The Complete Reference



Part III

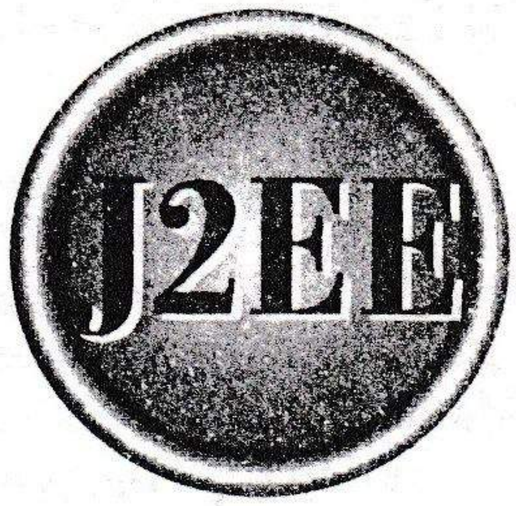
J2EE Databases

At the center of nearly every J2EE application is a repository of information that is accessed and manipulated by service-side components such as web services and client-side applications. A repository is a database management system that stores, retrieves, and maintains the integrity of information stored in its databases.

A J2EE application uses Java data objects, JDBC, and other technology necessary in interacting with a database management systems, to provide information to the J2EE application.

In Part II you'll learn database concepts in relationship to Java data objects. You'll also explore the details of JDBC, which is used to connect to and interact with popular—and some not so popular—database management systems. And you'll also learn how to create and send requests for information and then integrate the results into your J2EE application.

The Complete Reference



Chapter 5

J2EE Database Concepts

A J2EE application is built by assembling components, many of which are shared with other applications, as you learned in Part I. Each component provides a J2EE application with a web service, and one of these is access to one or more databases.

A database is a collection of data that is managed by a database management system (DBMS). Many corporations use one of several commercially available DBMSs such as Oracle, DB2, Sybase, and Microsoft Access, which is used for small data collections.

Components that provide database access to a J2EE application interact with commercial DBMSs by using a combination of Java data objects that are defined in the Java Database Connection (JDBC) specification and by using the Structured Query Language (SQL). Java data objects form a communications link with a DBMS while SQL is the language used to construct the message (called a query) that is sent to the DBMS to request data, update data, delete data, and otherwise manipulate data in the DBMS.

This is the first of three chapters that show how to incorporate database interactions into a J2EE application. This first chapter covers database fundamentals, where you learn how to transform data elements of a business system into a relational database, which is a common task in development of J2EE applications. You can skip this chapter if you are familiar with this technique.

The next chapter focuses on Java data objects—what they are, how they work, and how to use them to communicate with a DBMS. The third chapter shows how to write and execute queries, and then interact with the results returned by the DBMS.

Data

You are probably familiar with the term "data" because you use data in everyday life, such as when you dial a telephone number or log into a computer network using a user ID and password. The telephone number, user ID, and password are types of data.

Yet the term "data" is commonly confused with the term "information." Although these terms have similar meaning in the vernacular, they are different when related to a DBMS. Information consists of one or more words that collectively infer a meaning, such as a person's address.

Data refers to an atomic unit that is stored in a DBMS and is sometimes reassembled into information. Examples of data are a person's street address, city, state, and ZIP code. Each of these is an atomic unit that is commonly found in a DBMS. A J2EE component can access one or multiple atomic units as is required by a J2EE application or another component. A database is an organization of data so a J2EE component can quickly find, retrieve, update, or delete one or more data elements.

Database

As mentioned previously in this chapter, a database in the purest sense is a collection of data. While you can use Java and Java's I/O classes to create your own database, you'll probably interact with a commercially available DBMS. A DBMS provides an industrial-

strength solution to data management and uses proprietary and public domain algorithms to assure fast and secure interaction with data stored in the database.

There are many different DBMSs, most of which adhere to a widely accepted relational database model. A database model is a description of how data is organized in a database. In a relational database model, data is grouped into tables using a technique called normalization, which you'll learn about later in this chapter. Tables are grouped into a database.

Once a database and at least one table are created, a J2EE component can send SQL statements to the DBMS to

- Save data
- Retrieve data
- Update data
- Manipulate data
- Delete data

Tables

A table is the component of a database that contains data in the form of rows and columns, very similar to a spreadsheet. A row contains related data such as a client's name and address. A column contains like data, such as a clients' first names.

Each column is identified by a unique name, called a column name, that describes the data contained in that column. For example, client first name is a likely choice as the name of a column that contains clients' first names.

In addition, columns are defined by attributes. An attribute describes the characteristic of data that can be stored in the column. Attributes include size, data type, and format. You'll learn about these and other attributes later in this chapter.

Database name, table name, column name, column attributes, and other information that describes database components are known as metadata. Metadata is data about data. For example, the size of the client's first name column describes the data contained within the column and therefore is referred to as metadata.

Metadata is used by J2EE components to identify database components without needing to know details of a column, the table, or the database. For example, a J2EE component can request from the DBMS the data type of a specific column. The column type is used by a J2EE component to copy data retrieved from the DBMS into a Java collection.

Database Schema

A database schema (see Figure 5-1) is a document that defines all components of a database such as tables, columns, and indexes. A database schema also shows relationships between tables, which are used to join together rows of two tables. You'll learn about indexes and relating tables later in this chapter.

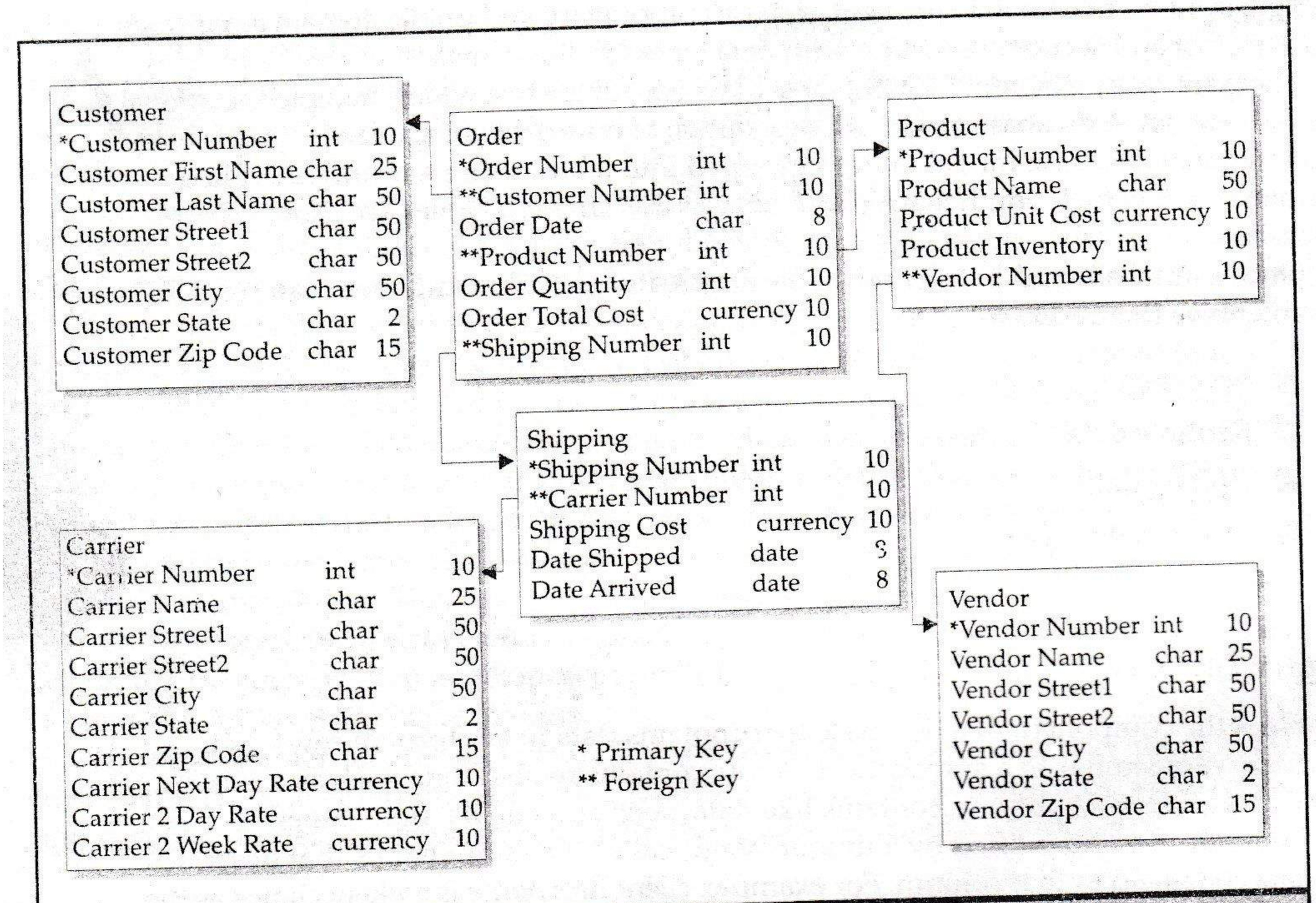


Figure 5-1. *A database schema diagrams the relationships among all components of a database.*

There are six steps that must be performed to create a database schema. These are

1. Identify information used in the existing system or legacy system that is being replaced by the J2EE application.
2. Decompose this information into data.
3. Define data.
4. Normalize data into logical groups.
5. Create primary and foreign keys.
6. Group data together into logical groups.

