mean we will now explain the code) and prefer to use the more direct "create an object from a class."

Inheriting Methods: Polymorphism

In general, methods are inherited by subclasses from their superclass. A subclass can use its base class's methods as its own. However, in some cases, it may not be possible to write a generic method that can be used by all subclasses. For example, assume that the ski rental shop's Merchandise Item class has a method named printCatalogEntry, the intent of which is to print a properly formatted entry for each distinct type of merchandise item. The subclasses of Merchandise Item, however, have attributes not shared by all subclasses and the printCatalogEntry method therefore must work somewhat differently for each subclass.

To solve the problem, the ski rental shop can take advantage of *polymorphism*, the ability to write different bodies for methods of the same name that belong to classes in the same inheritance hierarchy. The Merchandise Item class includes a *prototype* for the printCatalogEntry method, indicating just the method's pubic interface. There is no body for the method, no specifications of how the method is to perform its work (a *virtual method*). Each subclass then redefines the method, adding the program instructions necessary to execute the method.

The beauty of polymorphism is that a programmer can expect methods of the same name and same type of output for all the subclasses of the same base class. However, each subclass can perform the method according to its own needs. Encapsulation hides the details from all objects outside the class hierarchy.

Note: It is very easy to confuse polymorphism and overloading. Just keep in mind that overloading applies to methods of the same class that have the same name but different signatures, whereas polymorphism applies to several subclasses of the same base class that have methods with

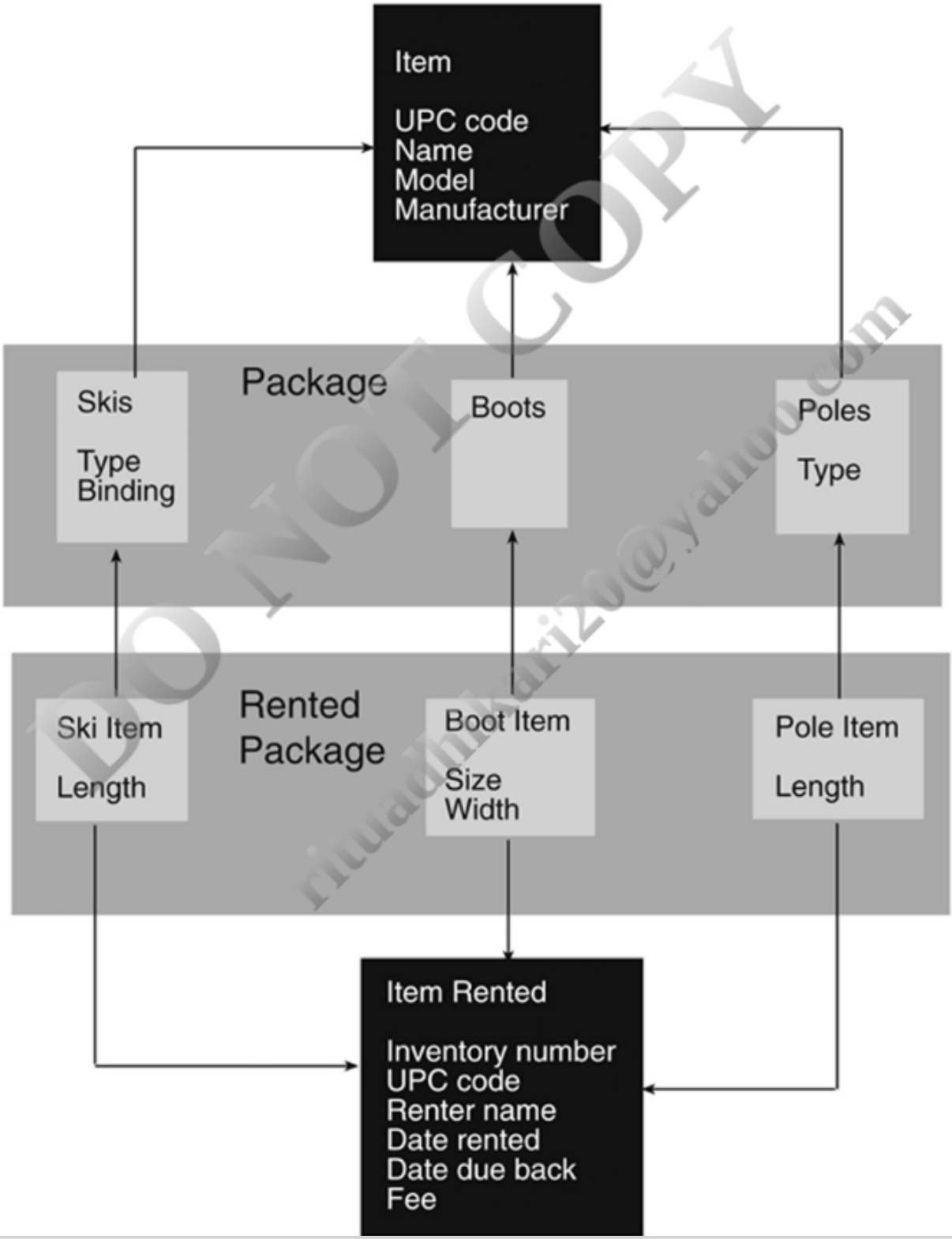
the same signature but different implementations.

Composition

Inheritance can be described as a general–specific relationship. In contrast, *composition* is a whole–part relationship. It specifies that one class is a component of another and is often read as "has a."

To help you understand how composition can be used, let's assume that the ski rental shop wants to offer packages of items for rent (skis, boots, and poles). The packages will come in three qualities—good, better, and best—based on the retail value of the items in the package.

As you can see in Figure 27.11, each package contains three types of merchandise items, so the package class "has a" boot, "has a" pole, and "has a" ski. An object created from this class would be used to indicate which types of items could be rented as a bundle. In contrast, the rented package class contains actual rental items and therefore indicates which specific inventory items have been rented together.



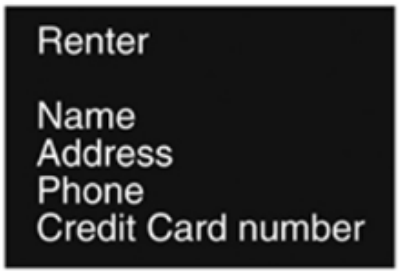


FIGURE 27.11 **Composition.**

Some pure object-oriented DBMSs take composition to the extreme. They provide simple data types such as integers, real numbers, characters, and Booleans. Everything else in the database—even strings—is built by creating classes from these simple data types, using those classes to build more complex classes, and so on.

Benefits of Object-Orientation

There are several reasons why the object-oriented paradigm has become so pervasive in programming. Among the perceived benefits are the following:

- An object-oriented program consists of modular units that are independent of one another. These units can therefore be reused in multiple programs, saving development time. For example, if you have a well-debugged employee class, you can use it in any of your business programs that require data about employees.
- As long as a class's public interface remains unchanged, the internals of the class can be modified as needed without requiring any changes to the programs that use the class. This can significantly speed up program modification. It can also make program modification more reliable, as it cuts down on many unexpected side effects of program changes.
- An object-oriented program separates the user interface from data handling, making it possible to modify one independent of the other.
- Inheritance adds logical structure to a program by relating classes in a general to specific manner, making the program easier to understand and therefore easier to maintain.

Where Objects Work Better Than Relations

There are some database environments—especially those involving a great deal of inheritance—in which object-orientation is easier to implement than a relational design. To see why this is so, let's look at a hobby environment that just happens to be one of the best examples of the situation in question that I've ever encountered.

