4.1 Review of Object-Oriented Concepts

Before introducing object-oriented database models, let us review the major object-oriented concepts themselves. Object-oriented programming has been widely regarded as a tool for better program organization and, ultimately, more reliable software implementation. First popularized in the language Smalltalk, object-oriented programming received a big boost with the development of C++ and the migration to C++ of much software development that was formerly done in C. More recently, the language Java, suitable for sharing programs across the World Wide Web, has also focused attention on object-oriented programming.

The database world has likewise been attracted to the object-oriented paradigm, particularly for database design and for extending relational DBMS's with new features. In this section we shall review the ideas behind object orientation:

1. A powerful type system.
2. *Classes*, which are types associated with an *extent*, or set of *objects* belonging to the class. An essential feature of classes, as opposed to conventional data types, is that classes may include *methods*, which are procedures that are applicable to objects belonging to the class.
3. *Object Identity*, the idea that each object has a unique identity, independent of its value.
4. *Inheritance*, which is the organization of classes into hierarchies, where each class inherits the properties of the classes above it.

4.1.1 The Type System

An object-oriented programming language offers the user a rich collection of types. Starting with *atomic types*, such as integers, real numbers, booleans, and character strings, one may build new types by using *type constructors*. Typically, the type constructors let us build:

1. *Record structures*. Given a list of types $T_1, T_2, \ldots, T_n$ and a corresponding list of *field names* (called *instance variables* in Smalltalk) $f_1, f_2, \ldots, f_n$, one can construct a record type consisting of n components. The ith component has type T_i and is referred to by its field name f_i. Record structures are exactly what C or C++ calls "structs," and we shall frequently use that term in what follows.
2. *Collection types*. Given a type T, one can construct new types by applying a *collection operator* to type T. Different languages use different collection operators, but there are several common ones, including arrays, lists, and sets. Thus, if T were the atomic type integer, we might build the collection types "array of integers," "list of integers," or "set of integers."

3. *Reference types.* A reference to a type T is a type whose values are suitable for locating a value of the type T. In C or C++, a reference is a "pointer" to a value, that is, the virtual-memory address of the value pointed to.

Of course, record-structure and collection operators can be applied repeatedly to build ever more complex types. For instance, a bank might define a type that is a record structure with a first component named `customer` of type string and whose second component is of type set-of-integers and is named `accounts`. Such a type is suitable for associating bank customers with the set of their accounts.

4.1.2 Classes and Objects

A *class* consists of a type and possibly one or more functions or procedures (called *methods*; see below) that can be executed on objects of that class. The objects of a class are either values of that type (called *immutable objects*) or variables whose value is of that type (called *mutable objects*). For example, if we define a class C whose type is "set of integers," then $\{2, 5, 7\}$ is an immutable object of class C, while variable s could be declared to be a mutable object of class C and assigned a value such as $\{2, 5, 7\}$.

4.1.3 Object Identity

Objects are assumed to have an *object identity* (OID). No two objects can have the same OID, and no object has two different OID's. Object identity has some interesting effects on how we model data. For instance, it is essential that an entity set have a key formed from values of attributes possessed by it or a related entity set (in the case of weak entity sets). However, within a class, we assume we can distinguish two objects whose attributes all have identical values, because the OID's of the two objects are guaranteed to be different.

4.1.4 Methods

Associated with a class there are usually certain functions, often called *methods*. A method for a class C has at least one argument that is an object of class C; it may have other arguments of any class, including C. For example, associated with a class whose type is "set of integers," we might have methods to sum the elements of a given set, to take the union of two sets, or to return a boolean indicating whether or not the set is empty.

In some situations, classes are referred to as "abstract data types," meaning that they *encapsulate*, or restrict access to objects of the class so that only the methods defined for the class can modify objects of the class directly. This restriction assures that the objects of the class cannot be changed in ways that were not anticipated by the designer of the class. Encapsulation is regarded as one of the key tools for reliable software development.

4.1.5 Class Hierarchies

It is possible to declare one class C to be a *subclass* of another class D. If so, then class C *inherits* all the properties of class D, including the type of D and any functions defined for class D. However, C may also have additional properties. For example, new methods may be defined for objects of class C, and these methods may be either in addition to or in place of methods of D. It may even be possible to extend the type of D in certain ways. In particular, if the type of D is a record-structure type, then we can add new fields to this type that are present only in objects of type C.

Example 4.1: Consider a class of bank account objects. We might describe the type for this class informally as:

```
CLASS Account = {accountNo: integer;
                 balance: real;
                 owner: REF Customer;
                }
```

That is, the type for the `Account` class is a record structure with three fields: an integer account number, a real-number balance, and an owner that is a reference to an object of class `Customer` (another class that we'd need for a banking database, but whose type we have not introduced here).

We could also define some methods for the class. For example, we might have a method

```
deposit(a: Account, m: real)
```

that increases the `balance` for `Account` object a by amount m.

Finally, we might wish to have several subclasses of the `Account` subclass. For instance, a time-deposit account could have an additional field `dueDate`, the date at which the account balance may be withdrawn by the owner. There might also be an additional method for the subclass `TimeDeposit`

```
penalty(a: TimeDeposit)
```

that takes an account a belonging to the subclass `TimeDeposit` and calculates the penalty for early withdrawal, as a function of the `dueDate` field in object a and the current date; the latter would be obtainable from the system on which the method is run. □

9.1 Introduction to OQL

OQL, the *Object Query Language*, gives us an SQL-like notation for expressing queries. It is intended that OQL will be used as an extension to some object-oriented *host* language, such as C++, Smalltalk, or Java. Objects will be manipulated both by OQL queries and by the conventional statements of the host language. The ability to mix host-language statements and OQL queries without explicitly transferring values between the two languages is an advance over the way SQL is embedded into a host language, as was discussed in Section ??.

