

Module 6

This is a single, concatenated file, suitable for printing or saving as a PDF for offline viewing. Please note that some animations or images may not work.

Module 6 Study Guide and Deliverables

Readings:**Required Reading:**

- CB6 chapter 27 (you can skim the following sections: 27.3.1, 27.4.5, 27.8, and 27.8.2)

Recommended Reading:

- CB6 chapters 27–28
- Loney chapters 38–41

Assignments: Extra Credit Assignment 6 due Wednesday, August 19 at 5:00 PM ET

Assessments: Quiz 6 due Wednesday, August 19 at 5:00 PM ET

Term Project Deliverable: If you have not already done so, during the last module you should complete your project presentation and report and submit them to your facilitator by the dates that you and your facilitator have chosen. There are three separate dropboxes where you can submit your Term Projects—one is for your presentations, one for your reports, and one is for your source code or other supporting data. There is considerable flexibility in the particular deliverables, depending on the design of your term project, so your approved

project proposal and plan may not have all of these deliverables. I have provided three dropboxes to accommodate the full range of possible deliverables. I do require that everyone have a presentation covering the central aspects of their term project in a form that their classmates, facilitator and I can experience, and that you submit a presentation document or a document containing a link and instructions—in the term project presentation dropbox.

Everything should be submitted by Wednesday, March 4 at 5:00 PM ET.

Live**Classrooms:**

Supplementary Live Session,
Wednesday, August 12, 8:00 PM-
10:00 PM ET

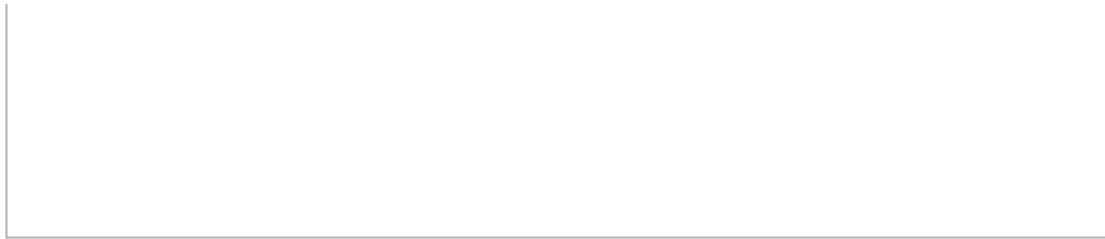
Current week's assignment review
and examples, Thursday, August 13,
8:00 PM - 9:00 PM ET

Live office help, Saturday, August
15, 11:00 AM - 12:00 PM ET

Live office help, Monday, August 17,
8:00 PM - 8:45 PM ET

Introduction

metcs779_wk5 video cannot be displayed here



Object-Oriented databases have dominated the Computer-aided Design (CAD) and Computer-aided Manufacturing database market niches for many years. In the last few years, the dominance of object-oriented languages for application development (particularly Java) has caused a resurgence in popularity of object-oriented databases, and new object-oriented databases such as DB4Objects (which is free) have increased the interest. Object-Oriented DBMS are usually faster than relational DBMS, and they are easier to integrate with object-oriented software, which can lower development and life cycle costs by over 10%.

Learning Outcomes

By reading this lesson, participating in discussions, and completing the assignments, you will be able to:

- Describe and use the basic concepts of object-oriented technology.
- Describe how object-oriented models are related to entity-relationship models.
- Use object-oriented ways of thinking in database design.
- Extend entity-relationship models into object models.
- Integrate object-oriented and relational models.
- Decide when to use object-oriented data modeling techniques and how to combine them with entity-relationship techniques.
- Describe the basic features of an object-oriented database management system (OODBMS).
- Design relational databases that efficiently support object-oriented applications, including the three ways to map specialization-generalization relationships to relational databases.
- Describe when and how to use the UNDER construct in object-relational databases to directly represent specialization-generalization relationships.
- Decide when it is appropriate to use an object-oriented database.
- Describe the relative strengths and weaknesses of ODBMS and RDBMS.
- Explain how object-oriented concepts are influencing the relational model.
- Explain why RDBMS vendors such as Oracle have extended their relational DBMS into object-relational DBMS (ORDBMS), and why recent ANSI/ISO SQL standards are object-relational.
- Understand and use the UML graphical object-oriented notation for database design.

■ Lecture 6A - Basic Object-Oriented Concepts

Basic Object-Oriented Concepts

We all have been using object-oriented ways of thinking since we were young children. In this section we will learn how to talk about and model basic concepts that we have all used since childhood. This is both exciting and sometimes a little difficult, simply because it is so fundamental. It is also extremely useful.

An object is a software entity that usually represents an entity in the real world. The object may represent something concrete such as a person, house, or vehicle, or it may represent something more abstract such as a particular bank account. Software objects include data that represents the properties of the real world objects. For example an object that represents a person may include a representation of the name of the person as data in the form of a text string. Similarly an object that represents a bank account may include data in the form of a number that represents account balance.

An object class is a software entity that represents a collection of objects that share common properties and behavior. Each software object is described as an *instance* of its object class. The real world classes that software object classes represent often have names such as *car*, *boat*, *person*, *student*, or *instructor*. Classes in our shared experience are essential for natural language, so we have all known how to work with classes since we learned to talk. Much of what we learn has to do with classes and their relationships. Software classes usually represent real world classes, and the software classes are usually given names that are the same as or similar to the words or phrases that we use to talk about the corresponding real world classes.

Abstraction is grouping objects or classes based on common characteristics or behavior. For example we may abstract the class *mammal* as applying to animals that have the common characteristics of having fur and being warm blooded, and the common behavior of giving birth to their offspring. There are different levels of abstraction. Data modeling languages which support objects, such as EERM and UML, explicitly model abstractions and relationships between object classes, and databases which support objects explicitly represent and operate on objects that participate in specialization/generalization relationships.

Test Yourself

Select all the answers that apply to basic object-oriented concepts.

An object class abstracts the common features of instances of the class.

This is true. The object class represents common data elements, constraints, and behaviors that apply to all instances of the class.

If an object class represents cookies, an example of an *abstraction* of that class would be chocolate chip cookies.

This is false. An abstraction groups objects or classes together based on common characteristics, so in this example, an abstraction of cookie would be something more general, like *Dessert* object.

A person's height might be represented as an attribute of the Person class of which the person is an instance.

This is true. If the model were to track time variations in height measurements this would still be true; the Person class height attribute would then consist of measurement records.

Specialization is the name for relationships between classes that differ in levels of abstraction. When we talk about specialization relationships between pairs of classes we talk about the more abstract and general class as

the *superclass*, and the less abstract and more specialized class as the *subclass*. For example a software class that represents mammals may be a subclass of a software class that represents animals. Similarly we say that a software class that represents (or implements) bank checking accounts may be a subclass of a software class that represents bank accounts. Abstraction requires that every subclass has all of the characteristics and behaviors of all of its superclasses. In English we use a phrase such as “is a kind of” or “is a” to describe the generalization/specialization relationship. For example we say that an automobile is a kind of motor vehicle and that a tandem is a kind of bicycle.

Object-Oriented modeling is the development of software representations for the entities, relationships and behavior in a problem domain. Object-oriented modeling works better than entity-relationship modeling for more complex problem domains such as enterprises, and most larger enterprises have developed enterprise object models to provide a unified framework that application developers use to organize and interface between application systems. These same organizations may use entity-relationship models based on those enterprise object models when designing and developing databases for simpler subsets of the enterprise problem domain. Object-oriented modeling is necessary for the development of software in object-oriented languages such as Java and C++. The most common graphical representation for object-oriented models is the [Unified Modeling Language](#), just as the most common graphical representation for entity-relationship models are variations of the Crow's Foot graphical notation. The two notations are influencing each other and notations are evolving that combine both.

Encapsulation is the organization of software and data into software modules that represent object classes and objects. *Hiding* refers to restricting access to the components of an object, usually based on its class. Within a software class the behaviors are modeled by procedures and functions that in the context of object classes are referred to as *methods*. The more data-like properties of objects are usually modeled as data. In the context of objects the data properties are referred to as *fields* or *attributes*.

Test Yourself

True or False: If Employee is an encapsulated class, a public method of Employee called `getEmployeeName` could be run from methods or other code that is not part of the Employee class.

This is true. Methods of an encapsulated class can be run on objects of that class, provided the methods have been assigned the correct access level (public, package or protected).

Object-Oriented database management systems (ODBMS or OODBMS) are DBMS that directly support and represent object classes, instances, specialization, data, and methods. A true ODBMS can fully support durability, search, update, and other DBMS functionality for a language such as Java or C++, and there are usually utilities for ODMS that take in descriptions of the entity classes in languages such as C++ or Java, produce the object-oriented schema, and even migrate data from earlier schemas to updated schemas.

Object-relational DBMS such as Oracle and DB2 have the functionality of relational DBMS and most of the functionality of ODBMS. The relational and object-oriented models and their database implementations are compatible, and it turns out that many of the features of the object-oriented technologies can produce faster,

more expressive databases when those features are added to relational databases. Relational technologies are also useful in object-oriented databases; for example, most modern commercial ODBMS support some SQL, though not necessarily very efficiently.

Test Yourself

True or False: An ODBMS has the functionality of both a relational DBMS and an object-oriented DBMS.

This is false. An ODBMS usually has the functionality of an object-oriented model, but does not contain the features of a relational database. An object relational database has the functionality of both.

The History of Object-Oriented Concepts

Pre-history and history—The ability to identify and use classes is fundamental to words and other symbolic language. Anthropologists have identified a flowering of human culture about thirty thousand years ago in cave paintings and in many other ways, and many conjecture that humankind developed symbolic language about thirty millennia ago. About five millennia ago humankind began to develop durable ways of recording and organizing our symbols for classes in the form of writing. The historical writings show a steady development in object-oriented ways of thinking. Bertrand Russell completed many of the philosophical foundations in the early twentieth century, and later in the twentieth century theoretical computer scientists such as Gödel discovered and formally defined such concepts as non-denumerable and non-decidable classes.

Early software development in the mid-twentieth century was mainly focused on getting computers to perform useful tasks by providing detailed step-by-step instructions to the computer, so it was natural that early program organization and ways of thinking about software were developed around ways to organize the functions that those instructions were designed to perform. The development of this intellectual thread led eventually to the disciplines of structured programming, which are based on the decomposition of programs into hierarchies of functionally-organized units. As programs became more complex programmers noticed that they often had to rewrite large parts of their programs when requirements changed, because the functional decomposition hierarchies often changed at a large scale when the requirements changed on a small scale. During this time it was normal that software development projects ran several times over budget and most software development projects failed, usually because changes in requirements, things learned about the software during the development, or discovery of new requirements forced so many rewrites of the software that people eventually gave up. The basic reason was that the functional decomposition principle is very brittle, meaning that defects in one area tend to propagate like cracks in tempered glass to shatter the whole software structure. This is because structured programming does not have a robust organizing principle that keeps the consequences of changing one part of the software from propagating to force changes in other parts of the software.

Simula—Software developers working in areas such as simulation found these problems with functional organization intolerable. Simulation engineers realized early in the 1960s that they needed a way to add new things (classes) as modules to their software, and explored organizing software into classes. By 1967 they had

developed languages such as Simula, which is arguably the first fully developed object-oriented programming system.

Object-Oriented Technology in Commercial Software—The software research community recognized the advantages of object organization very quickly, but the commercial software community was very slow to catch on. I remember the 1980s when I was at BBN developing software systems using object-oriented languages such as Flavors in a matter of weeks that major programs had failed to develop in many years. Old ways died hard, and in the commercial software world most projects still continued to fail. Eventually the commercial world realized that something important had happened, and efforts began to add object-oriented organizing principles to functionally-oriented languages such as C, and to retread software engineers by teaching them object-oriented ways of thinking. By the dawn of the 21st century the huge advantages of object-oriented software organization had become clear enough to management that organizations such as IBM mandated the use of languages such as Java for all application development.

Relational databases—A few years after Simula, in the late 1960s and early 1970s, Codd and others at IBM's San Jose Research Center published papers that defined relational databases and SQL. The database community recognized that the ability of relational databases to support ad hoc query without difficult programming was a huge advantage, but they were skeptical that these databases could be made to run fast enough. Over the 1970s many small companies were founded to develop relational databases, and the technology was developed to make relational databases faster, more scalable, and easier to use. The ability to use SQL to ask a database any reasonable question and to usually get an answer in an acceptable time proved so valuable that SQL databases quickly came to dominate in most application areas, and continue to dominate the database marketplace today. Relational databases represent classes as tables, and their lack of explicit abstraction, specialization or hiding was not usually a problem, mainly because most database designs are much simpler than corresponding software designs.

Object-Oriented databases—Software engineers working in computer aided design (CAD) and computer aided manufacturing (CAM) of complex systems such as automobiles and aircraft found that the organizing principles of relational databases were too brittle for their needs, and that relational databases were too slow to support the complex models that they needed. Consequently they adopted object-oriented representations in both their design systems and in their databases. This led to the development of many object-oriented databases, including Object Store and Versant. Today the object-oriented database market represents only a few percent of the total database market. Today application engineers will occasionally find the higher performance (often more than ten times higher), ease of interfacing to object-oriented software, and improved expressive power and conceptual integration to be compelling reasons to use an object-oriented database, so they choose to use an object-oriented database even though it is not as mainstream as SQL databases in most application communities. Object-Oriented database vendors also added ad hoc query to their databases, originally by supporting SQL, and more recently by supporting interactive native language query in languages such as Java.

Object-relational databases—Relational database vendors such as Oracle have long known the importance of some of the features of object-oriented databases, such as abstraction, methods, and the higher performance of navigational query. Oracle had implemented object-oriented extensions to its RDBMS as early as 1985, and worked for a decade to improve the performance of their emerging object-relational architecture. When Oracle rewrote their RDBMS kernel for Oracle 8 they added the object-oriented extensions that they found most useful that would improve performance without degrading it. In the rewrite they about tripled the speed of the database

when it was functioning as a purely relational database, and provided more than an order of magnitude speedup for many applications that could make good use of the object extensions. These object extensions have been in the ANSI SQL standards, since SQL 2003. We will study object-relational, LOB, and other extensions to the relational model in the second lecture this week.

Entity-Relationship and Object-Oriented Models

People communicate mainly in terms of entities (objects), relationships between objects, and actions by and upon objects. The best way for people to communicate about complex things is often to draw pictures, in part because our visual system uses much more of our brains than our natural language system. The relational algebra and calculus were fine as a theoretical foundation for relational databases, but they didn't help application developers design their relational databases. Chen and others developed graphical languages that represented classes (entities) as rectangles with text inside, and represented relationships between classes as diamonds or lines. The Chen notation, which made relationships into graphical objects, proved to not be sufficiently dense on the page, and difficult to implement in tools, so it was succeeded by Crow's foot notations, which represent the relationships by lines with annotations and decorations. It is frequently important to represent abstractions, specialization, composition, and methods, so the Crow's foot notations have been extended to form Extended Entity-Relationship (EER) graphical modeling languages. Today the best practice is usually to use UML class notation with extensions, and this is used in our main Connolly and Begg text.

Object-Oriented Programming

Those of you who have learned Java, C++, C#, Smalltalk or another object-oriented programming language are already familiar with object-oriented programming, so this section should be mainly a review. For those of you who haven't mastered object-oriented programming this section is a basic introduction to it, so that you can begin to understand why object-oriented databases are useful, particularly to implement persistent objects in object-oriented programming.

Object-Oriented programming was born in the synthesis of functional decomposition ways of thinking with object-oriented ways of thinking. The result was that almost all application programming languages in use today are based on or incorporate the object-oriented programming models. There are different object-oriented programming models, but essentially all modern application development uses object-oriented programming. Functional organization is a better fit to lower-level programming such as operating system and database kernels, so these are today built using a combination of C and C++, which are both optimized for performance, though these lower level modules are gradually being converted to languages such as C++, which have object-oriented extensions.