Identifying Information

The initial step to define a database schema is to identify all information that is used by the system that is being converted to J2EE technology. Information is associated with objects, also known as entities of the system.

An entity is, for example, an order form, a product, a customer, or a sales representative. Figure 5-2 illustrates the entities for an order system. Each entity is

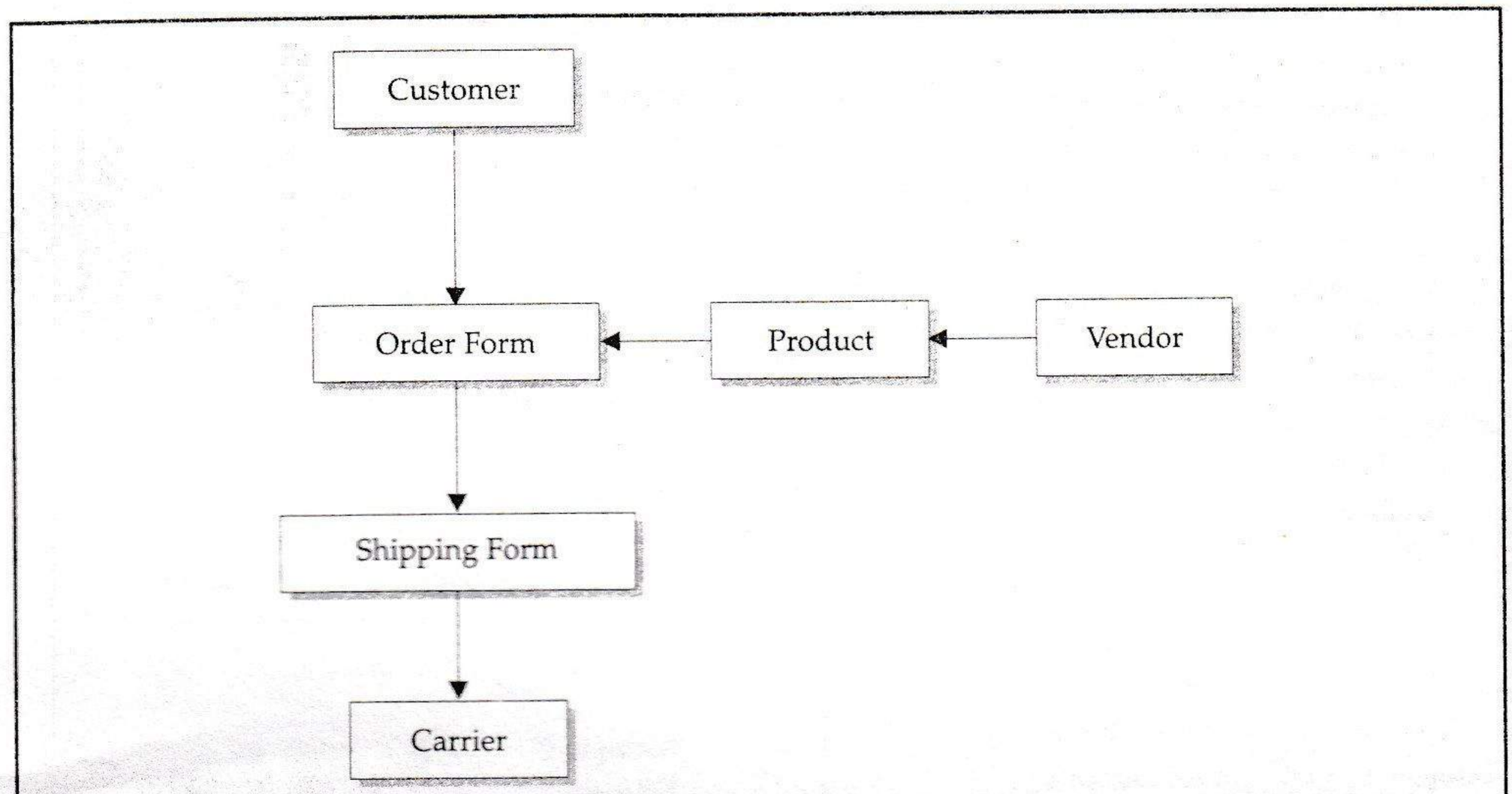


Figure 5-2. *Entities of the order system*

defined by attributes. An attribute is information that describes an entity such as a customer name for a customer entity.

Don't confuse an entity attribute with data attributes, because an entity attribute can be different than data attributes. An entity attribute provides general information about an entity while a data attribute provides information about data that is used by the entity. Data, as you'll recall, is the atomic level of information. An attribute of an entity is at a more general level than the atomic level. For example, a customer name is an entity attribute, and a customer first name and customer last name are data attributes.

Identifying attributes is intuitive most times because an attribute is information commonly used to describe an entity. For example, a customer name and address are information normally used to describe a customer. Therefore, customer name and address are easily recognizable as attributes of a customer entity. Figure 5-3 contains attributes for entities in the order system. Notice how attributes of an entity uniquely identify the entity.

The best way to identify attributes of an entity is by analyzing instances of the entity. An entity is like an empty order form and an instance is an order form that contains order information. Looking at instances of an entity helps to identify attributes because you are viewing a real entity. That is, instead of looking at a blank order form you are looking at an order form that represents a real order. You'll find instances of an entity in the existing system.

