Appendix G

Object-Oriented Databases

Preview

Object-oriented (OO) technology draws its strength from powerful programming and modeling techniques and advanced data-handling capabilities. Because OO technology has become an important contributor to the evolution of database systems, this appendix explores the characteristics of OO systems and how those characteristics affect data modeling and design. This appendix also investigates how, through the creation of what are known as extended relational or object/relational databases, relational database vendors have responded to the demand for databases capable of handling increasingly complex data types. You will see how some of those features are implemented in Oracle.

Data Files and Available Formats

	MS Access	Oracle	MS SQL	My SQL
Avion_Sales	✓			
RC_Stores	✓			
RC_Systems	✓			
RRE_Trucking	✓			

Data Files Available on cengagebrain.com

G-1 Object Orientation and Its Benefits

object orientation
A set of modeling and development principles focused on an autonomous entity with embedded intelligence to interact with other objects and itself.

Object orientation is a modeling and development methodology based on object-oriented (OO) concepts. More precisely, **object orientation** is defined as a set of design and development principles based on conceptually autonomous computer structures known as objects. Each object represents a real-world entity with the ability to act upon itself and to interact with other objects.

Considering that definition, it does not require much imagination to see that using objects makes modularity almost inevitable. Object orientation concepts have been widely applied to many computer-related disciplines, especially those involving complex programming and design problems. Table G.1 summarizes some object orientation contributions to computer-related disciplines.

TABLE G.1

OBJECT ORIENTATION CONTRIBUTIONS

COMPUTER-RELATED AREA	OO CONTRIBUTIONS
Programming languages	Reduces the number of lines of code Decreases development time Enhances code reusability Makes code maintenance easier Enhances programmer productivity
Graphical user interfaces (GUIs)	Enhances ability to create easy-to-use interfaces Improves system end-user friendliness Makes it easy to define standards
Databases	Supports abstract data types Supports complex objects Supports multimedia data types
Design	Captures more of the data model's semantics Represents the real world more accurately Supports complex data manipulations in specialized applications that target graphics, imaging, mapping, financial modeling, telecommunications, geospatial applications, medical applications, and so on
Operating systems	Enhances system portability by creating layers of abstractions to handle hardware-specific issues Facilitates system extensibility through the use of inheritance and other object-oriented constructs

G-2 The Evolution of Object-Oriented Concepts

object-oriented programming (OOP)
An alternative to conventional programming methods based on object oriented concepts. It reduces programming time and lines of code, and increases programmers' productivity.

Object-oriented concepts stem from **object-oriented programming (OOP)**, which was developed as an alternative to traditional programming methods. In an OOP environment, the programmer creates or uses objects (self-contained, reusable modules that contain data as well as the procedures used to operate on the data).

OO concepts first appeared in programming languages such as Ada, ALGOL, LISP, and SIMULA. Those programming languages set the stage for the introduction of more refined OO concepts. Smalltalk, C++, and Java are popular **object-oriented programming languages (OOPLs)**. Java is used to create web applications that run on the Internet and are independent of operating systems.

OOPLs were developed to:

- Provide an easy-to-use software development environment.
- Provide a powerful software modeling tool for application development.
- Decrease development time by reducing the amount of code.
- Improve programmer productivity by making the code reusable.

OOP changes not only the way in which programs are written but also how those programs behave. In the object-oriented view of the world, each object can manipulate the data that are part of the object. In addition, each object can send messages to change the data of other objects. Consequently, the OO environment has several important attributes:

- The data set is no longer passive.
- Data and procedures are bound together, creating an object.
- The object has an innate ability to act on itself.

An object can interact with other objects to create a system. Because such objects carry their own data and code, it becomes easier to produce reusable modular systems. It is precisely that characteristic that makes OO systems seem natural to those with little programming experience, but confusing to many whose traditional programming expertise has trained them to split data and procedures. It is not surprising that OO notions became more viable with the advent of personal computers because personal computers are typically operated by end users rather than by programmers and systems design specialists.

OO programming concepts have also had an effect on most computer-based activities, including those based on databases. Because a database is designed to capture data about a business system, it can be viewed as a set of interacting objects. Each object has certain characteristics (attributes) and has relationships with other objects (methods). Given that structure, OO systems have an intuitive appeal for those doing database design and development. As database designers rather than programmers, you cannot afford to ignore the OO revolution.

G-3 Object-Oriented Concepts

Although OO concepts have their roots in programming languages, and the programmers among you will recognize basic programming elements, *you do not need to know anything about programming to understand these concepts*.

G-3a Objects: Components and Characteristics

In OO systems, everything you deal with is an object, whether it is a student, an invoice, an airplane, an employee, a service, a menu panel, or a report. Some objects are tangible, and some are not. An **object** can be defined as an abstract representation of a real-world entity that has a unique identity, embedded properties, and the ability to interact with other objects and act upon itself.

Note the difference between *object* and *entity*. An entity has data components and relationships, but lacks manipulative ability. Other differences will be identified later.

A defining characteristic of an object is its *unique identity*. To emphasize this point, let's examine the real-world objects displayed in Figure G.1. As you examine those simple objects, note that the student named J. D. Wilson has a unique (biological) identity and therefore constitutes a different object from M. R. Gonzalez or V. K. Spelling. Note also that although they share common *general* characteristics such as name, Social Security number, address, and date of birth, each object exists independently in time and space.

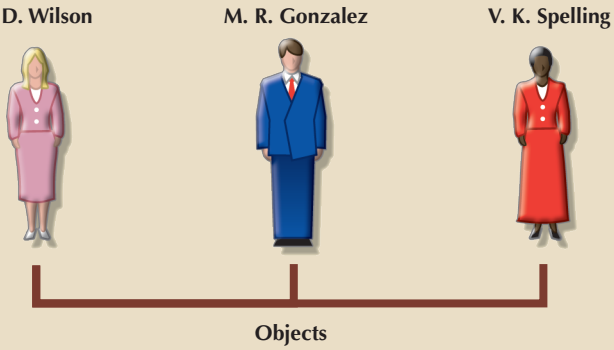
object-oriented programming languages (OOPLs)
A programming language based on object oriented concepts.

object
An abstract representation of a real-world entity that has a unique identity, embedded properties, and the ability to interact with other objects and itself.

Note

Current data privacy and security standards do not recommend the use of Social Security numbers as database identifiers. However, Social Security information is still stored by organizations using strict security measures such as encryption, secure access rules, and active data monitoring and protection.

FIGURE G.1 REAL-WORLD STUDENT OBJECTS



G-3b Object Identity

object ID (OID)
A system-generated object identifier that is independent of the object state and any physical address in memory.

The object's identity is represented by an **object ID (OID)**, which is unique to the object. The OID is assigned by the system at the moment of the object's creation *and cannot be changed under any circumstances*.

Do not confuse the relational model's primary key with an OID. In contrast to the OID, a primary key is based on *user-given* values of selected attributes and can be changed at any time. The OID is assigned by the system, does not depend on the object's attribute values, and cannot be changed. *The OID can be deleted only when the object is deleted, and that OID cannot be reused.*

G-3c Attributes (Instance Variables)

instance variables
In the object-oriented model, another term for an attribute. See *attribute*.

base data types
A term used to describe the data types frequently used in traditional programming languages. Base data types include *real*, *integer*, and *string*.

conventional data types
See *base data types*.

domain
In data modeling, the construct used to organize and describe an attribute's set of possible values.

Objects are described by their attributes, known as **instance variables** in an OO environment. For example, the student John D. Smith may have the attributes (instance variables) shown in Table G.2. Each attribute has a unique name and a data type associated with it. In Table G.2, the attribute names are SOCIAL_SECURITY_NUMBER, FIRST_NAME, MIDDLE_INITIAL, LAST_NAME, and so on. Traditional data types, also known as **base data types** or **conventional data types**, are used in most programming languages and include real, integer, string, and so on.

Attributes also have a domain. The **domain** logically groups and describes the set of all possible values that an attribute can have. For example, the possible values of SEMESTER_GPA (see Table G.2) can be represented by the real number base data type. But that does not mean that any real number is a valid GPA. Keep in mind that base data types define base domains; that is, *real* represents all real numbers, *integer* represents all integers, *date* represents all possible dates, *string* represents any combination of characters, and so on. However, base data type domains are the building blocks used to construct more restrictive *named* domains at a higher logical level. For example, to define

TABLE G.2

OBJECT ATTRIBUTES

ATTRIBUTE NAME	ATTRIBUTE VALUE
SOCIAL_SECURITY_NUMBER	414-48-0944
FIRST_NAME	John
MIDDLE_INITIAL	D
LAST_NAME	Smith
DATE_OF_BIRTH	11/23/1966
MAJOR *	Accounting
SEMESTER_GPA	2.89
OVERALL_GPA	3.01
COURSES_TAKEN *	ENG201;MATH243;HIST201;ACCT211;ECON210;ECON212; ACCT212;CIS220;ENG202;MATH301;HIST202;CIS310; ACCT343;ACCT345; ENG242;MKTG301;FIN331;ACCT355
ADVISOR*	Dr. W. R. Learned
* Represents an attribute that references one or more other objects	

the domain for the GPA attribute precisely, a domain named GPA must be created. Every domain has a name and a description, including the base data type, size, format, and constraints for the domain's values. Therefore, the GPA domain can be defined as "any positive number between 0.00 and 4.00 with only two decimal places." In this case, there is a domain name "GPA," a base data type "real," a constraint rule "any positive number between 0.00 and 4.00," and a format "with only two decimal places." The GPA domain will provide the values for the SEMESTER_GPA and OVERALL_GPA attributes. Domains can also be defined as lists of possible values separated by commas. For example the GENDER domain can be defined as "Male, Female" or as "M, F."

It is important to note that the relational database model also supports domains. In fact, C. J. Date, one of the relational database model's "parents," presents domains as the way in which relational systems are able to support abstract data types, thus providing the same functionality as object-oriented databases.[1]

Just as in the ER model, an object's attribute can be *single-valued* or *multivalued*. Therefore, the object's attribute can draw a single value or multiple values from its domain. For example, the SOCIAL_SECURITY_NUMBER attribute takes only one value from its domain because the student can have only one Social Security number. In contrast, an attribute such as LANGUAGE or HOBBY can have many values because a student might speak many languages or have many hobbies.

Object attributes may reference one or more other objects. For example, the attribute MAJOR refers to a Department object, the attribute COURSES_TAKEN refers to a list (or collection) of Course objects, and the attribute ADVISOR refers to a Professor object. At the implementation level, the OID of the referenced object is used to link both objects, thus allowing the implementation of relationships between two or more objects. Using the example in Table G.2, the MAJOR attribute contains the OID of a Department object (Accounting) and the ADVISOR attribute contains the OID of a Professor object (Dr. W. R. Learned). The COURSES_TAKEN attribute contains the OID of an object that contains a list of Course objects; such an object is known as a **collection object**.

collection object
An object that contains one or more objects.

[1] See C. J. Date and Hugh Darwen, *Foundation for Object/Relational Databases: The Third Manifesto*, Addison Wesley, 1998.

Note

Observe the difference between the relational and OO models at this point. In the relational model, a table's attribute may contain only a value used to join rows in different tables. The OO model does not need such joins to relate objects to one another.

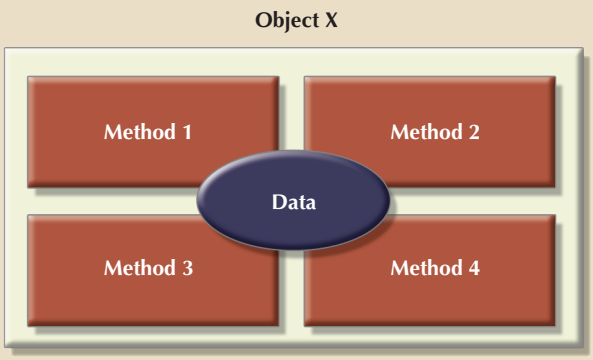
G-3d Object State

The **object state** is the set of values that the object's attributes have at any given time. Although the object's state can vary, its OID remains the same. If you want to change the object's state, you must change the values of the object's attributes. To change the object's attribute values, you must send a message to the object. This *message* will invoke a *method*.

G-3e Messages and Methods

A **method** is the code that performs a specific operation on the object's data. Methods protect data from direct and unauthorized access by other objects. To help you understand messages and methods, imagine that the object is a nutshell. The nutshell's nucleus (the nut) represents the object's data structure, and the shell represents its methods. (See Figure G.2.)

FIGURE G.2 DEPICTION OF AN OBJECT



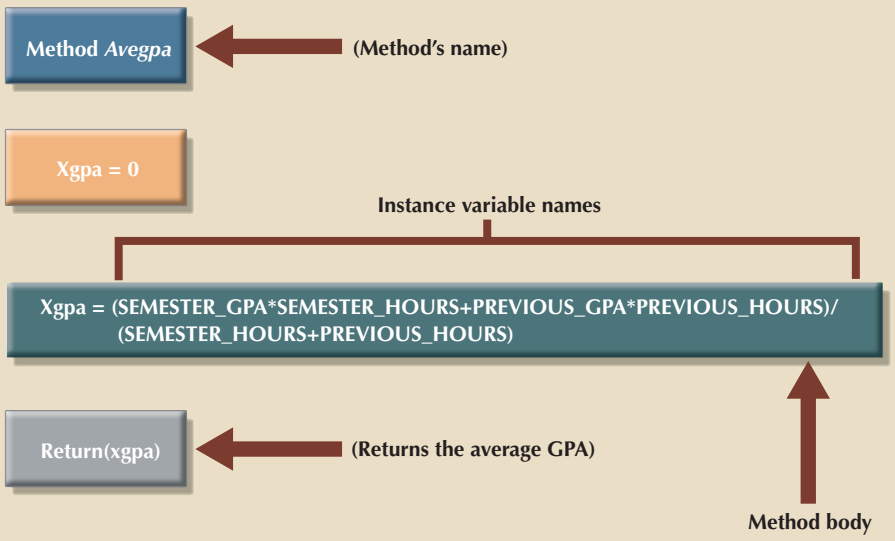
object state
The set of values that the object's attributes have at a given time.

method
In the object-oriented data model, a named set of instructions to perform an action. Methods represent real-world actions, and are invoked through messages.

Every operation performed on an object must be implemented by a method. Methods are used to change the object's attribute values or to return the value of selected object attributes. Methods represent real-world actions, such as changing a student's major, adding a student to a course, or printing a student's name and address. In effect, *methods are the equivalent of procedures in traditional programming languages*. In OO terms, methods represent the object's behavior.

Every method is identified by a *name* and has a *body*. The body is composed of computer instructions written in some programming language to represent a real-world action. For example, using the object attributes described in Table G.2, you can define a method *Avegpa* that will return the average GPA of a student by using the object's attributes SEMESTER_GPA and OVERALL_GPA. Thus, the method named *Avegpa* may be represented by the transformation shown in Figure G.3.

FIGURE G.3 METHOD COMPONENTS



As you examine Figure G.3, note that the Return(xgpa) would yield (3.2 * 15) + (3.0 * 60)/(15 + 60) = (48 + 180)/75 = 3.04 for a student with the following characteristics:

- Current semester GPA is 3.2.
- Current class load is 15 semester hours.
- Previous GPA was 3.0 earned for a total of 60 hours.

As you examine that example, note that a method can access the instance variables (attributes) of the object for which the method is defined.

To invoke a method, you send a message to the object. A **message** is sent by specifying a receiver object, the name of the method, and any required parameters. The internal structure of the object cannot be accessed directly by the message sender, which is another object. Denial of access to the structure ensures the integrity of the object's state and hides the object's internal details. The ability to hide the object's internal details (attributes and methods) is known as **encapsulation**.

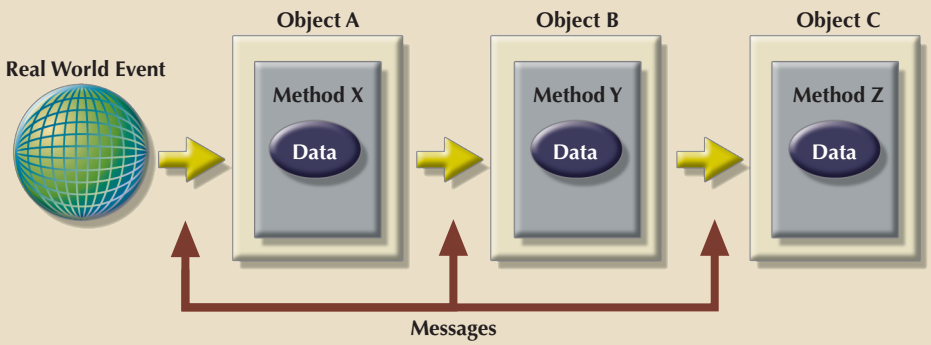
An object may also send messages to change or interrogate another object's state. (To **interrogate** means to ask for the interrogated object's instance variable value or values.) To perform such object-change and interrogation tasks, the method's body can contain references to other objects' methods (send messages to other objects), as depicted in Figure G.4.

message
In the OO data model, the name of a method sent to an object in order to perform an action. A message triggers the object's behavior. See *method*.

encapsulation
A feature by which the object can hide the internal data representation and method's implementation from external objects. Characteristic of an object oriented data model.

interrogate
To ask for the interrogated object's instance variable value or values. An object may send messages to interrogate another object's state.

FIGURE G.4 OBJECTS SEND MESSAGES TO EACH OTHER



G-3f Classes

OO systems classify objects according to their similarities and differences. Objects that share common characteristics are grouped into classes. In other words, a **class** is a collection of similar objects with shared structure (attributes) and behavior (methods).

A class contains the description of the data structure and the method implementation details for the objects in the class. Therefore, all objects in a class share the same structure and respond to the same messages. In addition, a class acts as a "storage bin" for similar objects. Each object in a class is known as a **class instance** or an **object instance**. (See Figure G.5.)

Using the example shown earlier in Table G.2, a class named Student can be defined to store student objects. All objects of the class Student shown in Figure G.6 share the same structure (attributes) and respond to the same messages (implemented by methods). Note that the *Avegpa* method was defined earlier; the *Enroll* and *Grade* methods shown in Figure G.6 have been added. Each instance of a class is an object with a unique OID, and each object knows to which class it belongs.

FIGURE G.5 CLASS ILLUSTRATION

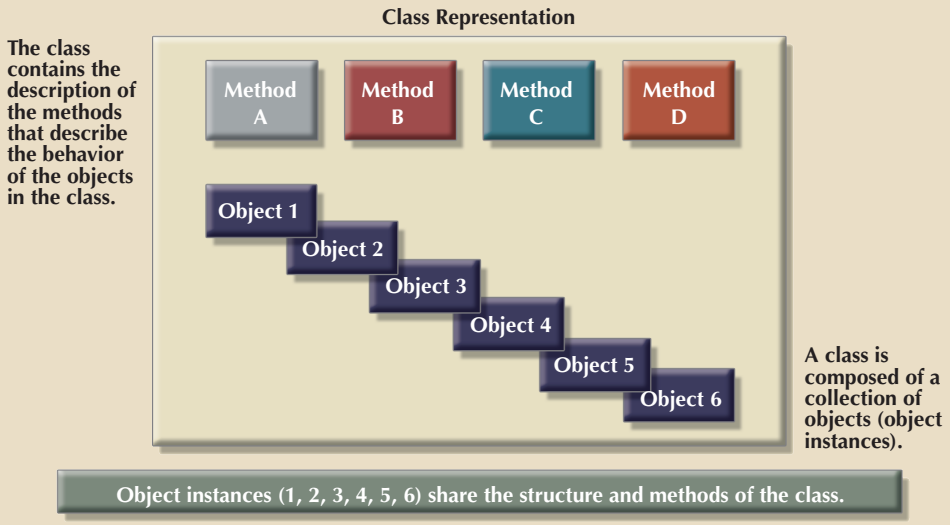
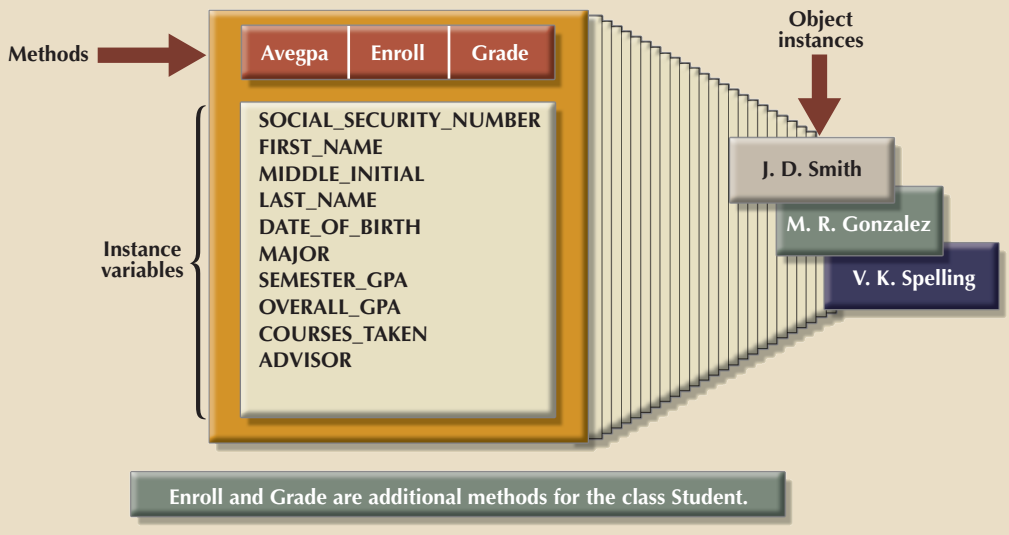


FIGURE G.6 REPRESENTATION OF THE CLASS STUDENT



class
A collection of similar objects with shared structure (attributes) and behavior (methods). A class encapsulates an object's data representation and a method's implementation. Classes are organized in a class hierarchy.