Most object-oriented programming languages are built on self-contained, reusable modules that contain data as well as the procedures. These modules correspond to object classes. The data and procedures in a software object class are termed *fields* and *methods*, respectively, and together the fields and methods of a class are termed its *members*. The class modules may be further organized into *packages*. The class modules include access controls that divide the data and procedures into categories which restrict the access to these data and procedures. Access controls are typically implemented as visibility controls, so that an unauthorized programmer or hacker cannot gain knowledge of private details of a class from access control error messages. The four visibility categories which are implemented in C++ are summarized in the following table.

Visibility Category	What Methods can Access Members of this Category
Private	Only visible to methods in the class
Protected	Only visible to methods in the class and subclasses.
Package scope	Only visible to methods in classes in the package
Public	Visible to any method that knows the name of the member

Test Yourself

To make a field or method available to only some classes in a package, which visibility control would you choose?

Private

This is false. A Private visibility control would prevent any other classes from accessing that method.

Protected

This is true. A Protected visibility control would allow a class and its subclasses access to the method, and hide it from other classes in the package.

Package

This is false. Setting the visibility control to Package would allow All classes in the package to access the method.

Public

This is false. Setting the visibility control to Public would allow All classes to access the method.

Visibility controls are not part of the relational model because the need for visibility controls was not recognized when the relational model was developed. Table-level access controls were added to the relational model in the 1970s in the form of privileges, and finer-grained access controls were added in the form of views, but the lack of effective access controls can make it difficult or effectively impossible to establish and maintain complex enterprise systems if hundreds or even thousands of applications are allowed to directly access the tables of a relational database. One common solution to this problem is to develop stored procedures that implement transactional interfaces between relational databases and their applications and to hide the underlying relational tables from the applications. What this approach does is provide a database interface organized according to the business transaction object classes, with the code of the stored procedures implementing any required security checks.

The advantages of object-oriented technology in programming languages are summarized in the following table, together with the corresponding advantages of object-oriented technology in databases.

Advantage of Object-Oriented Technology in Programming Languages	Corresponding Advantages of Object-Oriented Technology in Databases
Provides a stable software organizing paradigm that reduces risk and life cycle cost.	Representations in the database that faithfully model the real world
Provide natural stable encapsulated interfaces to other systems and users	Provides ideal interfaces for object-oriented software
Supports the features needed to directly implement natural domain models	Directly supports natural enterprise and other domain models
Decreases development effort by reducing the amount of code	Reduce development effort to almost nothing, because the schema is the same as the application class model
Encapsulates code in natural domain-based models, greatly improving the opportunities for code reuse	Supports direct schema reuse based on enterprise and industry object models
Integrates data and procedures with their object classes	Integrates behavior with the data rather than in separate stored procedures
Directly supports both abstract and concrete classes as software modules	Supports both abstract and concrete durable classes and objects

Definition of Domain

The term *domain* (sometimes termed *problem domain*) refers to a business area or activity with established processes, entities, roles and practices. Examples of problem domains include retail banking, post-graduate education, private health clubs, retail restaurant chains, and commercial airlines. Complex domains often have well-defined sub-domains. For example, flight operations and aircraft maintenance are sub-domains of the airline domain and graduate education is a sub-domain of the education domain.

Problem domains are important because the object models of mature domains are uniform and stable, even though the details of the procedures and applications software in those domains may vary. This uniformity of the object models is the basis for reuse of object-oriented software within problem domains and the success of object-oriented XML standards within problem domains.

Object Identity

Object Instance

An object instance represents a particular instance of a class. For example, you are an instance of the class *graduate student*.

Object-oriented and object-relational databases have one concept that is not explicitly present in object-oriented programming—the concept of an *object identity*. An object identity is a read-only data attribute associated with each object instance. Each object identity (OID) is unique to the particular object instance with which it is associated. The OID is assigned by the ODBMS when the object instance is created in the database. It cannot be changed by the software or even by the object methods. An OID can never be reused or destroyed. It must remain unique for all time, and in distributed object-oriented databases and applications, each OID must be unique over all time and all database instances. This is difficult to guarantee in practice, and it is not surprising that OIDs are many bytes long. A few years ago OIDs included the static IP of the platform of the ODBMS that created the object, but this was found to present security problems, so the standards were changed so that OIDs are now generated in ways such that the IP of the source ODBMS cannot be determined. These new algorithms do not positively guarantee uniqueness of OIDs over all instances and all time, but they have been engineered so that the probability that two distinct objects have the same OID is very low.

Test Yourself

Select all the answers that are true:

An object identity is the name of an object instance. For example, the string “Ford Focus” could be the object identity for an instance of the Ford_Focus object class.

This is false. The object identity is automatically generated by the system when the instance is created. Object IDs must be with high probability unique for all databases for all time and they are usually numeric.

An OID must remain unique in the database for all time.

This is true. OIDs are generated with unique numbers, and can never be reused or destroyed.

Object identities are unique for each object.

This is false. Object identities are assigned for each object instance, not at the object level.

Object Attributes and Object State

Definition of Combinatorially Large

The term *combinatorially large* refers to the fact that the number of combinations of different values of an object increases faster when additional attributes are added than functions such as polynomials. Combinatorially large is really large. Consider that the number of possible states of an object with a few dozen floating point attributes is larger than the number of atoms in the universe. Combinatorially large is qualitatively much larger than counting *actual* states of objects. Combinatorially large numbers only occur when counting *possible* states of objects, which is what the domain of an object is.

The data members of objects are called *attributes*. An object class may have attributes such as a count of the number of instances of the object created in an application system, or constants, but most attributes are associated with object instances. For example, an automobile has some attributes, such as its gross vehicle weight rating, that are based on its class, and other attributes, such as its VIN and current license plate number, that are based on its instance. Each attribute of an object has a domain, which is the set of all legal values that the attribute can take on. For example, the *balance* attribute of a bank account object may have a domain that includes only non-negative numeric values with a precision of one cent and some specified maximum value.

The *state* of an object is just the set of legal values of all of its attributes taken together. A change to the value of any one of the attributes of an object changes the object's state, even if all of the other attributes remain unchanged. The number of possible values for the state of ordinary objects can be combinatorially large.

It is usually desirable to hide (not make visible) the attributes of an object, and to provide methods that read and change the attributes. These methods can log, control, or otherwise extend the primitive read and write operations on data, and these extensions are under the control of the object, which has responsibility for its own integrity and consistency.

Test Yourself

Select all possible states of an assignment in Blackboard:

Posted

This is true. This might be the state after the assignment has been uploaded by the instructor.

Submitted

This is true. If the assignment has been submitted, the value of one or more attributes changes so the assignment is in a new state.

Graded

This is true. If the assignment has been graded, the value of the grade changes and the assignment is no longer in the queue to be graded, so the assignment is in a new state.

Viewed

This is false. Viewing the assignment does not change the value of any attributes, so the state does not change.

Properties of Object-Oriented Databases

Methods and Messages

The term *method* refers to a procedure, function, or similar encapsulation of behavior that is associated with an object class or instance. In good design a method usually performs some operation that corresponds to a similar operation on or by the corresponding object in the problem domain. For example, an *Account* class may include a method that changes the *balance* attribute of an instance of the *Account* class and logs the transaction.

Definition of Message Passing

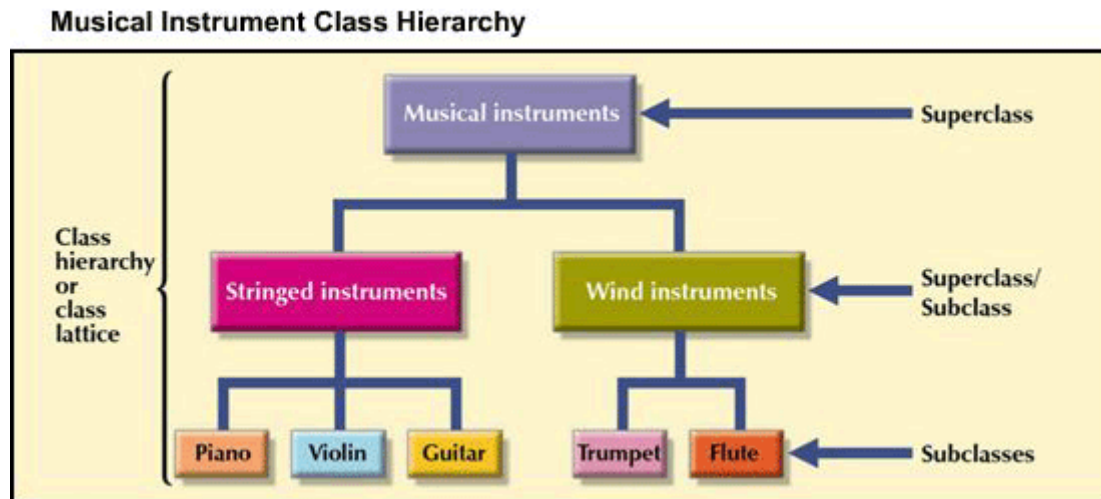
Message passing refers to the transfer of a structured block of data called a *message* between objects. The block of data usually specifies the object that sends the message, the object that is to receive the message, the method to be invoked when the message is received, and any parameter values to be passed to that method. Message passing is useful when building parallel or distributed object-oriented systems. In languages such as C++ or Java when one object invokes a method in a second object this is compiled into a direct procedure call to the method in the second object. This avoids the overheads of message passing, but loses some of the advantages of message passing, including parallelism, asynchrony, dynamic message routing, and message queuing.

Some ways of thinking about object-oriented programming systems, and some implementations of object-oriented programming systems use message passing to communicate between objects. This approach helps many people understand how objects communicate by passing messages between methods in one object and methods in another object, but it is not commonly used at runtime in most application systems, because message passing has more overhead than direct procedure calls. Message passing can allow the passer of the message to continue executing after a message is sent, without waiting for the recipient to finish processing the message, so message passing is more common in operating systems and application systems designed for high degrees of parallelism.

Class Hierarchies or Taxonomies

Specialization-generalization relationships can be extended beyond pairs of object classes to form hierarchies called *taxonomies*. Students of classical biology may remember the famous taxonomies used to organize species of plants and animals with the *plant* and *animal* kingdoms at the top and individual species at the bottom. Most complex problem domains can be organized in this way. Ordinary human thought makes extensive use of taxonomies. For example, if I asked you if a canary has skin you would probably answer correctly after a few seconds during which you would realize that a chicken is a bird and that birds have skin. You can answer this question correctly even though you have probably not experienced a canary's skin, because you know the taxonomic relationship between canaries and birds, and you know quite a bit about the common characteristics

encapsulated in the abstract class *bird*. The following figure from the text illustrates a simplified taxonomy of stringed and wind instruments.



The figure above illustrates single inheritance, meaning that each class has at most one superclass of which it is a specialization. In programming language such as C++ a class may have more than one superclass, implementing *multiple inheritance*. Multiple inheritance is much less commonly useful than single inheritance, in addition, it can slow down the runtime execution containing the instance variables of the classes that are formed using multiple inheritance, so recently developed object-oriented languages such as Java support only single inheritance.

Test Yourself

Select all that are true about inheritance:

Members of a subclass can add attributes to the attributes contained in the superclass.

This is true. The subclass inherits the attributes from the superclass, but can augment them with specific attributes for that subclass.

Multiple inheritance means that a subclass `ChocolateChipCookie` could inherit attributes and methods from both superclasses `Cookie` and `Chocolate_dessert`.

This is true. The subclass would inherit from both superclasses.

Advanced Topic: How Multiple Inheritance Slows Execution

The efficiency problem with multiple inheritance arises because methods of superclasses need to be able to efficiently access the corresponding inherited instance variables in the data frame of the subclass. With single inheritance the instance variables of the single superclass are allocated with the same offsets in the data frame of the subclass, with additional instance variables added by the subclass occupying addresses with larger offsets. With multiple inheritance, there will be conflicts

where instance variables from multiple superclasses have the same offset. As a result with multiple inheritance the offsets must be computed using a lookup mechanism, which is slower than the compiled-in register-relative fixed offset data access used with single inheritance. This is important because methods most frequently access instance data members.

The basic principle of taxonomic relationships is that the subclasses must have all of the attributes and behavior of their superclasses. In most modern object-oriented programming languages a subclass may replace a method inherited from a superclass, but it cannot delete an inherited method or an attribute. Subclasses may add methods or attributes of their own to those that they inherit. For example, we know that a canary has skin because we know that all birds have skin and that a canary is a kind of bird. In childhood most people generalize from their experience and assume that all birds can fly, then later they learn of penguins, ostriches and other flightless birds, and so subdivide their bird class into a class for birds that fly and a class for flightless birds. Note that this subdivision is based on the behavior of flying, not on data attributes of birds.

Polymorphism

Definition of Polymorphism

Polymorphism means “many forms” which indicates that the method has many implementations which share one common interface.

Polymorphism is just a fancy name for a feature of most object-oriented programming systems that relieves the programmer from some of the responsibility for knowing the precise class of every object that they operate on. Polymorphism operates by permitting a programmer to invoke a method of an object that the programmer knows is inherited from a superclass, even though the programmer does not know the subclass of the object. The runtime system determines the subclass of the object and invokes the appropriate method associated with the object. Polymorphism is particularly useful in complex problem domains. For example the programmer may know that an object is an instance of the *icon* class, and the programmer may know that *icon* implements a *draw* method. The programmer can call the *draw* method of an *icon* subclass, and the appropriate *draw* method will be called, even though the subclasses replaced (overrode) the *draw* method in the parent *icon* class.

Abstract Data Types

Abstract data types (ADTs) are important in both programming languages and databases. Abstract data types implement some but not all of the functionality of object-oriented classes. Abstract data types permit the user to define new data types as composites of the basic data types in the programming languages or database. For example, the programmer or database designer can define an *address* abstract data type as containing an *address1* line, an *address2* line, a *city*, a *state_code*, a *country_code*, and a *postal_code*. The programmer or database designer can then use that ADT in defining other ADTs such as *person*, or to define concrete classes or tables such as *employee*. We will study Oracle's ADT implementation, which is the basis of the standards, in the next lecture.

Object-Oriented Models

Object-Oriented data models represent the data, behaviors, and other characteristics of objects, and their relationships between object classes and object instances. The standard graphical notation for object models is the OMG's UML language (www.omg.org). Appendix G of the text, which is included as a course resource, introduces the UML class diagrams, which are the main part of UML that is used to model databases. The main characteristics of object-oriented models are that they:

- Support the representation of complex objects and complex class relationships
- Are extensible
- Are capable of defining new classes, including their attributes and behaviors
- Support encapsulation with different categories of hiding
- Support inheritance
- Support inter-object references and object identity

The following table summarizes the relationships between entity-relational modeling and object-oriented modeling:

Entity-Relational Model	Object-Oriented Model
Entity set (table)	Object class or type
Entity (row)	Object instance
Attribute (column)	Instance field
<no representation>	Class field
Foreign key	<no direct representation, but they can be built into classes>
Primary key	<no direct representation, but they can be built into classes>
<no representation>	Specialization/generalization class relationships
ER diagram	UML class diagram or EER diagram

Extended Entity Relationship Modeling—The last item above introduces Extended Entity Relationship (EER) modeling. ER modeling represents entities and their relationships while OO modeling includes these primitives plus explicit representation of generalization-specialization relationships and composition, so it is not surprising that the evolutionary addition of specialization-generalization and composition to ER modeling, forming extended ER modeling works well. EER modeling is covered in CS669. If you are not familiar with UML class notation and its use in modeling databases, please read Chapter 12 of our Connelly and Begg 6th edition textbook (CB6).

Early and Late Binding

Binding refers to the association of some variable such as an attribute with a particular data type or data value.