9.1.1 An Object-Oriented Movie Example

In order to illustrate the dictions of OQL, we need a running example. It will involve the familiar classes `Movie`, `Star`, and `Studio`. We shall use the definitions of `Movie`, `Star`, and `Studio` from Fig. ??, augmenting them with key and extent declarations. Only `Movie` has methods, gathered from Fig. ??. The complete example schema is in Fig. 9.1.

9.1.2 Path Expressions

We access components of objects and structures using a dot notation that is similar to the dot used in C and also related to the dot used in SQL. The general rule is as follows. If a denotes an object belonging to class C, and p is some property of the class — either an attribute, relationship, or method of the class — then $a.p$ denotes the result of "applying" p to a. That is:

1. If p is an attribute, then $a.p$ is the value of that attribute in object a.
2. If p is a relationship, then $a.p$ is the object or collection of objects related to a by relationship p.
3. If p is a method (perhaps with parameters), then $a.p(\cdots)$ is the result of applying p to a.

Example 9.2: Let `myMovie` denote an object of type `Movie`. Then:

- The value of `myMovie.length` is the length of the movie, that is, the value of the `length` attribute for the `Movie` object denoted by `myMovie`.
- The value of `myMovie.lengthInHours()` is a real number, the length of the movie in hours, computed by applying the method `lengthInHours` to object `myMovie`.
- The value of `myMovie.stars` is the set of `Star` objects related to the movie `myMovie` by the relationship `stars`.

```
class Movie
    (extent Movies key (title, year))
{
   attribute string title;
   attribute integer year;
   attribute integer length;
   attribute enum Film {color,blackAndWhite} filmType;
   relationship Set<Star> stars
                inverse Star::starredIn;
   relationship Studio ownedBy
                inverse Studio::owns;
   float lengthInHours() raises(noLengthFound);
   void starNames(out Set<String>);
   void otherMovies(in Star, out Set<Movie>)
                raises(noSuchStar);
};


class Star
    (extent Stars key name)
{
    attribute string name;
    attribute Struct Addr
        {string street, string city} address;
    relationship Set<Movie> starredIn
                 inverse Movie::stars;
};


class Studio
    (extent Studios key name)
{
    attribute string name;
    attribute string address;
    relationship Set<Movie> owns
                 inverse Movie::ownedBy;
};
```

Figure 9.1: Part of an object-oriented movie database

Arrows and Dots

OQL allows the arrow `->` as a synonym for the dot. This convention is partly in the spirit of C, where the dot and arrow both obtain components of a structure. However, in C, the arrow and dot operators have slightly different meanings; in OQL they are the same. In C, expression `a.f` expects `a` to be a structure, while `p->f` expects `p` to be a pointer to a structure. Both produce the value of the field `f` of that structure.

- Expression `myMovie.starNames(myStars)` returns no value (i.e., in C++ the type of this expression is `void`). As a side effect, however, it sets the value of the output variable `myStars` of the method `starNames` to be a set of strings; those strings are the names of the stars of the movie.

□

If it makes sense, we can form expressions with several dots. For example, if `myMovie` denotes a movie object, then `myMovie.ownedBy` denotes the `Studio` object that owns the movie, and `myMovie.ownedBy.name` denotes the string that is the name of that studio.

9.1.3 Select-From-Where Expressions in OQL

OQL permits us to write expressions using a select-from-where syntax similar to SQL's familiar query form. Here is an example asking for the year of the movie *Gone With the Wind*.

```
SELECT m.year
FROM Movies m
WHERE m.title = "Gone With the Wind"
```

Notice that, except for the double-quotes around the string constant, this query could be SQL rather than OQL.

In general, the OQL select-from-where expression consists of:

1. The keyword `SELECT` followed by a list of expressions.
2. The keyword `FROM` followed by a list of one or more variable declarations. A variable is declared by giving
   (a) An expression whose value has a collection type, e.g. a set or bag.
   (b) The optional keyword `AS`, and
   (c) The name of the variable.

Typically, the expression of (a) is the extent of some class, such as the extent `Movies` for class `Movie` in the example above. An extent is the analog of a relation in an SQL `FROM` clause. However, it is possible to use in a variable declaration any collection-producing expression, such as another select-from-where expression.

3. The keyword `WHERE` and a boolean-valued expression. This expression, like the expression following the `SELECT`, may only use as operands constants and those variables declared in the `FROM` clause. The comparison operators are like SQL's, except that `!=`, rather than `<>`, is used for "not equal to." The logical operators are `AND`, `OR`, and `NOT`, like SQL's.

The query produces a bag of objects. We compute this bag by considering all possible values of the variables in the `FROM` clause, in nested loops. If any combination of values for these variables satisfies the condition of the `WHERE` clause, then the object described by the `SELECT` clause is added to the bag that is the result of the select-from-where statement.

Example 9.3: Here is a more complex OQL query:

```
SELECT s.name
FROM Movies m, m.stars s
WHERE m.title = "Casablanca"
```

This query asks for the names of the stars of *Casablanca*. Notice the sequence of terms in the `FROM` clause. First we define `m` to be an arbitrary object in the class `Movie`, by saying `m` is in the extent of that class, which is `Movies`. Then, for each value of `m` we let `s` be a `Star` object in the set `m.stars` of stars of movie `m`. That is, we consider in two nested loops all pairs (`m`, `s`) such that `m` is a movie and `s` a star of that movie. The evaluation can be sketched as:

```
FOR each m in Movies DO
    FOR each s in m.stars DO
        IF m.title = "Casablanca" THEN
            add s.name to the output bag
```

The `WHERE` clause restricts our consideration to those pairs that have `m` equal to the `Movie` object whose title is *Casablanca*. Then, the `SELECT` clause produces the bag (which should be a set in this case) of all the name attributes of star objects `s` in the (`m`, `s`) pairs that satisfy the `WHERE` clause. These names are the names of the stars in the set m_c`.stars`, where m_c is the *Casablanca* movie object. □

Alternative Form of FROM Lists

In addition to the SQL-style elements of `FROM` clauses, where the collection is followed by a name for a typical element, OQL allows a completely equivalent, more logical, yet less SQL-ish form. We can give the typical element name, then the keyword `IN`, and finally the name of the collection. For instance,

```
FROM m IN Movies, s IN m.stars
```

is an equivalent `FROM` clause for the query in Example 9.3.