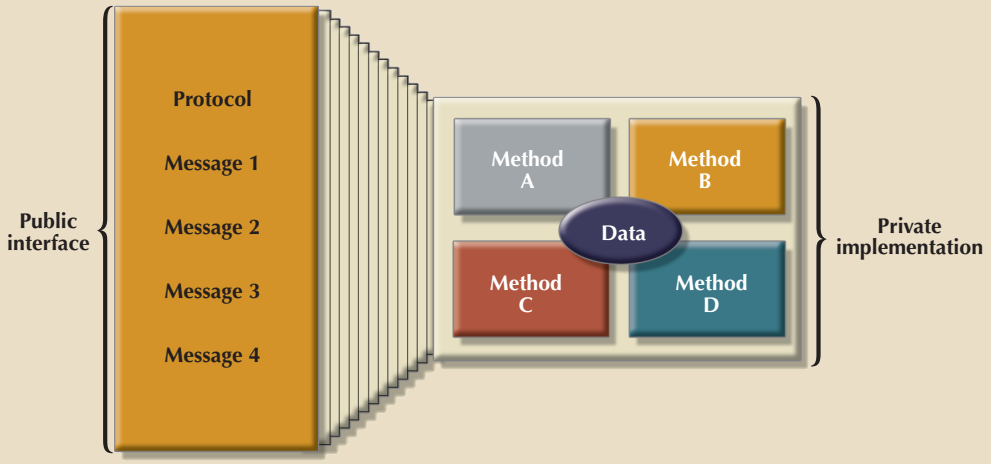
class instance
Each individual object stored in a class. Each class instance must share the same structure and respond to the same messages if they are located in the same class. Also known as *object instance*.

object instance
Each particular object belonging to a class.

G-3g Protocol

The class's collection of messages, each identified by a message name, constitutes the object or class *protocol*. The **protocol** represents an object's public aspect, that is, how it is known by other objects as well as end users. In contrast, the implementation of the object's structure and methods constitutes the object's *private aspect*. Both are illustrated in Figure G.7. For example, Figure G.6 shows three methods (*Avegpa*, *Enroll*, *Grade*) that represent the public aspect of the Student object. Because those methods are public, other objects communicate with the Student object, using any of the methods. The internal representation of the methods (see Figure G.3) yields the private aspect of the object. The private aspect of an object is not available for use by other objects.

FIGURE G.7 PUBLIC AND PRIVATE ASPECTS OF AN OBJECT



Usually, a message is sent to an object instance. However, it is also possible to send a message to the class rather than to the object. When the receiver object is a class, the message will invoke a *class method*. One example of a class method is *new*. The *new* class method creates a new object instance (with a unique OID) in the receiver class. Because the object does not exist yet, the message *new* is sent to the class and not to the object.

The preceding discussions have laid the foundation for your understanding of object-oriented concepts. Figure G.8 is designed to put together all of the pieces of this part of the OO puzzle, so examine it carefully before you continue.

G-3h Superclasses, Subclasses, and Inheritance

Classes are organized into a class *hierarchy*. A **class hierarchy** resembles an upside-down tree in which each class has only one parent class. The class hierarchy is known as a **class lattice** when its classes can have multiple parent classes. Class is used to categorize objects into groups of objects that share common characteristics. For example, the class *automobile* includes large luxury sedans as well as compact cars, and the class *government* includes federal, state, and local governments. Figure G.9 illustrates that the generalization *musical instruments* includes stringed instruments as well as wind instruments.

As you examine Figure G.9, note that Piano, Violin, and Guitar are **subclasses** of Stringed instruments, which is, in turn, a subclass of Musical instruments. Musical instruments defines the **superclass** of Stringed instruments, which is, in turn, the superclass of the Piano, Violin, and Guitar classes. As you can see, the superclass is a more general

protocol
A specific set of rules to accomplish a specific function. In the object oriented data model, protocol refers to a collection of messages to which an object responds.

class hierarchy
The organization of classes in a hierarchical tree in which each parent class is a *superclass* and each child class is a *subclass*. See also *inheritance*.

class lattice
The class hierarchy is known as a class lattice if its classes can have multiple parent classes.

subclasses
See *class hierarchy*.

superclass
In a class hierarchy, the superclass is the more general classification from which the subclasses inherit data structures and behaviors.

classification of its subclasses, which, in turn, are more specific components of the general classification.

The class hierarchy introduces a powerful OO concept known as *inheritance*. **Inheritance** is the ability of an object within the hierarchy to inherit the data structure and behavior (methods) of the classes above it. For example, the Piano class in Figure G.9 inherits its data structure and behavior from the superclasses Stringed instruments and Musical instruments. Thus, Piano inherits the strings and its sounding board characteristic from the Stringed instruments superclass and the musical scale from its Musical instruments superclass. *It is through inheritance that OO systems can deliver code reusability*.

In OO systems, all objects are derived from the superclass Object, or the Root class. Therefore, all classes share the characteristics and methods of the superclass Object. The inheritance of data and methods goes from top to bottom in the class hierarchy. There are two types of inheritance: single inheritance and multiple inheritance.

FIGURE G.8 SUMMARY OF OBJECT CHARACTERISTICS

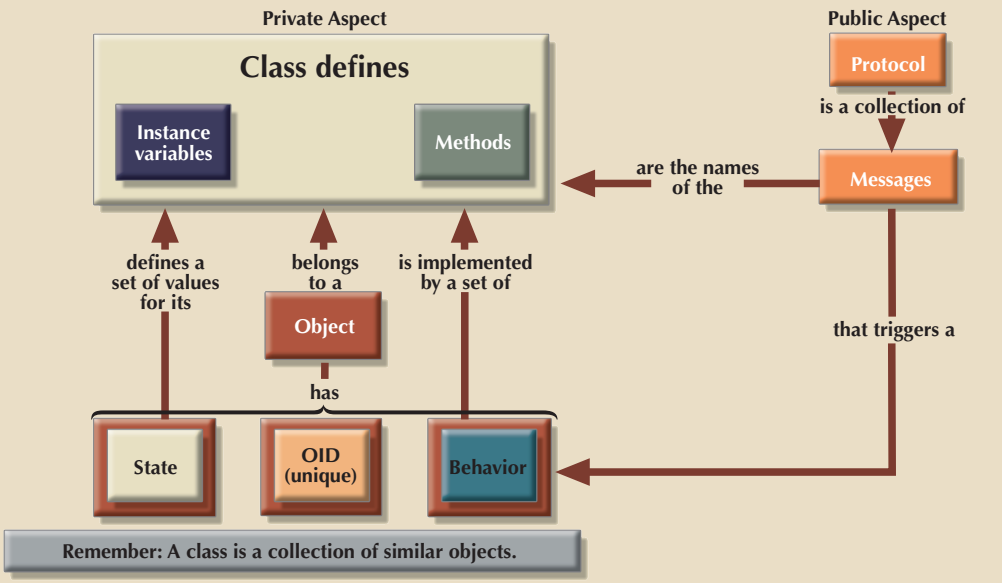
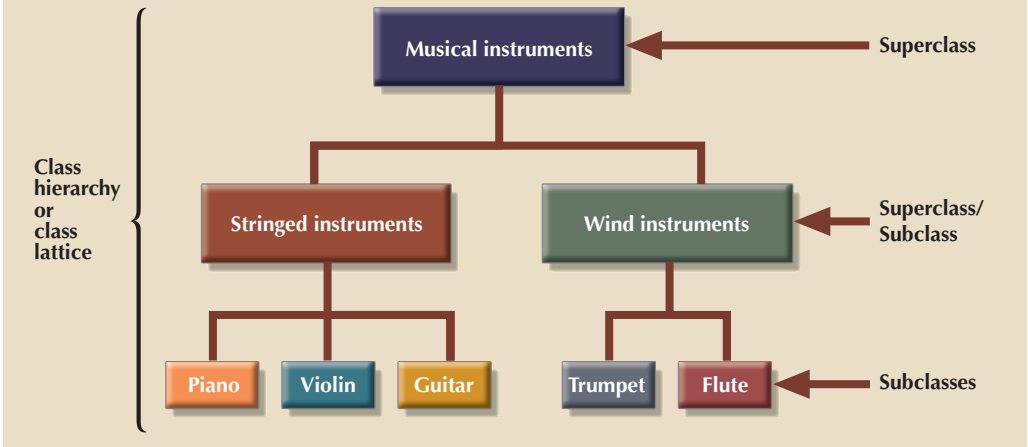


FIGURE G.9 MUSICAL INSTRUMENTS CLASS HIERARCHY



inheritance
In the object-oriented data model, the ability of an object to inherit the data structure and methods of the classes above it in the class hierarchy. See also *class hierarchy*.

Single Inheritance

Single inheritance exists when a class has only one immediate (parent) superclass above it. Such a condition is illustrated by the Stringed instruments and Wind instruments classes in Figure G.9. Most of the current OO systems support single inheritance. When the system sends a message to an object instance, the entire hierarchy is searched for the matching method, using the following sequence:

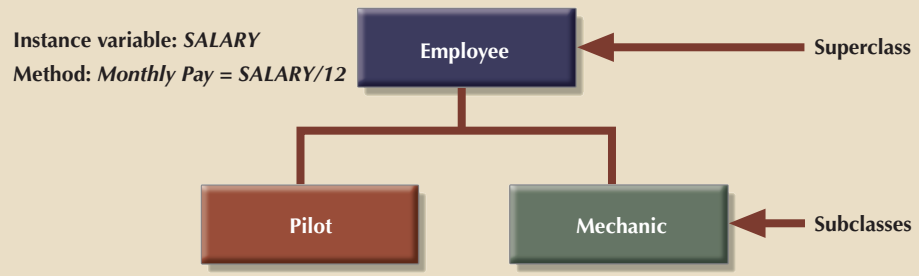
1. Scan the class to which the object belongs.
2. If the method is not found, scan the superclass.

The scanning process is repeated until either one of the following occurs:

- The method is found.
- The top of the class hierarchy is reached without finding the method. The system then generates a message to indicate that the method was not found.

For an illustration of the scanning process, let's examine the Employee class hierarchy shown in Figure G.10. If the *monthPay* message is sent to a pilot's object instance, the object will execute the *monthPay* method defined in its Employee superclass. Note the code reusability benefits obtained through object-oriented systems: the *monthPay* method's code is available to both the Pilot and Mechanic subclasses.

FIGURE G.10 SINGLE INHERITANCE

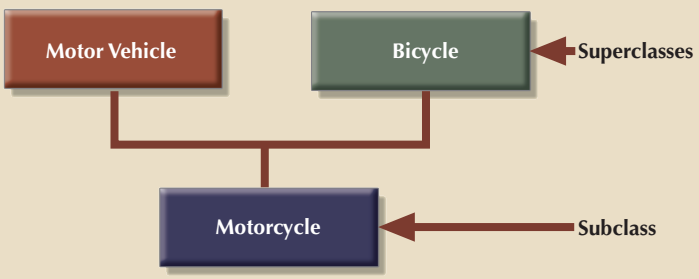


Multiple Inheritance

Multiple inheritance exists when a class can have more than one immediate (parent) superclass above it. Figure G.11 provides an example of multiple inheritance, illustrating that the Motorcycle subclass inherits characteristics from both the Motor Vehicle and Bicycle superclasses. From the Motor Vehicle superclass, the Motorcycle subclass inherits:

- Characteristics such as fuel requirements, engine pistons, and horsepower.
- Behavior such as start motor, fill gas, and depress clutch.

FIGURE G.11 MULTIPLE INHERITANCE



single inheritance
In the object oriented data model, the property of an object that allows it to have only one parent superclass from which it inherits its data structure and methods. See also *inheritance, multiple inheritance*.

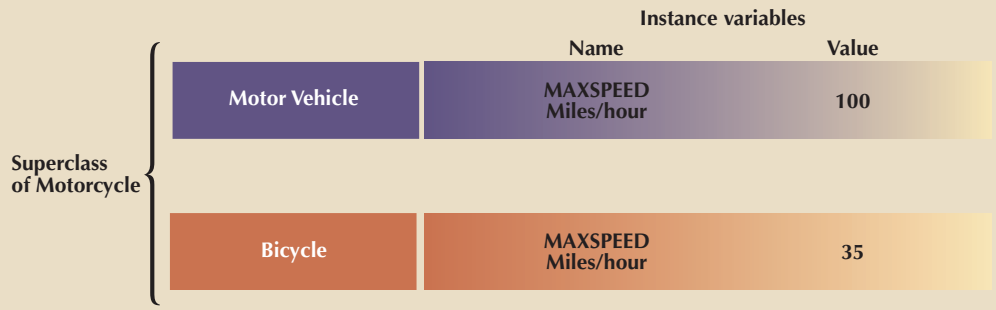
multiple inheritance
Exists when a class can have more than one immediate (parent) superclass above it.

From the Bicycle superclass, the Motorcycle subclass inherits:

- Characteristics such as two wheels and handlebars.
- Behavior such as straddle the seat and move the handlebar to turn.

The assignment of instance variable or method names must be treated with some caution in a multiple inheritance class hierarchy. For example, if you use the same name for an instance variable or method in each of the superclasses, the OO system must be given some way to decide which method or attribute to use. To illustrate that point, let's suppose that both the Motor Vehicle and Bicycle superclasses shown in Figure G.12 use a MAXSPEED instance variable.

FIGURE G.12 MOTOR VEHICLE AND BICYCLE INSTANCE VARIABLES



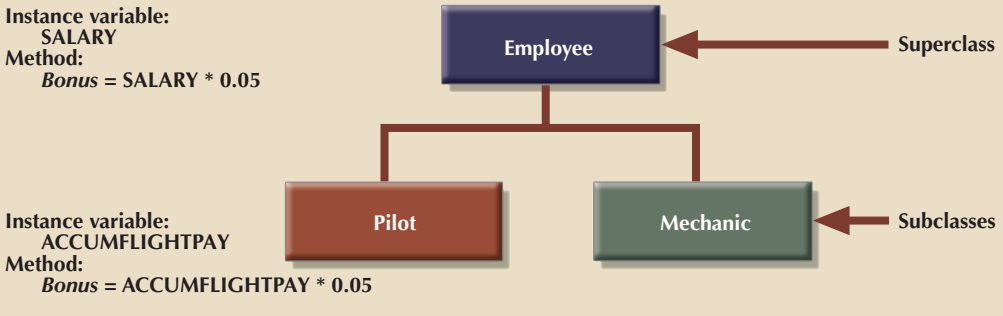
Which version of the MAXSPEED instance variable will be inherited by the Motorcycle's method in this case? A human being would use judgment to correctly assign the 100 miles/hour value to the motorcycle. The OO system, however, cannot make such value judgments and might:

- Produce an error message in a pop-up window, explaining the problem.
- Ask the end user to supply the correct value or to define the appropriate action.
- Yield an inconsistent or unpredictable result.
- Use user-defined inheritance rules for the subclasses in the class lattice. These inheritance rules govern a subclass's inheritance of methods and instance variables.

G-3i Method Overriding and Polymorphism

You may override a superclass method definition by redefining the method at the subclass level. For an illustration of the method override, look at the Employee class hierarchy depicted in Figure G.13.

FIGURE G.13 EMPLOYEE CLASS HIERARCHY METHOD OVERRIDE

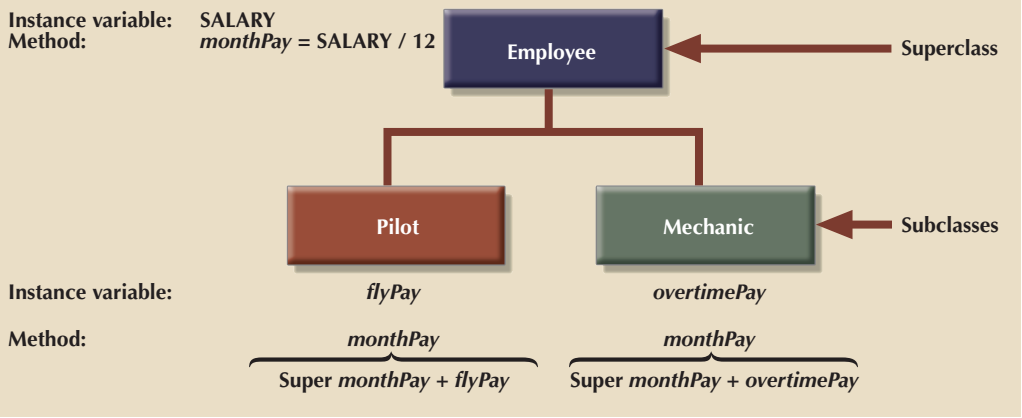


As you examine the summary presented in Figure G.13, note that a *Bonus* method has been defined to compute a Christmas bonus for all employees. The specific bonus computation depends on the type of employee. In this case, *with the exception of pilots*, an employee receives a Christmas bonus equal to 5 percent of his/her salary. Pilots receive a Christmas bonus based on accumulated flight pay rather than annual salary. By defining the *Bonus* method in the Pilot subclass, you are overriding the Employee Bonus method for all objects that belong to the Pilot subclass. However, the Pilot subclass bonus redefinition does not affect the bonus computation for the Mechanic subclass. In contrast to method overriding, polymorphism allows different objects to respond to the same message in different ways. Polymorphism is a very important feature of OO systems because it allows objects to behave according to their specific characteristics. In OO terms, **polymorphism** means that:

- You may use the same name for a method defined in different classes in the class hierarchy.
- The user may send the same message to different objects that belong to different classes and yet will always generate the correct response.

To illustrate the effect of polymorphism, let's examine the expanded Employee class hierarchy shown in Figure G.14. Using the class hierarchy in Figure G.14, the system computes a pilot's or mechanic's monthly pay by sending the same message, *monthPay*, to the pilot or mechanic object. The object will return the correct monthly pay amount even though the *monthPay* includes *flyPay* for the pilot object and *overtimePay* for the mechanic object. The computation of the regular monthly salary payment for both subclasses (Pilot and Mechanic) is the same: the annual salary divided by 12 months.

FIGURE G.14 EMPLOYEE CLASS HIERARCHY POLYMORPHISM



Note

Figure G.14 used Smalltalk syntax: **Super** *monthPay* in the pilot's *monthPay* method indicates that the object inherits its superclass *monthPay* method. Other OOPLs, such as C++, use the *Employee.monthPay* syntax.

As you examine the polymorphism example in Figure G.14, note that:

- The Pilot *monthPay* method definition overrides and expands the Employee *monthPay* method defined in the Employee superclass.
- The *monthPay* method defined in the Employee superclass is reused by the Pilot and Mechanic subclasses.

polymorphism
An object oriented data model characteristic by which different objects can respond to the same message in different ways.