Early binding refers to making that association early in the processing. Most programming languages and

databases bind variables as early as possible. *Late binding* refers to deferring the binding of data types and values until late in the processing. Late binding is very flexible, but more difficult to control in production, so it is mainly used in prototyping and design systems. Early binding in databases takes place when the schema designer chooses the data types for columns and the DBMS then generates errors when an application or user attempts to create a value in those columns that cannot be represented in the data type that the designer selected. We mention late binding because it is useful in CAD/CAM, so object oriented databases sometimes support late binding. No commercial relational or object-relational databases support late binding, though users of relational databases could sometimes use late binding, for example when they need to store attribute name-value pairs in a relational database and the data types of the attributes is variable or not known when the schema is designed. Languages such as PHP and Perl include name-value representations.

Test Yourself

True or False: Early binding is preferred over late binding in most business applications because early binding can detect attempts to create data that is inconsistent with the data type constraints specified by the designer.

This is true. Late binding is mainly used in design, prototyping, and similar systems where the data type may not be known when the schema is designed.

Versioning

Versioning means that a data storage system is able to retain both current and previous named states of the data repository. Versioning is normal in source control systems for software development, where it allows developers to revert to any identified previous state of the source code, for example, to determine where a bug crept into an application, or to revert to a previous working version of a module. Versioning is very useful in computer-aided design, so vendors of object-oriented databases were led to include support for versioning in their products. There is no fundamental reason why versioning is associated with object-oriented databases, and it can be built into relational databases as well. In fact since Oracle 9i the Oracle DBMS supports *flashback queries*, which allow you to query the previous states of tables during a defined retention window, using retained undo logs.

Flashback Queries and Databases

Flashback queries are useful for restoring a table to a previous state. Flashback databases allow you to restore an entire database to a prior state. In Oracle 12c when you request that a table be dropped it is actually retained and you can recover it using the RECYCLEBIN data dictionary view. These topics are covered in Loney Chapter 30. You are not required to understand Oracle's flashback or recycle bin capabilities for this course, though you may find them useful.

Test Yourself

Select all that are true about versioning:

Versioning is often used for software development source control.

This is true. Github is a popular versioning tool.

Flashback queries implement a kind of versioning.

This is true. A flashback query will show the previous version of the database tables by using retained undo logs.

Object-Oriented Database Management Systems

ODBMS combine object-oriented OO features, such as:

- Class abstraction, specialization and inheritance
- Instantiation with instance relations and reference-based navigational query
- Encapsulation with hiding
- Polymorphism

with database features, such as:

- Durability
- Ad hoc query
- Access controls
- Data integrity enforcement
- Security
- Transactions
- Backup and recovery

Combining these features has been a goal for more than two decades. In particular the combination of the flexible ad hoc query of relational databases and the fast navigational query of object-oriented databases is now available in object-relational DBMS such as Oracle and DB2 and in recent ANSI standards for ORDBMS.

The ODBMS Market and Frontiers

ODBMS Market—ODBMS have for two decades occupied a strong niche market next to the fifty-fold larger RDBMS market. ODBMS have been a catalyst for innovation, and are now being explored by many application developers who appreciate the significantly lower effort required to build an ODBMS to support an object-oriented application, and the much higher performance obtainable with object-oriented databases. In the past decade, a number of ODBMS have emerged with the modern hybrid model involving open-source software backed by commercial-quality support for a fee.

ODBMS Frontiers—Database management systems have been under development for more than fifty years, with hundreds of millions of dollars per year invested in their development. DBMS technology appears to still be in its infancy. Radical changes are still underway, such as development of columnar databases such as Vertica, and the development of technology to efficiently store and query unstructured big dataTrends that we see

developing are the increasing movement to open-source and hybrid commercial software models, as typified by the growth of the MySQL RDBMS and the interest in the DB4Objects ODBMS. One thing that is clear is that RDBMS vendors need to provide ways to reduce the engineering effort to design and support relational databases for object-oriented software. Their ODBMS competitors require little more than annotations in the application source code to provide object-oriented applications with a database that makes the objects durable and queryable. Software engineers and application development firms are increasingly looking to ODBMS as a way to reduce their development costs by over 10% while speeding the application's database operations and reducing life cycle costs.

Summary

Object-Oriented database management technology has been driven by the almost universal adoption of object-oriented technology for applications development, by the development of object-oriented enterprise models and industry standards, by the failure of the relational model to handle the complexity of applications such as CAD/CAM and simulation, and by the higher performance of ODBMS. Object-oriented modeling has largely replaced entity-relationship modeling, even for relational databases, where object-oriented modeling has been best practice for decades. Object-Oriented modeling allows the designer to more faithfully model real world objects and their relationships, so object-oriented models are more stable than other models, and object-oriented models have been developed for most enterprises and business domains. The ODBMS technology is still evolving rapidly, with the integration of object and relational technology and the ability to handle unstructured data being two areas where development is still rapid. ODBMS permit the database to represent more semantic information, including behavior, support for complex objects and collections, and abstract data and object types. ODBMS retain the performance advantages of navigational query, compared to the more flexible but slower query performance of the relational model. It is comparatively easy to develop an object-oriented database schema to provide database functionality behind an object-oriented application.

Lecture 6B - Enhanced Entity-Relationship Modeling

Overview

This lecture describes the Enhanced Entity-Relationship model (EERM). We begin by reviewing specialization/generalization concepts superclasses, subclass relationships, and attribute inheritance. Finally, we discuss aggregation and composition. The goal of this lecture is to provide you with an opportunity to learn the concepts of EERM to model more complex applications.

Enhanced Entity-Relationship Modeling is introduced in Chapter 13 of the sixth edition of Connolly and Begg. You are responsible for the material in this lecture and in Connolly and Begg.

Learning Objectives

By reading this lecture you will be able to:

- Understand and create Enhanced Entity-Relationship Models
- Describe specialization/generalization
- Explain and use attribute inheritance

- Explain and use the constraints on specialization/generalization
- Describe and use aggregation and composition

Introduction to EERM

There are many advanced database systems for which simple entity-relationship modeling has proven unworkable, because the models are unmanageably complex and difficult to maintain. The reason is that entity-relationship modeling is basically object-oriented modeling without the representational power and efficiency of specialization/generalization relationships or composition. The EERM has two common diagrammatic representations—the extended entity-relationship diagram notation (EERD) and the UML class diagram notation. We encountered EERD in CS 669. UML class diagrams are covered in MET CS682 *Systems Analysis and Design* and are used in our Connolly and Begg text. UML class diagram notation is usually preferred, because it is understood by both database people and application programmers. Standard UML class notation is purely object-oriented, so it doesn't include primary keys, foreign keys, nullability and other features unique to the relational model. The UML standard is extensible. There is an industry-standard extension to the UML class diagram notation which includes these relational features; that extended UML notation is used in our textbook.

Specialization/Generalization

The specialization/generalization relationship is a relationship between two entity classes, where one of the classes is more general or abstract than the other. We all learn about specialization/generalization in childhood. For example, children know that jellybeans and chocolates are kinds of candy; candy is the more general class and jellybeans and chocolates are the more specialized classes.

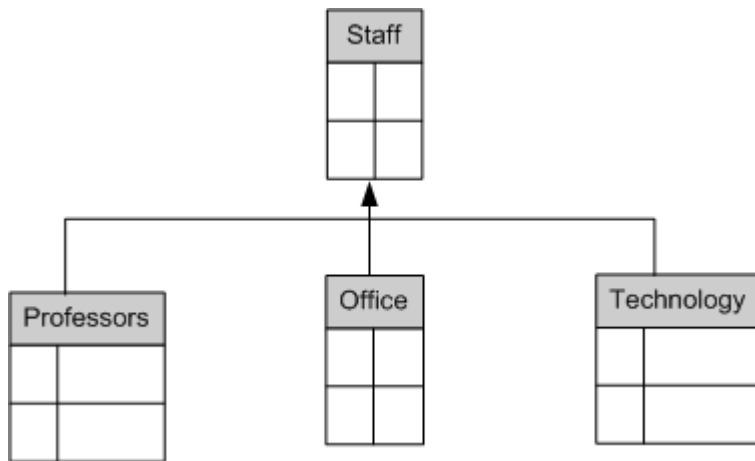
In specialization/generalization relationships the more general class is termed the *superclass* and the more specialized classes are termed *subclasses*. These are explained below.

A **Superclass** is an entity type that abstracts (represents) the common characteristics of one or more distinct subgroupings of its occurrences. For example, modern textbooks have a superclass which represents the title, author, and publisher and often two subclasses which represent the paper edition and the digital edition. Superclasses may be *concrete*, meaning that they may have instances, or *abstract*, meaning that they do not have sufficient information to be instantiated. For example, the class *mammal* is too abstract to be instantiated, but the *mammal* subclass *blue whale* is concrete.

A **Subclass** is a distinct subset of occurrences of an entity type (the superclass). For example, the classes *odd_integer* and *even_integer* are subclasses of the class *integer*. Subclasses may have further subclasses. For example, the class *integer* is a subclass of the class *number*. Classes may thus be organized into hierarchies based on their specialization/generalization relationships. Such hierarchies are termed *taxonomies*.

Attribute Inheritance

The attributes of superclasses are inherited by the subclasses. For example, consider the different types of Staff at a college. If the Staff supertype has an attribute *employee_ID*, then all of the subtypes have or *inherit*, this attribute.



The subclasses: *Professor*, *Office_staff*, and *Technology_staff* inherit all the attributes of the superclass *Staff*.

The subclasses are the different more specialized subtypes of the more general type staff.

Aggregation and Composition

Aggregation represents a "has-a" or "is-part-of" relationship between entity types, where one represents the "whole" and the others the "parts." An example of an aggregation is the players on the Red Sox baseball team. The players have a lifetime and existence apart from the team, yet they are together represented as the collection of players for the team.

Composition is a specific form of aggregation that represents an association between entities, where there is a strong ownership and coincidental lifetime between the "whole" and the "part." An example of composition is the relationship between fingers and the hand of which they are a part.

Test Yourself

Would you define the relationship between an order and its line items as aggregation or composition?

Composition.

The order lines would not exist separately without the order.

Participation and Disjointness Constraints

We may know two additional things about a relationship between a superclass and its subclasses. One of these is whether instances of the supertypes must be instances of one of the subtypes in the model, or whether there can be instances of the supertype that are not instances of one of the listed subtypes. This is termed a *participation constraint*. Participation can be *mandatory* or *optional*. An example of a participation constraint is a table that represents numbers, with subtypes even integers and odd integers. If zero is taken as even all integers are either even integers or odd integers, and not some other kind of integer, so this is a mandatory participation. In extended UML notation participation is indicated by a label in braces on the relationship, where the label is either *MANDATORY*, or *OPTIONAL*.

Something else that we may know is whether an instance of one of the subtypes can also be an instance of another subtype. This is termed a *disjointness constraint*. In the previous example of modeling integers there is a disjointness constraint in the relationship between integers and the subclasses odd integers and even integers, because an integer cannot be both odd and even. Disjointness is indicated on UML class diagrams by a word in braces, where the word is either *AND* or *OR*. In EERD notation disjointness is represented as (d), meaning a member of the superclass can only belong to one subclass, like the example with the integers

If a member of the superclass can be a member of one of the represented subclasses or can be a member of a different subclass, it could be represented with {o}. For instance, suppose there is a superclass for "Sports_player", with subclasses "Soccer_player" and "Baseball_player". Because it is possible for someone to play both soccer and baseball, it is possible that one "Sport_player" member could belong to either or both subclasses. This case represents an "overlapping" or {AND} relationship.

Test Yourself

Which participation and disjointness constraints would you include for the relationship between the superclass *IT Help Center Staff* and the subclasses *Student*, and *Full-Time IT Professional* where an *IT Help Center* staff member would have to belong to one of those two subclasses but could be both a *Student* and a *Full-time IT Professional*?

{MANDATORY, AND}

This is true.

{MANDATORY, OR}

This is false.

{OPTIONAL, AND}

This is false.

{OPTIONAL, OR}

This is false.

Lecture 6C - Abstract Data Types and Methods

Object-Relational Databases

Object-relational DBMS such as Oracle and DB2 combine most of the best features of ODBMS and RDBMS. Clients of mine have improved the performance of their Oracle DBMS by making effective use of the object extensions. In this lecture, you will learn about these extensions, using Oracle Examples. Most of these features have been adopted by the ANSI SQL standards and other DBMS vendors. The organization of this lecture

generally follows Loney Chapters 38 through 41. You can consult Loney for additional information, but you are not required to know more than the contents of this lecture.

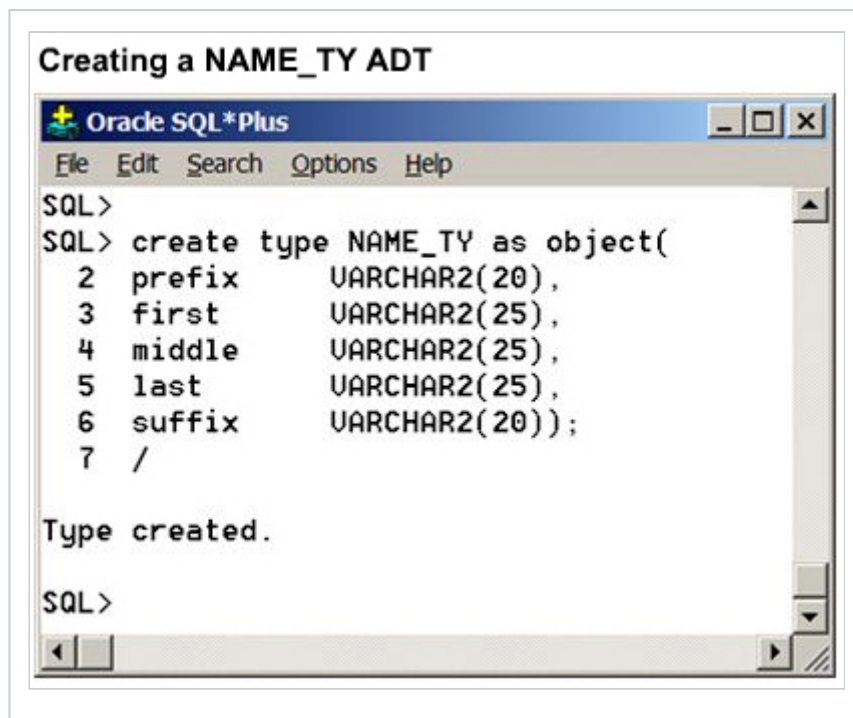
Learning Outcomes

By reading this lesson, participating in discussions, and completing the assignments, you will be able to:

- Explain abstract data types (ADTs) and use them in Oracle and other ORDBMS that support the ANSI/ISO standards.
- Explain and use methods as parts of object types in Oracle and other ORDBMS.
- Explain, create and use tables created with ADTs.

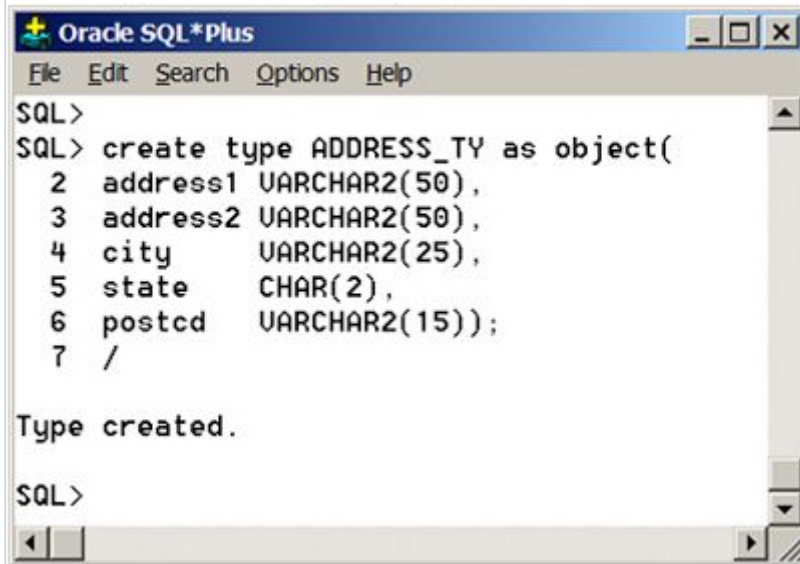
Abstract Data Types and Methods

An abstract data type (ADT) is a named user-defined object that combines base data types and other abstract data types into representations that can be used as parts of tables and additional abstract data types. The following figure illustrates the creation of the abstract data type NAME_TY in Oracle 10g. You could use this NAME_TY to represent the way that we store names in your databases.