9.1.4 Modifying the Type of the Result

A query like Example 9.3 produces a bag of strings as a result. That is, OQL follows the SQL default of not eliminating duplicates in its answer unless directed to do so. However, we can force the result to be a set or a list if we wish.

- To make the result a set, use the keyword `DISTINCT` after `SELECT`, as in SQL.
- To make the result a list, add an `ORDER BY` clause at the end of the query, again as in SQL.

The following examples will illustrate the correct syntax.

Example 9.4: Let us ask for the names of the stars of Disney movies. The following query does the job, eliminating duplicate names in the situation where a star appeared in several Disney movies.

```
SELECT DISTINCT s.name
FROM Movies m, m.stars s
WHERE m.ownedBy.name = "Disney"
```

The strategy of this query is similar to that of Example 9.3. We again consider all pairs of a movie and a star of that movie in two nested loops as in Example 9.3. But now, the condition on that pair (`m`, `s`) is that "Disney" is the name of the studio whose `Studio` object is `m.ownedBy`. □

The `ORDER BY` clause in OQL is quite similar to the same clause in SQL. Keywords `ORDER BY` are followed by a list of expressions. The first of these expressions is evaluated for each object in the result of the query, and objects are ordered by this value. Ties, if any, are broken by the value of the second expression, then the third, and so on. By default, the order is ascending, but a choice of ascending or descending order can be indicated by the keyword `ASC` or `DESC`, respectively, following an attribute, as in SQL.

Example 9.5: Let us find the set of Disney movies, but let the result be a list of movies, ordered by length. If there are ties, let the movies of equal length be ordered alphabetically. The query is:

```
SELECT m
FROM Movies m
WHERE m.ownedBy.name = "Disney"
ORDER BY m.length, m.title
```

In the first three lines, we consider each `Movie` object m. If the name of the studio that owns this movie is "Disney," then the complete object m becomes a member of the output bag. The fourth line specifies that the objects m produced by the select-from-where query are to be ordered first by the value of `m.length` (i.e., the length of the movie) and then, if there are ties, by the value of `m.title` (i.e., the title of the movie). The value produced by this query is thus a list of `Movie` objects. □

9.1.5 Complex Output Types

The elements in the `SELECT` clause need not be simple variables. They can be any expression, including expressions built using type constructors. For example, we can apply the `Struct` type constructor to several expressions and get a select-from-where query that produces a set or bag of structures.

Example 9.6: Suppose we want the set of pairs of stars living at the same address. We can get this set with the query:

```
SELECT DISTINCT Struct(star1: s1, star2: s2)
FROM Stars s1, Stars s2
WHERE s1.address = s2.address AND s1.name < s2.name
```

That is, we consider all pairs of stars, `s1` and `s2`. The `WHERE` clause checks that they have the same address. It also checks that the name of the first star precedes the name of the second in alphabetic order, so we don't produce pairs consisting of the same star twice and we don't produce the same pair of stars in two different orders.

For every pair that passes the two tests, we produce a record structure. The type of this structure is a record with two fields, named `star1` and `star2`. The type of each field is the class `Star`, since that is the type of the variables `s1` and `s2` that provide values for the two fields. That is, formally, the type of the structure is

```
Struct{star1: Star, star2: Star}
```

The type of the result of the query is a set of these structures, that is:

```
Set<Struct{star1: Star, star2: Star}>
```

□

SELECT Lists of Length One Are Special

Notice that when a `SELECT` list has only a single expression, the type of the result is a collection of values of the type of that expression. However, if we have more than one expression in the `SELECT` list, there is an implicit stucture formed with components for each expression. Thus, even had we started the query of Example 9.6 with

```
SELECT DISTINCT star1:  s1, star2:  s2
```

the type of the result would be

```
Set<Struct{star1: Star, star2: Star}>
```

However, in Example 9.4, the type of the result is `Set<String>`, not `Set<Struct{name: String}>`.

9.1.6 Subqueries

We can use a select-from-where expression anywhere a collection is appropriate. We shall give one example: in the `FROM` clause. Several other examples of subquery use appear in Section 9.2.

In the `FROM` clause, we may use a subquery to form a collection. We then allow a variable representing a typical element of that collection to range over each member of the collection.

Example 9.7: Let us redo the query of Example 9.4, which asked for the stars of the movies made by Disney. First, the set of Disney movies could be obtained by the query, as was used in Example 9.5.

```
SELECT m
FROM Movies m
WHERE m.ownedBy.name = "Disney"
```

We can now use this query as a subquery to define the set over which a variable `d`, representing the Disney movies, can range.

```
SELECT DISTINCT s.name
FROM (SELECT m
      FROM Movies m
      WHERE m.ownedBy.name = "Disney") d,
     d.stars s
```

This expression of the query "Find the stars of Disney movies" is no more succinct than that of Example 9.4, and perhaps less so. However, it does

illustrate a new form of building queries available in OQL. In the query above, the `FROM` clause has two nested loops. In the first, the variable `d` ranges over all Disney movies, the result of the subquery in the `FROM` clause. In the second loop, nested within the first, the variable `s` ranges over all stars of the Disney movie `d`. Notice that no `WHERE` clause is needed in the outer query. □

9.1.7 Exercises for Section 9.1

Exercise 9.1.1: In Fig. 9.2 is an ODL description of our running products exercise. We have made each of the three types of products subclasses of the main `Product` class. The reader should observe that a type of a product can be obtained either from the attribute `type` or from the subclass to which it belongs. This arrangement is not an excellent design, since it allows for the possibility that, say, a PC object will have its `type` attribute equal to `"laptop"` or `"printer"`. However, the arrangement gives you some interesting options regarding how one expresses queries.