Thus, polymorphism augments method override to enhance the code reusability so prized in modular programming and design.

G-3j Abstract Data Types

A data type describes a set of objects with similar characteristics. All conventional programming languages use a set of predefined base data types (real, integer, and string or character). Base data types are subject to a predefined set of operations. For example, the integer base data type allows operations such as addition, subtraction, multiplication, and division.

Conventional programming languages also include type constructors, the most common of which is the record type constructor. For example, a programmer can define a CUSTOMER record type by describing its data fields. The CUSTOMER record represents a new data type that will store CUSTOMER data, and the programmer can directly access that data structure by referencing the record's field names. A record data type allows operations such as WRITE, READ, or DELETE. However, new operations cannot be defined for base data types.

Like conventional data types, an **abstract data type (ADT)** describes a set of similar objects. However, an abstract data type differs from a conventional data type in that:

- The ADT's operations are user-defined.
- The ADT does not allow direct access to its internal data representation or method implementation. In other words, the ADT encapsulates its definition, thereby hiding its characteristics.

Some OO systems, such as Smalltalk, implement base data types as ADTs.

To create an abstract data type, you must define:

- Its name.
- The data representation or instance variables of the objects belonging to the abstract data type; each instance variable has a data type that may be a base data type or another ADT.
- The abstract data type operations and constraints, both of which are implemented through methods.

You might have noted that the abstract data type definition resembles a class definition. Some OO systems differentiate between class and type, using *type* to refer to the *class data structure and methods* and *class* to refer to the *collection of object instances*. A type is a more static concept, while a class is a more run-time concept. In other words, when you define a new class, you are actually defining a new type. The type definition is used as a pattern or template to create new objects belonging to a class at run time.

A simple example will help you understand the subtle distinction between OO type and class. Suppose you bought a cross-stitch pattern with which to create pillow covers. The pattern you bought includes the *description* of its structure as well as *instructions* about its use. That pattern will be the type definition. The collection of all actual pillow covers, each with a unique serial number or OID, that you create with the help of that pattern constitutes the class.

Together with inheritance, abstract data types provide support for complex objects. A **complex object** is formed by combining other objects in a set of complex relations. An example of such a complex object might be found in a security system that uses different data types, such as:

- Conventional (tabular) employee data; for example name, phone, or date of birth.
- Bitmapped data to store the employee's picture.
- Voice data to store the employee's voice pattern.

abstract data type (ADT)
Data type that describes a set of similar objects with shared and encapsulated data representation and methods. An abstract data type is generally used to describe complex objects. See also *class*.

complex object
An object formed by several different objects in complex relationships. See also *abstract data types*.

The ability to deal in a relatively easy manner with such a complex data environment gives OO credibility in today's database marketplace.

G-3k Object Classification

An object can be classified according to the characteristics (simple, composite, compound, hybrid, and associative) of its attributes. A **simple object** is an object that contains only single-valued attributes and has no attributes that refer to another object. For example, an object that describes the current semester can be defined as having the following single-valued attributes: SEM_ID, SEM_NAME, SEM_BEGIN_DATE, and SEM_END_DATE.

A **composite object** is an object that contains at least one multivalued attribute and has no attributes that refer to another object. An example of a composite object would be a MOVIE object in a movie rental system. For example, MOVIE might be defined as having the following attributes: MOVIE_ID, MOVIE_NAME, MOVIE_PRICE, MOVIE_TYPE, and MOVIE_ACTORS. In that case, MOVIE_ACTORS is a multivalued attribute that tracks the many performers in the movie.

A **compound object** is an object that contains at least one attribute that references another object. An example is the STUDENT object in Table G.2. In that example, the ADVISOR attribute refers to the PROFESSOR object.

A **hybrid object** is an object that contains a repeating group of attributes, at least one of which refers to another object. A typical example of a hybrid object is the invoice example introduced in Chapter 3, The Relational Database Model, in Figure 3.30. In that case, an invoice contains many products, and each product has a quantity sold and a unit price. The object representation of the invoice contains a repeating group of attributes that represent the product, quantity sold, and unit price (PROD_CODE, LINE_UNITS, and LINE_PRICE) for each product sold. Therefore, the object representation of the invoice does not require a new INV_LINE object as in the ER model representation.

Finally, an **associative object** is an object used to represent a relationship between two or more objects. The associative object can contain its own attributes to represent specific characteristics of the relationship. A good example of an associative object is the Enroll example in Chapter 3, Figure 3.26. In that case, the ENROLL object relates to a STUDENT and a CLASS object and includes an ENROLL_GRADE attribute that represents the grade earned by the student in the class.

In real-world data models, you find fewer simple and composite objects and more compound, hybrid, and associative objects. Those types of objects will be discussed in greater detail in Section G-4c.

simple object
An object that contains only single-valued attributes and has no attributes that refer to another object.

composite object
An object that contains at least one multivalued attribute and has no attributes that refer to another object.

compound object
An object that contains at least one attribute that references another object.

hybrid object
The type of object classification that contains and represents a repeating group of attributes. One or more of the attributes reference another object that usually summarizes the contents of the hybrid object.

associative object
An object used to represent a relationship between two or more objects.

G-4 Characteristics of an Object-Oriented Data Model

The object-oriented concepts described in previous sections represent the core characteristics of an **object-oriented data model (OODM)**, also known as an object data model, or ODM. At the very least, an object-oriented data model must:

- Support the representation of complex objects.
- Be **extensible**; that is, it must be capable of defining new data types as well as the operations to be performed on them.
- Support encapsulation; that is, the data representation and the method's implementation must be hidden from external entities.

object-oriented data model (OODM)
A data model whose basic modeling structure is an object.

extensible
Capable of being extended by adding new data types and the operations to be performed on them.

- Exhibit inheritance; an object must be able to inherit the properties (data and methods) of other objects.
- Support the notion of object identity (OID) described earlier in this appendix.

For instructional purposes and to the extent possible, OODM component descriptions and definitions will be used to correspond to the entity relationship model components described in Chapter 3. Although most of the basic OODM components were defined earlier in this appendix chapter, a quick summary may help you read the subsequent material more easily:

- The OODM models real-world entities as *objects*.
- Each object is composed of *attributes* and a set of *methods*.
- Each attribute can *reference* another object or a set of objects.
- The attributes and the methods' implementation are hidden, *encapsulated*, from other objects.
- Each object is identified by a unique *object ID* (*OID*), which is independent of the value of its attributes.
- Similar objects are described and grouped in a *class* that contains the description of the data (attributes or instance variables) and the methods' implementation.
- The class describes a *type* of object.
- Classes are organized in a *class hierarchy*.
- Each object of a class *inherits* all properties of its superclasses in the class hierarchy.

Armed with that summarized OO component description, note the comparison between the OO and ER model components presented in Table G.3.

TABLE G.3

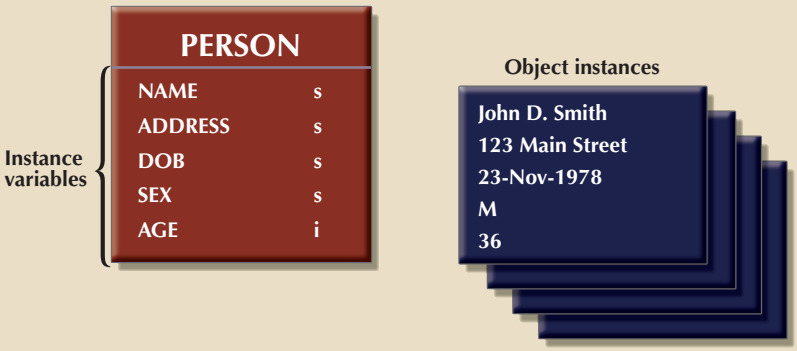
COMPARING THE OO AND ER MODEL COMPONENTS

OO DATA MODEL	ER MODEL
Type	Entity definition
Object	Entity
Class	Entity set
Instance variable	Attribute
N/A	Primary key
OID	N/A
Method	N/A
Class hierarchy	ER diagram

G-4a Object Schemas: The Graphical Representation of Objects

A graphical representation of an object resembles a box, with the instance variable names inside the box. Generally speaking, the object representation is shared by all objects in the class. Therefore, you will discover that the terms *object* and *class* are often used interchangeably in the illustrations. With that caveat in mind, let's begin by examining the illustration based on the Person class, shown in Figure G.15. In that case, the instance variables NAME, ADDRESS, DOB, and SEX use a string base data type, and the AGE instance variable uses an integer base data type.

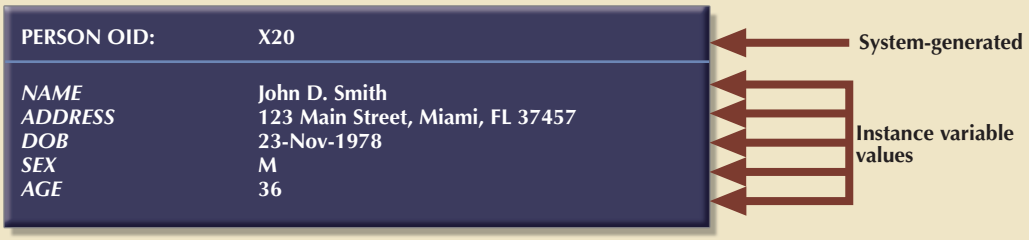
FIGURE G.15 SHARED REPRESENTATION FOR ALL OBJECTS OF THE CLASS PERSON



Next, let's examine the *state* of a Person object instance. (See Figure G.16.) As you examine Figure G.16, note that the AGE instance variable can also be viewed as a *derived* attribute. Derived attributes may be implemented through *methods*. For instance, a method named *Age* can be created for the Person class. That method will return the difference in years between the current date and the date of birth (DOB) for a given object instance. Aside from the fact that methods can generate derived attribute values, methods have the added advantages of encapsulation and inheritance.

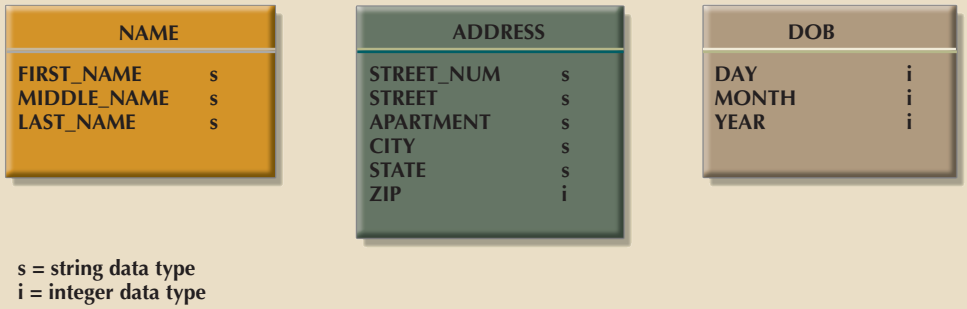
object space
The equivalent of the database schema, as seen by the designer in an object oriented database.

FIGURE G.16 STATE OF A PERSON OBJECT INSTANCE



Keep in mind that the OO environment allows you to create abstract data types from base data types. For example, NAME, ADDRESS, and DOB are composite attributes that can be implemented through classes or ADTs. To illustrate that point, Name, Address, and DOB have been defined to be abstract data types in Figure G.17.

FIGURE G.17 DEFINING THREE ABSTRACT DATA TYPES

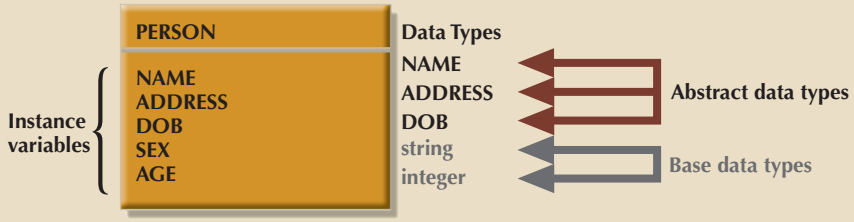


As you examine Figure G.17, note that the Person class now contains attributes that point to objects of other classes or abstract data types. The new data types for each instance variable of the class Person are shown in Figure G.18.

The **object space**, or **object schema**, is used to represent the composition of the state of an object at a given time.

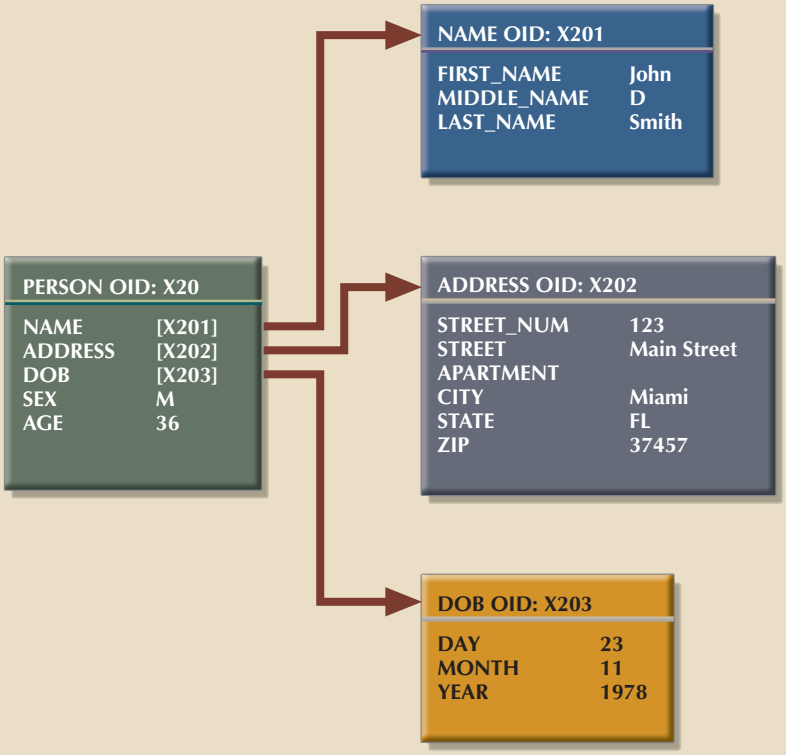
object schema
See *object space*.

FIGURE G.18 OBJECT REPRESENTATION FOR INSTANCES OF THE CLASS PERSON WITH ADTs



The object's state for an instance of class Person is illustrated in Figure G.19. As you examine Figure G.19, note the use of OIDs to reference other objects. For example, the attributes NAME, ADDRESS, and DOB now contain an OID of an instance of their respective class or ADT instead of the base value. The use of OIDs for object references avoids the data inconsistency problem that would appear in a relational system if the end user were to change the primary key value when changing the object's state. That is because the OID is independent of the object's state.

FIGURE G.19 OBJECT STATE FOR AN INSTANCE OF THE CLASS PERSON, USING ADTs



referential object sharing
When an object instance is referenced by other objects. That is, two or more different objects point to the same object instance, a change in the referenced object instance values is automatically reflected in all other referring objects.

To illustrate this point further, a rental property application will be used by which many rental properties and the persons living in them are tracked. In that case, two people living at the same address are likely to reference the same Address object instance. (See Figure G.20.) This condition is sometimes labeled as **referential object sharing**. A change in the Address object instance will be reflected in both Person instances.

Figure G.20 illustrates the state of two different object instances of the class Person; both object instances reference the same Address object instance. Note that Figure G.20 depicts four different classes or ADTs: Person (two instances), Name (two instances), Address, and DOB (two instances).

G-4b Class-Subclass Relationships

Do you remember that classes inherit the properties of their superclasses in the class hierarchy? That property leads to the use of the label "is a" to describe the relationship between the classes within the hierarchy. That is, an employee *is a* person, and a student *is* a person. This basic idea is sufficiently important to warrant a more detailed illustration based on the class hierarchy. (See Figure G.21.)

FIGURE G.20 REFERENTIAL OBJECT SHARING

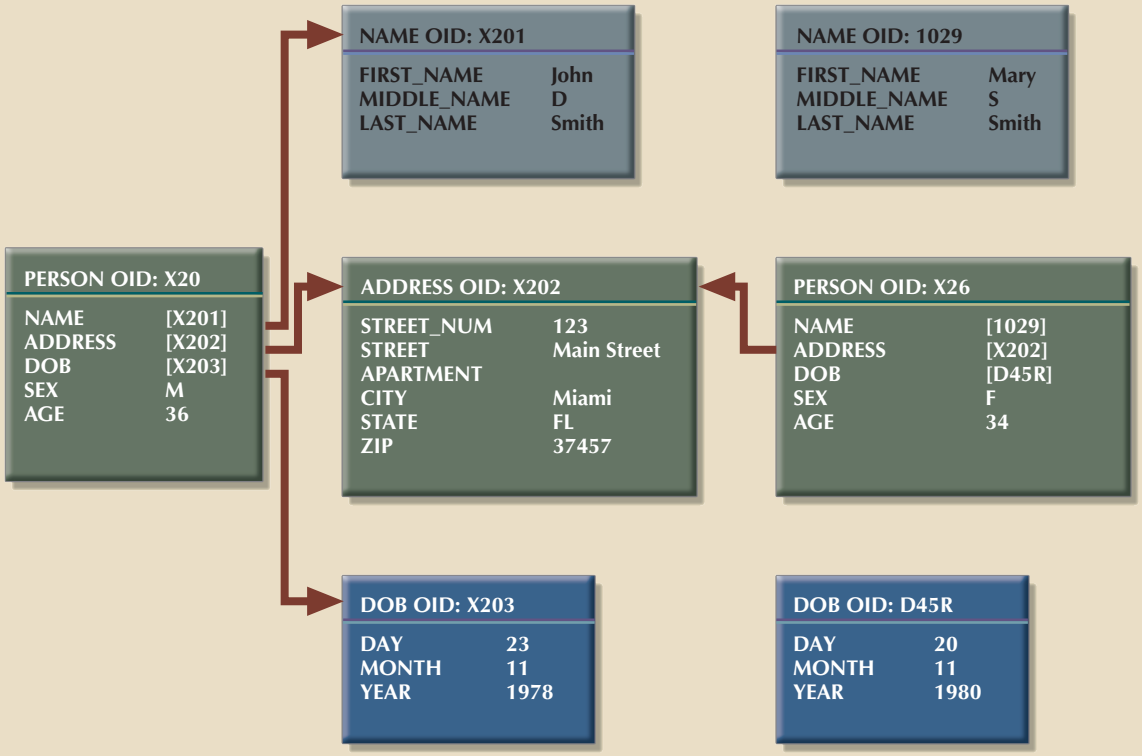
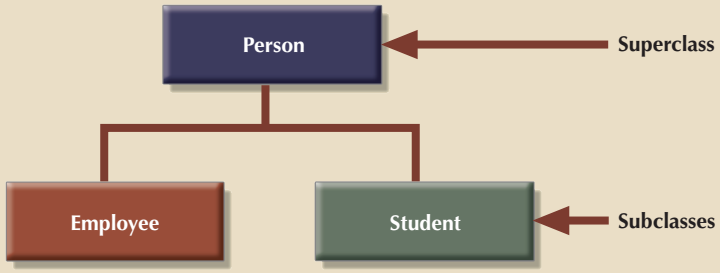
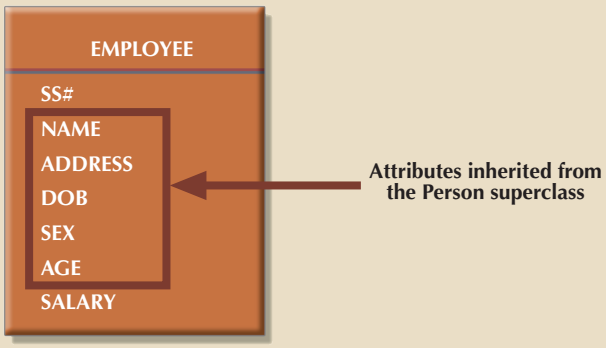


FIGURE G.21 CLASS HIERARCHY



In the hierarchy shown in Figure G.21, the Employee object is described by seven attributes, shown in Figure G.22. Social Security number (SS#) is recorded as a string base data type, and SALARY is recorded as an integer base data type. The NAME, ADDRESS, DOB, SEX, and AGE attributes are all inherited from the Person superclass.