The database catalogs Yu-Gi-Oh cards (one of those animé related trading card games). The collector for whom this database and its application were developed has thousands of cards, some of which are duplicates. They are stored in three binders. Within each binder there may be several sections; the pages are numbered within each section.

There are three major types of cards: monsters, spell, and traps. The monster card in Figure 27.12 is fairly typical. Notice that it has a title, an "attribute" at the top right, a "level" (count the circles below the name), an "edition" (first, limited, or other), a set designation, a type (and optionally two subtypes), a description, attack points, and defense points. At the bottom left, there may be a code number, which is missing from some of the early cards.

FIGURE 27.12 A typical Yu-Gi-Oh monster card.

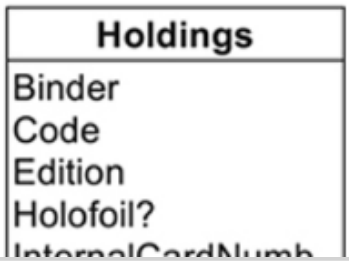
A card with the same name may appear in many sets and the same set may have more than one card of the same name. What distinguishes them is their "rarity," determined by how the title is printed (black or white for common cards and silver or gold for rare cards) and how the image is printed (standard color printing or holofoil printing). There are a variety of combinations of printing to generate common, rare, super rare, ultra rare, and ultimate rare cards.

Note: If you want an interesting challenge before you see the relational design for this database, try to figure out the primary key for a card!

Most cards can be used for game play, but some have been banned from specific types of games. Others have caveats ("rulings") attached to them by the game's governing board that affect how the card can be used in a game.

Spell cards that, as you might expect, can be used in the game to cast spells, share a number of attributes with the monster card, but don't have things such as type and subtypes. The third type of card, a trap, also shares some attributes with monsters, but is missing others and has a property that is unique to this type of card. Spells also have properties, but the list of possible properties differs between spells and traps.

You can find an ER diagram for the card database in Figure 27.13. As you might have guessed, there is an entity for the card, which has three subclasses, one for each specific type of card. There are also many holdings for each card.



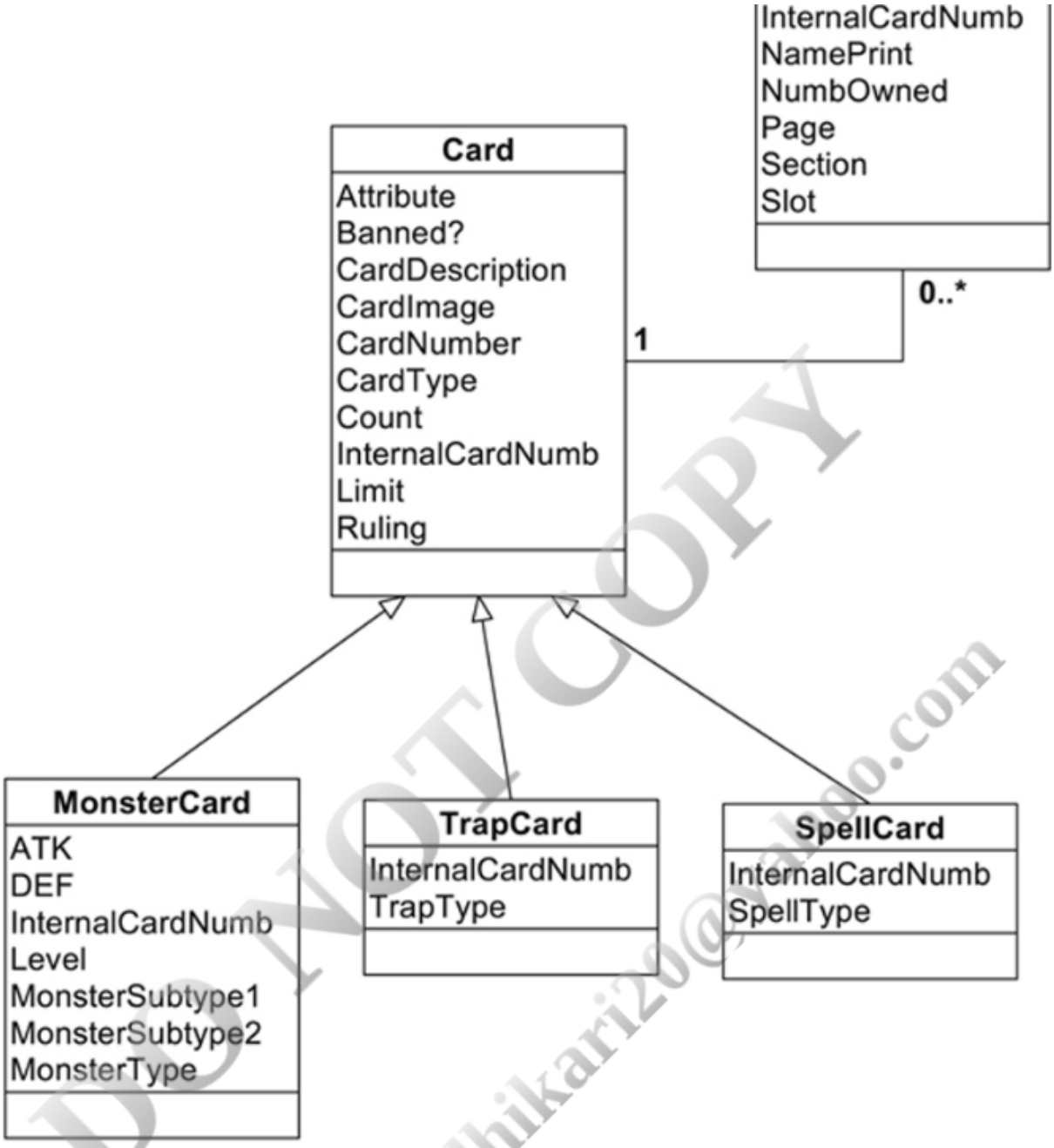


FIGURE 27.13 The ER diagram for the card database.

To design the relational database, we create one relation for each entity, including the superclass (in this example, Card) and its subclasses (Monster card, Trap card, and Spell card). With an object-oriented DMBS, we would create objects only from the subclasses; no object would ever be created from the superclass. The subclasses "inherit" the attributes of their parent. For the relational database, we have to do that manually, using some type of primary key–foreign key relationship to connect the subclass tables to the parent table. Differences in where cards of a given name appear and how they are printed are handled by Holdings. Therefore, the design of the card database looks like this:

```
Card (InternalCardNUMB, Attribute, Banned?, CardDescription?,
     CardImage, CardName, CardNumber, CardType, Count, Limit,
     Ruling)

Monster card (InternalCardNumb, ATK, DEF, Level,
     MonsterSubtype1, MonsterSubtype2, MonsterType)

Trap card (InternalCardNumb, TrapType)
```

```
Spell card (InternalCardNumb, SpellType)

Holdings (InternalCardNumb, Code, Edition, Holofoil?,
     NamePrint, NumberOwned, Binder, Page, Section, Slot)
```

Why have both the trap and spell card relations if they have exactly the same attributes? At the current time, they could certainly be maintained in one relation. However, there are several reasons to keep them separate. First, there is no way to guarantee that they will always have the same attributes. If they are separated from the start, it will be easier to add attributes to one or the other if needed at some later date.

Second, the major reason this type of design doesn't perform as well as it might is because the details about a card always need to be looked up in another relation, joining on the internal card number. If we keep spell and trap data separate, the relations will remain smaller, and the joins will perform better.