Because `type` is inherited by `Printer` from the superclass `Product`, we have had to rename the `type` attribute of `Printer` to be `printerType`. The latter attribute gives the process used by the printer (e.g., laser or inkjet), while `type` of `Product` will have values such as PC, laptop, or printer.

Add to the ODL code of Fig. 9.2 method signatures (see Section ??) appropriate for functions that do the following:

* a) Subtract x from the price of a product. Assume x is provided as an input parameter of the function.

* b) Return the speed of a product if the product is a PC or laptop and raise the exception `notComputer` if not.

c) Set the screen size of a laptop to a specified input value x.

! d) Given an input product p, determine whether the product q to which the method is applied has a higher speed and a lower price than p. Raise the exception `badInput` if p is not a product with a speed (i.e., neither a PC nor laptop) and the exception `noSpeed` if q is not a product with a speed.

Exercise 9.1.2: Using the ODL schema of Exercise 9.1.1 and Fig. 9.2, write the following queries in OQL:

* a) Find the model numbers of all products that are PC's with a price under $2000.

b) Find the model numbers of all the PC's with at least 128 megabytes of RAM.

*! c) Find the manufacturers that make at least two different models of laser printer.

```
class Product
    (extent Products
     key model)
 {
    attribute integer model;
    attribute string manufacturer;
    attribute string type;
    attribute real price;
 };


class PC extends Product
    (extent PCs)
 {
    attribute integer speed;
    attribute integer ram;
    attribute integer hd;
    attribute string rd;
 };


class Laptop extends Product
    (extent Laptops)
 {
    attribute integer speed;
    attribute integer ram;
    attribute integer hd;
    attribute real screen;
 };


class Printer extends Product
    (extent Printers)
 {
    attribute boolean color;
    attribute string printerType;
 };
```

Figure 9.2: Product schema in ODL

d) Find the set of pairs (r, h) such that some PC or laptop has r megabytes of RAM and h gigabytes of hard disk.

e) Create a list of the PC's (objects, not model numbers) in ascending order of processor speed.

! f) Create a list of the model numbers of the laptops with at least 64 megabytes of RAM, in descending order of screen size.

Exercise 9.1.3: In Fig. 9.3 is an ODL description of our running "battleships" database. Add the following method signatures:

a) Compute the firepower of a ship, that is, the number of guns times the cube of the bore.

b) Find the sister ships of a ship. Raise the exception `noSisters` if the ship is the only one of its class.

c) Given a battle b as a parameter, and applying the method to a ship s, find the ships sunk in the battle b, provided s participated in that battle. Raise the exception `didNotParticipate` if ship s did not fight in battle b.

d) Given a name and a year launched as parameters, add a ship of this name and year to the class to which the method is applied.

! **Exercise 9.1.4:** Repeat each part of Exercise 9.1.2 using at least one subquery in each of your queries.

Exercise 9.1.5: Using the ODL schema of Exercise 9.1.3 and Fig. 9.3, write the following queries in OQL:

a) Find the names of the classes of ships with at least nine guns.

b) Find the ships (objects, not ship names) with at least nine guns.

c) Find the names of the ships with a displacement under 30,000 tons. Make the result a list, ordered by earliest launch year first, and if there are ties, alphabetically by ship name.

d) Find the pairs of objects that are sister ships (i.e., ships of the same class). Note that the objects themselves are wanted, not the names of the ships.

! e) Find the names of the battles in which ships of at least two different countries were sunk.

!! f) Find the names of the battles in which no ship was listed as damaged.

```
class Class
    (extent Classes
     key name)
 {
    attribute string name;
    attribute string country;
    attribute integer numGuns;
    attribute integer bore;
    attribute integer displacement;
    relationship Set<Ship> ships inverse Ship::classOf;
 };

class Ship
    (extent Ships
     key name)
 {
    attribute string name;
    attribute integer launched;
    relationship Class classOf inverse Class::ships;
    relationship Set<Outcome> inBattles
                 inverse Outcome::theShip;
 };

class Battle
    (extent Battles
     key name)
 {
    attribute string name;
    attribute Date dateFought;
    relationship Set<Outcome> results
                 inverse Outcome::theBattle;
 };

class Outcome
    (extent Outcomes)
 {
    attribute enum Stat {ok,sunk,damaged} status;
    relationship Ship theShip inverse Ship::inBattles;
    relationship Battle theBattle inverse Battle::results;
 };
```

Figure 9.3: Battleships database in ODL

9.2 Additional Forms of OQL Expressions

In this section we shall see some of the other operators, besides select-from-where, that OQL provides to help us build expressions. These operators include logical quantifiers — for-all and there-exists — aggregation operators, the group-by operator, and set operators — union, intersection, and difference.

9.2.1 Quantifier Expressions

We can test whether all members of a collection satisfy some condition, and we can test whether at least one member of a collection satisfies a condition. To test whether all members x of a collection S satisfy condition $C(x)$, we use the OQL expression:

$$\texttt{FOR ALL}\ x\ \texttt{IN}\ S\ :\ C(x)$$

The result of this expression is `TRUE` if every x in S satisfies $C(x)$ and is `FALSE` otherwise. Similarly, the expression

$$\texttt{EXISTS}\ x\ \texttt{IN}\ S\ :\ C(x)$$

has value `TRUE` if there is at least one x in S such that $C(x)$ is `TRUE` and it has value `FALSE` otherwise.