FIGURE G.22 EMPLOYEE OBJECT REPRESENTATION



That example is based on the fact that the OODM supports the *class-subclass relationship*, for which it enforces the necessary integrity constraints. Note that the relationship between a superclass and its subclasses is 1:M; that is, if you assume single inheritance, each superclass can have many subclasses and each subclass is related to only one superclass.

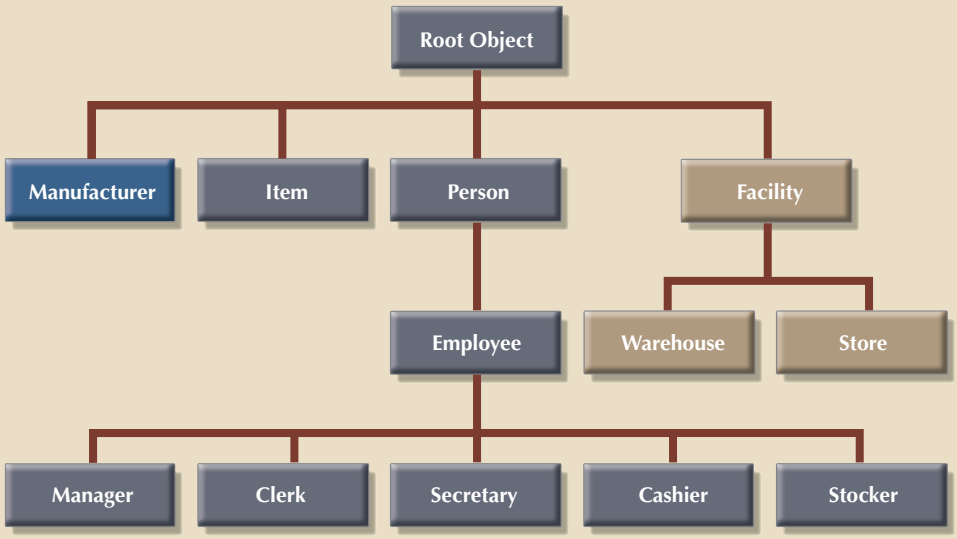
interobject relationship
An attribute-class relationship created when an object's attribute references another object of the same or a different class.

G-4c Interobject Relationships: Attribute-Class Links

In addition to supporting the class-subclass relationship, the OODM supports the attribute-class relationship. An attribute-class relationship, or **interobject relationship**, is created when an object's attribute references another object of the same or different class.

The interobject relationship is different from the class-subclass relationship explored earlier. To illustrate this difference, let's examine the class hierarchy for the EDLP (Every Day Low Prices) Retail Corporation, shown in Figure G.23.

FIGURE G.23 CLASS HIERARCHY FOR THE EDLP RETAIL CORPORATION

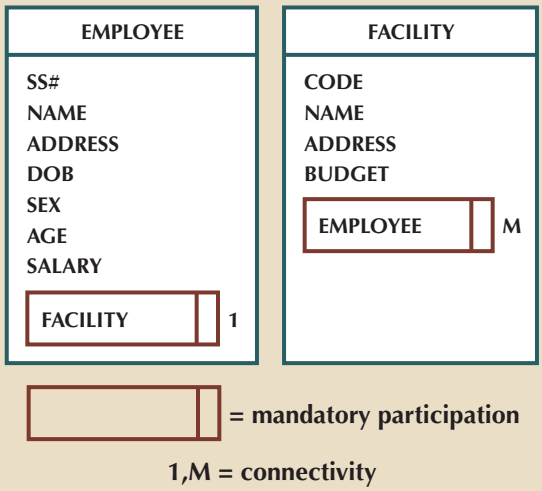


As you examine Figure G.23, note that all classes are based on the Root Object superclass. The class hierarchy contains the classes Manufacturer, Item, Person, and Facility.

The Facility class contains the subclasses Warehouse and Store. The Person class contains the subclass Employee, which, in turn, contains the subclasses Manager, Clerk, Secretary, Cashier, and Stocker. The following discussion will use the simple class hierarchy shown in Figure G.23 to illustrate basic 1:M and M:N relationships.

Representing 1:M Relationships Based on the class hierarchy in Figure G.23, a one-to-many relationship exists between Employee and Facility: each Employee works in only one Facility, but each Facility has several Employees. Figure G.24 shows how that relationship may be represented.

FIGURE G.24 REPRESENTING A 1:M RELATIONSHIP



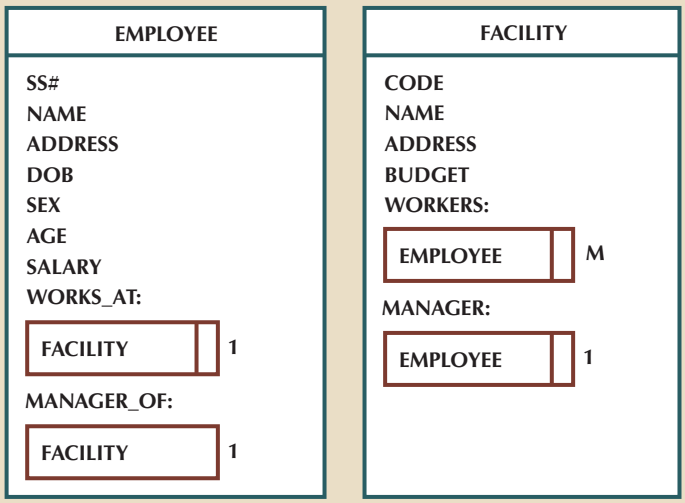
As you examine the relationship between Employee and Facility portrayed in Figure G.24, note that the Facility object is included within the Employee object and vice versa; that is, the Employee object is also included within the Facility object. The following techniques will be used to examine the relationships in greater detail:

- Related classes are enclosed in boxes to make relationships more noticeable.
- The double line on the boxes' right side indicates that the relationship is mandatory.
- Connectivity is indicated by labeling each box. In this case, a *1* was put next to Facility in the Employee object to indicate that each employee works in only one facility. The *M* beside Employee in the Facility object indicates that each facility has many employees.

Note that the ER notation is used to represent a mandatory entity and to indicate the connectivity of a relationship (1:M). The purpose of the notation is to maintain consistency with earlier diagrams.

Rather than just include the object box within the class, the preference is to use a name that is descriptive of the class *characteristic* being modeled. That procedure is especially useful when two classes are involved in more than one relationship. In those cases, the attribute's name should be written above the class box, and the class box should be indented to indicate that the attribute will reference the class. For example, two relationships between Employee and Facility can be represented by using WORKS_AT and WORKERS, as indicated in Figure G.25. Note that two relationships exist:

FIGURE G.25 REPRESENTING 1:1 AND 1:M RELATIONSHIPS



***Note*: the Manager attribute indicates the facility's general manager**

1. The 1:M relationship is based on the business rule "each facility employs many employees, and each employee is employed by only one facility."

2. The 1:1 relationship is based on the business rule "each facility is managed by only one employee, and each manager manages only one facility."

As you examine Figure G.25, note that the relationships are represented in both participating classes. That condition allows you to invert the relationship, if necessary. For example, the Facility object within the Employee object represents the "Manager_of" relationship. In this case, the Facility object is optional and has a maximum connectivity of 1. The Employee and Facility objects are examples of compound objects. Another type of 1:M relationship can be illustrated by examining the relationship between employees and their dependents. To establish that relationship, you first create a Dependent subclass, using Person as its superclass. *Note that a Dependent subclass cannot be created by using Employee as its superclass because the class hierarchy represents an "is a" relationship.* In other words, each Manager is an Employee, each Employee is a Person, each Dependent is a Person, and each Person is an Object in the object space—but each Dependent is not an Employee. Figure G.26 shows the proper presentation of the relationship between Employee and Dependent.

As you examine Figure G.26, note that Dependent is optional to Employee and that Dependent has a 1:M relationship with Employee. However, Employee is mandatory to Dependent. *The weak entity concept disappears in the OODM because each object instance is identified by a unique OID.*

Representing M:N Relationships Using the same EDLP Retail Corporation class hierarchy, a many-to-many (M:N) relationship can be illustrated by exploring the relationship between Manufacturer and Item, as represented in Figure G.27. Figure G.27 depicts a condition in which each Item may be produced by many Manufacturers, and each Manufacturer may produce many Items. Thus, Figure G.27 represents a conceptual view of the M:N relationship between Item and Manufacturer. In this representation, Item and Manufacturer are both compound objects.

Also note that the CONTACT attribute in the Manufacturer class in Figure G.27 references only one instance of the Person class. A slight complication arises at this point. It is likely that each contact (person) has a phone number, yet no phone number attribute

FIGURE G.26 EMPLOYEE-DEPENDENT RELATIONSHIP

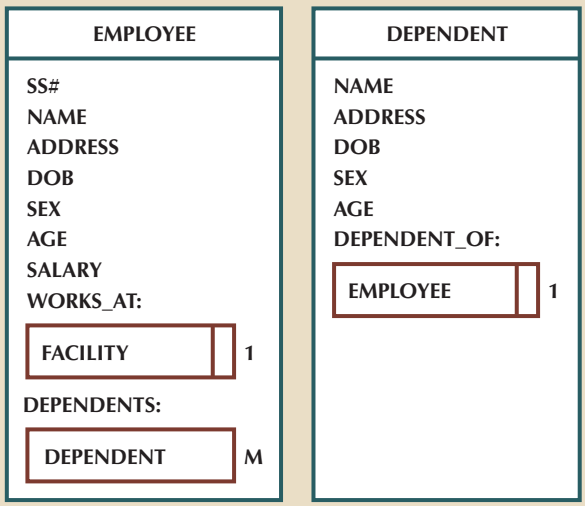
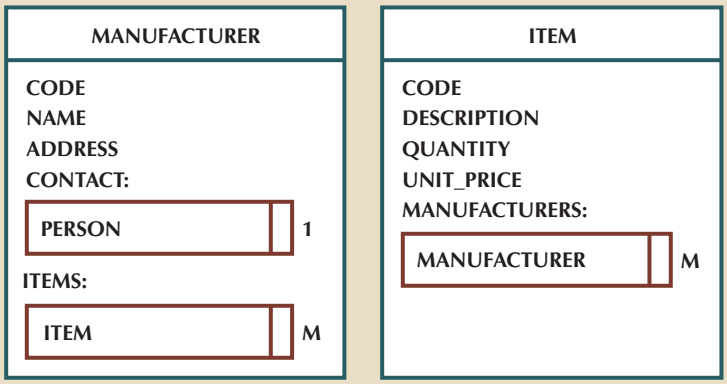


FIGURE G.27 REPRESENTING THE M:N RELATIONSHIP



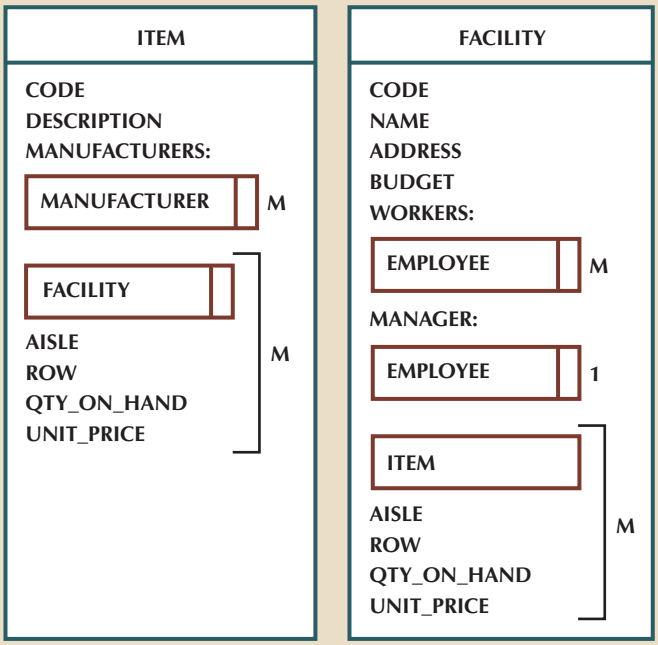
was included in the Person class. In that case, the designer may add the attribute so it will be available to all Person subclasses.

Representing M:N Relationships with an Intersection Class

Suppose you add a condition to the just-explored ITEM class that allows you to track additional data for each item. For example, let's represent the relationship between Item and Facility so that each Facility may contain several Items, and each Item may be located at several Facilities. In addition, you want to track the quantity and location (aisle and row) of an Item at each Facility. Those conditions are illustrated in Figure G.28. The right square bracket "]" in Figure G.28 indicates that the included attributes are treated as one logical unit. Therefore, each Item instance may contain several occurrences of Facility, each accompanied by related values for the AISLE, ROW, QTY_ON_HAND, and UNIT_PRICE attributes. The inverse is true for each instance of Facility. The Item and Facility objects in this relationship are hybrid objects with a repeating group of attributes. Note that the semantic requirements for this relationship indicate that the Item or Facility objects are accessed first so the aisle, row, and quantity on hand are known for each item.

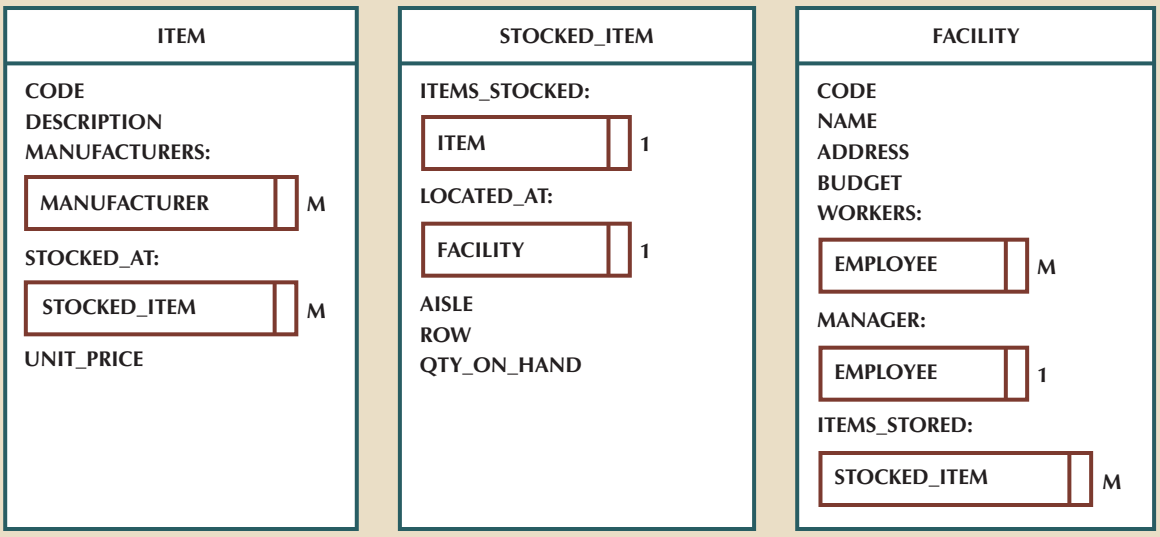
To translate the preceding discussion to a more relational view of the M:N scenario, you would have to define an **intersection** (bridge) **class** to connect both Facility and

FIGURE G.28 REPRESENTING THE M:N RELATIONSHIP WITH ASSOCIATED ATTRIBUTES



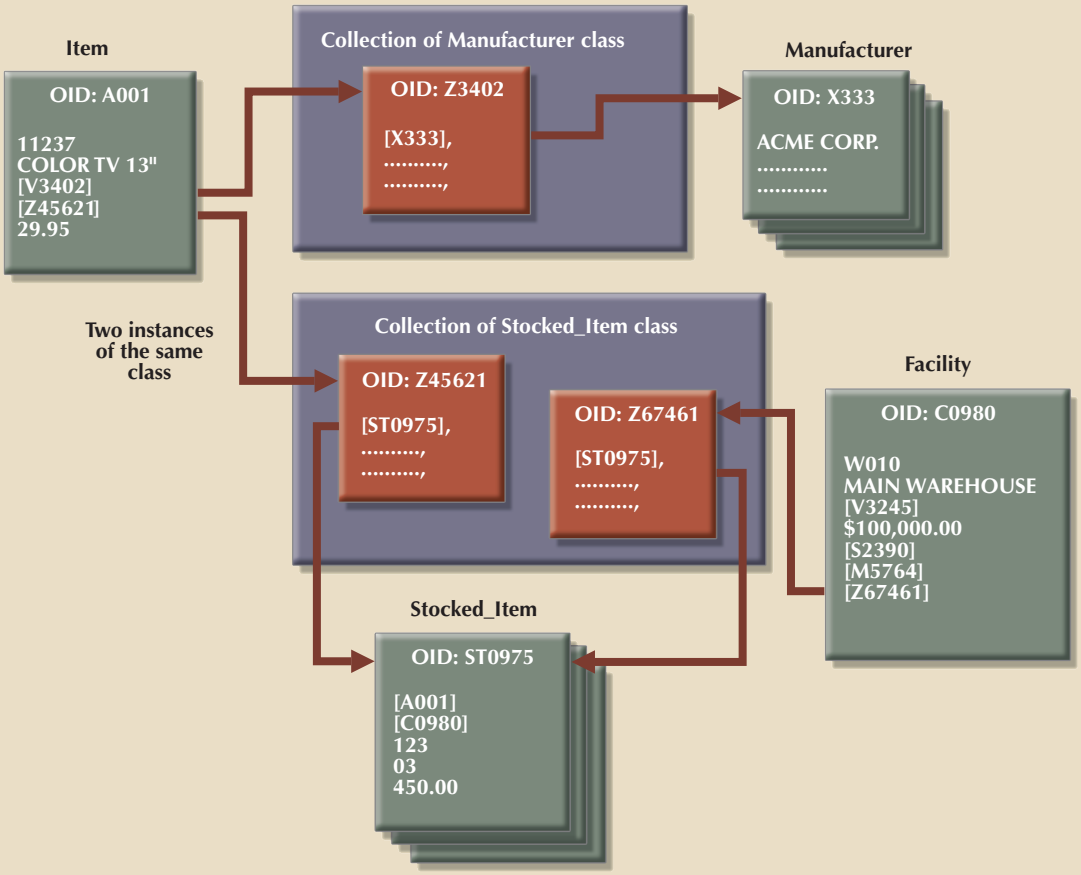
Item and store the associated attributes. In that case, you might create a Stocked_Item associative object class to contain the Facility and Item object instances and the values for each of the AISLE, ROW, QTY_ON_HAND, and UNIT_PRICE attributes. Such a class is equivalent to the Interclass_Connection construct of the Semantic Data Model. Figure G.29 shows how the Item, Facility, and Stocked_Item object instances might be represented.

FIGURE G.29 REPRESENTING THE M:N RELATIONSHIP WITH INTERSECTION CLASS



Having examined the depiction of the basic OO relationships, you can represent the object space as shown in Figure G.30.

FIGURE G.30 OBJECT SPACE REPRESENTATION



Because Figure G.30 contains much critical design information, you should examine the following points in particular:

- The Stocked_Item associative object instance contains references to an instance of each related (Item and Facility) class. The Stocked_Item intersection class is necessary only when you must keep track of the additional information referred to earlier.
- The Item object instance in this object schema contains the collection of Stocked_Item object instances, each one of which contains a Facility object instance. The inverse of that relationship is also true: a Facility object instance contains the collection of Stocked_Item object instances, each one of which contains an Item object instance. *You should realize that those two relationships represent two different application views of the same object schema*. It is desirable for a data model to provide such flexibility.
- The interobject references use the OID of the referenced objects in order to *access* and *include* them in the object space.
- The values inside square brackets "[ ]" represent the OID of an object instance of some class. *The "collection of" classes represent a class of objects in which each object instance contains a collection of objects of some class*. For example, the Z3402 and Z45621 OIDs reference objects that constitute a collection of Manufacturers and a collection of Stocked_Items, respectively.

- In the relational model, the ITEM table would not contain any data regarding the MANUFACTURERs or the STOCKED_ITEMs in its structure. To provide (combined) information about ITEM, STOCKED_ITEM, and FACILITY, you would have to perform a relational join operation. The OODM does not need joins to combine data from different objects because the Item object *contains* the references to the related objects; those references are automatically brought into the object space when the Item object is accessed.