The create type statement looks a lot like a create table statement, with named and typed column definitions. Next we create an ADDRESS_TY ADT to represent the ways that we represent addresses in our database.

Creating an ADT (object type)



```
Oracle SQL*Plus
File Edit Search Options Help
SQL>
SQL> create type ADDRESS_TY as object(
  2 address1 VARCHAR2(50),
  3 address2 VARCHAR2(50),
  4 city      VARCHAR2(25),
  5 state     CHAR(2),
  6 postcd    VARCHAR2(15));
  7 /

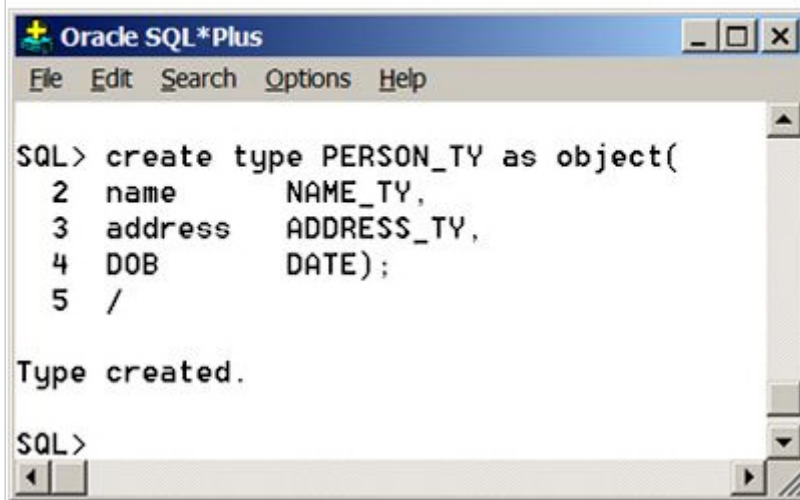
Type created.

SQL>
```

The use of such ADTs throughout a schema simplifies the problem of maintaining uniformity of representations for things like names and addresses in the schema.

We can use the NAME_TY and ADDRESS_TY ADTs that we just created to create a new PERSON_TY, where the new type includes both the NAME_TY and ADDRESS_TY columns.

Creating PERSON_TY using NAME_TY and ADDRESS_TY



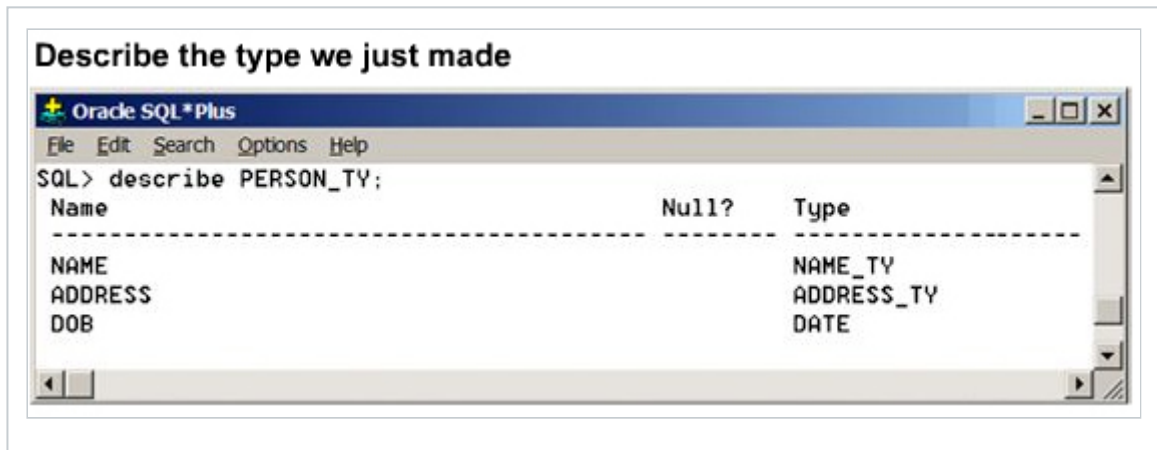
```
Oracle SQL*Plus
File Edit Search Options Help
SQL>
SQL> create type PERSON_TY as object(
  2 name      NAME_TY,
  3 address   ADDRESS_TY,
  4 DOB       DATE);
  5 /

Type created.

SQL>
```

Note that we used the NAME_TY and ADDRESS_TY ADTs just as if they were data types such as CHAR or NUMBER. This is what it means that an ADT is the abstraction for a data type. An ADT is not a base data type. We have extended the data type model of our DBMS by adding these new abstract types, which are composed of the base data types in the DBMS, and previously defined ADTs.

We next use the describe command to determine the structure of the ADT that we just created:



Something magic has happened. The DOB column has the base data type that we expected. The name and address columns are described as being of the new NAME_TY and ADDRESS_TY ADTs that we just created. We have extended the type system of the database!

Test Yourself

Select all that are true about abstract data types (ADTs):

An ADT is a data type that is comprised of both data types and other ADTs.

This is true. An ADT can contain both base data types and other ADTs.

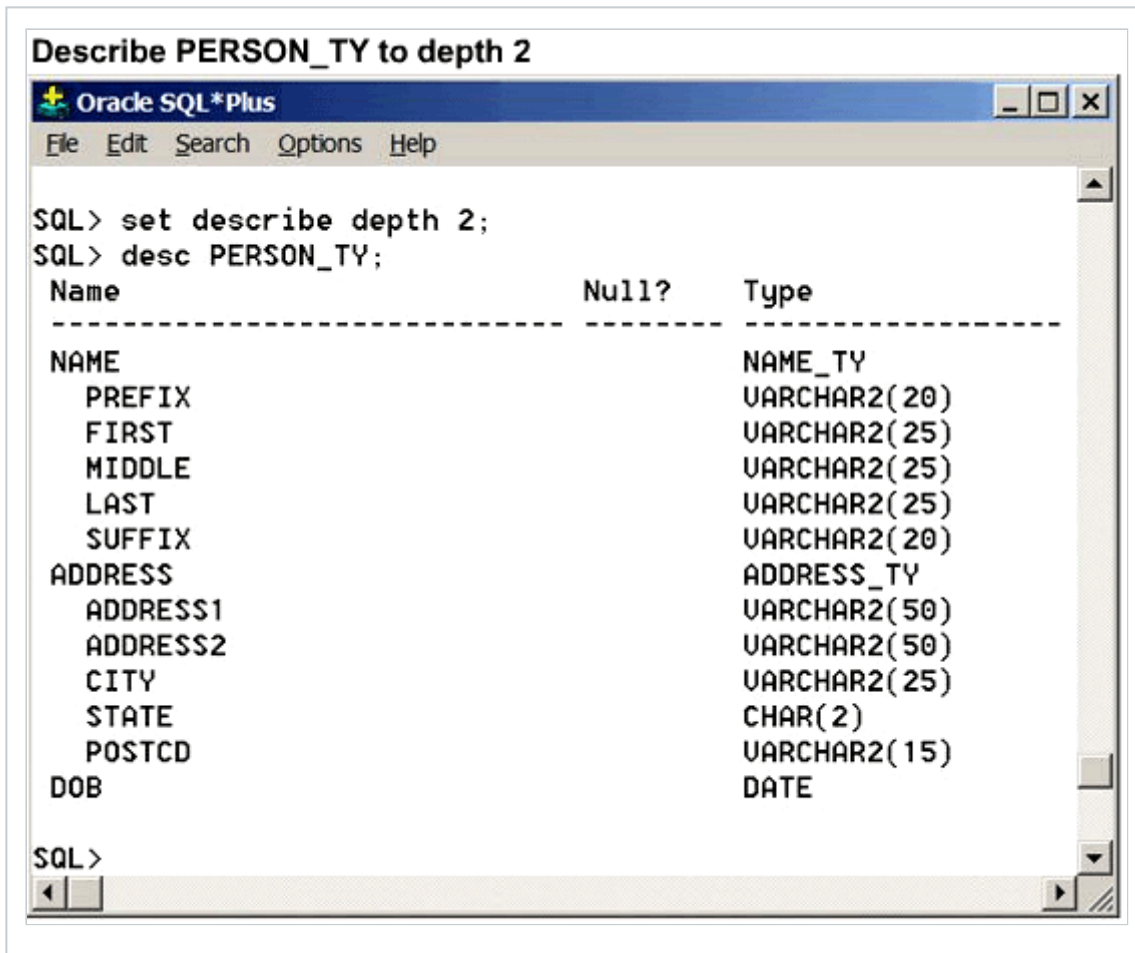
An ADT is itself a base data type to which we can add other ADTs.

This is false. A base data type is a data type that is primitive to the DBMS such as VARCHAR or INTEGER. An ADT is a user-defined type like PERSON_TY or ADDRESS_TY.

The Oracle Describe Command

What do we do if we want to look inside the NAME_TY and ADDRESS_TY that we used to create the PERSON_TY? The Oracle describe command has a describe depth parameter that defaults to 1. We can set it to a higher value to look inside the ADTs that appear when we describe something. Let's set it to 2 and look at the structure of PERSON_TY again:

Describe PERSON_TY to depth 2

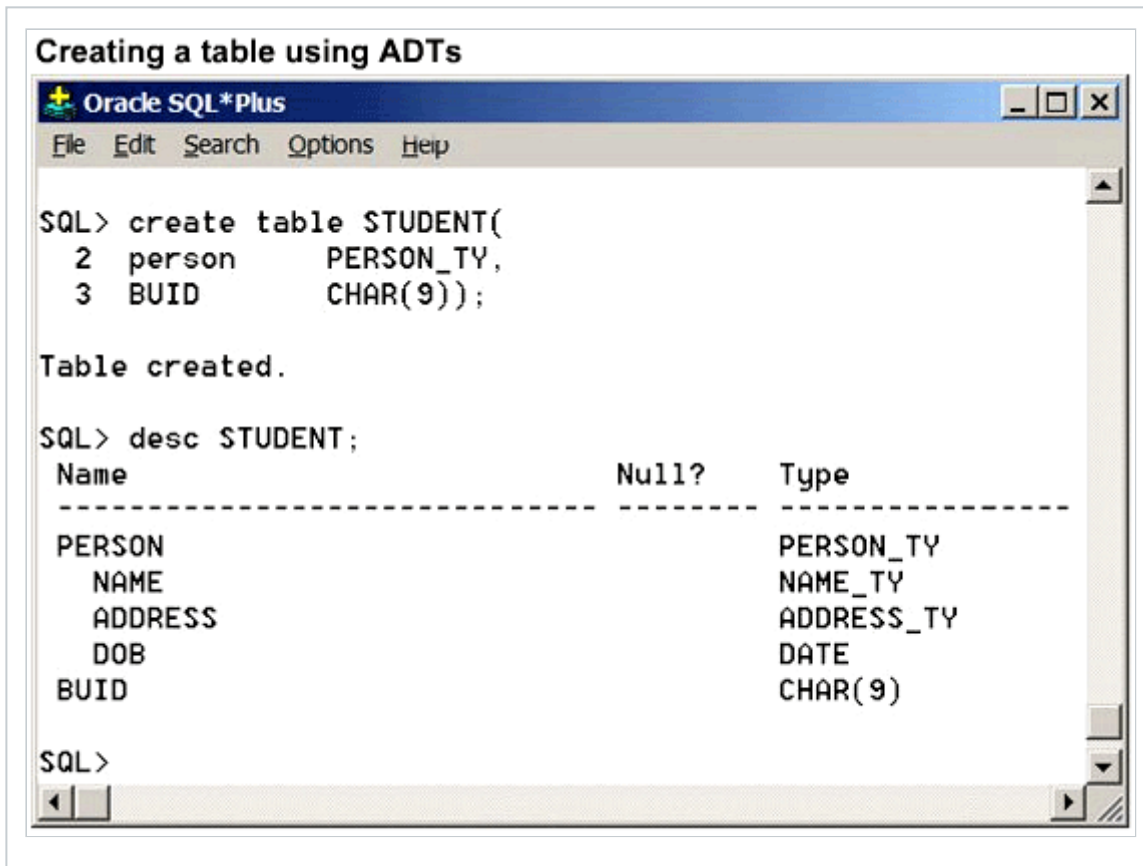


```
SQL> set describe depth 2;
SQL> desc PERSON_TY;
Name                                Null?      Type
-----
NAME                                NAME_TY
  PREFIX                           VARCHAR2(20)
  FIRST                            VARCHAR2(25)
  MIDDLE                           VARCHAR2(25)
  LAST                             VARCHAR2(25)
  SUFFIX                           VARCHAR2(20)
ADDRESS                             ADDRESS_TY
  ADDRESS1                         VARCHAR2(50)
  ADDRESS2                         VARCHAR2(50)
  CITY                             VARCHAR2(25)
  STATE                            CHAR(2)
  POSTCD                           VARCHAR2(15)
DOB                                DATE
```

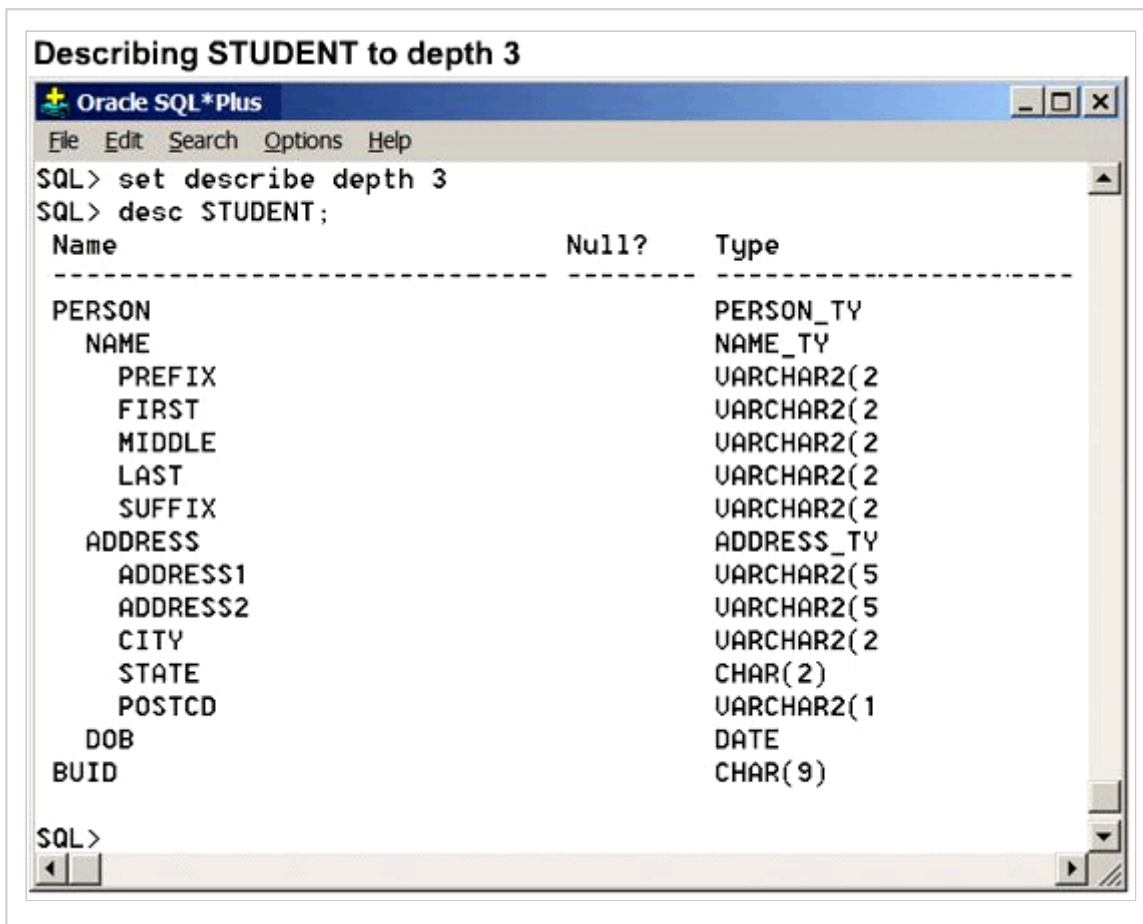
SQL>

Now Oracle showed us the data types of which NAME_TY and ADDRESS_TY are composed.