Example 9.8: Another way to express the query "find all the stars of Disney movies" is shown in Fig. 9.4. Here, we focus on a star s and ask if they are the star of some movie m that is a Disney movie. Line (3) tells us to consider all movies m in the set of movies `s.starredIn`, which is the set of movies in which star s appeared. Line (4) then asks whether movie m is a Disney movie. If we find even one such movie m, the value of the `EXISTS` expression in lines (3) and (4) is `TRUE`; otherwise it is `FALSE`. □

```
1)  SELECT s
2)  FROM Stars s
3)  WHERE EXISTS m IN s.starredIn :
4)     m.ownedBy.name = "Disney"
```

Figure 9.4: Using an existential subquery

Example 9.9: Let us use the for-all operator to write a query asking for the stars that have appeared only in Disney movies. Technically, that set includes "stars" who appear in no movies at all (as far as we can tell from our database). It is possible to add another condition to our query, requiring that the star appear in at least one movie, but we leave that improvement as an exercise. Figure 9.5 shows the query. □

```
SELECT s
FROM Stars s
WHERE FOR ALL m IN s.starredIn :
    m.ownedBy.name = "Disney"
```

Figure 9.5: Using a subquery with universal quantification

9.2.2 Aggregation Expressions

OQL uses the same five aggregation operators that SQL does: `AVG`, `COUNT`, `SUM`, `MIN`, and `MAX`. However, while these operators in SQL may be thought of as applying to a designated column of a table, the same operators in OQL apply to all collections whose members are of a suitable type. That is, `COUNT` can apply to any collection; `SUM` and `AVG` can be applied to collections of arithmetic types such as integers, and `MIN` and `MAX` can be applied to collections of any type that can be compared, e.g., arithmetic values or strings.

Example 9.10: To compute the average length of all movies, we need to create a bag of all movie lengths. Note that we don't want the *set* of movie lengths, because then two movies that had the same length would count as one. The query is:

```
AVG(SELECT m.length FROM Movies m)
```

That is, we use a subquery to extract the length components from movies. Its result is the bag of lengths of movies, and we apply the `AVG` operator to this bag, giving the desired answer. □

9.2.3 Group-By Expressions

The `GROUP BY` clause of SQL carries over to OQL, but with an interesting twist in perspective. The form of a `GROUP BY` clause in OQL is:

1. The keywords `GROUP BY`.
2. A comma-separated list of one or more *partition attributes*. Each of these consists of
   (a) A field name,
   (b) A colon, and
   (c) An expression.

That is, the form of a `GROUP BY` clause is:

`GROUP BY` f_1:e_1, f_2:e_2, ..., f_n:e_n

Each `GROUP BY` clause follows a select-from-where query. The expressions $e_1, e_2, \ldots, e_n$ may refer to variables mentioned in the `FROM` clause. To facilitate the explanation of how `GROUP BY` works, let us restrict ourselves to the common case where there is only one variable x in the `FROM` clause. The value of x ranges over some collection, C. For each member of C, say i, that satisfies the condition of the `WHERE` clause, we evaluate all the expressions that follow the `GROUP BY`, to obtain values $e_1(i), e_2(i), \ldots, e_n(i)$. This list of values is the group to which value i belongs.

The Intermediate Collection

The actual value returned by the `GROUP BY` is a set of structures, which we shall call the *intermediate collection*. The members of the intermediate collection have the form

$$\texttt{Struct(}f_1\texttt{:}v_1\texttt{, }f_2\texttt{:}v_2, \ldots, f_n\texttt{:}v_n\texttt{, partition:}P\texttt{)}$$

The first n fields indicate the group. That is, $(v_1, v_2, \ldots, v_n)$ must be the list of values $\big(e_1(i), e_2(i), \ldots, e_n(i)\big)$ for at least one value of i in the collection C that meets the condition of the `WHERE` clause.

The last field has the special name `partition`. Its value P is, intuitively, the values i that belong in this group. More precisely, P is a bag consisting of structures of the form `Struct(x:i)`, where x is the variable of the `FROM` clause.

The Output Collection

The `SELECT` clause of a select-from-where expression that has a `GROUP BY` clause may refer only to the fields in the structures of the intermediate collection, namely $f_1, f_2, \ldots, f_n$ and `partition`. Through `partition`, we may refer to the field x that is present in the structures that are members of the bag P that forms the value of `partition`. Thus, we may refer to the variable x that appears in the `FROM` clause, but we may only do so within an aggregation operator that aggregates over all the members of a bag P. The result of the `SELECT` clause will be referred to as the *output collection*.

Example 9.11: Let us build a table of the total length of movies for each studio and for each year. In OQL, what we actually construct is a bag of structures, each with three components — a studio, a year, and the total length of movies for that studio and year. The query is shown in Fig. 9.6.

To understand this query, let us start at the `FROM` clause. There, we find that variable m ranges over all `Movie` objects. Thus, m here plays the role of x in our general discussion. In the `GROUP BY` clause are two fields `stdo` and `yr`, corresponding to the expressions `m.ownedBy.name` and `m.year`, respectively.