Note: Here's the answer to the primary key challenge: A Holding actually represents one or more physical cards in the inventory. It has a name

(represented by the internal card number) and a set designation. When cards of the same name are printed in different ways in the same set, they have different set designations. Therefore, the concatenation of the internal card number and the set designation uniquely identifies a card (although not a physical card in the inventory, given that there may be duplicate cards in the inventory). The only other alternative is to assign unique inventory numbers to each physical card and to use them. For some collectors, this may make sense, given that they would want to track the condition of each and every card.

There is an alternative design for this type of database: Rather than use one relation for each entity, create only a single relation in which many attributes may be null, depending on the type of card. Such a relation might look like this:

```
Card (InternalCardNumb, Attribute, Banned?, CardDescription,
      CardImage, CardName, CardNumber, CardType, Count,
      Limit, Ruling, ATK, DEF, Level, MonsterSubtype1,
      MonsterSubtype2, MonsterType, TrapType, spellType)
```

The CardType attribute indicates which of the type-specific attributes should have data. For example, if CardType contained "M" for Monster, you would expect to find data in the ATK, DEF, and level attributes but not the spell type or trap type. The supposed benefit of this design is that you avoid the joins to combine the separate relations of the earlier design. However, when a DBMS retrieves a row from this relation, it pulls in the entire row, empty fields and all. Practically, in a performance sense, you haven't gained much, and you're stuck with a design that can waste disk space.

Note: Personally, I prefer the multiple relation design because it's cleaner, wastes less space, and is much more flexible as the design of the relations need to change over time.

A pure-object oriented design for the same database would include the five entity classes, although the Card class would be an abstract class. It would also include a class to aggregate all objects created from subclasses of the Card class, letting users handle all cards, regardless of type, as a single type of object. The nature of these data—the major need for inheritance—suggests that an object-oriented database may well perform better than a relational database.

Limitations of Pure Object-Oriented DBMSs

When object-oriented DBMSs first appeared in the 1980s, some people predicted that they would replace relational DBMSs. That has not occurred for a number of reasons:

- Not all database environments can be represented efficiently by an object-oriented design.
- When implemented in a DBMS, object-oriented schemas are significantly more difficult to modify than relational schemas.
- There are no standards for object-oriented DBMSs, which means that each product has its own way of querying a database.
- Most object DBMSs do not have interactive query languages, which means that there is little support for ad-hoc queries.
- Because there are no standards for object-oriented DBMSs, moving from one DBMS to another often means redoing everything, from the design to application programs.

The Object-Relational Data Model

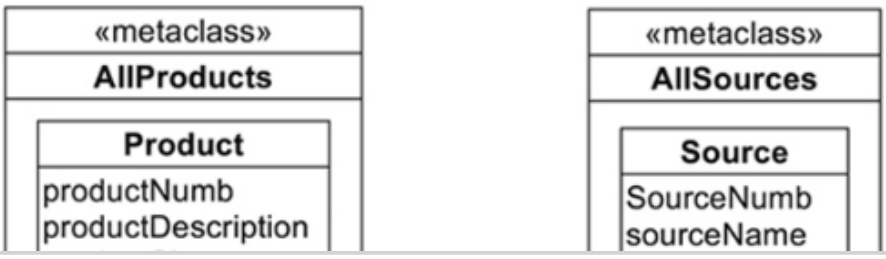
The *object-relational* (OR) data model—one of those known as *post-relational*—is a combination of the relational data model and some of the object-oriented concepts that—in the opinion of some database theorists and users—make up for shortcomings in the relational data model. The purpose of this discussion is to help you understand how OR designs differ from pure relational designs. With that in hand, you will be able to understand the strengths and weaknesses of SQL's support for object-related structures that are discussed later in this chapter.[1]

ER Diagrams for Object-Relational Designs

The Information Engineering, or IE, type of ER diagram does not lend itself to the inclusion of objects because it has no way to represent a class. Therefore, when we add objects to a relational database, we have to use another ERD style.

Although there are many techniques for object-oriented ERDs, one of the most commonly used is the Unified Modeling Language (UML). When used to depict a post-relational database design, UML looks a great deal like the IE style, but indicates relationships in a different way.

An example of an ER diagram using UML can be found in Figure 27.14. This design is of a purely object-oriented database and includes some elements that therefore won't appear in a hybrid design. It has been included here to give you an overview of UML so that you can better understand the portions of the modeling tool that we will be using later in this chapter.



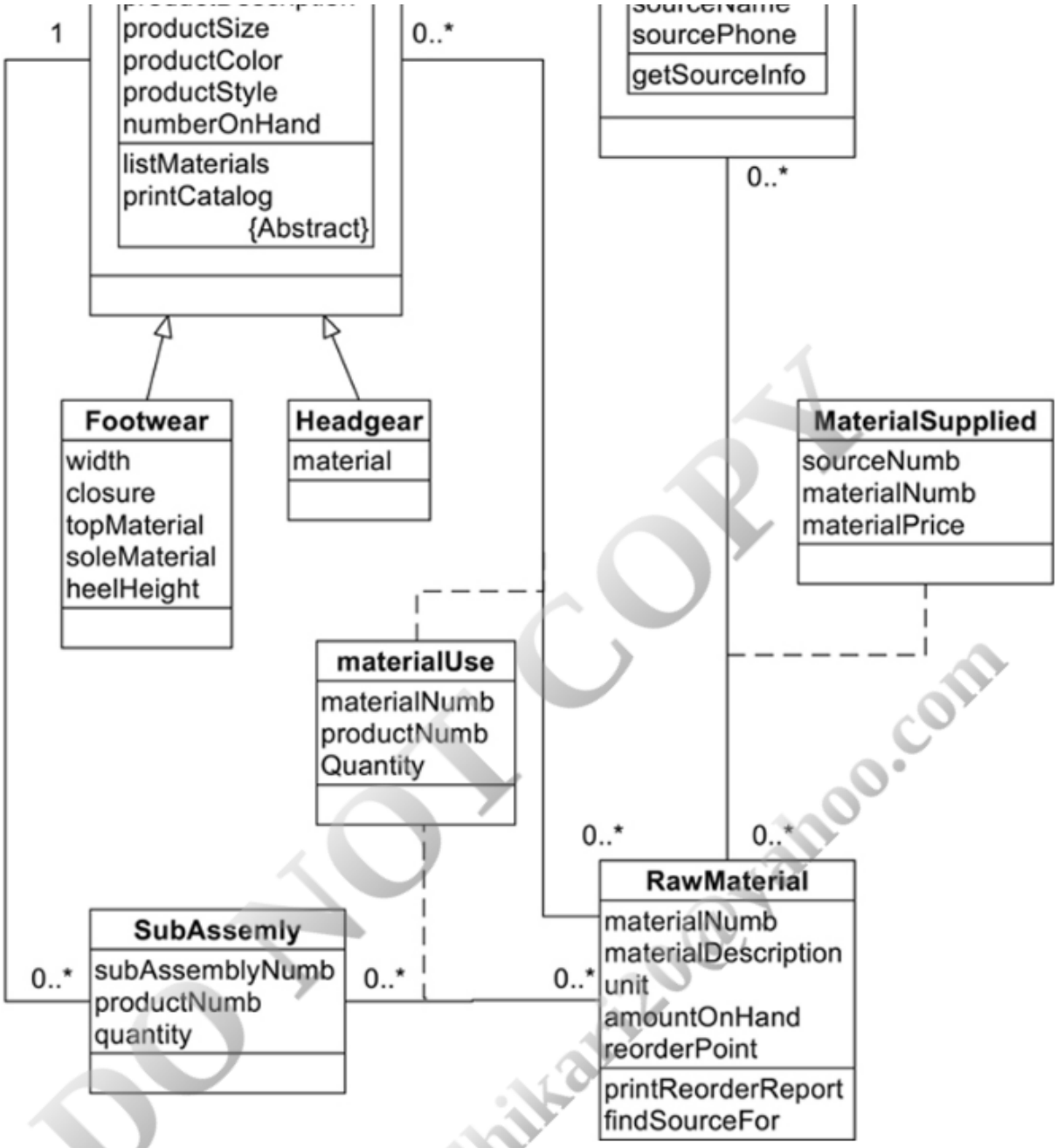


FIGURE 27.14 **An object-oriented database design using UML.**

The basic features of UML include the following:

- A regular class is represented by a rectangle, divided into three parts (name, attributes, procedures).
- An aggregate class (a *metaclass* in the diagram)—a class that collects all objects of a given class—is represented by a rectangle containing its name and the rectangles of the classes whose objects it aggregates. For example, in Figure 27.14, the Product and Source classes are within their aggregate classes, AllProducts and AllSources, respectively.
- Relationships between entities are shown with lines with plain ends. The cardinality of a relationship is represented as *n*, *n*..*n*, or *n*..*. For example, if the cardinality is 1, it is simply written as 1. If the cardinality is 0 or more, it appears as 0..*; 1 or more appears as 1..*. Notice in Figure 27.14 that there are several direct many-to-many relationships, shown with 0..* at either end of the association line.
- Inheritance is shown by a line with an open arrow pointing toward the base class. In Figure 27.14, the Footwear and Headgear classes have such arrows pointing toward Product.
- What we call composite entities in a relational database are known as *association classes*. They are connected to the relationship to which they apply with a dashed line. As you can see in Figure 27.14, the MaterialSupplied and MaterialUse classes are each connected to at least one many-to-many relationship by the required dashed line.

In addition to the basic features shown in Figure 27.14, UML diagrams can include any of the following:

- An attribute can include information about its visibility (public, protected, or private), data type, default value, and domain. In Figure 27.15, for example, you can see four classes and the data types of their attributes. Keep in mind that, in an object-oriented environment, data types can be other classes. Therefore, the Source class uses an object of the TelephoneNumber class for its phoneNumber attribute, and an object of the Address class for its sourceAddress attribute. In turn, Source, Address, and TelphoneNumber all contain attributes that are objects of the String class.
- Procedures (officially called *operations* by UML) can include their complete program signature and return data type. If you look at

Figure 27.15, for example, you can see each operation's name followed by the type of data it requires to perform its job (*parameters*). Together, the procedure's name and parameters make up the procedure's signature. If data are returned by the operation, then the operation's signature is followed by a colon and the data type of the return value, which may be an object of another class or a simple data type such as an integer.

- Solid arrows can be used at the end of associations to indicate the direction in which a relationship can be navigated.

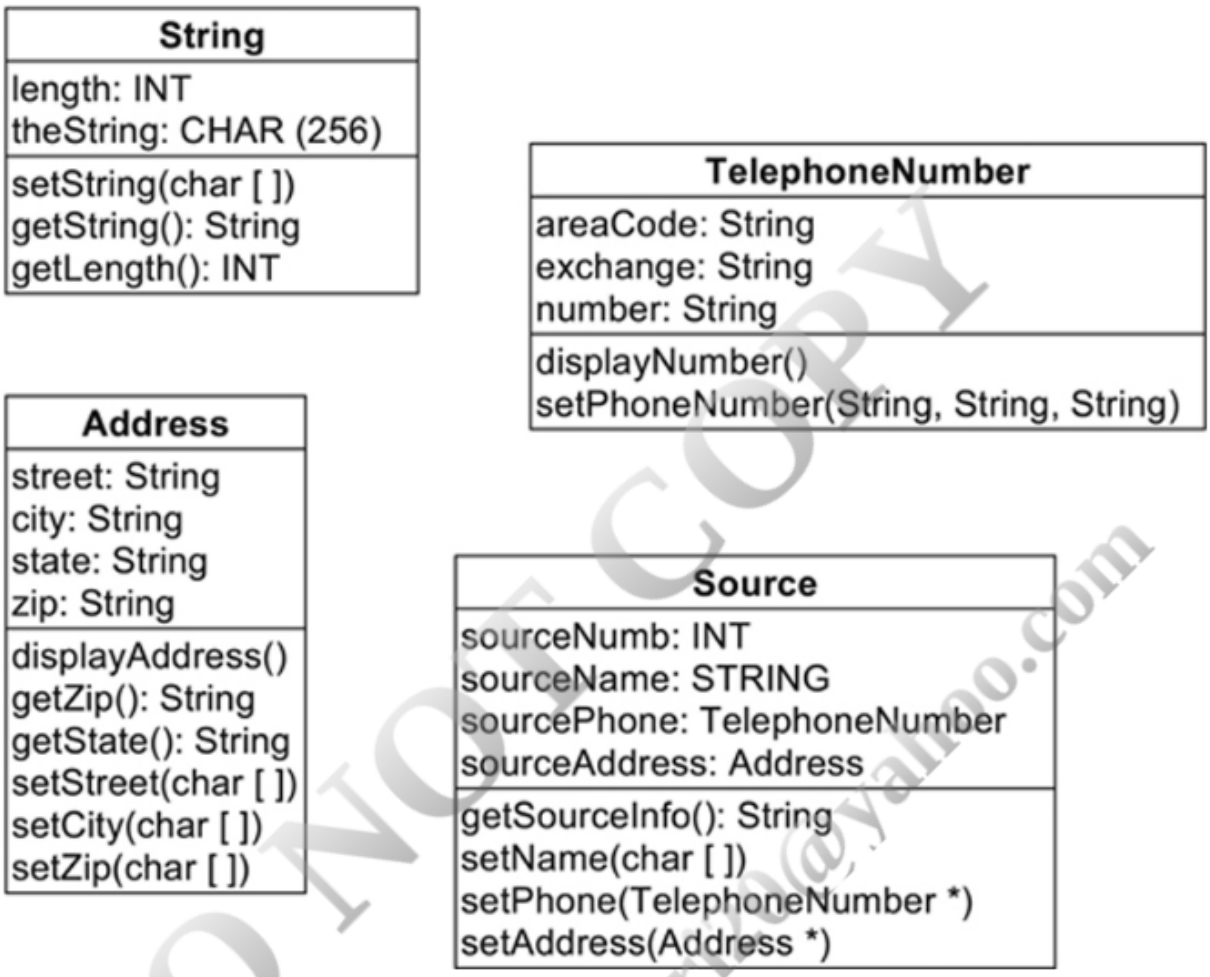


FIGURE 27.15 **UML classes showing attribute data types.**

Note: Pure object-oriented databases are navigational, meaning that traversal through the database is limited to following predefined relationships. Because of this characteristic, some theorists feel that the object-oriented data model is a step backwards rather than forward and that the relational data model continues to have significant advantages over any navigational data model.

There are three possible ways to use the arrows:

- ❑ Use arrows on the ends of all associations where navigation is possible. If an association has a plain end, then navigation is not possible in that direction. This would indicate, for example, a relationship between two objects that is not an inverse relationship, where only one of the two objects in a relationship contains the object identifier of a related object.
- ❑ Show no arrows at all, as was done in Figure 27.15. In that case, the diagram provides no information about how the database can be navigated.
- ❑ Show no arrows on associations that can be navigated in both directions, but use arrows on associations that can be navigated in only one direction. The drawback to this approach is that you cannot differentiate associations that can be navigated in both directions from associations that cannot be navigated at all.