Now let's use the PERSON_TY ADT to create a real table in which we can store rows of data. Let's create a STUDENT table that represents a student at BU. Boston University students all have a Boston University ID or BUID, so we give this student the properties of a PERSON_TY plus a BUID:



Note that we still have describe depth set to 2, so Oracle shows us the insides of PERSON_TY. Let's try cranking describe depth up to 3 to look inside of NAME_TY and ADDRESS_TY too:



Now Oracle expands the NAME_TY and ADDRESS_TY ADTs too, so that we get to see all of the base data types.

Test Yourself

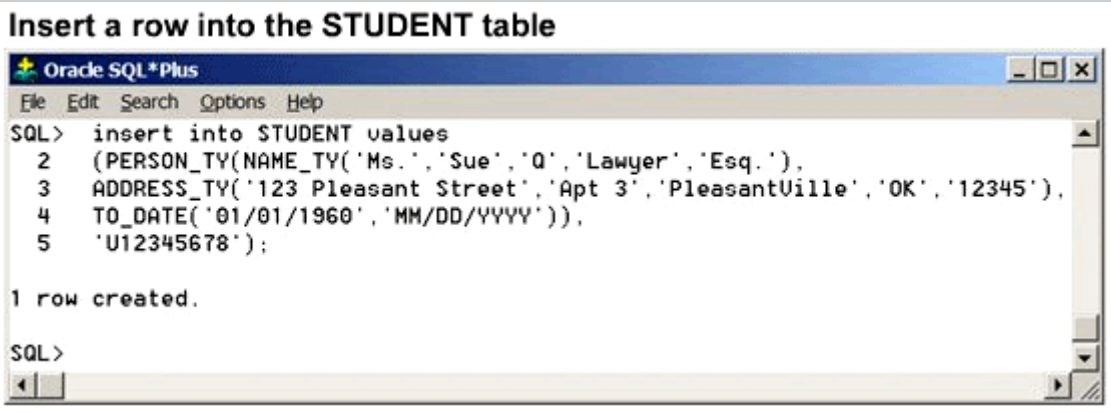
The professor asks a question: “How many physical columns are there in the STUDENT table?”

The STUDENT table has 12 columns. Eleven of these are based on PERSON_TY (5 from NAME_TY, 5 from ADDRESS_TY, and 1 for DOB), and one (BUID) is part of the definition of the Student table itself.

The Role of Constructors and Unique Naming

Now let's put some data in to the STUDENT table that we just created:

Insert a row into the STUDENT table



```

Oracle SQL*Plus
File Edit Search Options Help
SQL> insert into STUDENT values
2  (PERSON_TY(NAME_TY('Ms.', 'Sue', 'Q', 'Lawyer', 'Esq.'),
3  ADDRESS_TY('123 Pleasant Street', 'Apt 3', 'Pleasantville', 'OK', '12345'),
4  TO_DATE('01/01/1960', 'MM/DD/YYYY')),
5  'U12345678');

1 row created.

SQL>
  
```

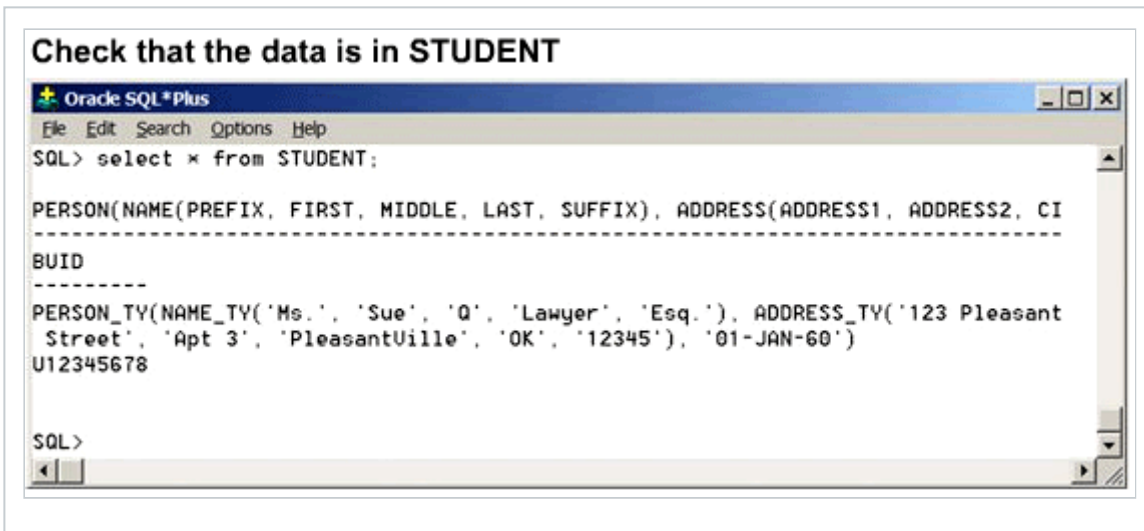
Test Yourself

True or False: Calling an ADT constructor creates an instance of the ADT.

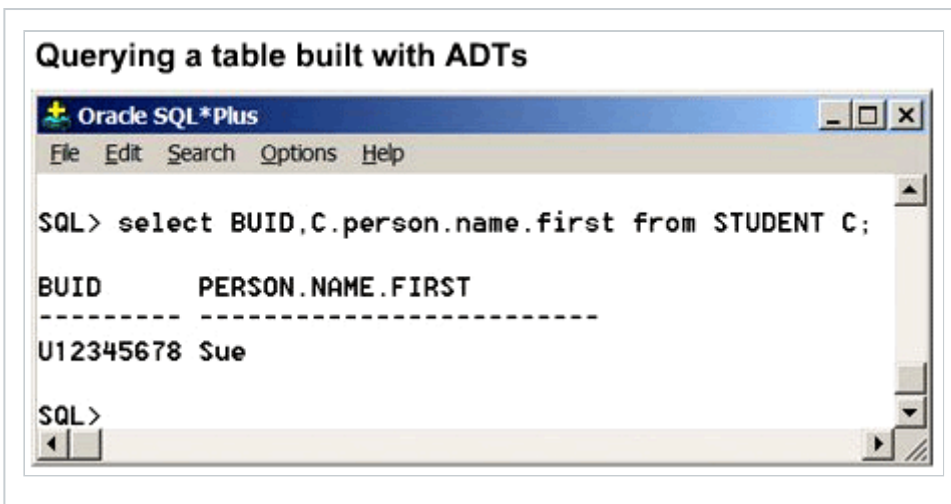
This is true. The purpose of the constructor is to assign the passed variables to the type elements in the ADT and to create an instance of the ADT object.

The values list looks a little weird, because it includes *constructors* for the PERSON_TY, NAME_TY, and ADDRESS_TY ADTs. Constructors are conceptual functions that take as arguments the values of the components of the ADT and construct and return the ADT to be inserted into the row. This may be a bit surprising. We used the ADTs to build the STUDENT table, and the describe command told us that Oracle had composed the table of the base types that we had composed in the nested ADTs. What we have now learned is that the ADTs must also be used in inserting rows into the table that was built with ADTs. We will discover that the ADT is part of all DML operations on the table.

Let's try querying the Student table to see what happens:



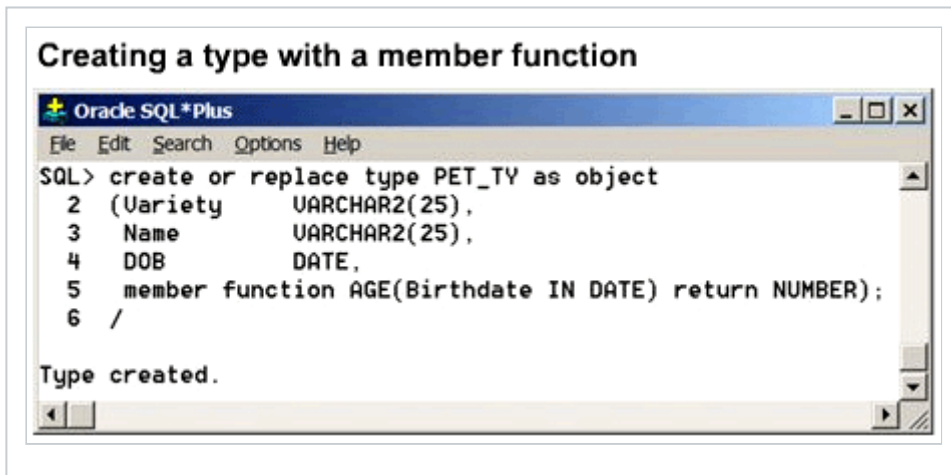
What we see is that the ADT names `PERSON_TY`, `NAME_TY` and `ADDRESS_TY` are part of the result set. This could be hard to work with if we wanted to work with this result set as if it came from an ordinary relational table. It seems that there must be a way to make this result set look more relational, and there is:



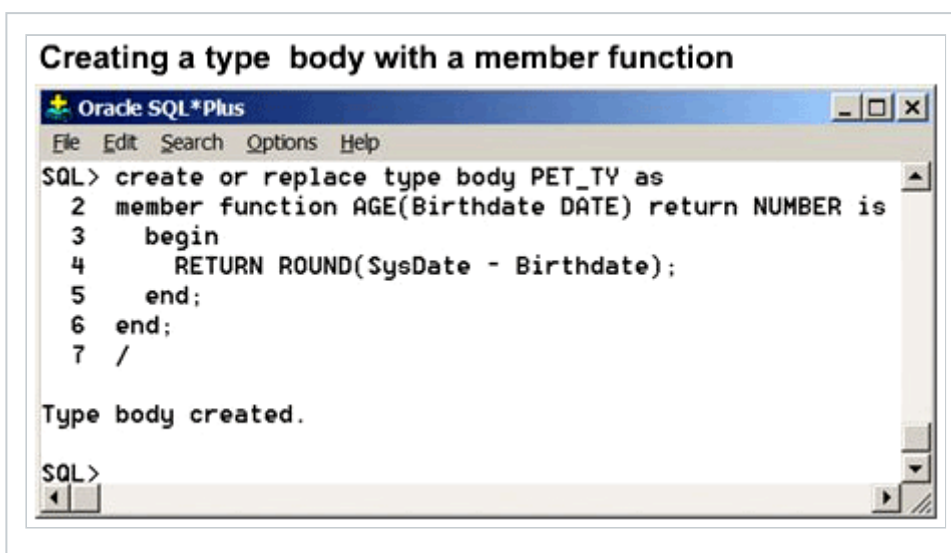
What we have done here is asked for the values of two columns from the `STUDENT` table—the `BUID` column which is part of the table itself, as a base data element, and the first name component of the `NAME_TY` ADT of the `PERSON_TY` ADT. Note that the result set included just the requested columns, with no ADT names. Note also that we refer to the ADTs by their names within the context of the `STUDENT` table. That is, we refer to `person.name.first` and not to `PERSON_TY.NAME_TY.first` or something similar. This is consistent with the way that we refer to columns within a table by their column names and not by the names of their data types. It is done this way to distinguish different uses of the same abstract or base data type within a table. For example, a person could have both a `home_address` and an `office_address`, and both could be defined in terms of `ADDRESS_TY`, so we could not uniquely identify either address within the context of the table by the name of the ADT. That is why uses of ADTs are given unique names within the table or ADT where they are used in the same way that columns are given unique names within the context of a table. Note that we use “C” as a *correlation* variable in this query.

Object Classes

Oracle's ADTs can also have methods like object classes, which is why the syntax says "as object." Let's create a pet object type with a method to return the pet's age in days:

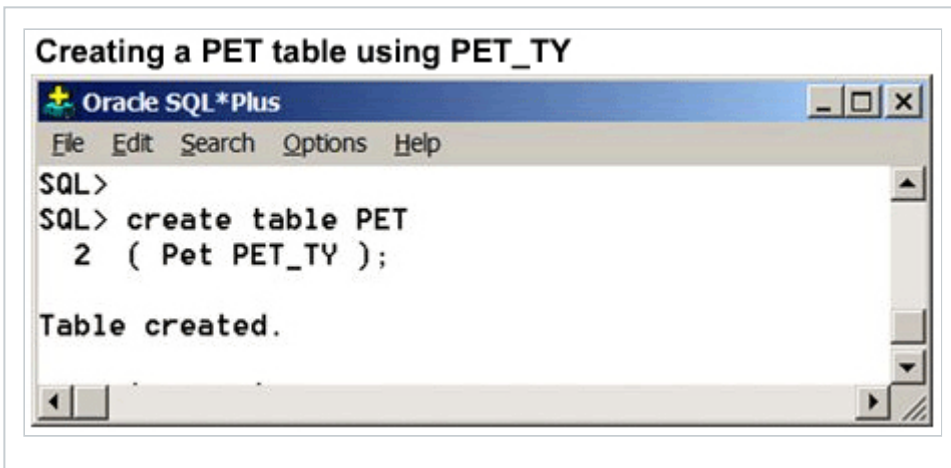


This statement only creates the type specification. There is no code for implementing the AGE method. If you recall this is how Oracle stored procedures work too. So now let's create the type body to go with this type specification...

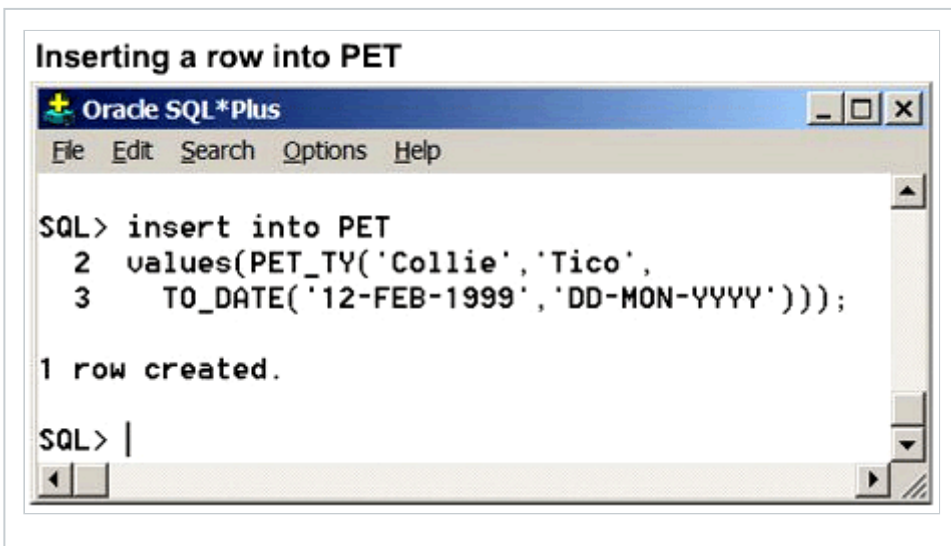


In Oracle when you subtract two values of type DATE such as the DOB and the built-in function SysDate, you get the difference in days, which makes sense, because Dates are in days. This is a PL/SQL function, but we could also have implemented the function in Java or C, though this is a little more complicated.