For instance, *Pretty Woman* is a movie made by Disney in 1990. When m is the object for this movie, the value of `m.ownedBy.name` is `"Disney"` and the value of `m.year` is 1990. As a result, the intermediate collection has, as one member, the structure:

```
SELECT stdo, yr, sumLength: SUM(SELECT p.m.length
                                FROM partition p)
FROM Movies m
GROUP BY stdo: m.ownedBy.name, yr: m.year
```

Figure 9.6: Grouping movies by studio and year

`Struct(stdo:"Disney", yr:1990, partition:`P`)`

Here, P is a set of structures. It contains, for example,

`Struct(m:`m_{pw}`)`

where m_{pw} is the `Movie` object for *Pretty Woman*. Also in P are one-component structures with field name `m` for every other Disney movie of 1990.

Now, let us examine the `SELECT` clause. For each structure in the intermediate collection, we build one structure that is in the output collection. The first component of each output structure is `stdo`. That is, the field name is `stdo` and its value is the value of the `stdo` field of the corresponding structure in the intermediate collection. Similarly, the second component of the result has field name `yr` and a value equal to the `yr` component of the intermediate collection.

The third component of each structure in the output is

```
SUM(SELECT p.m.length FROM partition p)
```

To understand this select-from expression we first realize that variable p ranges over the members of the `partition` field of the structure in the `GROUP BY` result. Each value of p, recall, is a structure of the form `Struct(m:`o`)`, where o is a movie object. The expression `p.m` therefore refers to this object o. Thus, `p.m.length` refers to the length component of this `Movie` object.

As a result, the select-from query produces the bag of lengths of the movies in a particular group. For instance, if `stdo` has the value `"Disney"` and `yr` has the value 1990, then the result of the select-from is the bag of the lengths of the movies made by Disney in 1990. When we apply the `SUM` operator to this bag we get the sum of the lengths of the movies in the group. Thus, one member of the output collection might be

`Struct(stdo:"Disney", yr:1990, sumLength:1234)`

if 1234 is the correct total length of all the Disney movies of 1990. □

Grouping When the `FROM` Clause has Multiple Collections

In the event that there is more than one variable in the `FROM` clause, a few changes to the interpretation of the query are necessary, but the principles remain the same as in the one-variable case above. Suppose that the variables appearing in the `FROM` clause are $x_1, x_2, \ldots, x_k$. Then:

1. All variables $x_1, x_2, \ldots, x_k$ may be used in the expressions $e_1, e_2, \ldots, e_n$ of the `GROUP BY` clause.

2. Structures in the bag that is the value of the `partition` field have fields named x_1, x_2, ..., x_k.

3. Suppose $i_1, i_2, \ldots, i_k$ are values for variables $x_1, x_2, \ldots, x_k$, respectively, that make the `WHERE` clause true. Then there is a structure in the intermediate collection of the form

   `Struct(`f_1`:`$e_1(i_1, \ldots, i_k)$`, ...,` f_n`:`$e_n(i_1, \ldots, i_k)$`, partition:P)`

   and in bag P is the structure:

   `Struct(`x_1`:`i_1`,` x_2`:`$i_2, \ldots,$ x_k`:`i_k`)`

9.2.4 HAVING Clauses

A `GROUP BY` clause of OQL may be followed by a `HAVING` clause, with a meaning like that of SQL's `HAVING` clause. That is, a clause of the form

`HAVING` <condition>

serves to eliminate some of the groups created by the `GROUP BY`. The condition applies to the value of the `partition` field of each structure in the intermediate collection. If true, then this structure is processed as in Section 9.2.3, to form a structure of the output collection. If false, then this structure does not contribute to the output collection.

Example 9.12: Let us repeat Example 9.11, but ask for the sum of the lengths of movies for only those studios and years such that the studio produced at least one movie of over 120 minutes. The query of Fig. 9.7 does the job. Notice that in the `HAVING` clause we used the same query as in the `SELECT` clause to obtain the bag of lengths of movies for a given studio and year. In the `HAVING` clause, we take the maximum of those lengths and compare it to 120. □

```
SELECT stdo, yr, sumLength: SUM(SELECT p.m.length
                                FROM partition p)
FROM Movies m
GROUP BY stdo: m.ownedBy.name, yr: m.year
HAVING MAX(SELECT p.m.length FROM partition p) > 120
```

Figure 9.7: Restricting the groups considered

9.2.5 Union, Intersection, and Difference

We may apply the union, intersection, and difference operators to two objects of set or bag type. These three operators are represented, as in SQL, by the keywords `UNION`, `INTERSECT`, and `EXCEPT`, respectively.

```
1)      (SELECT DISTINCT m
2)       FROM Movies m, m.stars s
3)       WHERE s.name = "Harrison Ford")
4)  EXCEPT
5)      (SELECT DISTINCT m
6)       FROM Movies m
7)       WHERE m.ownedBy.name = "Disney")
```

Figure 9.8: Query using the difference of two sets

Example 9.13: We can find the set of movies starring Harrison Ford that were not made by Disney with the difference of two select-from-where queries shown in Fig. 9.8. Lines (1) through (3) find the set of movies starring Ford, and lines (5) through (7) find the set of movies made by Disney. The `EXCEPT` at line (4) takes their difference. □

We should notice the `DISTINCT` keywords in lines (1) and (5) of Fig. 9.8. This keyword forces the results of the two queries to be of set type; without `DISTINCT`, the result would be of bag (multiset) type. In OQL, the operators `UNION`, `INTERSECT`, and `EXCEPT` operate on either sets or bags. When both arguments are sets, then the operators have their usual set meaning.

However, when both arguments are of bag type, or one is a bag and one is a set, then the bag meaning of the operators is used. Recall Section ??, where the definitions of union, intersection, and difference for bags was explained.

For the particular query of Fig. 9.8, the number of times a movie appears in the result of either subquery is zero or one, so the result is the same regardless of whether `DISTINCT` is used. However, the *type* of the result differs. If `DISTINCT` is used, then the type of the result is `Set<Movie>`, while if `DISTINCT` is omitted in one or both places, then the result is of type `Bag<Movie>`.