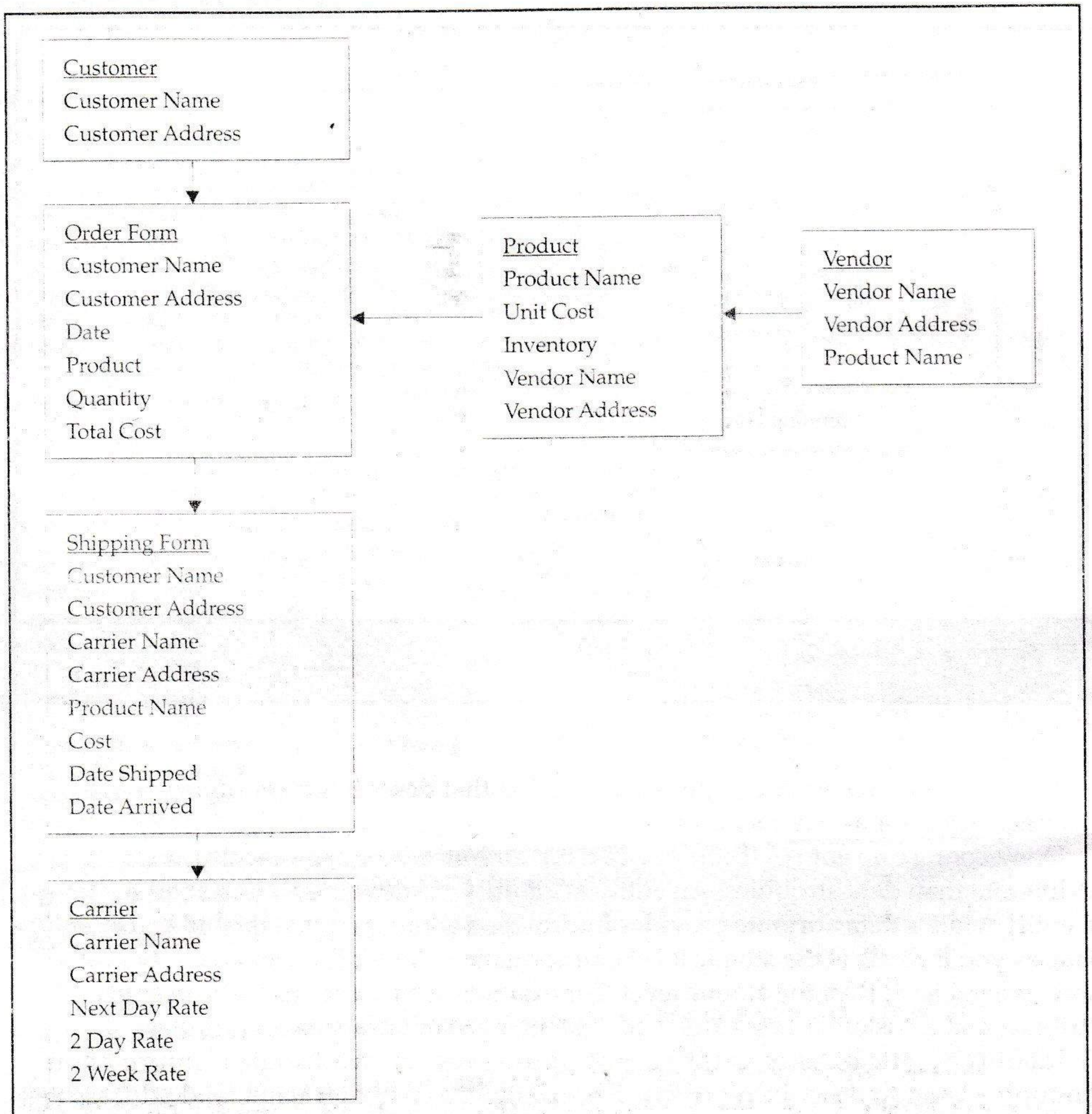


Figure 5-3. *Attributes of entities of the order system*

Once attributes are identified, you must describe the characteristics of each attribute (see Figure 5-4). Here are common characteristics found in many attributes.

- **Attribute name** The name of an attribute uniquely identifies the attribute from other attributes of the same entity. "First name" is an attribute name.

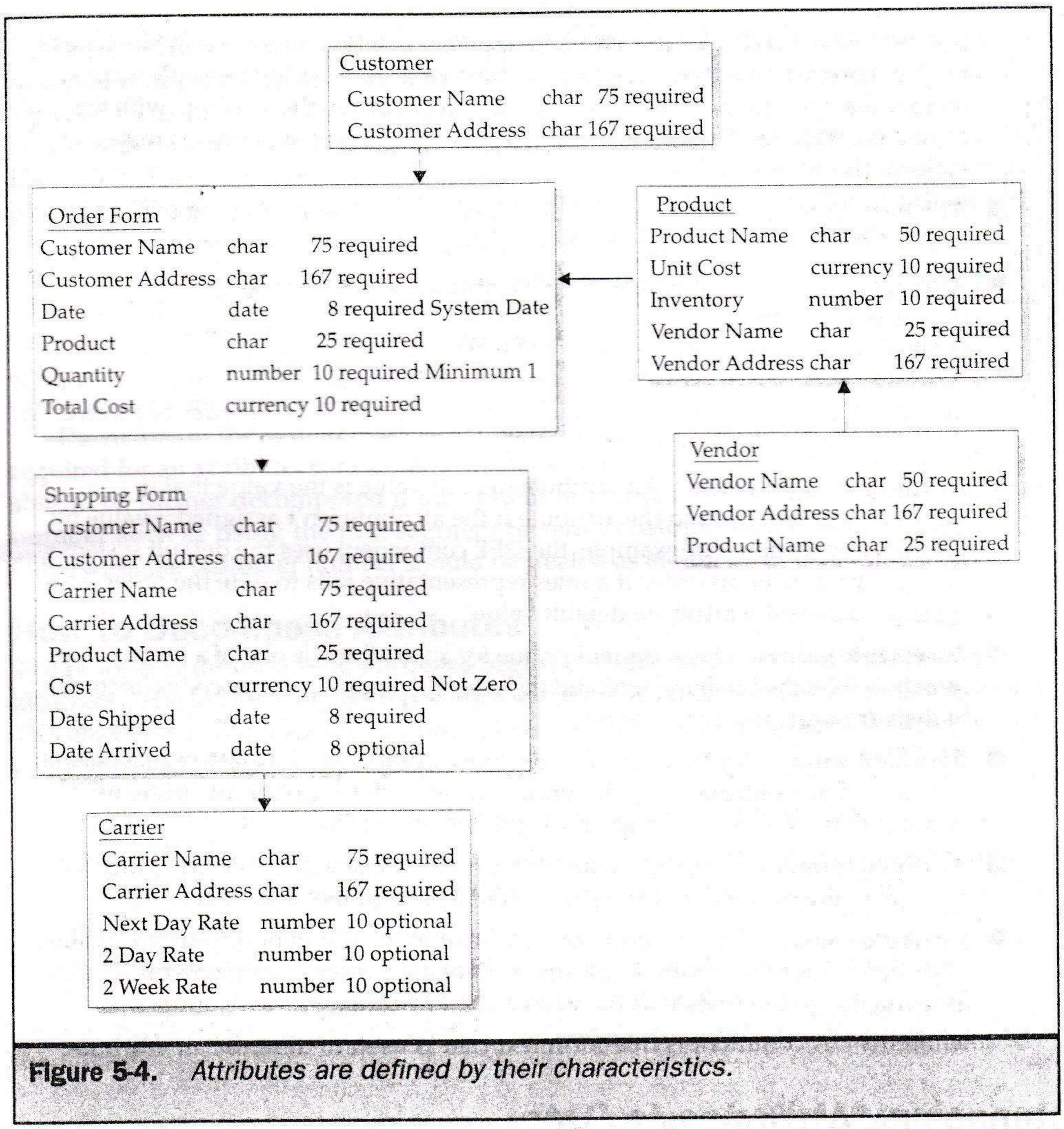


Figure 5-4. *Attributes are defined by their characteristics.*

Duplicate attribute names within the same entity are prohibited. However, two entities can use the same attribute name. That is, the customer entity and the sales representative entity both can have an attribute called first name.

- **Attribute type** An attribute type is nearly identical to the data type of a column in a table. Common attribute types include numeric, character, alphanumeric, date, time, Boolean, integer, float, and double, among other attribute types. However, unlike a data type for a column, you do not have to use precise terminology to describe an attribute type. That is, you might call an attribute type "sales in

dollars," which is not a valid Java data type but is sufficient to convey the type of information assigned to the entity. It is also advisable to include sample values that are assigned to the entity. This enables you to match the attribute with the most advantageous data type for the DBMS you are using when you create a table for the system.

- **Attribute size** The attribute size describes the number of characters used to store values of the attribute. This is similar to the size of a column in a table.
- **Attribute range** An attribute range is minimum and maximum values that can be assigned to an attribute. For example, the value of the "total amount" attribute of an order entity is likely to be greater than zero and less than 10,000 assuming that no order has ever been received that had a total amount of more than 9,999. This range is then used to throw an error should an order be received with a total amount outside of this range.
- **Attribute default value** An attribute default value is the value that is automatically assigned to the attribute if the attribute isn't assigned a value by the J2EE application. For example, the J2EE component uses the default system's date for the date of an order if a sales representative fails to date the order. The system's date is the attribute default value.
- **Acceptable values** An acceptable value for an attribute is one of a set of values established by the business unit and includes ZIP codes, country codes, methods of delivery, and simply yes or no.
- **Required value** An attribute may require a value before the attribute is saved to a table. For example, an order entity has an order number attribute, which requires that an order number be assigned to the attribute.
- **Attribute format** The attribute format consists of the way in which an attribute appears in the existing system, such as the format of the data.
- **Attribute source** The attribute source identifies the origin of the attribute value. Common sources are from data entry and the J2EE component generated such as using the system's date as the value of the attribute.
- **Comments** A comment is free-form text that is used to describe an attribute.

Decomposing Attributes to Data

Once attributes of entities are identified, attributes must be reduced to data elements. This process is called decomposing. For the most part, decomposing is an intuitive process because you can easily recognize whether or not an attribute is already at an atomic level, such as is illustrated in the customer name and customer address attributes. Both attributes are not atomic, however atomic level is obvious—first name, last name, city, state, and ZIP code.

However, decomposing other attributes might be less intuitive. For example, should a customer number attribute be decomposed? At first glance, the response is no because typically a customer number is already atomic.

Consider the following customer number that consists of three numbered segments: 12-24-1001. The first segment (12) represents the sales region within which the customer is located. The second segment (24) is the branch within the sales region that handles relationships with the customer. And the final segment (1001) is the number that identifies the customer within the branch and region.

Since the customer number is in three segments, should the customer number attribute be decomposed into each segment? This isn't an easy question to answer since in some systems it makes sense to further decompose the customer number attribute and in other systems no further decomposition of the customer number is necessary.