- An association that ends in a filled diamond indicates a whole–part relationship. For example, if you were representing a spreadsheet in a database, the relationship between the spreadsheet and its cells could be diagrammed as in Figure 27.16. The filled diamond can also be used to show aggregation, instead of placing one object within another, as was done in Figure 27.14.
- When an association is between more than two objects, UML uses a diamond to represent the relationship. If an association is present, it will be connected to the diamond, as in Figure 27.17. The four classes in the illustration represent entities from a poetry reading society's database. A "reading" occurs when a single person reads a single poem that was written by one or more poets. The association entity indicates when and where the reading took place.

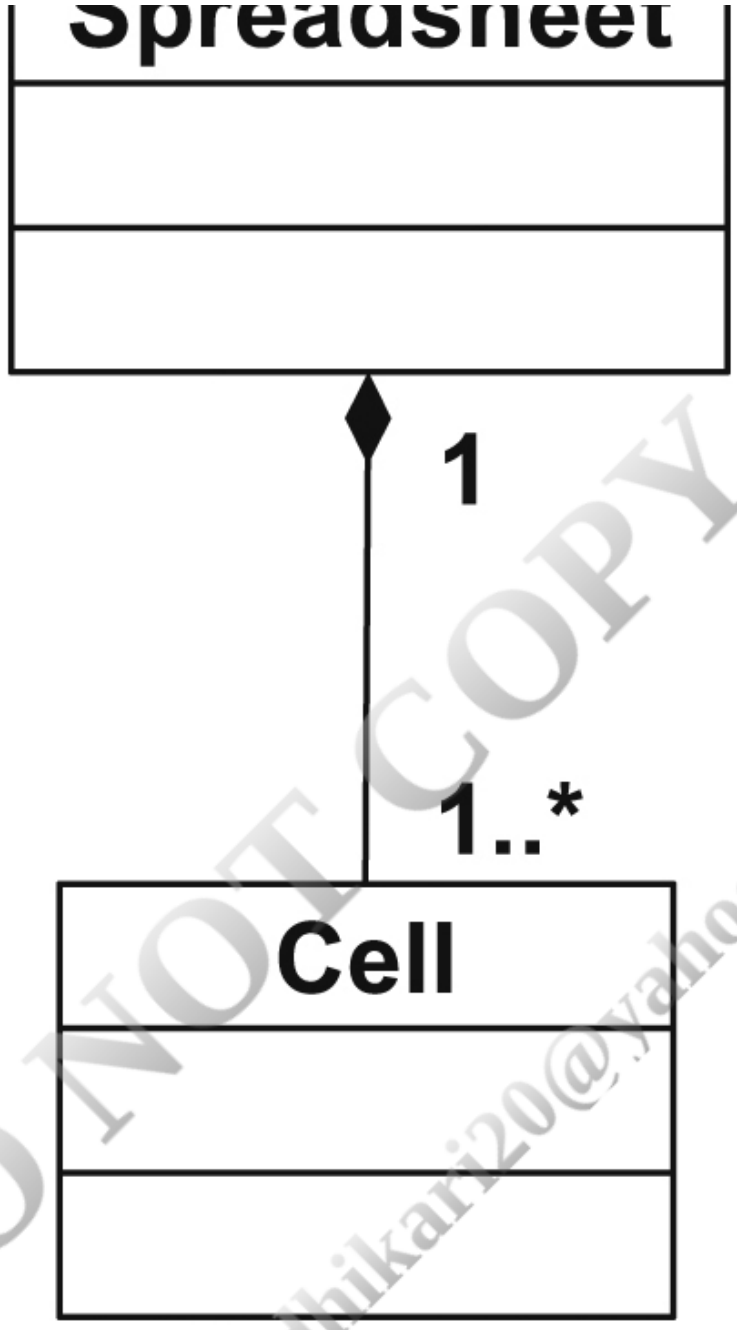


FIGURE 27.16 A UML representation of a whole–part relationship.

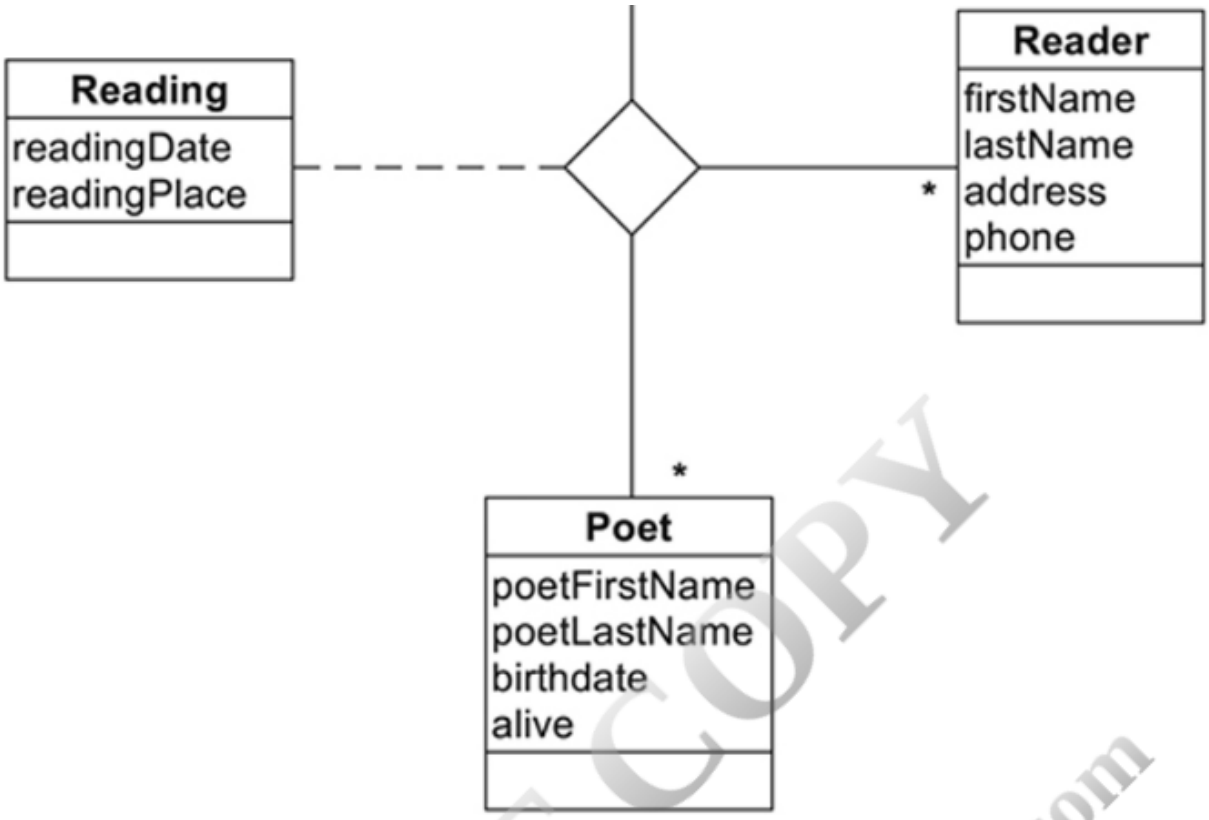


FIGURE 27.17 The UML representation of a relationship between more than two classes.