9.2.6 Exercises for Section 9.2

Exercise 9.2.1: Using the ODL schema of Exercise 9.1.1 and Fig. 9.2, write the following queries in OQL:

* a) Find the manufacturers that make both PC's and printers.

* b) Find the manufacturers of PC's, all of whose PC's have at least 20 gigabytes of hard disk.

c) Find the manufacturers that make PC's but not laptops.

* d) Find the average speed of PC's.

* e) For each CD or DVD speed, find the average amount of RAM on a PC.

! f) Find the manufacturers that make some product with at least 64 megabytes of RAM and also make a product costing under $1000.

!! g) For each manufacturer that makes PC's with an average speed of at least 1200, find the maximum amount of RAM that they offer on a PC.

Exercise 9.2.2: Using the ODL schema of Exercise 9.1.3 and Fig. 9.3, write the following queries in OQL:

a) Find those classes of ship all of whose ships were launched prior to 1919.

b) Find the maximum displacement of any class.

! c) For each gun bore, find the earliest year in which any ship with that bore was launched.

*!! d) For each class of ships at least one of which was launched prior to 1919, find the number of ships of that class sunk in battle.

! e) Find the average number of ships in a class.

! f) Find the average displacement of a ship.

!! g) Find the battles (objects, not names) in which at least one ship from Great Britain took part and in which at least two ships were sunk.

! **Exercise 9.2.3:** We mentioned in Example 9.9 that the OQL query of Fig. 9.5 would return stars who starred in no movies at all, and therefore, technically appeared "only in Disney movies." Rewrite the query to return only those stars who have appeared in at least one movie and all movies in which they appeared where Disney movies.

! **Exercise 9.2.4:** Is it ever possible for `FOR ALL` x `IN` S : $C(x)$ to be true, while `EXISTS` x `IN` S : $C(x)$ is false? Explain your reasoning.

9.3 Object Assignment and Creation in OQL

In this section we shall consider how OQL connects to its host language, which we shall take to be C++ in examples, although another object-oriented, general-purpose programming language (e.g. Java) might be the host language in some systems.

9.3.1 Assigning Values to Host-Language Variables

Unlike SQL, which needs to move data between components of tuples and host-language variables, OQL fits naturally into its host language. That is, the expressions of OQL that we have learned, such as select-from-where, produce objects as values. It is possible to assign to any host-language variable of the proper type a value that is the result of one of these OQL expressions.

Example 9.14: The OQL expression

```
SELECT DISTINCT m
FROM Movies m
WHERE m.year < 1920
```

produces the set of all those movies made before 1920. Its type is `Set<Movie>`. If `oldMovies` is a host-language variable of the same type, then we may write (in C++ extended with OQL):

```
oldMovies = SELECT DISTINCT m
            FROM Movies m
            WHERE m.year < 1920;
```

and the value of `oldMovies` will become the set of these `Movie` objects. □

9.3.2 Extracting Elements of Collections

Since the select-from-where and group-by expressions each produce collections — either sets, bags, or lists — we must do something extra if we want a single element of that collection. This statement is true even if we have a collection that we are sure contains only one element. OQL provides the operator `ELEMENT` to turn a singleton collection into its lone member. This operator can be applied, for instance, to the result of a query that is known to return a singleton.

Example 9.15: Suppose we would like to assign to the variable `gwtw`, of type `Movie` (i.e., the `Movie` class is its type) the object representing the movie *Gone With the Wind*. The result of the query

```
SELECT m
FROM Movies m
WHERE m.title = "Gone With the Wind"
```

is the bag containing just this one object. We cannot assign this bag to variable `gwtw` directly, because we would get a type error. However, if we apply the `ELEMENT` operator first,

```
gwtw = ELEMENT(SELECT m
               FROM Movies m
               WHERE m.title = "Gone With the Wind"
       );
```

then the type of the variable and the expression match, and the assignment is legal. □

9.3.3 Obtaining Each Member of a Collection

Obtaining each member of a set or bag is more complex, but still simpler than the cursor-based algorithms we needed in SQL. First, we need to turn our set or bag into a list. We do so with a select-from-where expression that uses `ORDER BY`. Recall from Section 9.1.4 that the result of such an expression is a list of the selected objects or values.

Example 9.16: Suppose we want a list of all the movie objects in the class `Movie`. We can use the title and (to break ties) the year of the movie, since `(title, year)` is a key for `Movie`. The statement

```
movieList = SELECT m
            FROM Movies m
            ORDER BY m.title, m.year;
```

assigns to host-language variable `movieList` a list of all the `Movie` objects, sorted by title and year. □

Once we have a list, sorted or not, we can access each element by number; the ith element of the list L is obtained by $L[i-1]$. Note that lists and arrays are assumed numbered starting at 0, as in C or C++.

Example 9.17: Suppose we want to write a C++ function that prints the title, year, and length of each movie. A sketch of the function is shown in Fig. 9.9.

Line (1) sorts the `Movie` class, placing the result into variable `movieList`, whose type is `List<Movie>`. Line (2) computes the number of movies, using the OQL operator `COUNT`. Lines (3) through (6) are a for-loop in which integer variable `i` ranges over each position of the list. For convenience, the ith element of the list is assigned to variable `movie`. Then, at lines (5) and (6) the relevant attributes of the movie are printed. □

```
1)  movieList = SELECT m
                FROM Movies m
                ORDER BY m.title, m.year;
2)  numberOfMovies = COUNT(Movies);
3)  for(i=0; i<numberOfMovies; i++) {
4)      movie = movieList[i];
5)      cout << movie.title << " " << movie.year << " "
6)          << movie.length << "\n";
    }
```

Figure 9.9: Examining and printing each movie

9.3.4 Constants in OQL

Constants in OQL (sometimes referred to as *immutable objects*) are constructed from a basis and recursive constructors, in a manner analogous to the way ODL types are constructed.