The nature of the system will determine whether or not additional decomposition is required for an attribute. For example, in this scenario the customer number probably should be further decomposed if the system references each segment of the customer number such as using the first segment to select customers of a particular region. Otherwise, the customer number should be treated as atomic.

How to Decompose Attributes

The process of decomposing attributes begins by analyzing the list of entities and its attributes. The list of attributes represents all the information used by the existing system. The objective is to reduce each attribute to a list of data that represents the atomic level of the attribute. Here's how to do this:

1. Look at each attribute and ask yourself if the attribute is atomic.
2. If the attribute isn't atomic, the attribute must be further decomposed. Create a list of data derived from the attribute below the attribute, as illustrated in Figure 5-5.
3. If the attribute is atomic, no further decomposition is necessary for that attribute. Place the name of the attribute on the data list.
4. Review the data list developed in step 2 and repeat the decomposition process until all attributes are atomic.

Tools for Analysis

Computer-aided software engineering (CASE) tools are designed to automate the process of analyzing a system. Oracle Designer/2000, Rational Rose, and Together J are three of the better CASE tools on the market.

A CASE tool transforms basic information that is entered into a CASE tool into entities, attributes, data, and relationships among entities. CASE tools are beyond the scope of this book. Visit the manufacturers' web sites for more information about CASE tools.

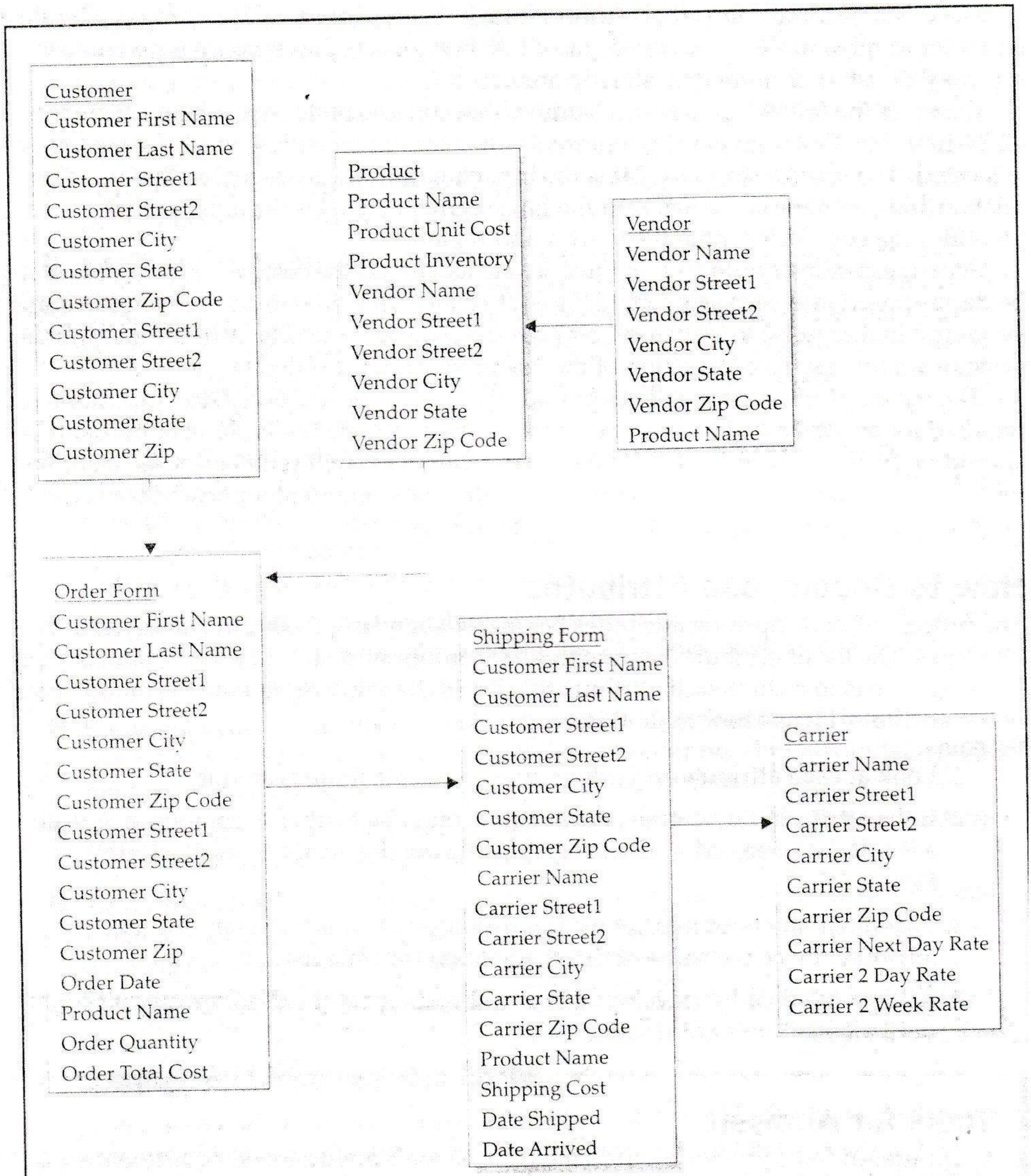


Figure 5-5. *Data elements for the customer name attribute*

Decomposing by Example

Let's work through an example of decomposing attributes using the customer address attribute of a customer entity. The following are attributes for the customer entity:

- Customer number
- Customer name
- Customer address
- Customer telephone number

Once the list of attributes is assembled, each attribute on the list must be decomposed. The customer address attribute is decomposed in this example. The same process can be applied to the other attributes.

Review the customer address attribute to determine if the attribute is atomic, which it isn't. Therefore, the customer address attribute must be decomposed into the following data elements:

- Street address 1
- Street address 2
- City
- State
- Country
- Country code
- Postal code
- Address type (that is, home or business)

Defining Data

Decomposing attributes results in the identification of data elements used by the existing system. Each data element must be defined using techniques similar to those used to describe an attribute. Here are common ways to define a data element:

- **Data name** The unique name given to the data element, which should reflect the kind of data (see "The Art of Choosing a Name" section).
- **Data type** A data type describes the kind of values that are associated with the data (see the "Data Types" section).
- **Data size** The size of text data is the maximum number of characters required to represent values of the data. The size of numeric data is usually either the number of digits or the number of bytes for binary representation (that is, smallint in DB2 is 2 bytes).

The Art of Choosing a Name

Picking a name for a data element might seem intuitive, but it can easily become tricky when you realize the name must describe the data, it must be unique, and it may experience size and character limitations depending on the DBMS.

The nature of the data provides a hint to the data name, such as "first name" being used as the name for data that is associated with a person's first name. However, sometimes the obvious choice isn't the best choice. For example, intuitively one might select "telephone number" as the name of data that is associated with a telephone number. However, an entity such as a customer might have more than one telephone number, such as a home telephone number and business telephone number. This means the name of the data element that contains the telephone number must reflect that kind of telephone number, too.

Another twist in selecting a name comes when more than one entity contains similar data. In this example, there could be a customer entity and a sales representative entity, both of which have home and business telephone numbers. The solution is to use the name of the entity as part of the data element name, such as customer home telephone number and customer business telephone number since this clearly identifies the data.

Keep the length of the name reasonable. The names of data elements typically become column names in a table. Some DBMSs restrict the length of column names and the types of characters that can be used as a column name.

Names of data should have as few characters as possible to identify the data. There are two reasons for this. First, the data name will probably conform to any restrictions imposed by the DBMS. Second, a programmer won't need to type long column names in when interacting with the database.

A data name can be abbreviated using components of the name. For example, customer home telephone number can be shortened to cust home phone.

When abbreviating data names, make sure the same style of abbreviation is used for naming data. This means that cust is the abbreviation for customer and should be used in other data of the customer entity (that is, cust bus phone).

Be aware that some DBMSs prohibit spaces in column names. You can avoid this problem by removing spaces in the data name. However, doing so might make the name unreadable. Readability can be enhanced by capitalizing the first letter of each word in the name or separating these words with hyphens or underlines, such as

- CustHomePhone
- cust-home-phone
- cust_home_phone

Data Types

As previously mentioned in this chapter, a data type describes the characteristics of data associated with a data element. For example, a street address is likely to be an alphanumeric data type because a street address has a mixture of characters and numbers.

It is important to use care when selecting the data type of a data element at this stage in the analysis because the data type that you choose typically becomes the data type of the column in the table that contains the data.

Where possible, limit the choice of data types to those that are common to commercial DBMSs. Not all DBMSs use the same data type. Some DBMSs enhance the standard data type offering.