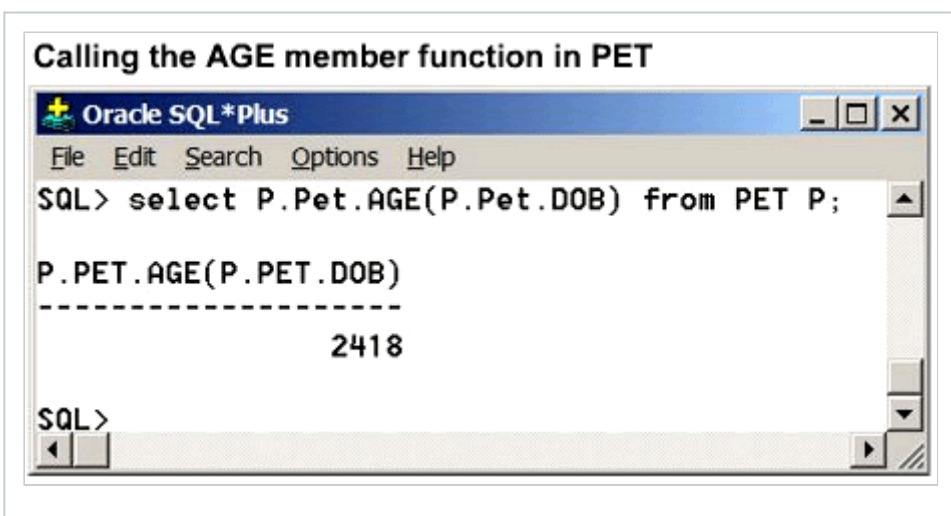
Next let's create a pet table using our new PET_TY so we can try out our new AGE method:



Now let's put a row in the new PET table.



Now let's invoke the age method to select the ages of the pets...



Notice that we used the correlation variable (also called an alias) P to tell Oracle that we want the Pet ADT from the PET table. There is only one table PET in the SELECT so we could conceivably have said something like select Pet.AGE(Pet.DOB) from PET, but Oracle requires that we tell it explicitly which table we are using with the correlation variable.

Notice that Tico is about $2418/365 = 6.6$ years old, which is about right. Notice that what we have done is created an AGE pseudo-column that we can use in queries. The old way to do this was to put in a real AGE column and sweat bullets trying to keep it up-to-date. With the AGE member function we don't need to keep an AGE column up-to-date, because it is recomputed correctly each time that we use it.

Test Yourself

Select all that are true of defining a method in an object type.

A method definition must be defined before it can be included as part of an object type.

This is true. If the method definition does not exist, the type will not be created.

Method parameters must be defined before it can be included as part of an object type.

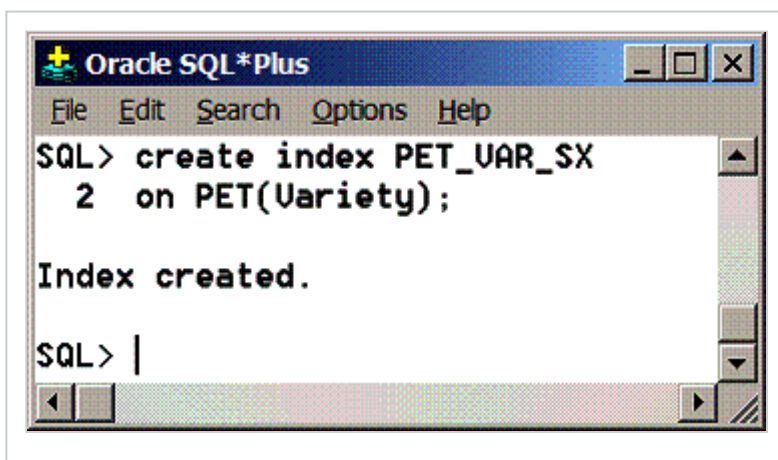
This is false. When inserting records, there is no need to include parameters for the method.

Selecting an object with a method definition requires both a correlation variable and fully qualified column names.

This is false. When executing the method, parameters need the correlation variable and also the name of the object (P.Pet.DOB), but not the fully-qualified name, which would include the name of the object type (PET_TY, in this case.)

Using a B-tree Index

Suppose that for some reason we have millions of rows in the PET table, and we frequently query for particular varieties and do ORDER BY Variety, so we need a B-tree index on the Variety column of the PET table. We do this as follows:



```
Oracle SQL*Plus
File Edit Search Options Help
SQL> create index PET_VAR_SX
      2  on PET(Variety);

Index created.

SQL> |
```

Note that we didn't use a correlation variable but we did use the name Pet that we gave to our use of PET_TY within the PET table.

Notice also that we are creating the index on the PET table, not on PET_TY. You will need to remember to create appropriate indexes on every column or columns in every table that uses ADTs and needs index(es). We covered indexing in more detail in Week 2 when we discussed database performance tuning.

In Oracle ADTs have no role when the queries are being executed. Oracle constructs tables defined using ADTs just like any other relational tables. This means that you get the full flexibility and all of the infrastructure of Oracle's powerful relational engine plus the convenience and schema control that comes with ADTs. The main cost is remembering that you need to use the name that you gave the use of the ADTs when querying the table. Oracle does keep the ADT definition for use in processing the incoming SQL, and Oracle won't let you drop an ADT if anything exists that used the ADT in its creation. This is an example of the performance tuning that went into Oracle's ADT design. For example if an ADT were a separate table there would be a performance penalty for the extra I/O to bring in the database blocks, and if ADTs were in the middle of query processing there would be at least additional CPU overhead. In Oracle's design the only overhead is our remembering when we write the SQL for a table that we used ADTs when we created the table

Oracle's ADT design implements object class composition (e.g., a person *has a* name) but not object class inheritance (a canary is *a kind of* pet). The technical reason for this is that it is difficult to implement inheritance within the context of a relational DBMS without incurring performance overheads. For example, if we implemented the class PET as a table and implement CANARY as a table with generalization link to PET, then many queries would probably need to fetch rows from both tables, which would increase the I/O.

Note: While the ADT implementation does not support inheritance, the full object-relational implementation does. To specify a type as a superclass and to give other objects permission to inherit from that type, use the clause "NOT FINAL". The "UNDER" clause associates the subclass with the superclass.

For example:

```

/*
 * super_class779 is a superclass.
 */
CREATE OR REPLACE TYPE super_class779 AS OBJECT
    (n NUMBER) NOT FINAL;
/*
 * sub_class779 is a subclass of super_class779
 */ CREATE OR REPLACE TYPE sub_class779 UNDER super_class779
    (n2 NUMBER);
/
/

```

Test Yourself

Select all that are true of ADTs.

A correlation variable is required for all queries involving ADTs.

This is false. While a correlation variable makes the query easier to read, it is possible to use fully qualified names instead.

Oracle will not drop an ADT if it is used in the definition of a table or other database object.

This is true. The DROP command will fail if the ADT is includedis used in other objects.

Including a method as part of an ADT type requires a separate CREATE TYPE METHOD block to define the method.

This is True.

Collectors: Nested Tables and VARRAYs

The material in this section is covered in Loney, Chapter 39.

The first normal form requirements of purely relational databases prohibit repeated groups or collections as part of a purely relational table. For example, we can't design a purely relational table that represents a standard invoice, because the line items are a repeated group, so we need to create something like a line item table which has foreign key references to the invoice table. This works, but it is more complex than necessary, and it creates a table to represent line items, which cannot exist apart from the invoice of which they are a part. This makes the database a less natural representation of an invoice, and it slows down the database, because rows have to be fetched from at least two tables to access the most important parts of the invoice.

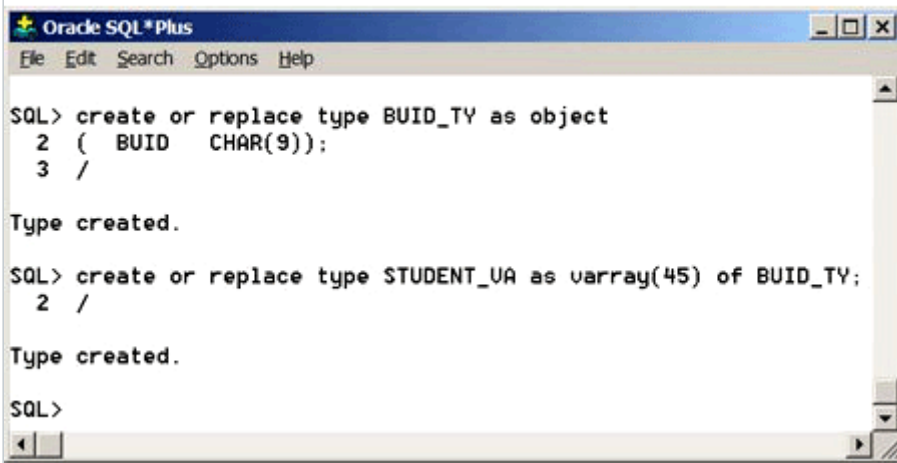
DBMS vendors have developed ways to overcome this limitation of purely relational databases by implementing data type extensions to represent collections. Most of these extensions, including those in Oracle, include a way to include collections as part of the same physical storage as the rest of the row, and a way to represent (typically larger) collections as separately stored database objects. In Oracle a collection stored with the row is called a VARRAY, and a collection with separate storage is called a NESTED TABLE.

VARRAYs

In Oracle a VARRAY is an integer indexed variable length array collection. To create a VARRAY we must first create an abstract data type that represents the elements of the array. These elements of the array can be any ADT, and they can include other ADTs in its definition. Thus the elements of an ARRAY can be like the rows of a relational object table. These are the same Oracle ADTs as we studied in the previous section.

Once you have created an ADT to represent an element of the array you then need to create a VARRAY type ADT to represent the collection itself. The following SQLPlus screen shot illustrates creation of an ADT to represent a Boston University student ID (BUID) and the use of that BUID_TY ADT to create a varying array ADT with elements of type BUID_TY.

Creating an ADT and using it to create a VARRAY type



```

Oracle SQL*Plus
File Edit Search Options Help

SQL> create or replace type BUID_TY as object
2  ( BUID  CHAR(9));
3  /

Type created.

SQL> create or replace type STUDENT_VA as varray(45) of BUID_TY;
2  /

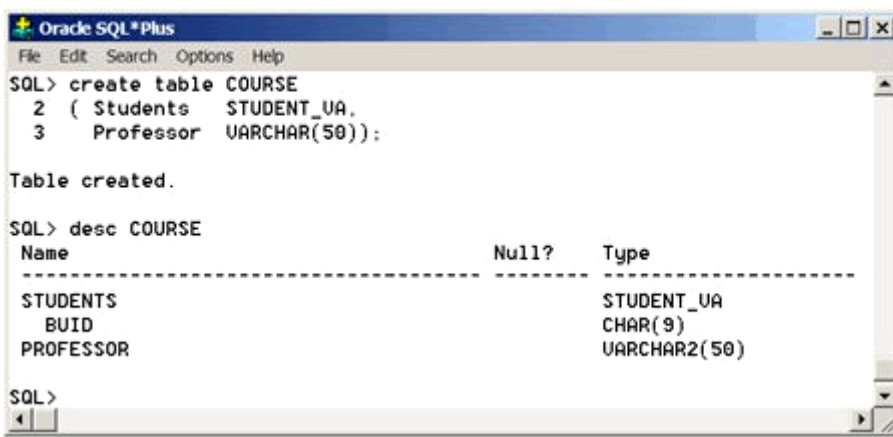
Type created.

SQL>

```

Now that we have created a STUDENT_VA varying array data type we can use it to create a COURSE table that includes a STUDENT_VA as a column:

Create a table with a VARRAY



```

Oracle SQL*Plus
File Edit Search Options Help

SQL> create table COURSE
2  ( Students  STUDENT_VA,
3  Professor  VARCHAR(50));

Table created.

SQL> desc COURSE

```

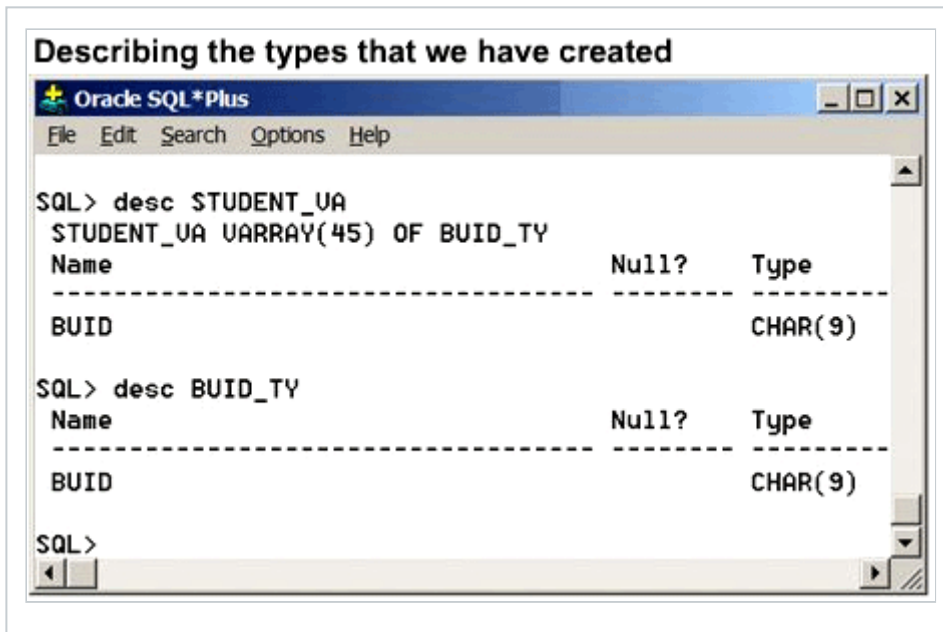
Name	Null?	Type
STUDENTS		STUDENT_VA
BUID		CHAR(9)
PROFESSOR		VARCHAR2(50)

```

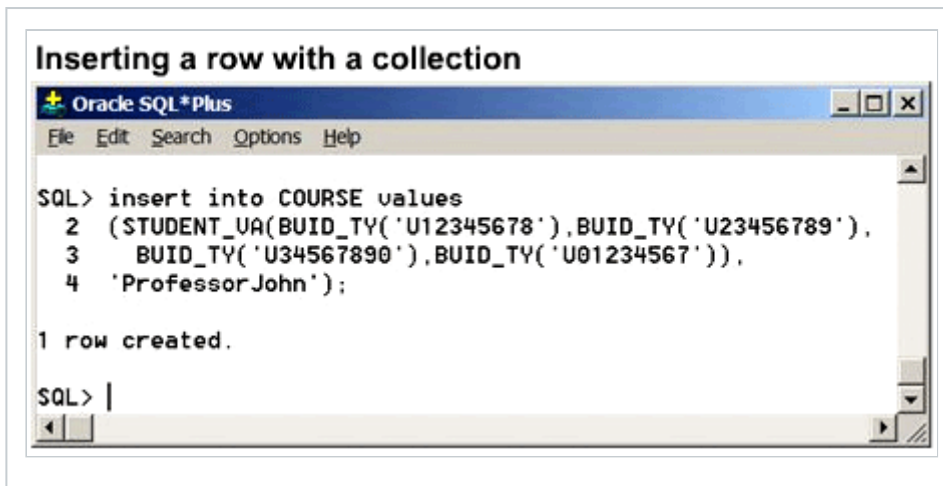
SQL>

```

Note that the DESCRIBE DEPTH is set to at least two, because when we describe COURSE we are told not only that the Students column is of type STUDENT_VA, but also that STUDENT_VA is made of type BUID, which is a char(9). In the next figure we describe the underlying types that we created:



We next insert a row into the COURSE table that we just created:



The values list includes what looks like a function call to the ADT STUDENT_VA. What is actually being called is a *constructor* for the STUDENT_VA abstract data type. The constructor STUDENT_VA has the same name as the abstract data type. This is the same way that it is done in object-oriented programming languages such as C++ and Java. Oracle creates the constructor for an ADT when you create the ADT. Note that the BUID constructor is called four times and the results are passed as arguments to the STUDENT_ID constructor. Each call to the BUID_TY constructor creates an object of type BUID, and the STUDENT_VA constructor assembles those four objects of type BUID_TY into a VARRAY of type STUDENT_VA to form the *Students* column of the COURSE table. Note that the values list also includes 'Professor John' as a nice familiar ordinary relational value for the *Professor* column.