G-4d Late and Early Binding: Use and Importance

A desirable OODM characteristic is its ability to let any object's attribute contain objects that define different data types (or classes) at different times. With that feature, an object can contain a *numeric value* for a given instance variable, and the next object (of the same class) can contain a *character value* for the same instance variable. That characteristic is achieved through late binding. With **late binding**, the data type of an attribute is not known until execution time or run time. Therefore, two different object instances of the same class can contain values of different data types for the same attribute.

In contrast to the OODM's ability to use late binding, a conventional DBMS requires that a base data type be defined for each attribute at the time of its creation. For example, suppose you want to define an INVENTORY to contain the following attributes: ITEM_TYPE, DESCRIPTION, VENDOR, WEIGHT, and PRICE. In a conventional DBMS, you create a table named INVENTORY and assign a base data type to each attribute, as shown in Figure G.31.

Recall from earlier chapters that when the designer is working with conventional database systems, (s)he must define the data type for each attribute *when the table structure is defined*. That approach to data type definition is called early binding. **Early binding** allows the database to check the data type for each of the attribute's values at compilation or definition time. For instance, the ITEM_TYPE attribute in Figure G.31 is limited to numeric values. Similarly, the VENDOR attribute may contain only numeric values to match the primary key of some row in a VENDOR table with the same numeric value restriction.

Now let's take a look at Figure G.32 to see how an OODM would handle this early-binding problem. As was true in the conventional database environment, the OODM allows the data types to be defined at creation time. However, quite *unlike* the conventional database, the OODM allows the data types to be user-defined ADTs. In this example of early binding, the abstract data types Inv_type, String_of_characters, Vendor, Weight, and Money are associated with the instance variables at definition time. Therefore, the designer may define the required operations for each data type. For example, the Weight data type can have methods to show the weight of the item in pounds or kilograms. Similarly, the Money data type may have methods to return the price as numbers or letters denominated in U.S. dollars, euros, or other currencies. (Remember that abstract data types are implemented through classes.)

In a late-binding environment, the object's attribute data type is not known prior to its use. Therefore, an attribute can have any type of value assigned to it. Using the same basic data set described earlier, Figure G.33 shows the attributes (instance variables) ITEM_TYPE, DESCRIPTION, VENDOR, WEIGHT, and PRICE without a prior data type definition. Because no data types are predefined for the class instance variables, two different objects of the Inventory class may have different value types for the same attribute. For example, ITEM_TYPE can be assigned a character value in one object instance and a numeric value in the next instance. Late binding also plays an important role in polymorphism, allowing the object to decide which implementation method to use at run time.

late binding
A characteristic in which the data type of an attribute is not known until execution time or run-time.

early binding
A property by which the data type of an object's attribute must be known at definition time, bonding the data type to the object's attribute. Characteristic of an object oriented data model. See also *late binding*.

FIGURE G.31 INVENTORY TABLE WITH PREDETERMINED (BASE) DATA TYPES

Table: INVENTORY

Attributes	Conventional (Base) Data Type
ITEM_TYPE	Numeric
DESCRIPTION	Character
VENDOR	Numeric
WEIGHT	Numeric
PRICE	Numeric

FIGURE G.32 INVENTORY CLASS WITH EARLY BINDING

Class: INVENTORY

Instance variables	Type
ITEM_TYPE	Inv_type
DESCRIPTION	String_of_characters
VENDOR	Vendor
WEIGHT	Weight
PRICE	Money

FIGURE G.33 OODM INVENTORY CLASS WITH LATE BINDING

Class: INVENTORY

Instance variables	Type
ITEM_TYPE	No data type is known when the class is created
DESCRIPTION	
VENDOR	
WEIGHT	
PRICE	

G-4e Support for Versioning

Versioning is an OODM feature that allows you to track the history of change in the state of an object. Versioning is a very powerful modeling feature, especially in computer-aided design (CAD) environments. For example, an engineer using CAD can load a machine component design in his/her workstation, make some changes, and see how those changes affect the component's operation. If the changes do not yield the expected results, the engineer can undo those changes and restore the component to its original state. Versioning is one of the reasons the OODBMS is such a strong player in the CAD and computer-aided manufacturing (CAM) arenas.

G-5 OODM and Previous Data Models: Similarities and Differences

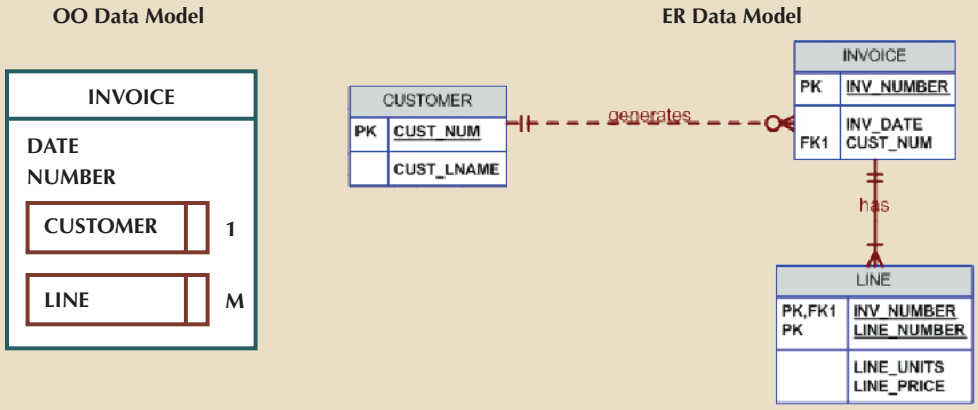
Although the OODM has much in common with relational and ER data models, the OODM introduces some fundamental differences. The following summary is designed to offer detailed comparisons to help clarify the OODM characteristics introduced in this appendix.

G-5a Object, Entity, and Tuple

versioning
A property of an OODBMS that allows the database to keep track of the different transformations performed on an object.

The OODM concept of *object* extends well beyond the concept of *entity* or *tuple* in other data models. Although an OODM object resembles the entity and the tuple in the ER and relational models, an OODM object has additional characteristics, such as behavior, inheritance, and encapsulation. Those OODM characteristics make OO modeling more natural than ER and relational modeling. In fact, the ER and relational models often force the designer to create new artificial entities to represent real-world entities. For example, in the ER model, an invoice is usually represented by two separate entities; the second (LINE) entity is usually weak because its existence depends on the first (INVOICE) entity and its primary key is partially derived from the INVOICE entity. (See Figure G.34.)

FIGURE G.34 AN INVOICE REPRESENTATION



As you examine Figure G.34, note that the ER approach requires the use of two different entities to model a single real-world INVOICE entity. That artificial construct is imposed by the relational model's inherent limitations. The ER model's artificial representation introduces additional overhead in the underlying system. In contrast, the OODM's INVOICE object is directly modeled as an object into the object space, or object schema.

G-5b Class, Entity Set, and Table

The concept of *class* can be associated with the ER and relational models' concepts of *entity set* and *table*, respectively. However, class is a more powerful concept that allows the description of not only the data structure but also the behavior of the class objects. A class also allows for both the concept and the implementation of abstract data types in the OODM. The ADT is a very powerful modeling tool because it allows the end user to create new data types and use them like any other base data type that accompanies a database. Thus, the ADT yields an increase in the *semantic* content of the objects being modeled.

G-5c Encapsulation and Inheritance

ADT brings two other OO features that are not supported in previous models: encapsulation and inheritance. Classes are organized in class hierarchies. An object belonging to a class inherits all properties of its superclasses. Encapsulation means that the data representation and the method's implementation are hidden from other objects and from the end user. In an OODM, only the methods can access the instance variables. In contrast, the conventional system's data components or fields are directly accessible from the external environment.

Conventional models do not incorporate the methods found in the OODM. The closest thing to methods is the use of triggers and stored procedures in SQL databases. However, because triggers do not include the encapsulation and inheritance benefits that are typical of the object model's methods, triggers do not yield the same functionality as methods.

G-5d Object ID

The object ID (OID) is not supported in either the ER or the relational model. Although database users may argue that Oracle Sequences and MS Access AutoNumber provide the same functionality as an OID, that argument is true *only* to the extent that they can be used to uniquely identify data elements. However, unlike the object model, in which the relationships are implicit, the relational model still uses value-based relationships such as:

```
SELECT    *
FROM      INVOICE, INV_LINE
WHERE     INVOICE.INV_ID = INV_LINE.INV_ID;
```

The hierarchical and CODASYL models support some form of ID that can be considered similar to the OID, thus supporting the argument presented by some researchers who insist that the OO evolution is a step back on the road to the old pointer systems. Therefore, OO-based systems return to the modeling and implementation complexities that were typical of the hierarchical and network models.

G-5e Relationships

The main property of any data model is found in its representation of relationships among the data components. The relationships in an OODM can be of two types: inter-class references or class hierarchy inheritance. The ER and the relational models use a *value-based* relationship approach. Using a value-based approach means that a relationship among entities is established through a common value in one or several of the entity attributes. In contrast, the OODM uses the object ID, which is identity-based, to establish relationships among objects, and *those relationships are independent of the state of the object*. (While that property makes it easy to deal with the database objects at the end-user applications level, you may have concluded that the price of the convenience is greater conceptual complexity.)

G-5f Access Methods

The ER and relational data models depend on the use of SQL to retrieve data from the database. SQL is a set-oriented query language that is based on a formally defined mathematical model. Given its set-oriented heritage and based on the value of some of its attributes, SQL uses associative access methods to retrieve related information from a database. For example, to retrieve a list of customer records based on the value of their year-to-date purchases, SQL would use:

```
SELECT    *
FROM      CUSTOMER
WHERE     CUS_YTD_BUYS >= 5000;
```

If no CUS_YTD_BUYS value parameter is specified, SQL "understands" that condition to mean "any value," thus reducing the query statement to:

```
SELECT    *
FROM      CUSTOMER;
```

As a consequence of having more semantics in its data model, the OODM produces a schema in which relations form part of the structure of the database. Accessing the structured object space resembles the record-at-a-time access of the old structured hierarchical and network models, especially if you use a 3GL or even the OOPL supported by the OODBMS. The OODM is suited to support both navigational (record-at-a-time) and set-oriented access. The navigational access is provided and implemented directly by the OODM through the OIDs. The OODM uses the OIDs to navigate through the object space structure developed by the designer.

Associative set-oriented access in the OODM must be provided through explicitly defined methods. Therefore, the designer must implement operations to manipulate the object instances in the object schema. The implementation of those operations will have an effect on performance and on the database's ability to manage data. This is where the main problem of the object model appears: the lack of a universally accepted underlying mathematical model for data manipulation. Not having a universal access standard hampers the OODM because it forces each implementation to create its own version of an **object query language (OQL)**. The OQL is the database query language used by an OODBMS. Of course, different vendors create different versions of OQL, and that, in turn, limits true interoperability. However, several groups, such as the Object Management Group (OMG, *www.omg.org*)[2], are currently working on the development of

object query language (OQL)
The database query language used by an object oriented database management system.

[2] Check the OMG main website (www.omg.org) for the most recent object standards developments. In spite of the fact that OMG develops standards, the authors see little evidence that the OMG standards are being widely adopted in the database modeling marketplace, which is the focus of their interest.

standards for object-oriented technology. For example, to facilitate modeling in an OO environment, the OMG has developed the Unified Modeling Language (UML) standard. UML represents an attempt to create a universal modeling notation to facilitate activities such as system development, data modeling, and network design. (See Appendix H to learn more about UML.)

Although there is no standard way to manipulate sets with an OQL, the relational model's SQL2 standard did not provide ways to manipulate objects in an object-oriented database, either. However, the mismatch between OQL and SQL was reduced with the publication of the SQL3, or SQL-99, standard in 1999 by the American National Standards Institute (ANSI). SQL3 paves the way toward the integration of object-oriented extensions within relational databases. For more information about this standard, see document number ANSI/ISO/IEC 9075, Parts 1–5, at *www.ansi.org*.

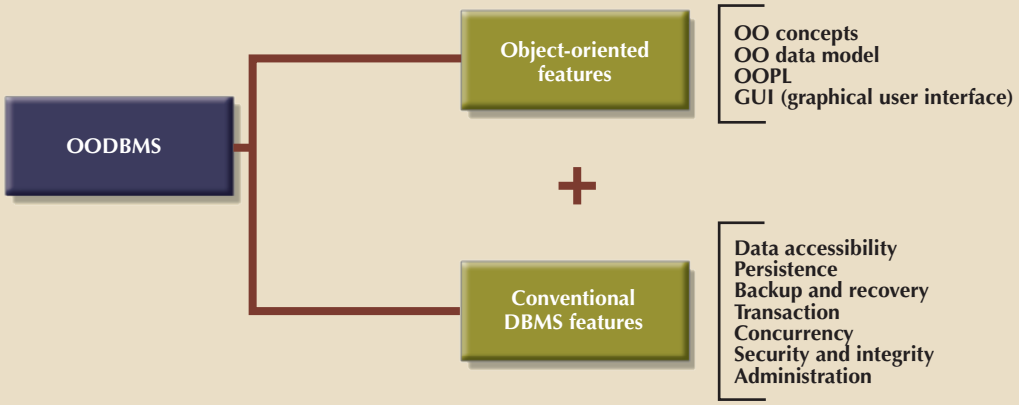
Previous sections discussed the object-oriented concepts that were derived from OOPLs. Those concepts were used to establish the characteristics of the object-oriented data model (OODM) and to study its graphical representation. This section compared the OODM to previous data models and explained that one of the major problems of the OODM is that it fails to conform to a universally accepted standard. Yet in spite of the lack of standards, there is agreement about *minimal* OODBMS characteristics. Those characteristics will be explored in the next section.

G-6 Object-Oriented Database Management Systems

object-oriented database management system (OODBMS)
Data management software used to manage data in an object-oriented database model.

During the past few years, the data management and application environment has become far more complex than the one envisioned by the creators of the hierarchical, network, or relational DBMSs. Those complex application environments may best be served by an **object-oriented database management system (OODBMS)**. The OODBMS is a database management system that integrates the benefits of typical database systems with the more powerful modeling and computational (programming) characteristics of the object-oriented data model. (See Figure G.35.)

FIGURE G.35 OBJECT-ORIENTED DATABASE MANAGEMENT SYSTEMS



Note

A DBMS based on the object model may be labeled an object-oriented database management system (OODBMS) or an object database management system (ODBMS). Given the frequent use of "OO" and "OODBMS" labels in the early stages of object-oriented research, the OODBMS label will be used here as a matter of personal preference.

OODBMS products are used to develop complex systems such as:

- Medical applications that handle digitized data such as x-rays, MRI scans, and ultrasounds, together with textual data used for medical research and patient medical history analysis.
- Financial applications in portfolio and risk management. These applications yield a real-time view of data that is based on multiple computations and aggregations applied to data acquired from complex stock transactions around the world. These applications can handle "time series" data as a user-defined data type with its own internal representation and methods.
- Telecommunications applications such as network configuration management applications that automatically monitor, track, and reconfigure communications networks based on hundreds of parameters in real time. Companies such as Ericsson, AT&T, and China Telecom use OODBMSs to support their telecommunications management applications. Motorola's Iridium global communications system manages its complex network of satellites and ground stations using an OODBMS.
- The BaBar physics experiment at the Stanford Linear Accelerator Center, which enters 1 terabyte of data per day into an OODBMS.
- Computer-aided design (CAD) and computer-aided manufacturing (CAM). These applications make use of complex data relations as well as multiple data types.
- Computer-aided software engineering (CASE) applications, which are designed to handle very large amounts of interrelated data.
- Multimedia applications, such as geographic information systems (GIS), that use video, sound, and high-quality graphics that require specialized data-management features such as intersect, inside, within, point, line, and polygon.

Many OODBMSs use a subset of the object-oriented data model features. Therefore, those who create the OODBMS tend to select the OO features that best serve the particular OODBMS's purpose, such as support for early or late binding of the data types and methods and support for single or multiple inheritance. Whatever the choices, the critical factor for a successful OODBMS implementation appears to be finding the best mix of OO and conventional DBMS features that will not sacrifice the benefits of either one.

G-6a Features of an Object-Oriented DBMS

As shown in Figure G.35, an OODBMS is the result of combining OO features such as class inheritance, encapsulation, and polymorphism with database features such as data integrity, security, persistence, transaction management, concurrency control, backup, recovery, data manipulation, and system tuning. The "Object-Oriented Database System Manifesto" (Atkinson et al., 1989)[3] was the first comprehensive attempt to define

[3] Malcolm Atkinson et al., "The Object-Oriented Database System Manifesto." This white paper first presented at the First International Conference on Deductive and Object-Oriented Databases, Kyoto, Japan, in 1989, may be downloaded from the Object Data Management Group website at *www.odmg.org*. Select "White papers," then "Database," then "OO Database System Manifesto." You will also find a series of follow-up white papers that explore the 1997–1999 OO rule extensions and standards. The extensions and standards are designed to augment the manifesto, and none claims to replace any part thereof.

OODBMS features. The document included 13 mandatory features as well as optional characteristics of the OODBMS. The 13 rules are divided into two sets: the first eight characterize an OO system, and the last five characterize a DBMS. The 13 rules are listed in Table G.4. Each rule will be discussed briefly.

TABLE G.4

THE 13 OODBMS RULES

	RULES THAT MAKE IT AN OO SYSTEM
Rule 1	The system must support complex objects.
Rule 2	Object identity must be supported.
Rule 3	Objects must be encapsulated.
Rule 4	The system must support types or classes.
Rule 5	The system must support inheritance.
Rule 6	The system must avoid premature binding.
Rule 7	The system must be computationally complete.
Rule 8	The system must be extensible.
	RULES THAT MAKE IT A DBMS
Rule 9	The system must be able to remember data locations.
Rule 10	The system must be able to manage very large databases.
Rule 11	The system must accept concurrent users.
Rule 12	The system must be able to recover from hardware and software failures.
Rule 13	Data query must be simple.

- *Rule 1. The system must support complex objects*. It must be possible to construct complex objects from existing objects. Examples of such object constructors are sets, lists, and tuples that allow the user to define aggregations of objects as attributes.
- *Rule 2. Object identity must be supported*. The OID must be independent of the object's state. This feature allows the system to compare objects at two different levels: comparing the OID (identical objects) and comparing the object's state (equal or shallow equal objects).

Note

If the objects have different OIDs but their attribute values are equal, the objects are not identical, but they are considered to be "shallow equal." To use an analogy, identical twins are alike, yet different.

- *Rule 3. Objects must be encapsulated*. Objects have a public interface, but private implementation of data and methods. The encapsulation feature ensures that only the public aspect of the object is seen, while the implementation details are hidden.
- *Rule 4. The system must support types or classes*. This rule allows the designer to choose whether the system supports types or classes. Types are used mainly at compile time to check type errors in attribute value assignments. Classes are used to store and manipulate similar objects at execution time. In other words, class is a more dynamic concept, and type is a more static one.

- *Rule 5. The system must support inheritance*. An object must inherit the properties of its superclasses in the class hierarchy. This property ensures code reusability.
- *Rule 6. The system must avoid premature binding*. This feature allows you to use the same method's name in different classes. Based on the class to which the object belongs, the OO system decides which implementation to access at run time. This feature is also known as late binding or dynamic binding.
- *Rule 7. The system must be computationally complete*. The basic notions of programming languages are augmented by features common to the database data manipulation language (DML), thereby allowing you to express any type of operation in the language.
- *Rule 8. The system must be extensible*. The final OO feature concerns the system's ability to define new types. There is no management distinction between user-defined types and system-defined types.
- *Rule 9. The system must be able to remember data locations*. The conventional DBMS stores its data permanently on disk; that is, the DBMS displays data persistence. OO systems usually keep the entire object space in memory; once the system is shut down, the entire object space is lost. Much of the OODBMS research has focused on finding a way to permanently store objects and to retrieve them from secondary storage (disk).
- *Rule 10. The system must be able to manage very large databases*. Typical OO systems limit the object space to the amount of primary memory available. For example, Smalltalk cannot handle objects larger than 64K. Therefore, a critical OODBMS feature is the ability to optimize the management of secondary storage devices by using buffers, indexes, data clustering, and access path selection techniques.
- *Rule 11. The system must support concurrent users*. Conventional DBMSs are especially capable in this area. The OODBMS must support the same level of concurrency as conventional systems.
- *Rule 12. The system must be able to recover from hardware and software failures*. The OODBMS must offer the same level of protection from hardware and software failures that the traditional DBMS provides; that is, the OODBMS must provide support for automated backup and recovery tools.
- *Rule 13. Data query must be simple*. Efficient querying is one of the most important features of any DBMS. Relational DBMSs have provided a standard database query method through SQL, and the OODBMS must provide an object query language (OQL) with similar capability.