However, if you are unsure of the data type to describe a data element, then describe the type of data in your own words and include an example of data values that are associated with the data element. You can refine your choice to available data types in the DBMS when you create the table used to store the data element.

Many commercially available DBMSs adopted a common set of data types that center on the SQL set of data types, which you'll learn about in detail in the next chapter. However, some of these are listed here:

- **Character** Also referred to as text. Stores alphabetical characters and punctuation.
- **Alpha** Stores only alphabetical characters.
- **Alphanumeric** Stores alphabetical characters, punctuation, and numbers.
- **Numeric** Stores numbers only.
- **Date/Time** Stores date and time values.
- **Logical (Boolean)** Stores one of two values—true or false, 0 or 1, or yes or no.
- **LOB (Large object)** Stores large text fields, images, and other binary data.

Normalizing Data

Normalization is the process of organizing data elements into related groups to minimize redundant data and to assure data integrity. Redundant data elements occur naturally since multiple entities have the same data elements. For example, an order form and invoice are both entities that contain a customer name and address. Therefore, customer name and address are redundant.

For transactional databases, redundant data makes a database complex and inefficient, and exposes the database to problems referred to as anomalies when the DBMS maintains the database. Anomalies occur whenever new data is inserted into the database and when existing data is either modified or deleted and can lead to a breach in referential integrity of the database (see "Referential Integrity" later in this chapter). However, for reporting data, the redundancy rules are bent a little because redundant data is more efficient since it minimizes the number of joins and allows data to be summarized into logical groups.

Errors caused by redundant data are greatly reduced, and possibly eliminated, by applying the normalization process to the list of data elements that describe all the entities in a system. This is called normalizing the logical data model of a system.

The normalization process consists of applying a series of rules, called *normal forms*, to the list of data elements to

- Remove redundant data elements.
- Reorganize data elements into groups.
- Define one data element of the group (called a primary key) to uniquely identify the group. Often, two or more data elements make up the primary key, which is referred to as a composite key.
- Make other data elements of the group (called nonkey data elements) functionally dependent on the primary key.
- Relate one group to another using the primary key.

For example, a customer number is the primary key of a group that contains customer information. Other data contained in the group such as the customer first name and last name are referred to as nonkey data elements. Nonkey data elements are functionally dependent on the primary key. That is, a customer name, address, and related information cannot exist in the customer group without being assigned a customer number.

The Normalization Process

There are five normal forms. However, many industry leaders conclude that the fourth and fifth normal forms are difficult to implement and unnecessary. Therefore, we'll confine ourselves to the first three normal forms:

- First normal form (1NF) requires that information be atomic, as discussed previously in this chapter.
- Second normal form (2NF) requires data to be in the first normal form. In addition, data elements are organized into groups, eliminating redundant data. Each group contains a primary key and nonkey data, and nonkey data must be functionally dependent on a primary key.
- Third normal form (3NF) requires that data elements be in the second normal form and nonkey data must not contain transitive dependencies.

Grouping Data

A common way to organize data elements into groups is to first assemble a list of all data elements, as discussed previously in this chapter. When this is done, you'll notice that some data elements are duplicated because they are used by more than one entity. Duplicate data elements must be removed from the list. Although this is an intuitive process, you must be careful because not all data elements with similar sounding names are duplicates.

Let's say there are two data elements, one called zip code and the other postal code. At first glance these appear to have the same meaning. A ZIP code is another term for postal code—or is it?

A ZIP code is a specific kind of postal code used in the United States. A postal code is a general term that also applies to postal codes used by countries other than the United States.

The difference is subtle, but could have an impact if you assume zip code and postal code are the same. For example, many relational databases contain a table of ZIP codes where only valid ZIP codes are stored. The table also contains the city and state that corresponds to a ZIP code.

Postal code data may or may not be contained in a relational database table since it could be difficult to identify all the postal codes from every country—and maintain this list as postal codes are modified.

Figure 5-6 contains a list of all data elements before redundant data elements are removed. Figure 5-7 shows the same list after redundant data is removed. Notice that the number of data elements on the list has been dramatically reduced. Figure 5-8 organizes related data into groups.

Creating Primary Keys

As discussed previously in this chapter, a primary key is a data element that uniquely identifies a row of data elements within a group. The data selected to become the primary key may or may not exist in the data list generated as the results of analyzing entities. Sometimes a data element such as an order number is used as the primary key. Other times, the DBMS can be requested to automatically generate a primary key whenever a column in the group isn't suitable to be designated the primary key. Figure 5-8 indicates the primary key of each group with an asterisk.

Let's use a customer entity as an illustration. In the purest sense, a customer has a name and address as attributes. These attributes decompose to first name, last name, street, city, state, and ZIP code. However, none of these data elements are suited to become a primary key because individually and collectively none uniquely identify a customer.

Intuitively, the customer first name and last name seem to uniquely identify a customer, but upon closer analysis you'll see that more than one customer might have the same first name and last name. Likewise, there might be two people at the same address having the same name, although somewhat unlikely.

If neither a single data element nor a combination of data elements uniquely identifies a row, then you must create another data element to serve as the primary key of the table, which is what is required in the previous example. Alternatively you can request the DBMS automatically generate a primary key.

Nearly all commercial DBMSs can generate primary keys to make the database thread safe and reliable. In contrast, a J2EE component that generates a key must contain the logic to be sure that none of the components running on different servers accidentally generate the same key.

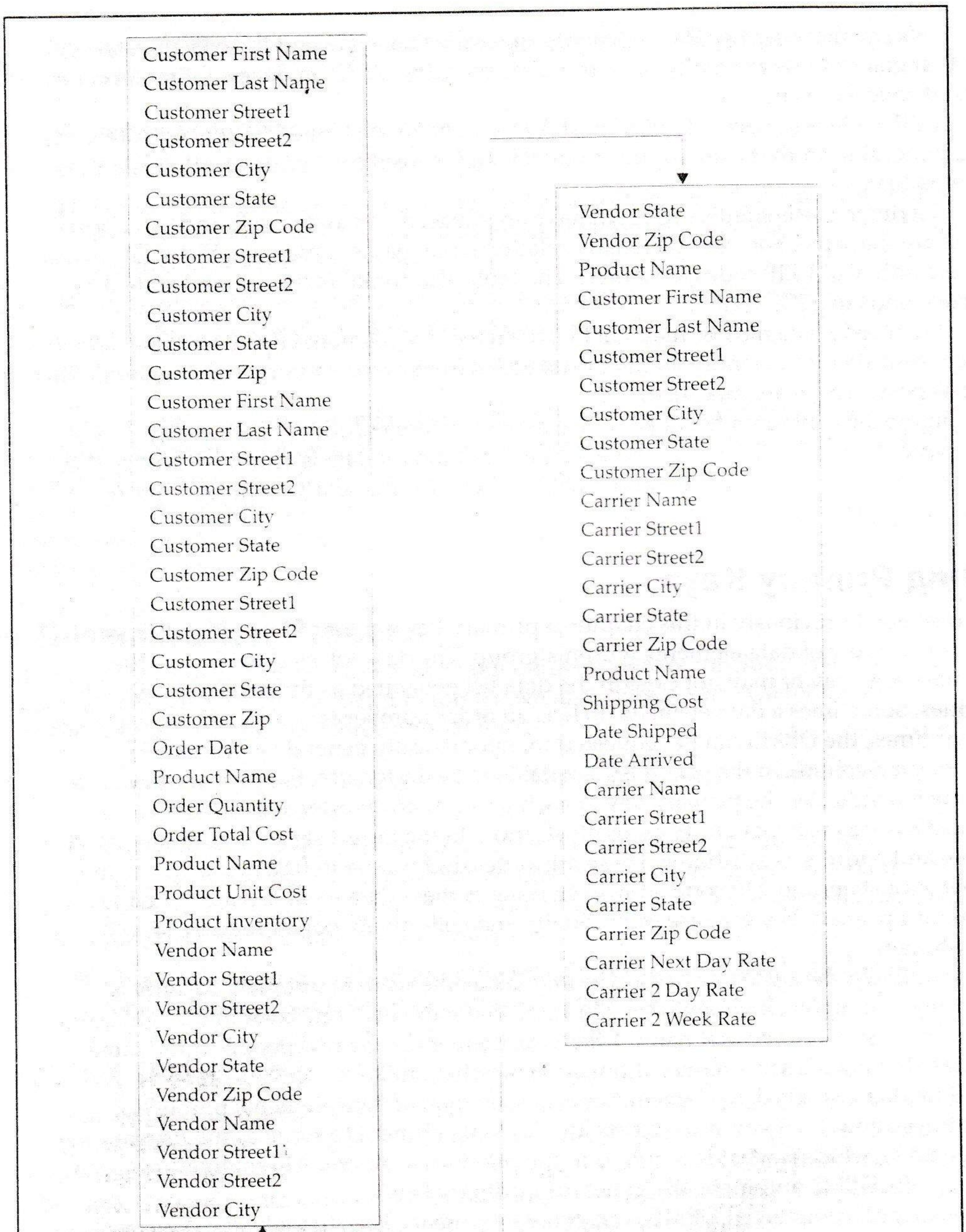


Figure 5-6. *Here is a list of data elements from the entities shown in Figure 5-5.*