Test Yourself

Select all that are true about VARRAYs.

A VARRAY makes it possible to model a multivalued relationship within an object.

This is true. A VARRAY will allow multiple values for each row of the object type.

If a type includes attributes with base types along with a VARRAY, the base type values will be repeated for every value included in the VARRAY. In other words, in the STUDENT_TY example above, the Professor's name would be stored 4 times for each of the BUIDs inserted into the array.

This is false. Base types are stored only once for each row.

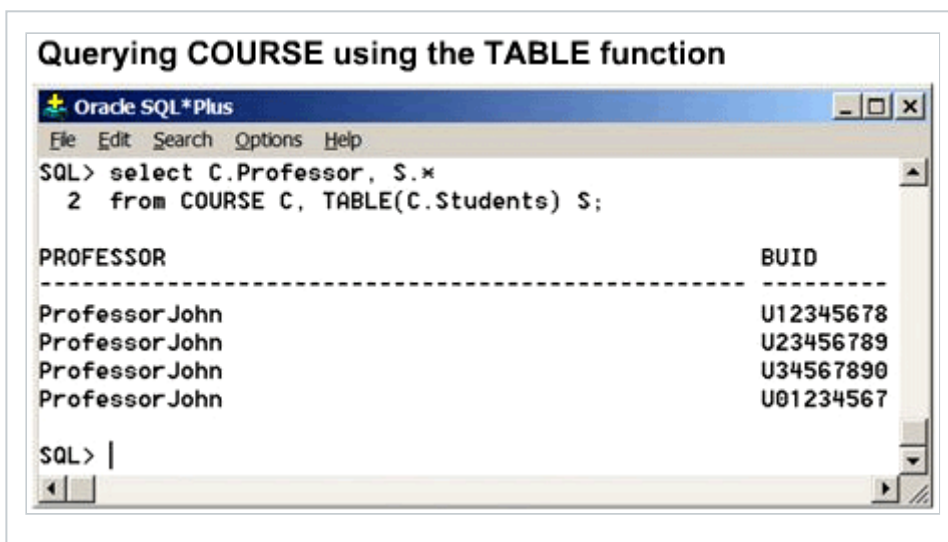
To create values for a VARRAY, you need to call the constructor method of the ADT.

This is True.

Querying a Table with a Varying Array

Let's reflect on what we have done. We have created a COURSE relational table that includes an ordinary relational *Professor* column and a *Students* column that is a collection of type STUDENT_TY. In a real-world example the COURSE table would probably have a few more columns, and the Professor column would probably be something like a BUID_TY too, but I wanted to keep this first example as simple as possible, because there is enough new material to understand with the two types and constructors.

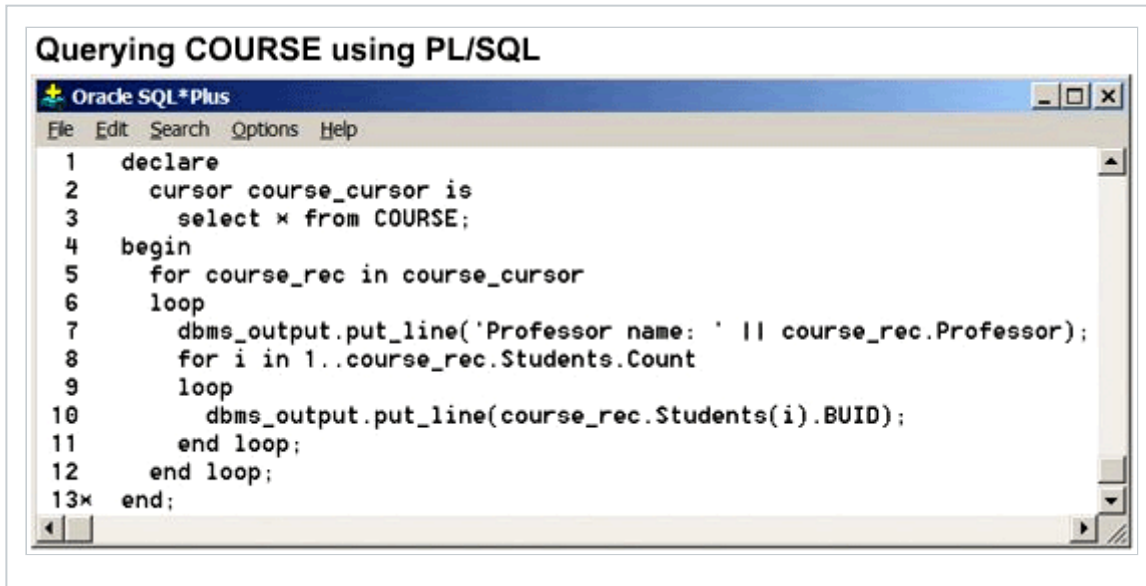
Querying a Table with a Varying Array—We next query the Students table. This is obviously going to be more complicated than querying an ordinary table, because the Students column doesn't have just one value the way it would have in a purely relational table. I will just show you how it is done in an example, and then explain what the parts of the example are doing.



The first things that we notice is that there are two *correlation variables* (also called *aliases*), C and S. C stands for the *COURSE* table and S stands for the *Students* column in the *COURSE* table. Both aliases are required. The TABLE function tells Oracle to convert each cell in the Students column to a table with one row for each element in the VARRAY. The resulting table is then joined with the COURSE table in a relational query, to retrieve both the Professor and the Students. What this is doing conceptually is extracting the Students VARRAY from the COURSE table, turning it into a table, and then joining that table with the COURSE table to extract the

Professor column. This is nifty, even if the syntax is at first unfamiliar. The TABLE function is very useful for querying collections using ordinary SQL.

Recall that database programming languages such as PL/SQL are good for processing tables a row at a time, so it is not surprising that they are also useful for processing varying arrays and other collections. The following example illustrates a simple PL/SQL procedure for printing the contents of our COURSE table.



Let's walk through this from top to bottom. At line 2 in the DECLARE section we declare a cursor that selects everything from the COURSE table. At line 5 the cursor **FOR** loop for course_rec **IN** course_cursor then implicitly declares course_rec as a record of the type of row in the COURSE table (this is the same as if we declared course_rec as COURSE%ROWTYPE). The for loop then walks down the rows of COURSE, storing each row in turn in course_rec. In line 7 the stored procedure put_line in the built-in package dbms_output then writes out the literal string "Professor name:" concatenated with the value of the Professor column in the current record of COURSE.

At line 8 we then come to the nested for loop that walks through the elements in the Students VARRAY column in the COURSE table. With the first iteration the value of the implicitly declared variable **i** is one, and it is incremented with each successive loop until it reaches the value of course_rec.Students.Count. Of course course_rec holds the current row of COURSE from the outer for loop started in line 5. Course_rec.Students refers to the current Students VARRAY, and course_rec.Students.Count invokes the built-in function which returns the number of elements in that particular Students VARRAY.

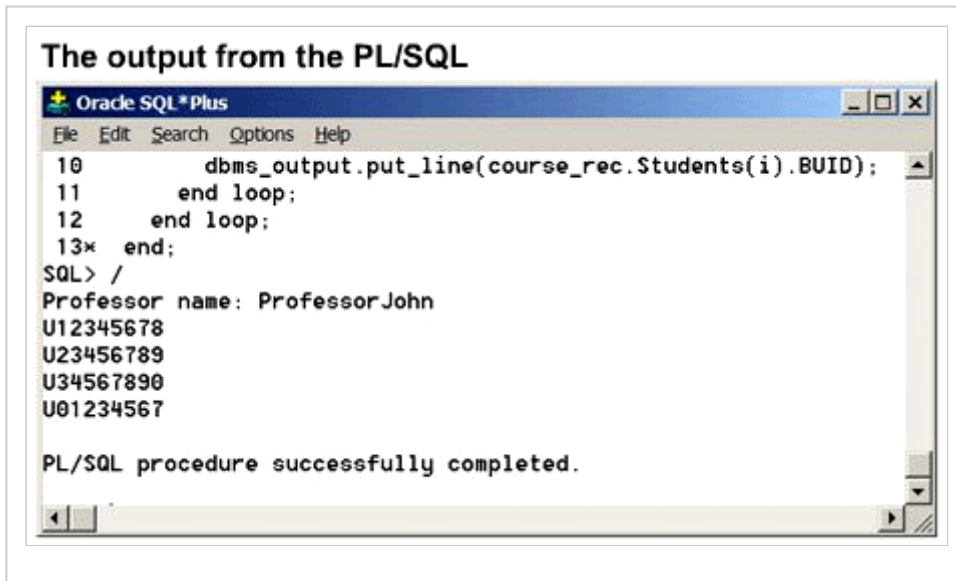
In line 10 we output the *i*th element of the current Students VARRAY. Then in line 11 we end the loop begun at line 8, and at line 12 we end the outer cursor for loop begun at line 5; see the tip below.

Cursor for loops open and close the cursors

There is no need to explicitly open or close the cursor with the cursor for loop syntax at line 5, because Oracle does this for us with this syntax. Cursor for loops such as this are simpler and safer, because we can't forget to close the cursor.

Executing VARRAYs

We next type a slash and a return to send the PL/SQL block to Oracle (see detail below). The expected result then appears in the SQLPlus window.



How to Execute SQL Requests

The forward slash followed by a return signals SQLPlus to execute the current contents of the buffer, which in this case is the lines above. In TOAD you would click the green execute triangle to do the same thing. Other Oracle client tools have different ways that you use to tell the client to send the SQL or PL/SQL to Oracle.

Where results appear in GUI clients

In graphical user interface database client tools such as TOAD the results appear in a separate pane, usually at the bottom.

VARRAYs are useful when the size of the collection is naturally bounded, such as the students in a course or the players on a baseball team. If the collection gets very large (think of more than a kilobyte or so) then it can begin to cause storage difficulties and inefficiencies. For example, suppose that we are looking through a large COURSE table to count the courses taught by a particular professor. If the Students arrays were large, then this would slow this search, because all of the Students' data would need to be fetched along with the Professor column.

Another limitation of VARRAYs is that they can't be indexed. Thus if there were many elements in a VARRAY it would slow processing, because the only way to search a VARRAY is to scan all of the elements. Again, the advantage of using a VARRAY is that the VARRAY data is retrieved along with the rest of the row, which can reduce the I/O and improve performance relative to an implementation with the collection stored separately. For example, if we are designing a BASEBALL_TEAM table that stores the current data for baseball teams it would

be natural to use a VARRAY for the Players collection column, because there are no more than a few dozen players, and many accesses to the BASEBALL_TEAM table would be to retrieve data about the players, so it is efficient to store the player data along with the rest of the baseball team data in the database rows.

Test Yourself

Select all that are true about VARRAYs.

VARRAYs of ordinary size reduce I/O because related items are stored together in a row.

This is true. VARRAYs reduce disk accesses which can help with performance.

VARRAYs can be indexed with B-tree indexes.

This is false. VARRAYs cannot be indexed, so the only way to search a VARRAY is to scan all the elements of the array.

VARRAYs are useful for storing large collections of data.

This is false. Large numbers of elements cause storage and performance issues.

Nested Tables

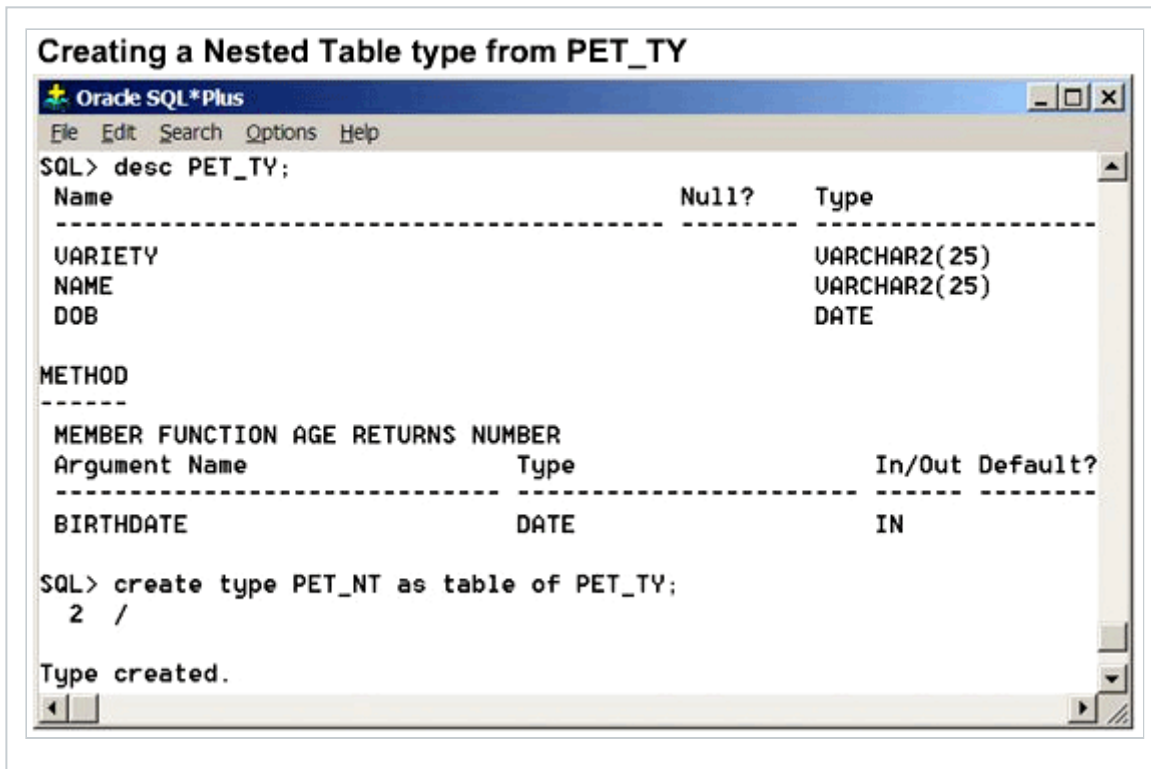
Nested tables are the complementary way of implementing collections in Oracle. Nested tables are stored separately from the rest of the row of the table. Nested tables are real tables, and you can create indexes or triggers on them. In the following examples we will create tables that include collections of pets that are implemented as nested tables. The following example reviews the PET_TY ADT that we created earlier in this lecture, by describing it, and then creates a new ADT that represents the type of a nested table consisting of the PET_TY ADT.

LOBS are also stored outside the row

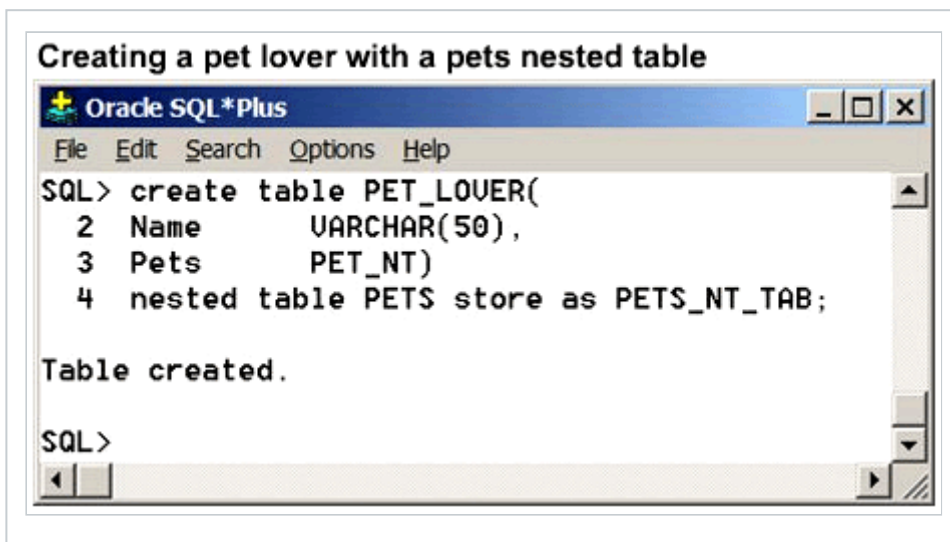
LOBS are stored separately from their tables, for the same reasons that nested tables are stored separately, and the same guidelines for the sizes and storage parameters apply to both LOBS and nested tables.