Optional OODBMS features include:

- *Support for multiple inheritance*. Multiple inheritance introduces greater complexity by requiring the system to manage potentially conflicting properties between classes and subclasses.
- *Support for distributed OODBMSs*. The trend toward systems application integration constitutes a powerful argument in favor of distributed databases. If the OODBMS is to be integrated seamlessly with other systems through networks, the database must support some degree of distribution.
- *Support for versioning*. Versioning is a new characteristic of the OODBMS that is especially useful in applications such as CAD and CAM. Versioning allows you to maintain a history that tracks all object transformations. Therefore, you can browse through all of the different object states, in effect letting you walk back and forth in time.

G-6b Oracle Object Examples

Oracle databases (since version 8) support object-oriented extensions. The extensions allow users to create object types, using DDL commands. Those object types are the equivalent of classes (see Section G-3f) or abstract data types (ADT) in the object model (see Section G-3j). Oracle supports various object types:

- *Column type*. This object type provides data type extensibility by allowing the user to define his/her own data types. A column object type is the equivalent of an abstract data type (ADT). An ADT can be used when defining a column data type within a relational table. An ADT can also have methods associated with it; those methods are implemented using PL/SQL or C++ or Java.
- *Row type*. A row type object is used to define an object table object. An **object table** is the equivalent of a relational table composed of many rows where each row is an object of the same type. Each row object has a unique system-generated object ID (OID), or object identifier. (See Section G-3b.)
- *Collection objects*. Oracle also provides support for two types of collection objects. (Note the discussion that accompanies Figure G.29 in Section G-4c.)
  - *Variable length arrays* (*VARRAY*). Enables the user to create an object type as an array of objects of a given type.
  - *Nested tables*. Allows the creation of a relational table in which one of the attributes is a table. Specifically, a relational table contains an attribute with an object table type.

To illustrate creating and using various object types, Oracle 11g will be used in the following discussion.

Column Type

A column type is basically a new data type (abstract data type) you can use when you define an attribute in a table. By creating a column type, you are defining a new class with shared attributes and methods. To illustrate the use of column types, let's create two column types named T_ADDRESS and T_JOB. The T_ADDRESS column type will contain the street, city, state, and zip code attributes. The T_JOB column type will be used to store data about a job (company name, start date, end date, and monthly salary). The T_JOB column type will have two methods: *monthsonjob*, which returns the number of months spent in a given job, and *totalearned*, which returns the result of the multiplication of the number of months employed and the monthly salary. Figure G.36 shows the use of the CREATE TYPE command to create the two column types.

In Figure G.36, note that the T_JOB column data type creation command includes references to the methods that are to be created. By using the MEMBER FUNCTION clause, you define the name of the method to be created, any optional parameters that may be required (shown in parentheses), and the type of value (such as number or character) to be returned. To actually create the methods, you use the CREATE TYPE BODY command, as shown in Figure G.37.

As you examine Figure G.37, note that the method definition uses standard PL/SQL commands. You might also define methods using other languages, such as C++ or Java. Each method definition starts with the MEMBER FUNCTION keywords. The actual method code is contained within the BEGIN and END clauses.

object table
The equivalent of a relational table composed of many rows, where each row is an object of the same type. Each row object has a unique system generated object ID (OID) or object identifier.

FIGURE G.36 CREATION OF THE T_ADDRESS AND T_ JOB COLUMN DATA TYPES

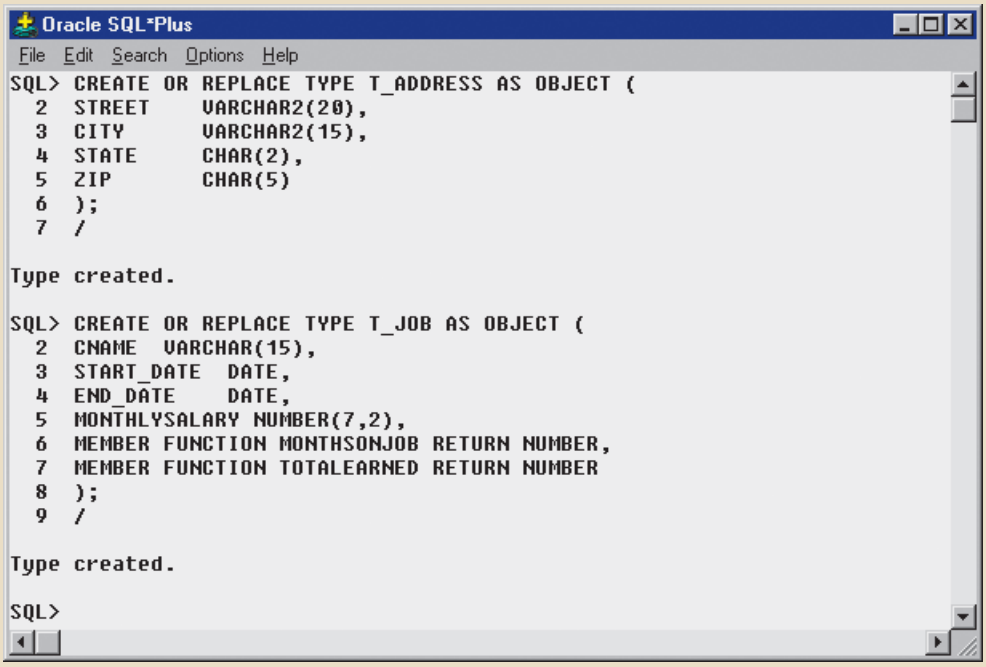


FIGURE G.37 CREATION OF THE T_ JOB METHODS

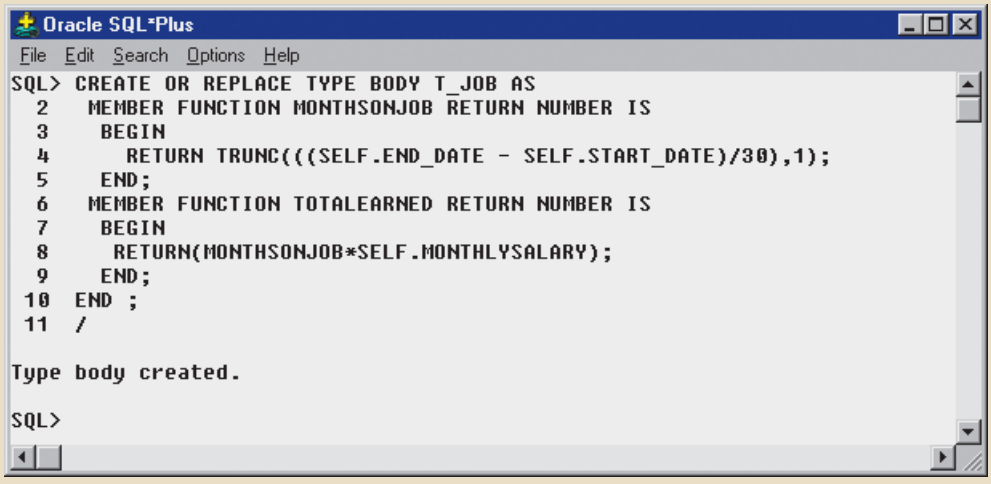
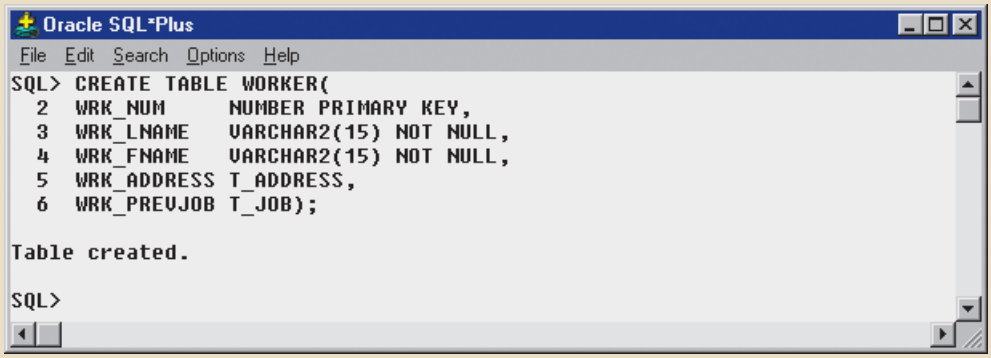


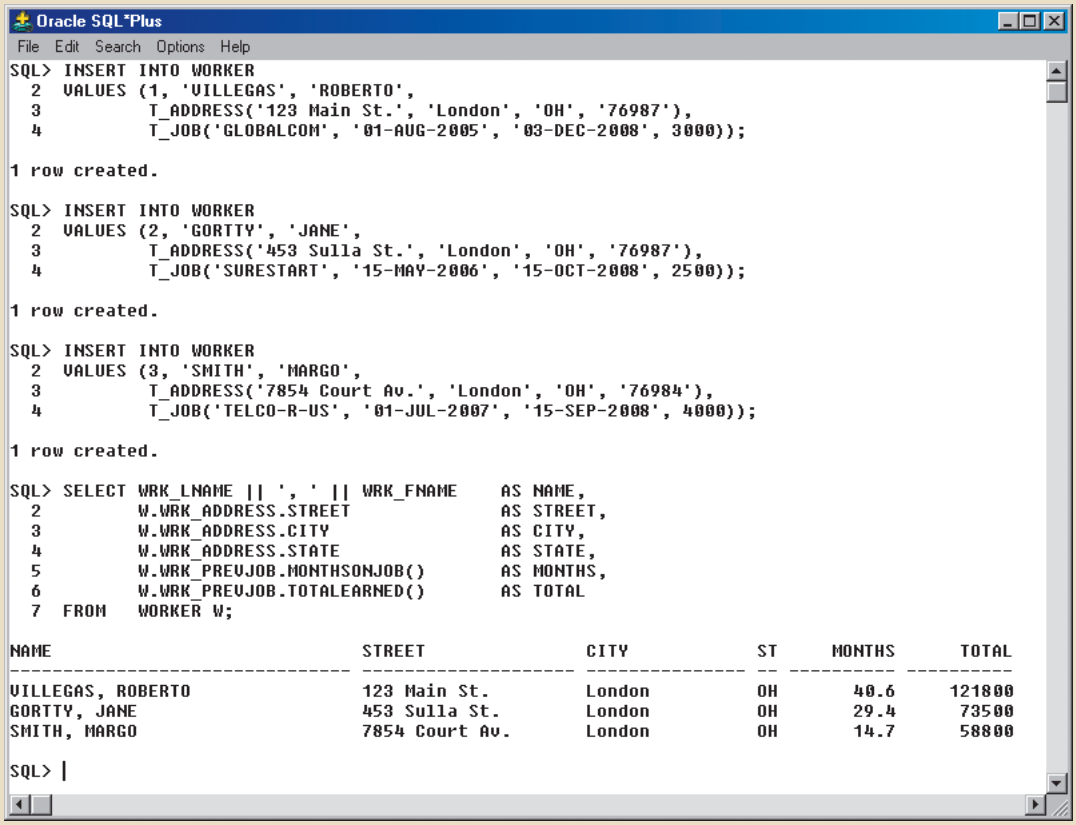
Figure G.38 shows the creation of a WORKER table that uses the T_ADDRESS and T_JOB column types defined earlier.

FIGURE G.38 CREATION OF THE WORKER TABLE, USING T_ADDRESS AND T_ JOB COLUMN TYPES



Once you have created the table, you use standard SQL commands to insert data. However, to enter data in a column type attribute, you must use the column type name, as indicated in Figure G.39, with the INSERT statements.

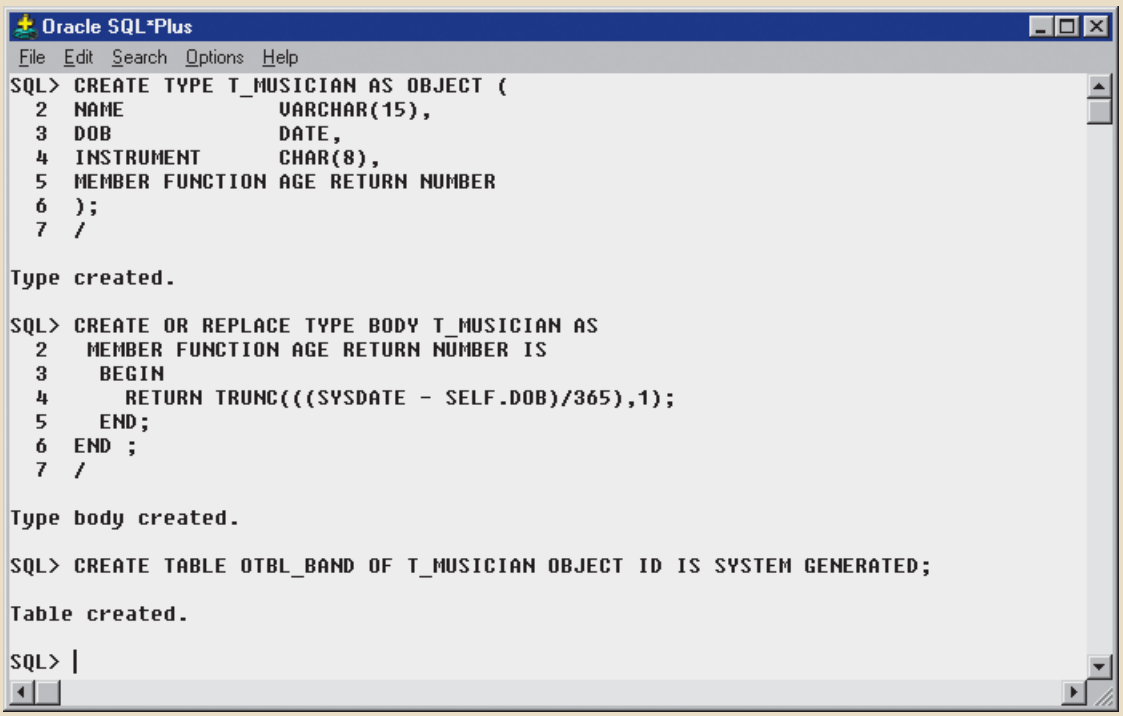
FIGURE G.39 WORKING WITH COLUMN TYPES IN THE WORKER TABLE



To retrieve data from a column type attribute using a SELECT statement, you must first declare an alias for the table. In this case, the alias W has been used. Next, you can refer to a column type attribute or method using a dot-separated notation such as W.WRK_ADDRESS.STREET or W.WRK_PREVJOB.TOTALEARNED(), as shown in Figure G.39.

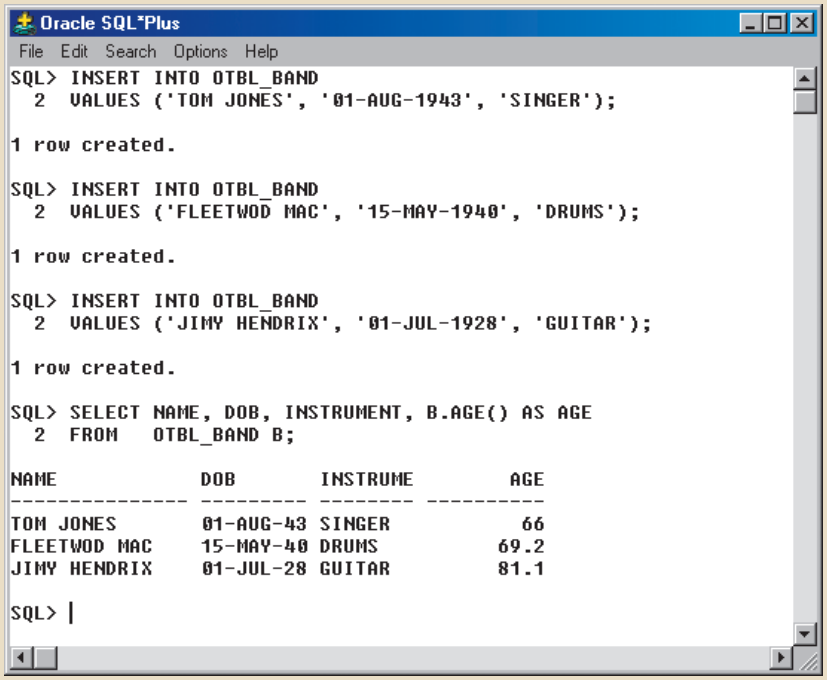
Row Type A row type enables you to create a table in which each row is an object instance. That table is called an object table to differentiate it from a relational table. To demonstrate the use of row types, let's create the OTBL_BAND object table in which each row is a Musician object. To accomplish that task, let's first create a T_MUSICIAN column type. (Remember that a column type is an object.) The T_MUSICIAN column type will have an AGE method that uses the system date and the musician's date of birth to return the age of each musician. To create the object table, you use the CREATE TABLE OF command, as shown in Figure G.40.

FIGURE G.40 CREATION OF THE OTBL_BAND OBJECT TABLE



To insert data to the newly created object table, you use the INSERT command, as shown in Figure G.41. Note that you do not have to identify the column type, as was required with the WORKER table. The difference is that the WORKER table is a relational table containing attributes with abstract data types. In such cases, you need to specify the abstract data (column type). In the case of the OTBL_BAND table, you are adding rows to an *object table*. However, you still must use an alias to invoke a method. (See Figure G.41.)

FIGURE G.41 WORKING WITH THE OTBL_BAND OBJECT TABLE



VARRAY Collection Type The variable length array creates a new object type that represents a collection of objects of a similar type (objects or base data types). For example, an employee may have multiple dependents. In that case, you can store all of the dependents in an array for each of the employees. Figure G.42 shows the commands required to create the T_DEPENDLIST variable array object type and the EMP3 table containing the E_DEPENDENTS attribute, which uses the T_DEPENDLIST data type. Note that the variable array has been defined to hold a maximum of 10 dependent names.

Nested Table Collection Type When you have related data that are more extensive than you would expect to find in an array, you can use a nested table. A nested table is created when an attribute within a relational table definition (CREATE TABLE) is assigned a table data type. For example, Figure G.43 shows the creation of the EMP4 table containing the E_DEPENDTS attribute, which uses the T_DEPTAB data type. In turn, the T_DEPTAB data type is defined as a table type. Conceptually speaking, the attribute is, in effect, a table.