Customer First Name
Customer Last Name
Customer Street1
Customer Street2
Customer City
Customer State
Customer Zip Code
Order Date
Product Name
Order Quantity
Order Total Cost
Product Unit Cost
Product Inventory
Vendor Name
Vendor Street1
Vendor Street2
Vendor City
Vendor State
Vendor Zip Code
Carrier Name
Carrier Street1
Carrier Street2
Carrier City
Carrier State
Carrier Zip Code
Shipping Cost
Date Shipped
Date Arrived
Carrier Next Day Rate
Carrier 2 Day Rate
Carrier 2 Week Rate

Figure 5-7. *Here is the list of data elements shown in Figure 5-6 after redundant data is removed.*

Functional Dependency

A functional dependency occurs when data depends on other data such as when nonkey data is dependent on a primary key. This means that all nonkey data has a functional dependency on the primary key within its group.

For example, order product quantity in the Orders group cannot exist unless there is an order number. The order number is the primary key of the group and the order

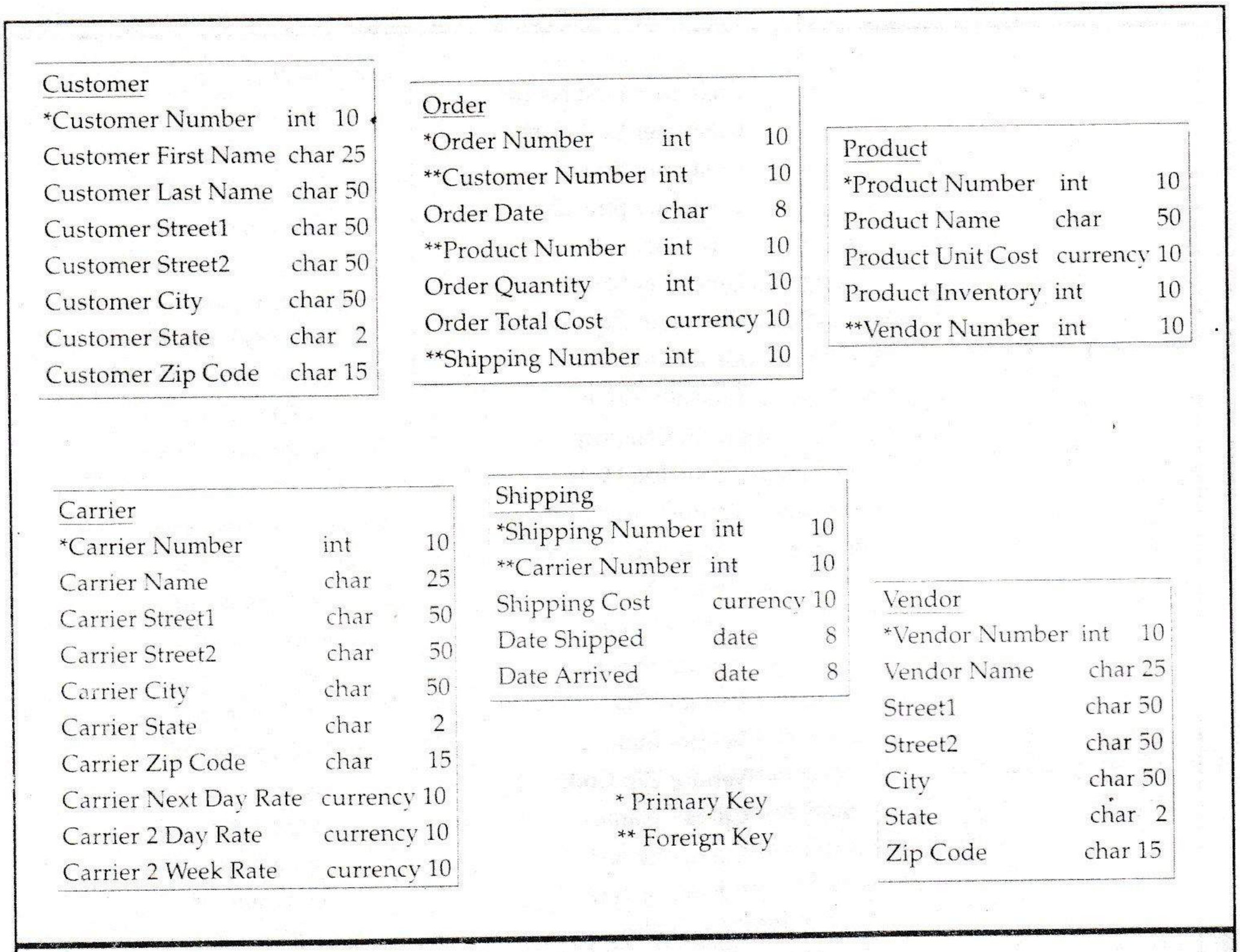


Figure 5-8. *Related data are organized into groups.*

product quantity is the nonkey data. You can say that order product quantity is functionally dependent on order number. This is noted by the expression order

```
product quantity -> order number.
```

Transitive Dependencies

A transitive dependency is a functional dependency between two or more nonkey data elements. This is difficult to understand at first, but an example will clearly illustrate transitive dependency.

The first grouping in Figure 5-9 shows the order entity with two data elements. These are sales rep number and region, which is the region to which the sales representative is assigned. Salesperson and region have a transitive dependency. The region is functionally dependent on the salesperson and the salesperson is functionally dependent on the order number. Since both salesperson and region are nonkey data, they are therefore functionally mutually dependent.

