Mary	Scott	842 Vine Ave.	Losantiville	Ohio
Sam	Jones	33 Elm St.	Paris	New York
Sarah	Ackerman	440 U.S. 110	Upton	Michigan

To explain what you just did, you asked for the all of data in the EmployeeAddressTable, and specifically, you asked for the *columns* called FirstName, LastName, Address, City, and State. Note that column names and table names do not have spaces...they must be typed as one word; and that the statement ends with a semicolon (;). The general form for a SELECT statement, retrieving all of the *rows* in the table is:

```
SELECT ColumnName, ColumnName, ...
FROM TableName;
```

To get all columns of a table without typing all column names, use:

```
SELECT * FROM TableName;
```

Each database management system (DBMS) and database software has different methods for logging in to the database and entering SQL commands; see the local computer "guru" to help you get onto the system, so that you can use SQL.

Conditional Selection

To further discuss the SELECT statement, let's look at a new example table (for hypothetical purposes only):

EmployeeStatisticsTable			
EmployeeIDNo	**Salary**	**Benefits**	**Position**
010	75000	15000	Manager
105	65000	15000	Manager
152	60000	15000	Manager
215	60000	12500	Manager
244	50000	12000	Staff
300	45000	10000	Staff
335	40000	10000	Staff
400	32000	7500	Entry-Level

result to use to evaluate the OR, which evaluates to true if either value is true). Mathematically, SQL evaluates all of the conditions, then evaluates the AND "pairs", and then evaluates the OR's (where both operators evaluate left to right).

To look at an example, for a given row for which the DBMS is evaluating the SQL statement Where clause to determine whether to include the row in the query result (the whole Where clause evaluates to True), the DBMS has evaluated all of the conditions, and is ready to do the logical comparisons on this result:

```
True AND False OR True AND True OR False AND False
```

First simplify the AND pairs:

```
False OR True OR False
```

Now do the OR's, left to right:

```
True OR False
True
```

The result is True, and the row passes the query conditions. Be sure to see the next section on NOT's, and the order of logical operations. I hope that this section has helped you understand AND's or OR's, as it's a difficult subject to explain briefly.

To perform OR's before AND's, like if you wanted to see a list of employees making a large salary ($50,000) or have a large benefit package ($10,000), and that happen to be a manager, use parentheses:

```
SELECT EMPLOYEEIDNO
FROM EMPLOYEESTATISTICSTABLE
WHERE POSITION = 'Manager' AND (SALARY > 50000 OR BENEFITS > 10000);
```

IN & BETWEEN

An easier method of using compound conditions uses *IN* or *BETWEEN*. For example, if you wanted to list all managers and staff:

```
SELECT EMPLOYEEIDNO
FROM EMPLOYEESTATISTICSTABLE
WHERE POSITION IN ('Manager', 'Staff');
```

All tables within a database must be created at some point in time...let's see how we would create the Orders table:

```
CREATE TABLE ORDERS
(OWNERID INTEGER NOT NULL,
ITEMDESIRED CHAR(40) NOT NULL);
```

This statement gives the table name and tells the DBMS about each column in the table. ***Please note*** that this statement uses generic data types, and that the data types might be different, depending on what DBMS you are using. As usual, check local listings. Some common generic data types are:

- Char(x) - A column of characters, where x is a number designating the maximum number of characters allowed (maximum length) in the column.
- Integer - A column of whole numbers, positive or negative.
- Decimal(x, y) - A column of decimal numbers, where x is the maximum length in digits of the decimal numbers in this column, and y is the maximum number of digits allowed after the decimal point. The maximum (4,2) number would be 99.99.
- Date - A date column in a DBMS-specific format.
- Logical - A column that can hold only two values: TRUE or FALSE.

One other note, the NOT NULL means that the column must have a value in each row. If NULL was used, that column may be left empty in a given row.

Altering Tables

Let's add a column to the Antiques table to allow the entry of the price of a given Item (Parentheses optional):

```
ALTER TABLE ANTIQUES ADD (PRICE DECIMAL(8,2) NULL);
```

The data for this new column can be updated or inserted as shown later.

Adding Data

To insert rows into a table, do the following:

```
INSERT INTO ANTIQUES VALUES (21, 01, 'Ottoman', 200.00);
```

This inserts the data into the table, as a new row, column-by-column, in the pre-defined order. Instead, let's change the order and leave Price blank:

though it is somewhat common to use other, more advanced forms (fourth, fifth, Boyce-Codd; see documentation).

First Normal Form refers to moving data into separate tables where the data in each table is of a similar type, and by giving each table a primary key.

Putting data in *Second Normal Form* involves removing to other tables data that is only dependent of a part of the key. For example, if I had left the names of the Antique Owners in the items table, that would not be in Second Normal Form because that data would be redundant; the name would be repeated for each item owned; as such, the names were placed in their own table. The names themselves don't have anything to do with the items, only the identities of the buyers and sellers.

Third Normal Form involves getting rid of anything in the tables that doesn't depend solely on the primary key. Only include information that is dependent on the key, and move off data to other tables that are independent of the primary key, and create a primary key for the new tables.

There is some redundancy to each form, and if data is in *3NF* (shorthand for 3rd normal form), it is already in *1NF* and *2NF*. In terms of data design then, arrange data so that any non-primary key columns are dependent only on the *whole primary key*. If you take a look at the sample database, you will see that the way then to navigate through the database is through joins using common key columns.