PET_NT isn't a table

The PET_NT nested table abstract data type is not a nested table, but only the abstraction of a PET_TY nested table. To create an actual PET_NT nested table we have to use PET_NT in creating a real physical nested table.



Now that we have an abstract data type for a PET_TY nested table we can use it in creating a real PET_LOVER table:



Notice that each pet lover has a name and a collection of pets, which are stored in a nested table called *Pets*. Note that the name of the nested table as a collection column within the context of the PET_LOVER table is *Pets*. The full name of the *Pets* nested table is owner.PET_LOVER.Pets. In line 4 we are giving this nested table the name PETS_NT_TAB in the larger context of the schema.

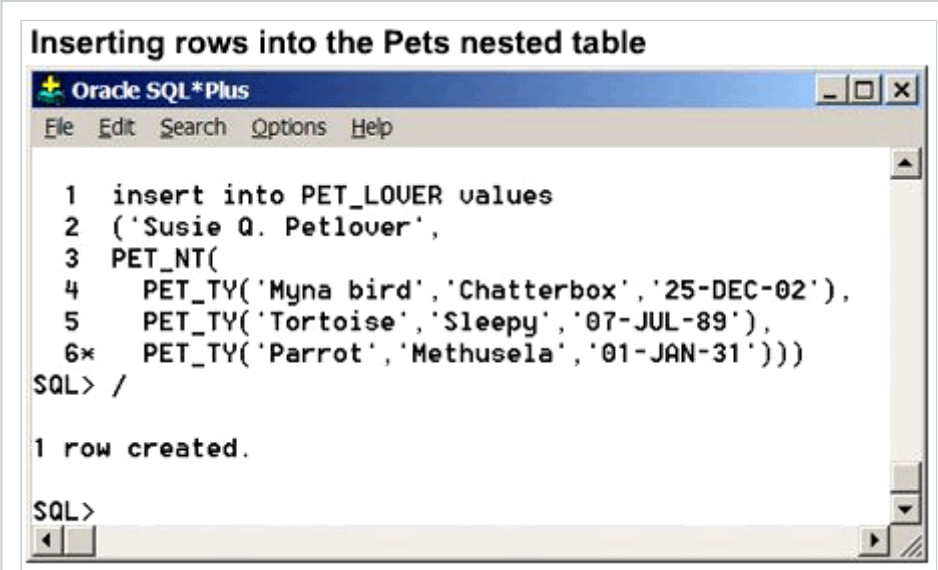
Naming Conventions

This is a good time to introduce common naming conventions for collection types. They are summarized in the following:

- Use plurals for the column names of collections (e.g., *Pets*). This identifies the logical column as a collection, either a VARRAY or a nested table.
- Name types *X_TY*
- Name VARRAYs *X_VA*
- Name nested tables *X_NT*
- Name NTs in storage clauses *X_NT_TAB*

Inserting Data into a Nested Table

The sample below illustrates how we insert a row into the PET_LOVER table, including giving the pet lover a collection of pets:



The screenshot shows the Oracle SQL*Plus interface with a window titled "Inserting rows into the Pets nested table". The SQL prompt is followed by an insert statement for the PET_LOVER table. The statement uses the PET_NT nested table type and the PET_TY constructor, which takes three arguments: a pet name, a pet type, and a date. The output shows "1 row created." and the SQL prompt is ready for the next command.

```
1 insert into PET_LOVER values
2 ('Susie Q. Petlover',
3  PET_NT(
4    PET_TY('Myna bird','Chatterbox','25-DEC-02'),
5    PET_TY('Tortoise','Sleepy','07-JUL-89'),
6    PET_TY('Parrot','Methusela','01-JAN-31'))))
SQL> /

1 row created.

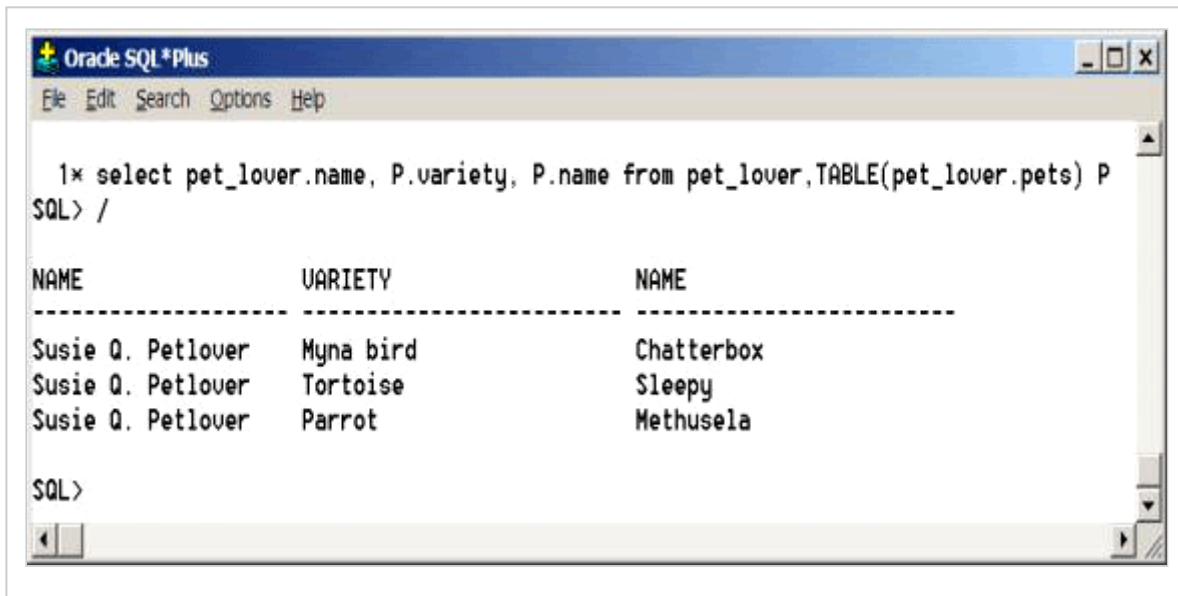
SQL>
```

This is the same syntax as used to insert rows into a table that contains a VARRAY, which is nice, because then we can change the implementation of the nested collections without changing the applications SQL that operates on the tables that include collection type columns. Also notice that the PET_TY is a little more complicated, having three columns, so the PET_TY constructor takes three arguments.

The Explicit Column List

You may notice that this isn't quite industrial-strength SQL, because there should be a column list after the name of the table PET_LOVER above. Recall that the reason for including the explicit column list in production code is so that if an alteration of the table changes the column order the SQL will still work correctly.

Now let's query the data that we inserted into PET_LOVER.



Note that we used the TABLE function, as we did in querying a table with a VARRAY. Note the use of the alias P to represent the result of the TABLE function call.

Note

We could have used a correlation variable to represent the PET_LOVER table, but instead we have just repeated the name of the pet_lover table in the clause pet_lover.name in the beginning of the query.

Inserting Rows into Nested Tables

Next we will insert a new row into the Pets nested table without inserting a new row in the PET_LOVER table. What we will do is give Susie a new collie named "Tico."



Remember that there is a whole Pets table for each row in PET_LOVER, so to insert a row into Susie's Pets nested table we have to tell Oracle which row (which pet owner, in this case Susie) we want to add the pet to. We do this with the clause:

```
SELECT Pets FROM pet_lover WHERE Name='Susie Q. Petlover'
```

Notice that the type returned by this query is PET_NT, so the query is returning a nested table. The TABLE function converts this nested table into a table that can be used in the ordinary SQL insert statement. By now you should not be surprised to see the call to the PET_TY constructor, which builds a row to be inserted into Susie's Pets nested table.

Now let's query PET_LOVER to see what we have inserted:



Note that Susie's collie Tico is now in her Pets table. Note that this is an ordinary result set that we can use in all of the usual ways. For example, we could use this as a subquery. Note that we have again provided an alias P for the nested table, and used it in the select list to reference the variety and names of the pets. Please note that in this table we see only one pet lover, Susie. If we had inserted more pet lovers we would see an additional row for each pet for each pet lover.

Choosing between VARRAYs, Nested Tables, and Non-nested Tables

In the prerequisite class you learned how to create and use ordinary relational tables to represent and store collections. You have now learned how to create and use VARRAYs and nested tables for collections that are always associated with some other object. We now discuss how we decide when to use which of these three ways of handling collections in databases such as Oracle and DB2. The basic criteria for choosing are summarized in the following:

- VARRAYs can't be indexed, so they don't scale well.
- VARRAYs have a predefined maximum array length, while the number of rows in a nested table is unlimited.
- Nested tables can be indexed.
- Nested tables can have methods.
- Ordinary relational tables are more easily related to other tables.
- VARRAYs are stored "in line" with the row, while nested tables are stored "out of line" with their own storage clause.
- VARRAYs and nested tables should not be used when the objects in the nested collection may survive the object of which they are a part. For example, a collection type would not be appropriate for representing a vehicle owner and their collection of vehicles, because the vehicles exist independently of their current owner.

Test Yourself

What are some of the differences between nested tables and VARRAYs (Choose all that apply).

A VARRAY has a predefined maximum number of elements.

This is true. The the size of an array is bounded when it is declared, but the size of a nested table is effectively unbounded.

It is possible to create triggers and functions on VARRAYs.

This is false. A nested table is a real table, so triggers and functions can be created on nested tables. VARRAYs are array collections, so they can't be indexed, and it is not possible to create triggers and functions on them.

It is possible to delete certain objects in a nested table without deleting the entire collection of objects.

This is true. With a VARRAY, an element can't be deleted, though they can be moved within the array, but with a nested table, it is possible to delete elements.

Large Objects

The material in this section can also be found in Loney Chapter 40.

Large objects, which are commonly called LOBs, are implemented in most modern DBMS. LOBs are used to store larger types of data such as audio, video or multimedia objects. LOBs are stored outside the row of the table in which they occur, and LOBs have separate storage clauses. We will explore Oracle's LOB implementation, which has been in production releases for many years.

LOBs are data types. They are used in defining columns of tables and in other places where data types are used. The following table summarizes the Oracle LOB data types.

LOB Type	What the name means	What the LOB stores
BLOB	Binary Large Object	Binary data such as a video
CLOB	Character Large Object	Large character data such as a novel
BFILE	Binary FILE	Read-only binary data such as a seismic record stored in a file outside the DB
NCLOB	National Character Large Object	Multi-byte UNICODE character set data such as Chinese text

The following figure illustrates how to create a table with a BLOB and a CLOB:



Storage Tip

In a production system for storing videos these would probably go in a different tablespace, perhaps on a different kind of storage, because videos are normally only read sequentially from the beginning.

Note that the first four lines look like a fairly normal CREATE TABLE statement that uses the BLOB and CLOB built-in Oracle data types. Then beginning on line 5 I specify special storage for the MPEG BLOB, which stores a video in the MPEG format. In line 6 I tell Oracle to put the videos in the USER tablespace.

Line 7 is a storage clause which provides the details of the storage parameters for the videos. There I tell Oracle to allocate the first extent at 100 kilobytes, and that each subsequent extent should be the same size, with no increase. In line 8 I tell Oracle that it should allocate 16KB for each operation on the MPEG LOB. The pctversion parameter tells Oracle to not overwrite older versions of the MPEG LOB until ten percent of the available LOB storage space has been used. Nochace tells Oracle not to cache the LOB, and nologging tells Oracle not to log changes to the LOB. Given the way videos are accessed caching and logging would only flush other data from the cache and clutter the logfiles with data that would never be used. I do not expect you to remember the details of these LOB storage parameters, but just to remember the kinds of things that they specify.

Large Objects, Continued

Next we will insert a row into this VIDEO table:

```
.
```

Here I provided the column list like a good database developer. The title is a VARCHAR so it is not surprising that the literal 'Starwars II' inserted into it just fine. The built-in functions EMPTY_BLOB and EMPTY_CLOB return a BLOB with nothing in it and a CLOB with nothing in it, respectively. The built-in functions EMPTY_NCHAR and EMPTY_BFILE perform similar functions.

Now let's update the row that we just inserted, by giving it a Description:

```
.
```

Notice that something wonderful happened. Oracle took the literal string 'This is the Starwars II description' and inserted it as the Description column, replacing the EMPTY_CLOB, without our doing anything special. The regular old SQL string operations work on CLOBs, just as if they were CHARs or VARCHARs. So now let's try querying the VIDEO table:

```
.
```

When we tried to select the MPEG BLOB from VIDEO we are told that SQLPlus doesn't know how to display a binary LOB. This is expected, because Oracle doesn't know that this is an MPEG, so that it should call an MPEG player to present the video. At the bottom we ask Oracle and SQLPlus to show us the title and description columns, and this works fine. The difference between BLOBs and the character LOB types is that the DBMS doesn't know anything about the internal structure of BLOBs, so any presentation or other operations on BLOBs are up to the application programs that use them.

Most operations on LOBS are performed using applications software. Support for many LOB operations is provided in Oracle's DBMS_LOB package. Similar support is provided for other DBMS that implement LOBs. A few of the functions in the DBMS_LOB package are summarized in the following table:

Function/Procedure Name	What it does
APPEND procedure	Appends one LOB to another
COMPARE function	Compares LOBs or parts of LOBs
COPY procedure	Copies all or parts of LOBs to other LOBs
CONVERTOCLOB procedure	Copies a CLOB or NCLOB to a BLOB, converting the character form to binary form
ERASE procedure	Deletes a LOB
LOADFROMFILE procedure	Loads a BFILE into an internal LOB where it can be operated on
READ procedure	Reads data from a LOB starting at a specified offset; this is how you get the LOB into your code
WRITE procedure	Writes data into a LOB

I do not expect you to remember the details of the DBMS_LOB package, but just remember the general kinds of things that you can do with this and similar LOB support software in other ORDBMS. Details on the DBMS_LOB package and how to use it from PL/SQL can be found in Loney Chapter 40.

Test Yourself

Select all that are true about LOB data types.

LOBs can be physically stored outside of the row and sometimes outside of the database.

This is true. Storage for a LOB is outside of the row or in a separate file

SQL operations performed on CLOBs and BLOBs work in the same way as those performed on other base data types.

This is false. Most operations that are performed as CLOBs are identical to those performed on base data types, but operations on BLOBs are different, because Oracle and other ORDBMS does not understand the format of the BLOB.

This concludes our study of the typical object-relational and LOB extensions found in modern commercial ORDBMS. We have emphasized the basic concepts such as Oracle's ADT implementation, because these have found their way into the ANSI and ISO standards, so other DBMS support them now or will within a few years. We have shown you a few of the details of Oracle-specific features such as the PL/SQL syntax, but have not emphasized them, because they are not part of the ANSI syntax, so a database other than Oracle will not support this syntax. However the other implementations will behave about the same way, with the main differences being slight differences in the syntax. Over the next decade I expect that the DBMS industry will adopt the ANSI database programming and object-relational extensions, so the same syntax and semantics (meaning) will apply across the main DBMS. One of the interesting possibilities is that Java will become the de facto portable database programming standard for applications that can tolerate the lower performance, because many DBMS support Java for database programming today.

Boston University Metropolitan College