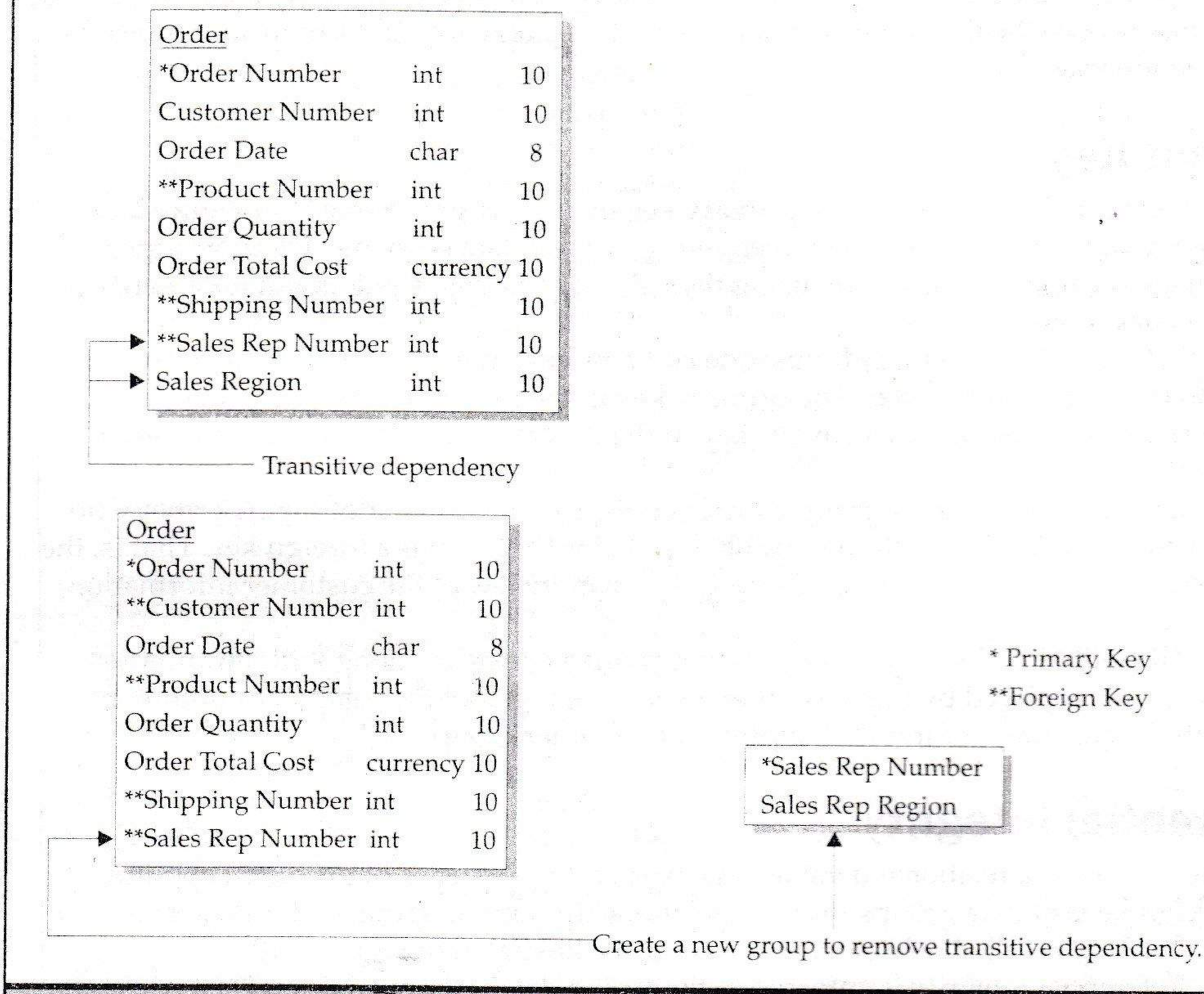


Figure 5-9. *Sales rep number and region (top) form a transitive dependency. Regrouping data elements (see bottom) resolves the transitive dependency problem.*

The problem lies with the fact that a salesperson cannot be relocated to a different region without having to modify the region data element in the order information group. Therefore, data elements must be regrouped to conform to the third normal form and eliminate transitive dependency.

The second grouping in Figure 5-9 illustrates the regrouping of the order entity to address the transitive dependency problem. Notice that a new group is formed that contains the sales rep number and the region.

The sales rep number and the region can be used as a composite key. A composite key, as you recall from an earlier discussion in this chapter, is a primary key that consists of two data elements.

In this example, the sales rep number and the region can be joined to form a composite key. If a sales representative is permitted to make sales in multiple regions, the composite key is used to associate the sales representative with one of many regions and a region to one of many sales representatives.

Identifying transitive dependencies is tricky. You have to carefully analyze the data elements once the list of data elements is in the second normal form to spot transitive dependencies.

Foreign Key

As you recall, a foreign key is a primary key of another group (see ** in Figure 5-8) used to draw a relationship between two groups of data elements. Relationships between two groups are made using the value of a foreign key and not necessarily the name of a foreign key.

Let's say there are two groups: one contains customer information and the other contains order information. The primary key in the customer information group is the customer number and the primary key in the order information group is the order number.

Each row in the order group contains the customer number of the customer who placed the order. The customer number in the order group is a foreign key. That is, the customer number in the order group is the primary key of the customer information group.

The DBMS is able to join information about a customer, along with information about orders placed by that customer, by joining together the customer number in both the customer information group and the order group.

Referential Integrity

The success of a relational database is based on the existence of primary keys and foreign keys of data groups to create relationships among groups. The existence of this relationship is called referential integrity, as is illustrated in Figure 5-10.

Referential integrity is enforced by imposing database constraints. This means the DBMS assures referential integrity by preventing primary and foreign keys from being modified or deleted. Likewise, database constraints prevent new rows from being inserted without maintaining referential integrity.

The Art of Indexing

An index is used to quickly locate information in a table similar to how information is located in a book. However, instead of page numbers, an index references the row number in the table that contains the search criteria.

Conceptually an index is a table that has two columns. One column is the key to the index and the other column is the number of the row in the corresponding table that contains the value of the key.

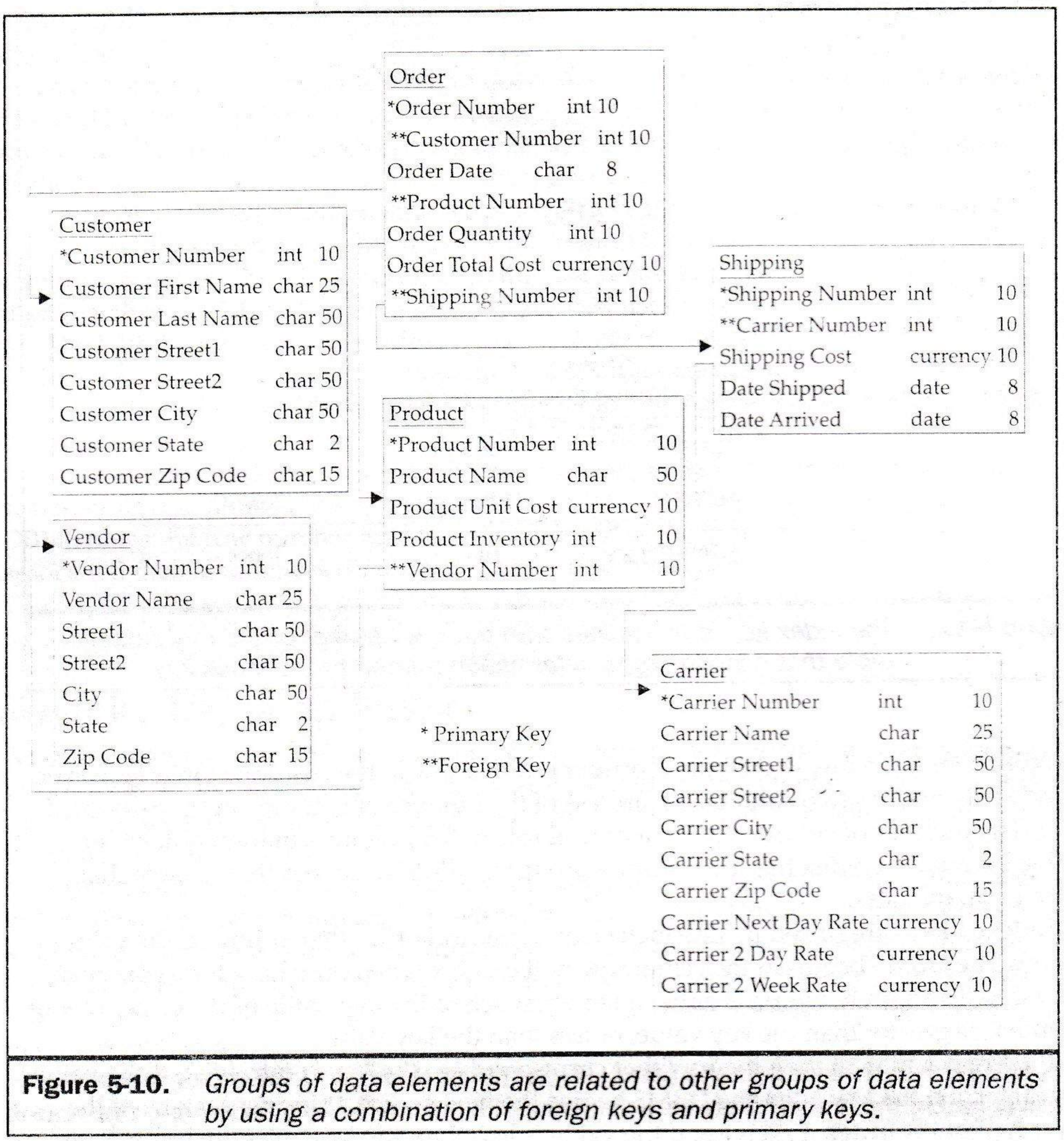


Figure 5-10. *Groups of data elements are related to other groups of data elements by using a combination of foreign keys and primary keys.*

Let's say that you created a table of customer information and use the customer number as the primary key to the table. Figure 5-11 contains such a table. An index is created using the primary key as the index key.