FIGURE G.42 CREATING AND WORKING WITH THE VARRAY OBJECT TYPE

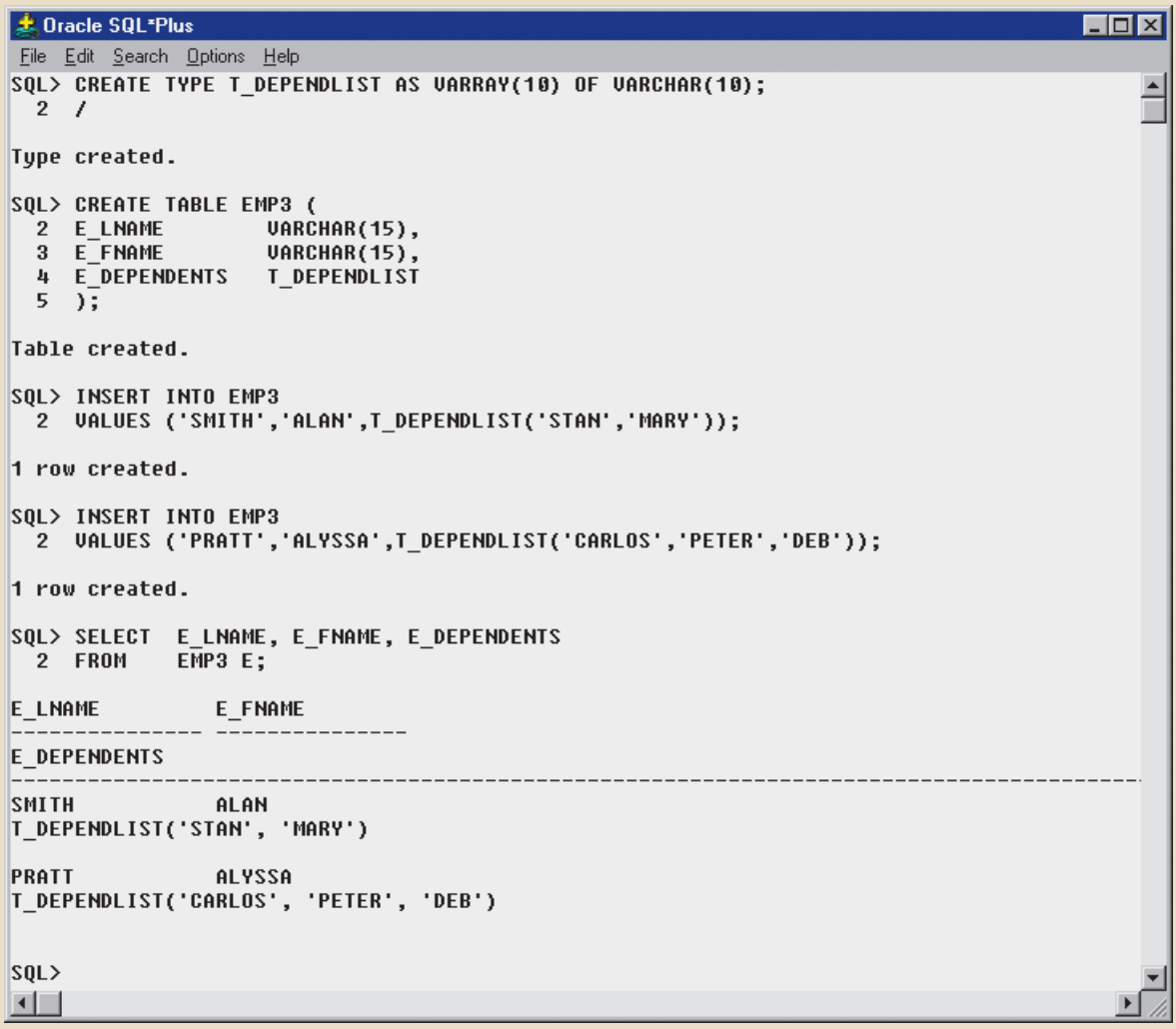
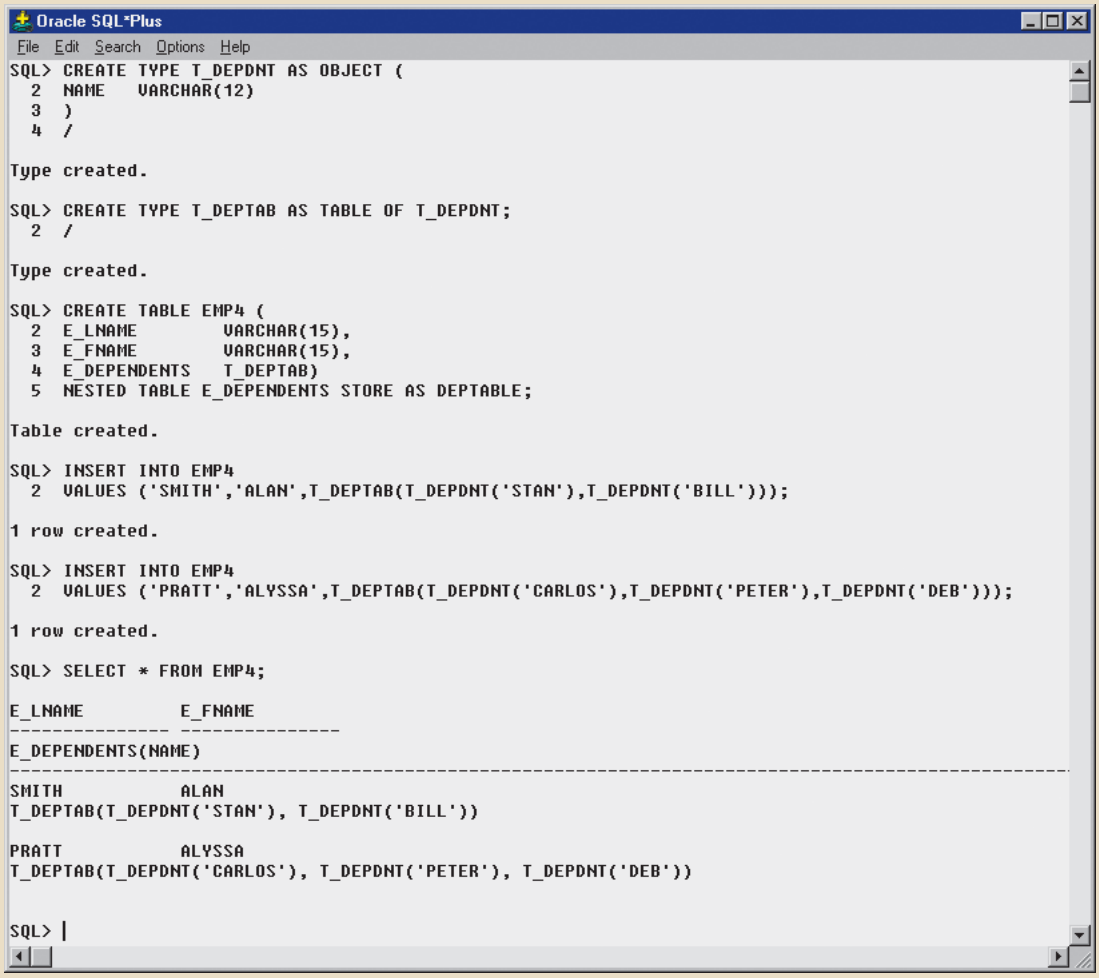


FIGURE G.43 CREATING AND WORKING WITH A NESTED TABLE OBJECT TYPE



G-7 How Object Orientation Affects Database Design

A conventional relational database design process involves the application of ER modeling and normalization techniques to develop and implement a design. During a design process, emphasis is placed on modeling real-world objects through simple tabular relations, usually presented in 3NF. Unfortunately, as you have already seen, sometimes the relational and ER models cannot adequately represent some objects. Consequently, the ER model makes use of constructs such as bridge (composite) entities that widen the semantic gap between the real-world objects and their corresponding representations.

You may have noticed the database design process generally focusing on identification of the data elements, rather than including data operations as part of the process. In fact, the definition of data constraints and data transformations is usually considered late in the database design process. Those definitions are implemented by external application program code. In short, operations are not a part of the database model.

Why does the conventional model tolerate and even require the existence of the data/procedures dichotomy? After all, the idea of object-oriented design had been contemplated even in the classical database environment. The reason is simple: until recently, database designers simply had no access to tools that bonded data and procedures.

The object-oriented database design approach answers the problem of a split between data and procedures by providing both data identification and the procedures or manipulations to be performed on the data. Object-oriented database design forces you to think of data and procedures as a self-contained entity. Specifically, the OO design requires the database description to include the objects and their data representation, constraints, and operations. That design can produce a more complete and meaningful description of the database than was possible in the conventional database design.

OO design is iterative and incremental in nature. The database designer identifies each real-world object and defines its internal data representation, semantic constraints, and operations. Next, the designer groups similar objects in classes and implements the constraints and operations through methods. At this point, the designer faces two major challenges:

1. Build the class hierarchy or the class lattice (if multiple inheritance is allowed), using base data types and existing classes. This task will define the superclass-subclass relationships.

2. Define the interclass relationships (attribute-class links), using both base data types and ADTs.

The importance of those tasks can hardly be overestimated because the better the use of the class hierarchy and the treatment of the interclass relationships, the more flexible and closer to the real world the final model will be.

Code reusability does not come easy. One of the hardest tasks in OODB design is creation of the class hierarchy, using existing classes to construct new ones. Future DBAs will have to develop specialized skills to perform that task properly and to incorporate code that represents data behavior. Thus, DBAs are likely to become surrogate database programmers who must define data-intrinsic behavior. The role of DBAs is likely to change when they take over some of the programming burden of defining and implementing operations that affect the data.

Both DBAs and designers face additional problems. In contrast to the relational or ER design processes, there are few computerized OODB design tools, and if the design is to be implemented in any of the conventional DBMSs, it must be translated carefully. The reason is that conventional databases do not support abstract data types, non-normalized data, inheritance, encapsulation, or other OO features.

As is true in any of the object-oriented technologies, the lack of standards also affects OO database design. There is neither a widely accepted standard methodology to guide the design process, nor a set of rules (like the normalization rules in the relational model) to evaluate the design. This situation is improving. The Object Management Group (OMG), mentioned earlier, produces vendor-independent standards and specifications for object-based systems and components. The OMG created the Unified Modeling Language (UML), a graphical language for the modeling, design, and visualization of object-oriented systems. UML is used to model not only the database component of a system but also its processes, modules, and network components and the interaction among them. OMG also created object standards that define the Object Management Architecture (OMA), which allows the interoperation of objects across diverse systems and platforms. The OMA standard includes the Common Object Request Broker Architecture (CORBA) and the Common Object Services Specifications (COSS). That framework is used by OODBMS vendors and developers to implement systems that are highly interoperable with other OODBMSs, as well as with RDBMSs and older DBMS systems.

Note

Appendix H provides an introduction to the Unified Modeling Language (UML).

Some vendors are already offering products that comply with the OMG's CORBA and COSS specifications, such as IBM's System Object Model (SOM) and HP's Object Request Broker (ORB). Other object architectures have emerged as alternatives to the concerted standards efforts, especially Microsoft's object linking and embedding (OLE) and Component Object Model (COM). Although the OLE/COM specification is not a standards-based effort, the sheer established market volume is making it the de facto object standard for the Microsoft Windows environment.

G-8 OODBMS: Advantages and Disadvantages

Compared to the RDBMS market share, OODBMSs have a long way to go before they can claim double-digit market percentage. In fact, at this point, the OODBMS occupies a strong niche market, as the Apple Mac does in the microcomputing arena. As with the Mac's impact on microcomputing, the OODBMS has been the vehicle for technological innovation, but it has not been the beneficiary of market share growth based on its technological innovations. Yet in spite of its lack of market share, the OODBMS is worth examining, especially because its OO features drive the changes in database technology that define today's object/relational DBMS.

One reason for the OODBMS's lack of market acceptance is that the RDBMS has incorporated many OO features while retaining its conceptual simplicity, thus diminishing the OODBMS's allure. Nevertheless, as long as the RDBMS does not incorporate C. J. Date's recommended domain implementation, the OODBMS offers benefits that are worth examining. Most of those benefits are expressed in terms of the complex object management capabilities you have explored in some detail. To obtain those benefits, the OODBMS depends on the use of an OOPL. That is why you examine some of the OODBMS's benefits with reference to programming issues.

G-8a Advantages

- OODBMSs allow the inclusion of more semantic information in the database, thus providing a more natural and realistic representation of real-world objects.
- OODBMSs provide an edge in supporting complex objects, which makes them especially desirable in specialized application areas. Conventional databases simply lack the ability to provide efficient applications in CAD, CAM, medical imaging, spatial imaging, and specialized multimedia environments.
- OODBMSs permit the extensibility of base data types, thereby increasing both the database functionality and its modeling capabilities.
- *If the platform allows efficient caching*, when managing complex objects, OODBMSs provide dramatic performance improvements compared to relational database systems.
- Versioning is a useful feature for specialized applications such as CAD, CAM, medical imaging, spatial imaging, engineering, text management, and desktop publishing.
- The reusability of classes allows for faster development and easier maintenance of the database and its applications.

- Faster application development time is obtained through inheritance and reusability. This benefit is obtained only after mastering the use of OO development features such as:
  - Proper use of the class hierarchy; for example, how to use existing classes to create new classes.
  - OO design methodology.
- The OODBMS provides a possible solution to the problem of integrating existing and future DBMSs into a single environment. This solution is based on the OODBMS's strong data-abstraction capabilities and its promise of portability.

G-8b Disadvantages

- OODBMSs face strong and effective opposition from the firmly established RDBMSs, especially when those RDBMSs—such as IBM's DB2 Universal Database and Oracle—incorporate many OO features that would otherwise have given the OODBMS the clear competitive edge in a complex data environment. Therefore, the OODBMS's design and implementation complexities become more difficult to justify.
- The OODBMS is based on the object model, which lacks the solid theoretical foundation of the relational model on which the RDBMS is built.
- In some sense, OODBMSs are considered a throwback to the traditional pointer systems used by hierarchical and network models. This criticism is not quite true when it associates the pointer system with the navigational data manipulation style and fixed access paths that led to the relational system's dominance. Nevertheless, the *complexity* of the OODBMS pointer systems cannot be denied.
- OODBMSs do not provide a standard ad hoc query language, as relational systems do. At this point, development of the object query language (OQL) is far from complete. Some OODBMS implementations are beginning to provide extensions to the relational SQL to make the integration of the OODBMS and RDBMS possible.
- The relational DBMS provides a comprehensive solution to business database design and management needs, supplying both a data model and a set of fairly straightforward normalization rules for designing and evaluating relational databases. OODBMSs do not yet provide a similar set of tools.
- The initial learning curve for the OODBMS is steep. If you consider the direct training costs and the time it takes to fully master the uses and advantages of object orientation, you will appreciate why OODBMSs seldom are rated as the first option when solutions are sought for noncomplex business-oriented problems.
- The OODBMS's low market presence, combined with its steep learning curve, means that few people are qualified to make use of the presumed power of OO technology. Most of the technology is currently focused on engineering application areas of software development. Therefore, only companies with the right mix of resources (money, time, and qualified personnel) can afford to invest in OO technology.
- The lack of compatibility between different OODBMSs makes switching from one piece of software to another very difficult. With RDBMSs, different products are very similar, and switching from one to another is relatively easy.

A few years ago, the authors speculated that future systems would manage objects with embedded data and methods, rather than with records, tuples, or files. The authors also suggested that although the portability details were not clear yet, they would have a major and lasting impact on how databases would be designed and used. Given the benefit of hindsight, the authors now know that the OODBMS's

reach has been limited by the object-relational DBMS's successful integration of many OO concepts. In any case, the OODBMS has had a major impact on how databases are viewed and managed, and the battle of the relational and object titans is far from over. Finally, because the object concepts are likely to remain the focus for future DBMS developments, they continue to be worth understanding.

G-9 How OO Concepts Have Influenced the Relational Model

Most relational databases are designed to serve general business applications that require ad hoc queries and easy interaction. The data types encountered in those applications are well defined and are easily represented in common tabular formats with equally common short and well-defined transactions. However, RDBMSs are not as well suited as OODBMSs to the complex requirements of some applications, and the RDBMS is beginning to reach its limits in a business data environment that is changing with the advent of mixed-media data storage and retrieval.

The fast-changing data environment has forced relational model advocates to respond to the OO challenge by extending the relational model's conceptual reach. The result of their efforts is usually referred to as the extended relational model (ERM) or, more appropriately, the object/relational model (O/RM). Although this O/RM effort is still a work in progress, its basic features provide support for:

- Extensibility of new user-defined (abstract) data types
- Complex objects
- Inheritance
- Procedure calls (rules or triggers)
- System-generated identifiers (OID surrogates)

That is not an exhaustive list of the extensions added to the relational model, nor do all extended relational models incorporate all of the listed additions. However, the list contains the most crucial and desirable extended relational features.

Note

It's worth noting again that C. J. Date's "Third Manifesto" is based on Date's observation that the relational model already contains the desired capabilities through its support of domains. Therefore, the implementation of that domain support will yield the benefits now claimed for the OO "extensions" of the relational model. However, the relational domain implementations have not (yet?) been developed commercially, while the OO "extensions" to the relational database model are a commercial fact of life.

The enhancements to the relational model are based on the following concepts:

- Semantic and object-oriented concepts are necessary to support the new generation of applications—especially if those applications will be deployed through the Internet.
- The concepts can and must be added to the relational model.
- The benefits of the relational model must be preserved to protect the investment in relational technology and to provide downward compatibility.

Most current extended relational DBMSs conform to the notions expressed in C. J. Date's "Third Manifesto." (See preceding note.) They also provide the following useful features:

- Oracle Corporation and IBM have developed suites of products marketed as Universal Database Servers. Although the Universal Database Server is not a pure object-oriented DBMS—it lacks the object storage component—this product supports complex data types such as multimedia data and spatial data, and it's Internet-ready. The Internet feature allows users to query the database using the World Wide Web (WWW). Oracle also includes support for object-oriented extensions and storage. IBM's DB2 Universal Database Server has similar capabilities.
- IBM's DB2 Universal Database system is a proven database that is used by many Fortune 1000 corporations. IBM's system supports digitized data (video and audio) as well as user-defined data types and procedures. The Universal Database is also being positioned as a key player in the Internet arena with its support for web access and Java programming interfaces.

G-10 The Next Generation of Database Management Systems

The adaptation of OO concepts in several computer-related areas has changed both systems design and system behavior. The next generation of DBMSs is likely to incorporate features borrowed from object-oriented database systems, artificial intelligence systems, expert systems, distributed databases, and the Internet.

OODBMSs represent only one step toward the next generation of database systems. The use of OO concepts will enable future DBMSs to handle more complex problems with normalized and non-normalized data. The extensibility of database systems is one of the many major object-oriented contributions that enable databases to support new data types such as sets, lists, arrays, video, bitmap pictures, voice, and maps. The SQL3 standard provides such extensibility by supporting user-defined data types in addition to its predefined data types (numeric, integer, string, and so on). For example, in SQL3, a DBA can create a new abstract data type that represents a collection of objects, then use that data type in a table definition. That procedure enables a database column to contain a *collection* of values instead of a single value.

Recent market history indicates that the OODBMS will probably continue to occupy a niche within the database market. That niche will be characterized by applications that require very large amounts of data with several complex relations and with specialized data types. For example, the OODBMS seems likely to maintain its standing in CAD, CAM, computer-integrated manufacturing, specialized multimedia applications, medical applications, architectural applications, mapping applications, simulation modeling, and scientific applications.

However, current market conditions seem to dictate that the object/relational databases will become dominant in most complex business applications. That conclusion is based on the need to maintain compatibility with existing systems, the universal acceptance of the relational model as a standard, and the sheer weight of the relational database's considerable market share.

Key Terms

abstract data type (ADT), G-14
associative object, G-15
base data types, G-4
class, G-8
class hierarchy, G-9
class instance, G-8
class lattice, G-9
collection object, G-5
complex object, G-14
composite object, G-15
compound object, G-15
conventional data types, G-4
domain, G-4
early binding, G-26
encapsulation, G-7
extensible, G-15
hybrid object, G-15
inheritance, G-10
instance variables, G-4
interobject relationship, G-20
interrogate, G-7
intersection class, G-23
late binding, G-26
message, G-7
method, G-6
multiple inheritance, G-11
object, G-3
object ID (OID), G-4
object instance, G-8
object orientation, G-2
object-oriented data model (OODM), G-15
object-oriented database management system (OODBMS), G-31
object-oriented programming (OOP), G-2
object-oriented programming languages (OOPLs), G-2
object query language (OQL), G-30
object space (object schema), G-17
object state, G-6
object table, G-35
polymorphism, G-13
protocol, G-9
referential object sharing, G-18
simple object, G-15
single inheritance, G-11
subclasses, G-9
superclass, G-9
versioning, G-28

Review Questions

1. Discuss the evolution of object-oriented concepts. Explain how those concepts have affected computer-related activities.
2. How would you define object orientation? What are some of its benefits? How are OO programming languages related to object orientation?
3. Define and describe the following:
   a Object
   b Attributes
   c Object state
   d Object ID (OID)
4. Define and contrast the concepts of method and message. What OO concept provides the differentiation between a method and a message? Give examples.
5. Explain how encapsulation provides a contrast to traditional programming constructs such as record definition. What benefits are obtained through encapsulation? Give an example.
6. Using an example, illustrate the concepts of class and class instances.