Features of the OR Data Model

There is no accepted standard for the object-relational data model. However, a commonly used model is based on the elements supported by recent SQL standards. As you will see, these features violate many of the rules applied to relational databases:

- A relational database should have no data structures other than tables. An OR database, however, allows an attribute to support a reference to a row in another table. These references are the internal object identifiers used by OO databases described earlier in this chapter. An OR database can use the relational concept of a primary key–foreign key relationship to indicate entity relationships. However, it is not required; references to rows can be used instead. The advantage to using row references rather than the relational method is improved performance because the joins needed to follow data relationships are unnecessary. The major drawback to using row references is that the integrity of the relationships can't be verified by the DBMS; referential integrity requires both primary and foreign key values.
- A relational database is limited to one value at the intersection of a single column and row. An OR database, however, can store more than one value in the same location. The values can be an array of the same type of data, a row of data (much like a table within a table), an unordered collection of data of different data types, or an entire object.
- Classes are implemented as *user-defined data types* (UDTs). A new UDT may inherit from an existing UDT, although multiple inheritance is not allowed. A UDT will have default accessor and mutator methods, as well as a default constructor, each of which can be overridden by a database programmer. There is nothing in the relational data model that prohibits UDTs. However, to be used in a relational database, a custom data type must hold only a single value.
- UDTs may have methods defined with them. Methods may be overloaded. Polymorphism is supported. Relational databases, however, have no concept of storing procedures with data.

SQL Support for the OR Data Model

The SQL:2003 standard introduced a variety of object-relational features. Although not all relational DBMSs support this part of the standard, SQL provides four column data types for OR storage, as well as support for UDTs. You will find at least some OR features in most of today's major DBMSs.

Note: There are some people who cling to the pure relational data model like a lifeline. However, in practice there is nothing that requires you to avoid SQL's OR features. If those features can help model your database environment, then those designing your database shouldn't be afraid to use them. Just be aware of the referential integrity issues that can arise when you store more than one piece of data in a single column in a single row.

Note: Some of the OR features covered in this chapter require programming. In those instances, this chapter assumes that you have programming experience. If you don't know how to program, then you can skim over that material.

An Additional Sample Database

For some of the examples in this chapter, we will be working with a classic home computer application: recipes. You can find the ERD in Figure 27.18. (It has been designed to illustrate OR concepts and therefore is probably missing elements that would be part of a commercial application.)

application.)

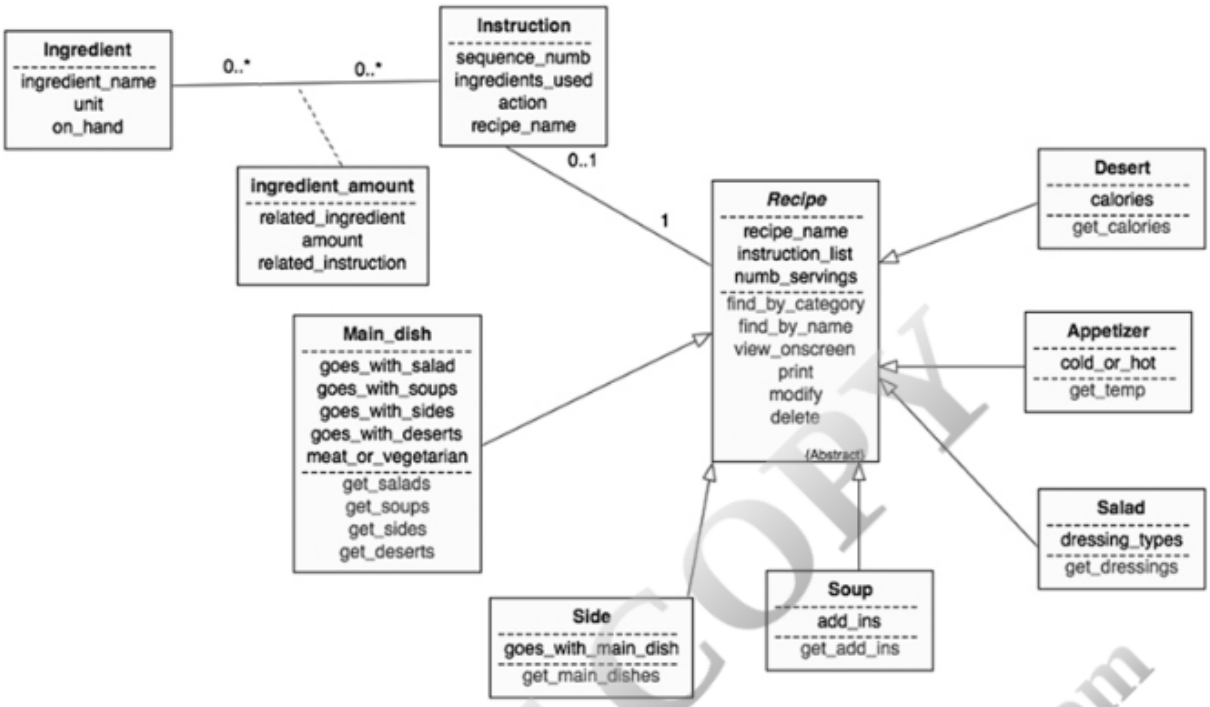


FIGURE 27.18 **An object-relational ERD.**

The *recipe* class is an abstract class that stores data common to all types of recipes. The six subclasses represent categories of recipes, each of which has at least one unique attribute.

The *ingredient*, *instruction*, and *ingredient_amount* classes are more traditional entities. A recipe has many instructions. Each instruction uses zero, one, or more ingredients. The *ingredient_amount* class therefore stores relationship data: the amount of a given ingredient used in a given instruction.

SQL Data Types for Object-Relational Support

SQL's OR features include three column data types for storing multiple values: ROW, ARRAY, and MULTISET. You can use these data types without using any of the other OR features (in particular, typed tables to store objects). Because they do not act as objects, these columns cannot have methods.

Row Type

A column declared as a ROW type holds an entire row of data (multiple pieces of data). This gives you the equivalent of a table within a table. The contents of the row—called *fields*—can be declared as built-in data types or UDTs.

As an example, we'll create a table for customers of the rare book store that stores the customer's address in a single column, using the ROW data type:

```
CREATE TABLE customer
     (first_name CHAR (20),
     last_name CHAR (20),
     address ROW (street CHAR (50), city CHAR (30),
          state CHAR (2), zip CHAR (10), phone CHAR (12));
```

Notice that the ROW column is given a name, just as a single-valued column. The data type is followed by the contents of the row in parentheses. The row's fields are declared in the same way as any other SQL column (name followed by data type).

We use "dot" notation to access the individual fields in a column of type ROW. For example, to reference the *street* field in the *address* column without qualifying the table name you would use

```
address.street
```

When the SQL statement requires a table name (for example, for use in a join or for display by a query in which the field appears in multiple tables), you precede the field reference with the table name, as in

```
customer.address.street
```

Inserting values into a row column is only a bit different from the traditional INSERT statement. Consider the following example:

```
INSERT INTO customer VALUES
     ('John','Doe',
      ROW ('123 Main Street','Anytown','ST','11224'),
      '555-111-2233');
```

The data for the *address* column are preceded by the keyword ROW. The values follow in parentheses.

Array Type

An ARRAY is an ordered collection of elements. Like arrays used in programming, they are declared to have a maximum number of elements of the same type. That type can be a simple data type or a UDT. For example, we might want to store order numbers as part of a customer's data in a *customer* table:

```
CREATE TABLE customer
     (first CHAR (20),
      last CHAR (20),
      orders INT ARRAY[100],
      numb_orders INT,
      phone CHAR (12);
```

The array column is given a name and a data type, which are followed by the keyword ARRAY and the maximum number of elements the array should hold (the array's *cardinality*) in brackets. The array's data type can be one of SQL's built in data types or a UDT.