1. *Basic values*, which are either

   (a) *Atomic values*: integers, floats, characters, strings, and booleans. These are represented as in SQL, with the exception that double-quotes are used to surround strings.

   (b) *Enumerations*. The values in an enumeration are actually declared in ODL. Any one of these values may be used as a constant.

2. *Complex values* built using the following type constructors:

   (a) `Set(...)`.

   (b) `Bag(...)`.

   (c) `List(...)`.

   (d) `Array(...)`.

   (e) `Struct(...)`.

   The first four of these are called *collection types*. The collection types and `Struct` may be applied at will to any values of the appropriate type(s), basic or complex. However, when applying the `Struct` operator, one needs to specify the field names and their corresponding values. Each field name is followed by a colon and the value, and field-value pairs are separated by commas. Note that the same type constructors are used in ODL, but here we use round, rather than triangular, brackets.

Example 9.18: The expression `Bag(2,1,2)` denotes the bag in which integer 2 appears twice and integer 1 appears once. The expression

```
Struct(foo: bag(2,1,2), bar: "baz")
```

denotes a structure with two fields. Field `foo`, has the bag described above as its value, and `bar`, has the string `"baz"` for its value. □

9.3.5 Creating New Objects

We have seen that OQL expressions such as select-from-where allow us to create new objects. It is also possible to create objects by assembling constants or other expressions into structures and collections explicitly. We saw an example of this convention in Example 9.6, where the line

```
SELECT DISTINCT Struct(star1: s1, star2: s2)
```

was used to specify that the result of the query is a set of objects whose type is `Struct{star1: Star, star2: Star}`. We gave the field names `star1` and `star2` to specify the structure, while the types of these fields could be deduced from the types of the variables `s1` and `s2`.

Example 9.19: The construction of constants that we saw in Section 9.3.4 can be used with assignments to variables, in a manner similar to that of other programming languages. For instance, consider the following sequence of assignments:

```
x = Struct(a:1, b:2);
y = Bag(x, x, Struct(a:3, b:4));
```

The first line gives variable `x` a value of type

```
Struct(a:integer, b:integer)
```

a structure with two integer-valued fields named `a` and `b`. We may represent values of this type as pairs, with just the integers as components and not the field names `a` and `b`. Thus, the value of x may be represented by $(1,2)$. The second line defines `y` to be a bag whose members are structures of the same type as x, above. The pair $(1,2)$ appears twice in this bag, and $(3,4)$ appears once. □

Classes or other defined types can have instances created by *constructor functions*. Classes typically have several different forms of constructor functions, depending on which properties are initialized explicitly and which are given some default value. For example, methods are not initialized, most attributes will get initial values, and relationships might be initialized to the empty set and augmented later. The name for each of these constructor functions is the name of the class, and they are distinguished by the field names mentioned in their arguments. The details of how these constructor functions are defined depend on the host language.

Example 9.20: Let us consider a possible constructor function for `Movie` objects. This function, we suppose, takes values for the attributes `title`, `year`, `length`, and `ownedBy`, producing an object that has these values in the listed fields and an empty set of stars. Then, if `mgm` is a variable whose value is the MGM `Studio` object, we might create a *Gone With the Wind* object by:

```
gwtw = Movie(title: "Gone With the Wind",
             year: 1939,
             length: 239,
             ownedBy: mgm);
```

This statement has two effects:

1. It creates a new `Movie` object, which becomes part of the extent `Movies`.
2. It makes this object the value of host-language variable `gwtw`.

□

9.3.6 Exercises for Section 9.3

Exercise 9.3.1: Assign to a host-language variable x the following constants:

* a) The set $\{1, 2, 3\}$.

b) The bag $\{1, 2, 3, 1\}$.

c) The list $(1, 2, 3, 1)$.

d) The structure whose first component, named `a`, is the set $\{1, 2\}$ and whose second component, named `b`, is the bag $\{1, 1\}$.

e) The bag of structures, each with two fields named `a` and `b`. The respective pairs of values for the three structures in the bag are $(1, 2)$, $(2, 1)$, and $(1, 2)$.

Exercise 9.3.2: Using the ODL schema of Exercise 9.1.1 and Fig. 9.2, write statements of C++ (or an object-oriented host language of your choice) extended with OQL to do the following:

* a) Assign to host-language variable x the object for the PC with model number 1000.

b) Assign to host-language variable y the set of all laptop objects with at least 64 megabytes of RAM.

c) Assign to host-language variable z the average speed of PC's selling for less than \$1500.

! d) Find all the laser printers, print a list of their model numbers and prices, and follow it by a message indicating the model number with the lowest price.

!! e) Print a table giving, for each manufacturer of PC's, the minimum and maximum price.

Exercise 9.3.3: In this exercise, we shall use the ODL schema of Exercise 9.1.3 and Fig. 9.3. We shall assume that for each of the four classes of that schema, there is a constructor function of the same name that takes values for each of the attributes and single-valued relationships, but not the multivalued relationships, which are initialized to be empty. For the single-valued relationships to other classes, you may postulate a host-language variable whose current value is the related object. Create the following objects and assign the object to be the value of a host-language variable in each case.

* a) The battleship Colorado of the Maryland class, launched in 1923.

b) The battleship Graf Spee of the Lützow class, launched in 1936.

c) An outcome of the battle of Malaya was that the battleship Prince of Wales was sunk.

d) The battle of Malaya was fought Dec. 10, 1941.

e) The Hood class of British battlecruisers had eight 15-inch guns and a displacement of 41,000 tons.