7. What is a class protocol, and how is it related to the concepts of methods and classes? Draw a diagram to show the relationships among these OO concepts: object, class, instance variables, methods, object state, object ID, behavior, protocol, and messages.
8. Define the concepts of class hierarchy, superclasses, and subclasses. Explain the concept of inheritance and the different types of inheritance. Use examples in your explanations.
9. Define and explain the concepts of method overriding and polymorphism. Use examples in your explanations.
10. Explain the concept of abstract data types. How do they differ from traditional or base data types? What is the relationship between a type and a class in OO systems?
11. What are the five minimum attributes of an OO data model?
12. Describe the difference between early and late binding. How does each of these affect the object-oriented data model? Give examples.
13. What is an object space? Using a graphic representation of objects, depict the relationship(s) that exist between a student taking several courses and a course taken by several students. What type of object is needed to depict that relationship?
14. Compare and contrast the OODM with the ER and relational models. How is a weak entity represented in the OODM? Give examples.
15. Name and describe the 13 mandatory features of an OODBMS.
16. What are the advantages and disadvantages of an OODBMS?
17. Explain how OO concepts affect database design. How does the OO environment affect the DBA's role?
18. What are the essential differences between the relational database model and the object database model?
19. Using a simple invoicing system as your point of departure, explain how its representation in an entity relationship model (ERM) differs from its representation in an object data model (ODM). (*Hint:* See Figure G.34.)
20. What are the essential differences between an RDBMS and an OODBMS?
21. Discuss the object/relational model's characteristics.

Problems

1. Convert the following relational database tables to the equivalent OO conceptual representation. Explain each of your conversions with the help of a diagram. (*Note:* The RRE Trucking Company database includes the three tables shown in Figure PG.1.)
2. Using the tables in Figure PG.1 as a source of information:
   a. Define the implied business rules for the relationships.
   b. Using your best judgment, choose the type of participation of the entities in the relationship (mandatory or optional). Explain your choices.
   c. Develop the conceptual object schema.

FIGURE PG.1 THE RRE TRUCKING COMPANY DATABASE

Table name: TRUCK **Database name: RRE_Trucking**

VEHICLE_NUM	BASE_CODE	TYPE_CODE	VEHICLE_MILES	VEHICLE_BUY_DATE	VEHICLE_VIN
5001	101	1	162123.50	08-Nov-03	AA-322-12212-W11
5002	102	1	276984.30	23-Mar-01	AC-342-22134-Q23
5003	101	2	212346.60	27-Dec-02	AC-445-78656-Z99
5004	101	1	99894.30	21-Feb-03	WQ-112-23144-T34
5005	103	2	245673.10	15-Apr-02	FR-998-32245-W12
5006	101	6	293245.70	30-Aug-01	AD-456-00845-R45
5007	102	3	132012.30	01-Dec-02	AA-341-96573-Z84
5008	102	3	144213.60	21-Sep-02	DR-559-22189-D33
5009	103	2	80932.90	16-Jan-04	DE-887-98456-E94
5010	104	1	34213.40	12-Apr-05	FD-221-21100-F32
5011	101	1	42326.80	09-Nov-05	DT-324-04056-H22
5012	104	6	152339.40	23-May-03	GF-657-22134-K48
5013	101	3	298145.80	11-Apr-01	HR-344-54560-J92
5014	104	2	8122.20	09-Sep-03	RW-289-38956-H87
5015	103	1	154667.90	26-Mar-03	HH-231-55498-K37
5016	105	1	1200.60	18-Mar-06	FR-332-23459-G55
5017	105	1	3345.50	23-Feb-06	DD-545-78896-X39

Table name: BASE

BASE_CODE	BASE_CITY	BASE_STATE	BASE_AREA_CODE	BASE_PHONE	BASE_MANAGER
101	Nashville	TN	615	123-4567	Andrea D. Gallagher
102	Lexington	KY	606	234-5678	George H. Delarosa
103	Kansas City	MO	573	345-6789	Maria J. Talindo
104	Athens	GA	901	456-7890	Peter F. McAfee
105	Gainesville	FL	904	678-6543	Lee A. Chau

Table name: TYPE

TYPE_CODE	TYPE_DESCRIPTION
1	Single box, double-axle
2	Single box, single-axle
3	Tandem trailer, single-axle
4	Tandem trailer, double-axle
5	Utility
6	Dump truck, single-axle
7	Dump truck, double-axle

3. Using the data presented in Problem 1, develop an object space diagram representing the object's state for the instances of Truck listed below. Label each component clearly with proper OIDs and attribute names.
   a. The instance of the class Truck with TRUCK_NUM = 5001.
   b. The instances of the class Truck with TRUCK_NUM = 5003 and 5004.
4. Given the information in Problem 1, define a superclass Vehicle for the Truck class. Redraw the object space you developed in Problem 3, taking into consideration the new superclass that you just added to the class hierarchy.
5. Assume the following business rules:
   - A course contains many sections, but each section has only one course.
   - A section is taught by one professor, but each professor may teach one or more different sections of one or more courses.
   - A section may contain many students, and each student is enrolled in many sections, but each section belongs to a different course. (Students may take many courses, but they cannot take many sections of the same course.)

- Each section is taught in one room, but each room may be used to teach several different sections of one or more courses.
- A professor advises many students, but a student has only one advisor.

Based on those business rules:

a. Identify and describe the main classes of objects.

b. Modify your description in (a) to include the use of abstract data types such as Name, DOB, and Address.

c. Use object representation diagrams to show the relationships between:
   - Course and Section.
   - Section and Professor.
   - Professor and Student.

d. Use object representation diagrams to show the relationships between:
   - Section and Students.
   - Room and Section.

What type of object is necessary to represent those relationships?

e. Using an OO generalization, define a superclass Person for Student and Professor. Describe this new superclass and its relationship to its subclasses.

6. Convert the following relational database tables to the equivalent OO conceptual representation. Explain each of your conversions with the help of a diagram. (*Note:* The R&C Stores database includes the three tables shown in Figure PG.6.)

FIGURE PG.6 THE R&C STORES DATABASE

Table name: EMPLOYEE **Database name: RC_Stores**

EMP_CODE	EMP_TITLE	EMP_LNAME	EMP_FNAME	EMP_INITIAL	EMP_DOB	EMP_SERVICE	STORE_CODE	EMP_AREACODE	EMP_PHONE	EMP_ADDRESS	EMP_CITY	EMP_STATE	EMP_ZIPCODE
101	Mr.	Williamson	John	W	21-Jun-1962	Yes	1	545	870-4567	2219 Orchard Road	Flagstaff	AZ	32119
102	Ms.	Ratula	Nancy		12-Mar-1967	Yes	2	545	873-5467	345 Lake Circle	Flagstaff	AZ	32117
103	Ms.	Greenboro	Lottie	R	02-Nov-1959	No	4	615	366-8567	Rt. 23, Box 123	Eagleville	TN	30123
104	Mrs.	Rumpersfro	Jennie	S	01-Jul-1969	No	5	901	224-8332	3425 NW 55th Terrace	Gainesville	FL	38155
105	Mr.	Smith	Robert	L	23-Dec-1957	No	3	615	123-7009	1234 Airport Road	Smyrna	TN	30118
106	Mr.	Renselaer	Cary	A	25-Jan-1964	Yes	1	545	870-0705	1108 Orchard Road	Flagstaff	AZ	32119
107	Mr.	Ogallo	Roberto	S	30-Aug-1960	No	3	615	876-1004	2345 Meadow View	Murfreesboro	TN	32130
108	Ms.	Johnsson	Elizabeth	I	11-Oct-1966	No	1	545	224-7531	1016 Orchard Road	Flagstaff	AZ	32119
109	Mr.	Eindsmar	Jack	W	19-May-1953	Yes	2	545	224-9246	9829 East Main Str.	Flagstaff	AZ	32120
110	Ms.	Jones	Rose	R	05-Apr-1964	No	4	703	123-9358	6543 Snowview Circle	Aspen	CO	41234
111	Mr.	Broderick	Tom		21-Nov-1970	No	3	615	123-2214	4256 Greenbriar Road	Murfreesboro	TN	32130
112	Mr.	Washington	Alan	Y	08-Oct-1972	No	2	545	875-4447	2896 Tall Pine Road	Flagstaff	AZ	32119
113	Mr.	Smith	Robert	N	25-Sep-1962	No	3	615	224-8999	4345 Oak Terrace	Murfreesboro	TN	32129
114	Mrs.	Smith	Sherry	H	24-Jun-1964	No	4	703	224-8999	2693 Edelweiss Lane	Aspen	CO	41234
115	Mr.	Olenko	Howard	U	24-Jun-1962	No	5	901	123-8878	2314 NW 23rd Place	Gainesville	FL	38152
116	Mr.	Archialo	Barry	V	04-Oct-1958	No	5	901	876-3428	6541 Clear Lake Drive	Lake City	FL	38167
117	Ms.	Grimaldo	Jeanine	K	12-Dec-1968	Yes	4	703	123-7890	4356 Snowflake Road	Aspen	CO	41234
118	Mr.	Rosenberg	Andrew	D	23-Feb-1969	Yes	4	703	123-5360	5112 Avalanche View	Aspen	CO	41235
119	Mr.	Rosten	Peter	F	03-Nov-1966	No	4	703	224-7211	3256 Tall Timber Lane	Aspen	CO	41234
120	Mr.	Zack	Robert	S	05-Apr-1968	Yes	1	545	873-2218	3567 Deep Water Drive	Flagstaff	AZ	32117
121	Ms.	Mcbee	Jennifer	A	10-Jan-1972	Yes	1	545	875-7768	3256 Treebranch Lane	Flagstaff	AZ	32120
122	Mr.	Ryan	Herman	G	06-Feb-1967	Yes	3	615	567-8903	4436 Hadley Ct.	Smyrna	TN	37123

Table name: STORE

STORE_CODE	STORE_NAME	STORE_YTD_SALES	REGION_CODE	EMP_CODE	STORE_ADDRESS	STORE_CITY	STORE_STATE	STORE_ZIP
1	Access Junction	1403456.00	2	108	1234 Cactus Circle	Flagstaff	AZ	32117
2	Database Corner	1821987.00	2	112	2345 Longview Pike	Flagstaff	AZ	32121
3	Tuple Charge	1386783.00	1	107	9876 Brandywood Road	Murfreesboro	TN	30130
4	Attribute Alley	1344569.00	2	103	7654 Mountainview Drive	Aspen	CO	40123
5	Primary Key Point	3330099.00	1	115	4567 Palmetto Road	Gainesville	FL	38762

Table name: REGION

REGION_CODE	REGION_LOCATION
1	East
2	West

7. Convert the following relational database tables to the equivalent OO conceptual representation. Explain each of your conversions with the help of a diagram. (*Note:* The Avion Sales database includes the tables shown in Figure PG.7.)

FIGURE PG.7 THE AVION SALES DATABASE

Table name: PRODUCT **Database name: Avion_Sales**

PROD_CODE	PROD_TYPE	PROD_SUBTYPE	PROD_MODE	PROD_MANUFACT	PROD_DESCRIPTION	PROD_COST	PROD_PRICE	PROD_QOH	PROD_MIN_QOH	PROD_LAST_ORDER
ADF-841	ADF	Standard	Panel	Narco	Digital display ADF, keep-alive memory/auto dim, combined l	$1,699.00	$2,595.00	11	5	11-Mar-2014
AIRMAP	GPS	Moving map	hand-held	Lowrance	High-density display with accu cartridge, auto zoom, backlit	$619.00	$895.00	34	15	07-Sep-2013
APOLLO2001	GPS	Standard	Panel	II Morrow	LED display, airspace alerts with user-programmable penetr	$1,529.00	$2,245.00	17	10	12-Dec-2013
APOLLO360	GPS	Moving map	Panel	II Morrow	Round GPS with moving map, standard 3-1/8" instrument ho	$1,229.00	$1,995.00	32	20	19-Jan-2014
APOLLO-920	GPS	Moving map	Hand-held	II Morrow	Auto zoom, 20 reversible 30-leg flight plans, nearest waypc	$816.00	$1,225.00	26	12	16-Feb-2014
AT-150	Transponder	Standard	Panel	Narco	Mode C TSO'd, 250-watt transmitter, compatible with all leac	$649.00	$950.00	16	10	24-Jun-2013
CP-136M	Audio panel	Standard	Panel	Narco	Pushbutton, LED, tuning function, internal marker beacon rec	$605.00	$980.00	15	5	26-May-2013
EC-10X	GPS	Moving map	Lap	Magellan	6x4.5-in backlit LCD, GPS/electronic chart	$1,545.00	$1,999.00	12	5	08-Jun-2013
FLTPRO	GPS	Standard	Hand-held	Trimble	Palm-sized GPS, 4-line interface, LCD, updatable through RS	$499.00	$725.00	12	5	20-May-2013
GPS-150	GPS	Standard	Panel	Garmin	Internal rechargeable battery, up to 4 hours use in the event	$1,349.00	$1,995.00	29	15	20-Nov-2013
GPS-155	GPS	Standard	Panel	Garmin	Front-loading data card, interfaces with fuel mgt, EFIS, HSI.	$3,888.00	$5,995.00	14	8	14-Jan-2014
GPS-55AVD	GPS	Standard	Hand-held	Garmin	Alkaline battery pack, yoke mount adapter, power/data cable	$439.00	$625.00	31	12	11-Dec-2013
GPS-95XL	GPS	Moving map	Hand-held	Garmin	CDI and jeppesen database, cigarette lighter adapter, remov	$699.00	$1,075.00	27	12	10-Feb-2014
KLN-89	GPS	Moving map	Panel	Bendix/King	VFR Moving map, 4-line gas discharge display, up to 500 us	$2,289.00	$3,195.00	18	12	11-Dec-2013
KLN-89B	GPS	Moving map	Panel	Bendix/King	IFR-certifiable, 4-line gas display, certified to TSO C 129 A-1	$3,899.00	$5,895.00	14	8	11-Dec-2013
KLN-90B	GPS	Moving map	Panel	Bendix/King	Updatable via PC-compatible computer or with exchangeable	$6,311.00	$8,400.00	11	5	22-Jun-2013
KMA-24	Audio Panel	Standard	Panel	Bendix/King	Push button selection and control for three transceivers and	$399.00	$599.00	17	12	12-Feb-2014
KMA-24H	Audio Panel	Standard	Panel	Bendix/King	Adds intercom to KMA-24. Hot mike, voice-activated, push to	$459.00	$699.00	12	10	12-Nov-2013
KN-62A	DME	Standard	Panel	Bendix/King	Solid state, 200-channel, distance, groundspeed, time-to-sta	$1,344.00	$1,899.00	12	5	09-Jun-2013
KX-155	Nav/Com	Standard	Panel	Bendix/King	Electronic digital display, 760 channel, 10 watt, 200 channel	$1,119.00	$1,599.00	26	12	12-Nov-2013
KX-165	Nav/Com	Standard	Panel	Bendix/King	Electronic tuning, digital flip-flop, 760-channel, built-in 40-cha	$1,299.00	$1,695.00	16	10	12-Nov-2013
KY-196A/197A	Com	Standard	Panel	Bendix/King	Simultaneous display of 2 preselected comm. freq., button-p	$659.00	$999.00	12	5	09-Jun-2013
SKYNAV5000	GPS	Standard	Panel	Magellan	High-contast wide-angle display, front-loading data card, 20	$1,249.00	$1,799.00	18	10	21-Dec-2013
TMA-350D	Audio panel	Standard	Panel	Terra	4-place voice-activated intercom built-in, 3-position toggle sv	$499.00	$850.00	15	10	17-Jan-2014
TNL1000(DC)	GPS	Standard	Panel	Trimble	Backlit 2x20-character display, 250 waypoint total, interface	$1,250.00	$1,695.00	35	10	10-Jan-2014
TNL-2000	GPS	Standard	Panel	Garmin	TSO'd for IFR. Uses Jeppesen NavData card.	$3,999.00	$6,650.00	12	5	11-Mar-2014
TNL-2000A	GPS	Standard	Panel	Trimble	Backlit LCD, vertical vav, interface to moving maps, autopilot	$1,999.00	$2,695.00	27	10	11-Jan-2014
TRT-250D	Transponder	Standard	Panel	Terra	Solid state, 750 mA @ 14v., gas discharge display, direct to	$699.00	$1,070.00	19	10	27-Jan-2014

Table name: CUSTOMER

	CUS_NUM	CUS_LNAME	CUS_FNAME	CUS_INITIAL	CUS_AREACODE	CUS_PHONE	CUS_YTD_BUY	CUS_CREDIT	CUS_BALANCE
+	10010	Ramas	Alfred	A	615	844-2573	2011.56	5000.00	0.00
+	10011	Dunne	Leona	K	713	894-1238	4613.29	5000.00	1708.51
+	10012	Smith	Kathy	W	615	894-2285	3217.94	5000.00	0.00
+	10013	Olowski	Paul	F	615	894-2180	10020.53	5000.00	9562.67
+	10014	Orlando	Myron		615	222-1672	14234.85	7500.00	4358.55
+	10015	O'Brian	Amy	B	713	442-3381	3005.77	5000.00	0.00
+	10016	Brown	James	G	615	297-1228	5432.76	5000.00	1217.84
+	10017	Williams	George		615	290-2556	1629.95	1000.00	1629.95
+	10018	Farriss	Anne	G	713	382-7185	3856.31	2500.00	1232.42
+	10019	Smith	Olette	K	615	297-3809	8938.43	7500.00	854.76

Table name: SELLER

EMP_NUM	SEL_YTD_SALES	SEL_PCT	SEL_YTD_COMMISSION
102	27037.65	8.00	2163.01
105	21945.51	8.00	1755.64
106	30543.82	10.00	3054.38
108	25139.94	5.00	1257.00
109	29515.68	8.00	2361.25
110	20012.45	5.00	1000.62

Table name: EMPLOYEE

EMP_NUM	EMP_TITLE	EMP_LNAME	EMP_FNAME	EMP_INITIAL	EMP_HIRE_DATE	EMP_DOB	JOB_CODE
100	Mr.	Kolmycz	Andrew	H	15-Mar-85	15-Sep-62	T
101	Dr.	Lewis	Barbara	K	25-Apr-90	01-Dec-83	T
102	Ms.	Vandam	Marie	L	20-Dec-93	24-Mar-77	S
103	Mr.	Jones	Robert		28-Aug-96	18-May-84	A
104	Dr.	Lange	William	B	20-Oct-97	03-Jul-65	A
105	Mr.	Williams	George	D	08-Nov-97	19-Nov-72	S
106	Mrs.	Duzak	Judith	E	05-Jan-89	28-Aug-68	S
107	Ms.	Diante	Melanie	K	02-Jul-97	07-Dec-59	T
108	Ms.	Wiesenbach	Roberta	F	18-Nov-95	18-May-81	S
109	Mr.	Travis	Peter		14-Apr-89	19-Sep-79	S
110	Mr.	Genkazi	George	R	01-Dec-03	30-Jan-75	S

Table name: INV_LINE

INV_NUM	INVLINE_NUM	PROD_CODE	INVLINE_UNITS	INVLINE_PRICE	INVLINE_TOTAL
10001	1	GPS-55AVD	1	625.00	625.00
10001	2	TMA-350D	2	850.00	1700.00
10002	1	KX-155	1	1599.00	1599.00
10003	1	TNL1000(DC)	1	1695.00	1695.00
10003	2	KMA-24	1	599.00	599.00
10003	3	CP-136M	4	980.00	3920.00
10003	4	TRT-250D	1	1070.00	1070.00

Table name: INVOICE

INV_NUM	CUS_NUM	EMP_NUM	INV_DATE	INV_SUB	INV_TAX8%	INV_TOTAL	INV_PYMT	INV_BALANCE
10001	10015	103	14-Jan-14	2325.00	186.00	2511.00	2511.00	0.00
10002	10018	105	14-Jan-14	1599.00	127.92	1726.92	1726.92	0.00
10003	10010	105	15-Jan-14	7284.00	582.72	7866.72	5000.00	2866.72

8. Using the ERD shown in Appendix C, The University Lab: Conceptual Design Verification, Logical Design, and Implementation, Figure C.22 (the Check_Out component), create the equivalent OO representation.
9. Using the contracting company's ERD in Chapter 6, Normalization of Database Tables, Figure 6.16, create the equivalent OO representation.