Access to values in an array is by the array's *index* (its position in the array order). Although you specify the maximum number of elements in an array counting from 1, array indexes begin at 0. An array of 100 elements therefore has indexes from 0 to 99. The sample *customer* table above includes a column (*numb_orders*) that stores the total number of elements in the array. The last used index will be *numb_orders – 1*.

You can input multiple values into an array at one time when you first insert a row:

```
INSERT INTO customer VALUES
     ('John','Doe', ARRAY (25,109,227,502,610),
     5, '555-111-2233');
```

```
3,  555-111-2233 ');
```

The keyword ARRAY precedes the values in parentheses.
You can also insert or modify a specify array element directly:

```
INSERT INTO customer (first, last, orders[0],numb_orders,
     phone)
     VALUES ('John','Doe',25,1, '555-111-2233');
```

When you query a table and ask for display of an array column by name, without an index, SQL displays the entire contents of the array, as in:

```
SELECT orders
FROM customer
WHERE first = 'John' AND last = 'Doe';
```

Use an array index when you want to retrieve a single array element. The query

```
SELECT orders [numb_orders - 1]
FROM customer
WHERE first = 'John' AND last = 'Doe';
```

displays the last order stored in the array.

Processing each element in an array requires programming (a trigger, stored procedure, or embedded SQL). Declare a variable to hold the array index, initialize it to 0, and increment it by 1 each time an appropriate loop iterates—the same way you would process all elements in an array using any high-level programming language.

Note: Although many current DBMSs support arrays in columns, not all automatically perform array bounds checking. In other words, they do not necessarily validate that an array index is within the maximum number specified when the table was created. Check your software's documentation to determine whether array bounds constraints must be handled by an application program or can be left up to the DBMS.

Manipulating Arrays

Restrictions of content and access notwithstanding, there two operations that you can perform on arrays:

- Comparisons: Two arrays can be compared (for example, in a WHERE clause) if the two arrays are created from data types that can be compared. When making the comparison, SQL compares the elements in order. Two arrays A and B, therefore, are equivalent if A[0] = B[0], A[1] = B[1], and so on; all comparisons between all the pairs of values in the array must be true. By the same token, A > B if A[0] > B[0] throughout the arrays.
- Concatenation: Two arrays with compatible data types (data types that can be converted into a single data type) can be concatenated with the concatenation operator (| |). The result is another array, as in Figure 27.19. Notice that the data from array A have been converted to real numbers, because SQL will always convert to the format that has the highest precision.

(A)		(B)		(C)
16		96.05		16.00
52		295.82		52.00
109		303.00		109.00
85	||	105.88	=	85.00
33		22.16		33.00
203		111.23		203.00
384		88.22		384.00
23		45.99		23.00
		18.62		96.05

```
35.88                295.82
                     303.00
                     105.88
                     22.16
                     111.23
                     88.22
                     45.99
                     18.62
                     35.88
```

FIGURE 27.19 **Concatenating arrays.**

Multiset Type

A *multiset* is an unordered collection of elements of the same type. The following table contains a multiset to hold multiple phone numbers:

```
CREATE TABLE customer
      (first CHAR (20),
       last CHAR (20),
       orders INT ARRAY[100],
       phones CHAR (20) MULTISET;
```

You specify the contents of a multiset when you insert a row into a table, much like you do for an array. The only difference is the use of the keyword MULTISET to indicate that the values in parentheses are intended as a single group:

```
INSERT INTO customer (first, last, orders[0],numb_orders,phones)
  VALUES ('John','Doe',25,1, MULTISET ('555-111-2233','555-222-
1122'));
```

Because a multiset is unordered, you cannot access individual elements by position, as you do with array elements. You can, however, display the entire contents of the multiset by using its name in a query:

```
SELECT phones
FROM customer
WHERE first = 'John' AND last = 'Doe';
```

Updating a multiset is an all or nothing proposition. In other words, you can't pull one value out, or put in a single value. An UPDATE statement such as

```
UPDATE customer
     SET phones = MULTISET ('555-111-2233','555-333-1122');
```

replaces the entire contents of the *phones* column.

Manipulating Multisets

Manipulating Multisets

As with arrays, there are a few operations that can be performed on multisets with compatible data types:

- Multisets can be compared, just as arrays. Multisets A and B will be true if they contain exactly the same elements.
- Union: The MULTISET UNION operator returns a multiset that contains all elements in the participating multisets. For example,

```
UPDATE some_table
  SET big_multiset = small_multiset1 MULTISET
      UNION small_multiset2
```

puts the two small multisets into the big multiset.

- Intersect: The MULTISET INTERSECT operator returns all elements that two multisets have in common. For example,

```
SELECT table1.multiset MULTISET
     INTERSECT table2.multiset
FROM table1 JOIN table2;
```

works on each row in the joined tables, returning the elements that the multisets in each row have in common.

- Difference: The MULTISET EXCEPT operation returns the difference between two multisets (all elements they *don't* have in common). The query

```
SELECT table1.multiset MULTISET
     EXCEPT table2.multiset
FROM table1 JOIN table2;
```

functions exactly like the previous example, but returns elements from each row that the multisets don't share.

The union, intersect, and difference operators have two options. If you include ALL after the operator, SQL includes duplicate elements in the result. To exclude duplicates, use DISTINCT.

User-Defined Data Types and Typed Tables

The more classic SQL object-oriented features are built from UDTs and typed tables. The UDT defines a class and the typed table provides a place to store objects from that class. What this means is that an instance of a UDT is not stored in the column to which the UDT has been assigned as a data type. Relations simply have no mechanism for handling multiple values at the intersection of a column and a row. Therefore, the objects are stored in their own typed table and a reference to those objects is placed in the relation using them, one reference per table row. Columns that will hold references to objects in another table are given a data type of REF.

In this section of this chapter you will first see UDTs used as domains, something you can do without using any other OR features in your database. Then, we will look at UDTs as classes and how references to objects are handled.

UDTs as Domains

A user-defined data type is a structured, named group of attributes of existing data types (either built-in types or other UDTs). In its simplest form, the UDT has the following general syntax:

```
CREATE TYPE type_name AS (column_definitions)
```

We could create a very simple type to hold a date, for example:

```
CREATE TYPE date_type AS
     (month int,
      day int,
      year int);
```

We could then specify *date_type* as the data type for a column in a table:

```
CREATE TABLE people
     (first CHAR (20),
      last CHAR (20),
      birthdate date_type);
```

You use dot notation to access the parts of the UDT. For example,

```
birthdate.year
```

refers to the year potion of the *date_type* UDT.

UDTs as Classes

More commonly, we use a UDT to define a class. For example, we could create a type for the *Ingredient* class with

```
CREATE TYPE ingredient_type AS OBJECT
     (ingredient_name CHAR (256),
      unit char (20),
      on_hand int);
```

Notice the AS OBJECT clause that has been inserted after the UDT's name. This indicates that rather than being used as the domain for a value in a table, this class will be used as the structure of a typed table.

Note: UDTs can have methods, just like a class created in an object-oriented programming language. We'll look at them at the end of this chapter.

Creating Typed Tables Using UDTs

Once you have created a class as a UDT, you then use that UDT to create a *typed table*:

```
CREATE TABLE table_name OF UDT_name
    REF IS reference_column_name (method_to_generate_row_ID)
```

SQL creates a table with one column for each column in the UDT on which the table is based, along with a column for the object ID. There are three options for creating the object ID of a row:

- The user generates the object ID (REF USING *existing_data_type*).
- The DBMS generates the object ID (REF IS *identifier_name* SYSTEM GENERATED).
- The object ID comes from the values in a list of attributes (REF FROM *attribute_list*).