Two other important points in database design are using good, consistent, logical, full-word names for the tables and columns, and the use of full words in the database itself. On the last point, my database is lacking, as I use numeric codes for identification. It is usually best, if possible, to come up with keys that are, by themselves, self-explanatory; for example, a better key would be the first four letters of the last name and first initial of the owner, like JONEB for Bill Jones (or for tiebreaking purposes, add numbers to the end to differentiate two or more people with similar names, so you could try JONEB1, JONEB2, etc.).

14. What is the difference between a *single-row query* and a *multiple-row query* and why is it important to know the difference? --First, to cover the obvious, a single-row query is a query that returns one row as its result, and a multiple-row query is a query that returns more than one row as its result. Whether a query returns one row or more than one row is entirely dependent on the design (or *schema*) of the tables of the database. As query-writer, you must be aware of the schema, be sure to include enough conditions, and structure your SQL statement properly, so that you will get the desired result (either one row or multiple rows). For example, if you wanted to be sure that a query of the AntiqueOwners table returned only one row, consider an equal condition of the primary key column, OwnerID.

    Three reasons immediately come to mind as to why this is important. First, getting multiple rows when you were expecting only one, or vice-versa, may mean that the query is erroneous, that the database is incomplete, or simply, you learned something new about your data. Second, if you are

key column, and that for every primary key value, there is **one** foreign key value. For example, in the first example, the EmployeeAddressTable, we add an EmployeeIDNo column. Then, the EmployeeAddressTable is related to the EmployeeStatisticsTable (second example table) by means of that EmployeeIDNo. Specifically, each employee in the EmployeeAddressTable **has** statistics (one row of data) in the EmployeeStatisticsTable. Even though this is a contrived example, this is a "1-1" relationship. Also notice the "has" in bold...when expressing a relationship, it is important to describe the relationship with a verb.

The other two kinds of relationships may or may not use logical primary key and foreign key constraints...it is strictly a call of the designer. The first of these is the *one-to-many relationship* ("1-M"). This means that for every column value in one table, there is **one or more** related values in another table. Key constraints may be added to the design, or possibly just the use of some sort of identifier column may be used to establish the relationship. An example would be that for every OwnerID in the AntiqueOwners table, there are one or more (zero is permissible too) Items **bought** in the Antiques table (verb: buy).

Finally, the *many-to-many relationship* ("M-M") does not involve keys generally, and usually involves identifying columns. The unusual occurrence of a "M-M" means that one column in one table is related to another column in another table, and for every value of one of these two columns, there are one or more related values in the corresponding column in the other table (and vice-versa), or more a common possibility, two tables have a 1-M relationship to each other (two relationships, one 1-M going each way). A [bad] example of the more common situation would be if you had a job assignment database, where one table held one row for each employee and a job assignment, and another table held one row for each job with one of the assigned employees. Here, you would have multiple rows for each employee in the first table, one for each job assignment, and multiple rows for each job in the second table, one for each employee assigned to the project. These tables have a M-M: each employee in the first table **has** many job assignments from the second table, and each job **has** many employees assigned to it from the first table. This is the tip of the iceberg on this topic...see the links below for more information and see the diagram below for a *simplified* example of an E-R diagram.

16. What term is used to describe the event of a database system automatically updating the values of foreign keys in other tables, when the value of a primary key is updated?
17. What database object provides fast access to the data in the rows of a table?
18. What type of SQL statement is used to change the attributes of a column?
19. In a Create Table statement, when a column is designated as NOT NULL, what does this mean?
20. If you wish to write a query that is based on other queries, rather than tables, what do these other queries need to be created as?

Answers (Queries may have more than one correct answer):

```
1. SELECT AntiqueOwners.OwnerLastName, AntiqueOwners.OwnerFirstName,
Orders.ItemDesired
FROM AntiqueOwners, Orders
WHERE AntiqueOwners.OwnerID = Orders.OwnerID;
or
SELECT AntiqueOwners.OwnerLastName, AntiqueOwners.OwnerFirstName,
Orders.ItemDesired
FROM AntiqueOwners RIGHT JOIN Orders ON AntiqueOwners.OwnerID = Orders.
OwnerID;
2. SELECT *
FROM EmployeeStatisticsTable
ORDER BY Position, EmployeeIDNo;
3. SELECT Sum(Benefits)
FROM EmployeeStatisticsTable;
4. SELECT OwnerLastName, OwnerFirstName
FROM AntiqueOwners, Antiques
WHERE Item In ('Chair')
AND AntiqueOwners.OwnerID = Antiques.BuyerID;
5. SELECT OwnerLastName, OwnerFirstName
FROM AntiqueOwners
WHERE OwnerID NOT IN
(SELECT OwnerID
FROM Orders);
6. SELECT DISTINCT OwnerLastName, OwnerFirstName
FROM Orders, AntiqueOwners
WHERE AntiqueOwners.OwnerID = Orders.OwnerID;
or to use JOIN notation:
SELECT DISTINCT AntiqueOwners.OwnerLastName, AntiqueOwners.
OwnerFirstName
FROM AntiqueOwners RIGHT JOIN Orders ON AntiqueOwners.OwnerID = Orders.
OwnerID;
7. DELETE FROM ORDERS
WHERE OWNERID = 02;
8. INSERT INTO ORDERS VALUES (21, 'Rocking Chair');
9. CREATE TABLE EMPLOYEES
```