You may want to use a primary key as a source for an object ID. Although this makes sense logically, it also provides the slowest retrieval performance.

By default, the object ID value is generated by the SQL command processor whenever a row is inserted into the typed table, using the method that was specified when the table was created. However, an insert operation can override the default object ID, placing a user-specified value into the ID column. Once created, the object ID cannot be modified.

To create the *ingredient* table, we could use

```
CREATE TABLE ingredient OF ingredient_type
    (REF IS ingredient_ID SYSTEM GENERATED);
```

Note: Only base tables or views can be typed tables. Temporary tables cannot be created from UDTs.

Inheritance

One of the most important OO features added to the SQL:2003 standard was support for inheritance. To create a *subtype* (a *subclass* or *derived class*, if you will), you create a UDT that is derived from another and then create a typed table of that subtype.

As a start, let's create the recipe type that will be used as the superclass for types of recipes:

```
CREATE TYPE recipe_type AS OBJECT
    (recipe_name CHAR (256),
     instruction_list instruction ARRAY[20],
     numb_servings INT)
     NOT INSTANTIABLE,
     NOT FINAL;
```

The two last lines in the preceding example convey important information about this class. *Recip_type* is an abstract class: Objects will never be created from it directly. We add NOT INSTANTIABLE to indicate this property.

By default, a UDT has a *finality* of FINAL. It cannot be used as the parent of a subtype. (In other words, nothing can inherit from it.) Because we want to use this class as a superclass, we must indicate that it is NOT FINAL.

To create the subtypes, we indicate the parent type, preceded by the keyword UNDER. The subtype declaration also includes any attributes (and methods) that are not in the parent type that need to be added to the subtype. For example, we could create the desert type with:

```
CREATE TYPE desert_type
    UNDER recipe_type (calories INT);
```

Because this type will be used to create objects, and because no other types will be derived from it, we can accept the defaults of INSTANTIABLE and FINAL.

Note: As you have just seen, inheritance can operate on UDTs. It can also be used with typed tables, where a typed table is created UNDER another.

Reference (REF) Type

Once you have a typed table, you can store references to the objects in that table in a column of type REF that is part of another table. For example, there is one REF column in the recipe database: the attribute in the *ingredient_amount* table (*related_ingredient*) that points to which ingredient is related to each occurrence of *ingredient_amount*.

To set up the table that will store that reference, use the data type REF for the appropriate column:

```
column_name REF UDT_being_referenced SCOPE IS typed_table_name
```

For example,

```
CREATE TABLE ingredient_amount
     (related_ingredient REF ingredient_type
     SCOPE IS ingredient, amount decimal (5,2));
```

creates a table with a column that stores a reference to an ingredient. The SCOPE clause specifies the table or view that is the source of the reference.

To insert a row into a table with a REF column, you must include a SELECT in the INSERT statement that locates the row whose reference is to be stored. As you would expect, the object being referenced must exist in its own table before a reference to it can be generated. We must therefore first insert an ingredient into the *ingredient* table:

```
INSERT INTO ingredient VALUES
     ('Unbleached flour', 'cups',25);
```

Then, we can insert a referencing row into *ingredient_amount*:

```
INSERT INTO ingredient_amount
     (SELECT REF (i)
     FROM ingredient i
     WHERE i.ingredient_name = 'Unbleached flour')
     VALUES (2.5);
```

Dereferencing for Data Access

An application program that is using the recipe database as its data store will need to use the reference stored in the *ingredient_amount* table to locate the name of the ingredient. The DEREF function follows a reference back to the table being referenced and returns data from the appropriate row. A query to retrieve the name and amount of an ingredient used in a recipe instruction could therefore be written:

```
SELECT DEREF(related_ingredient).ingredient_name, amount
FROM ingredient_amount
WHERE DEREF(related_instruction).recipe_name = 'French toast';
```

Note that the DEREF function accesses an entire row in the referenced table. If you don't specify otherwise, you will retrieve the values from every column in the referenced row. To retrieve just the value of a single column, we use "dot" notation. The first portion—

```
DEREF(related_ingredient)
```

—actually performs the dereference. The portion to the right of the dot specifies the column in the referenced row.

—actually performs the dereference. The portion to the right of the dot specifies the column in the referenced row.
Some DBMSs provide a dereference operator (- >) that can be used in place of the DEREF function. The preceding query might be written:

```
SELECT related_ingredient->ingredient_name, amount
FROM ingredient_amount;
```

Methods

The UDTs that we have seen to this point have attributes, but not methods. It is certainly possible, however, to declare methods as part of a UDT and then to use SQL programming to define the body of the methods. Like classes used by OO programming languages such C + +, in SQL the body of a method is defined separately from the declaration of the UDT.

Note: SQL extensions for writing methods appear in Appendix B. To get the most out of it, you need to know a high-level programming language. If you don't program, you can just skim the rest of this section.

You declare a method after declaring the structure of a UDT. For example, we could add a method to display the instructions of a recipe with

```
CREATE TYPE recipe_type AS OBJECT
     (recipe_name CHAR (256),
      instruction_list instruction ARRAY[20],
      numb_servings INT)
      NOT INSTANTIABLE,
      NOT FINAL
      METHOD show_instructions ();
```

This particular method does not return a value and the declaration therefore does not include the optional RETURNS clause. However, a method to compute the cost of a recipe (if we were to include ingredient costs in the database) could be declared as

```
CREATE TYPE recipe_type AS OBJECT
     (recipe_name CHAR (256),
      instruction_list instruction ARRAY[20],
      numb_servings INT)
      NOT INSTANTIABLE,
      NOT FINAL
      METHOD show_instructions ()
      METHOD compute_cost ()
           RETURNS DECIMAL (5,2));
```

Methods can accept input parameters within the parentheses, following the method name. A method declared as

```
METHOD scale_recipe (IN numb_servings INT):
```

accepts an integer value as an input value. The parameter list can also contain output parameters (OUT) and parameters used for both input and output (INOUT).

Defining Methods

As mentioned earlier, although methods are declared when UDTs tables are declared, the bodies of methods are written separately. To define a method, use the CREATE METHOD statement:

```
CREATE METHOD method_name FOR UDT_name
BEGIN
     //body of method
END
```

A SQL-only method is written using the language constructs discussed in Appendix B

A SQL-only method is written using the language constructs discussed in Appendix B.

Executing Methods

Executing a method uses notation:

```
typed_table_name.method_name (parameter_list );
```

Such an expression can be, for example, included in an INSERT statement to insert the method's return value into a column. It can also be included in another SQL method, trigger, or stored procedure. Its return value can then be captured across an assignment operator. Output parameters return their values to the calling routine, where they can be used as needed.

For Further Reading

Brown P. *Object-Relational Database Development: A Plumber's Guide*. Prentice Hall PTR; 2000.
Date CJ, Darwen H. *Foundation for Object/Relational Databases: The Third Manifesto*. Addison-Wesley; 1998.
Dietrich SW, Urban SD. *Fundamentals of Object Databases: Object-Oriented and Object-Relational Designs*. Morgan and Claypool; 2011.
Melton J. *Advanced SQL: 1999: Understanding Object-Relational and Other Advanced Features*. Morgan Kaufmann; 2002.

[1] There have been some DBMSs that use only an object-oriented data model. However, over time they have not had significant market penetration. For more information, see "Whatever Happened to Object-Oriented Databases" (http://leavcom.com/articles/db_08_00.htm).