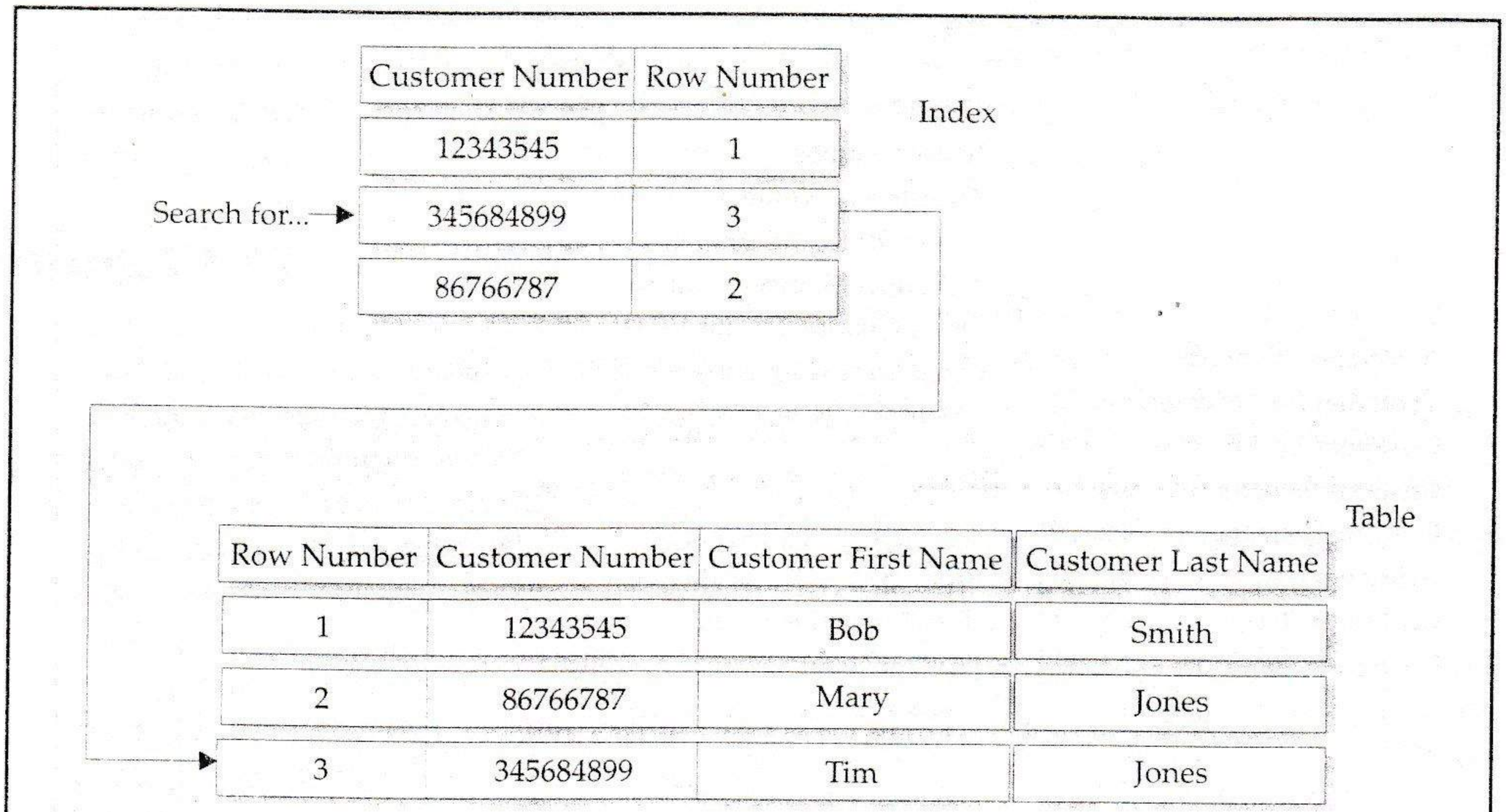


Figure 5-11. *The index key is associated with the row number of the row in the table that contains other information related to the index key.*

Notice that each row in the index corresponds to a row in the customer table. However, an index always has two columns regardless of the number of columns in the associated table. The number of columns plays a critical role in finding information quickly in a table because an index has less information for a DBMS to search than a table that contains all the data.

Indexed keys are sorted in alphabetical or numerical order depending on the value of the key. The DBMS begins at the center row of the index when searching for a particular key value. The search criteria is either an exact match to the key value of the center row of the index, is greater than the key value, or less than the key value.

If there is a match, then the row that corresponds to the key is retrieved. If the value is greater than the key, then the DBMS begins the next search at the center row of the lower half of the index. Likewise, if the value is less than the key, the search continues at the center row of the upper half of the index. The process is repeated until either a match is found or the search criteria cannot be found.

An Index in Motion

Unlike an index of a book, a table can be associated with multiple indexes each of which contains a different key. For example, a customer information table might have an index based on a ZIP code and another on a customer number. The designer of the database determines the number of indexes that are associated with a table, although

some DBMSs create their own indexes to speed searches if an appropriate index doesn't exist.

A J2EE component sends the DBMS a query that contains search criteria for information required by the component. Instead of searching the table that contains the search criteria, the DBMS compares the search criteria to keys of indexes, looking for an index to use in the search.

Let's say that a J2EE component needs to retrieve information about a customer. The request for information, called a query, is sent to the DBMS along with the customer number. The DBMS recognizes that the request contains a customer number, then searches a catalog of indexes for an index that uses customer number as its key.

The DBMS always uses an index, if one exists, to locate search criteria. If an index doesn't match the search criteria, then the DBMS either creates a temporary index as part of the search or sequentially searches the table. The method used with your DBMS depends on how the manufacturer designed the DBMS.

Once an index is located or a temporary index is created, the DBMS compares the search criteria contained in the request to the index key. When a match is found, the DBMS notes the row number in the second column of the index and then opens the associated table and moves directly to the row that corresponds to the row number. The DBMS then selects columns from the row that contains customer information requested by the J2EE component.

Drawbacks Using an Index

An index offers an unparalleled advantage for finding information in a table quickly. However, there is a drawback when too many indexes are used with one table. That is, an unacceptable delay can occur whenever a row is inserted into or deleted from the table.

Once an index is built, the DBMS is responsible for automatically maintaining the index whenever a row is inserted or deleted from the table. This means that each index associated with a table must be modified whenever a row is inserted or deleted from the table, which can cause performance degradation if a table is associated with many indexes. The tradeoff of using multiple indexes is the time necessary for the DBMS to maintain each index.

Performance degradation can be minimized by using a publisher-consumer database design for applications where rows are frequently inserted, deleted, or modified, such as in an order processing system.

The publisher-consumer database design consists of two or more databases that contain the same information. One database, called the publisher, receives a request from the J2EE component to insert a new row, or modify or delete an existing row. J2EE components don't use the publisher to retrieve information; therefore, the publisher database isn't indexed.

A consumer database receives instructions to insert a new row, or modify or delete an existing row from the publisher. However, the consumer receives requests for information from a J2EE component. Therefore, the consumer database is indexed.

Once the publisher passes data to the consumer, the publisher is free to process the next incoming data, during which time the DBMS is updating the consumer's indexes. There are two major benefits to this design.

First, a bottleneck is avoided when many requests are received by the DBMS to insert new information or update existing information. This is because the publisher has two tasks to perform: insert or modify the database, then pass along those changes to the consumer.

The other benefit is that the database becomes scalable. This means that the database can be adjusted to handle an increased volume of requests. Let's say a company begins with the publisher and one consumer. Over time there is an increase in the number of requests for information, which can overwhelm the DBMS and cause performance degradation. This can be remedied by creating a second consumer and routing requests to one or the other consumer, reducing response time by half.

Clustered Keys

A clustered key is an index key whose value represents data contained in multiple columns of the corresponding table. Although there is only one column in the index to store the index key, that value can be a combination of data from multiple columns of the table.

Let's say a customer name is used as the index key. The customer name is composed of two data elements: the customer first name and the customer last name. Since there is only one column for the index key, the customer first name and the customer last name are concatenated into one value. That is, the DBMS takes one data element and places it behind the other data element to create a new data element that becomes the index key.

Let's say that the customer first name is Tom and last name is Jones. Here's what they look like when they are concatenated: **TomJones**. Notice there isn't a space between the first and last name. This is because concatenating places the character of the second value immediately following the last character of the first value.

Clustered keys add overhead to a DBMS because columns are delimited and the delimiter must be escaped if the delimiter appears in the data. This extra step might impede performance of the DBMS.

The DBMS treats the index key of a clustered index as it does any index key and uses a combination of trees and hashing algorithms to locate search criteria in the index.

Concatenating data elements

- Create an index key that uses two or more columns to uniquely identify rows in a table.
- Facilitate searching for values of multiple columns such as a customer name using one index.

Derived Keys

A derived key consists of a value that represents part of the value of a column rather than the entire value of the column. Let's say that an order number is comprised of

three components. The first component represents the sales region within which the order was placed. The second component is the sales representative's number. And the last component is a unique number that identifies the order.

Although the order number appears in one column of the order table, a component of the order number can be used as an index key. For example, the DBMS can be instructed to derive an index key from the order number by using the first component. This means the index can be used to search for all orders placed within a region that is specified in the search criteria.

Selective Rows

Typically, all the rows in a table are represented in an index associated with that table. However, an index can be created that references a subset of rows in a table. The subset is determined when the index is created.

Let's return to the order number example used in the previous section to illustrate this feature. There might be tens of thousands of orders, each having a row in the order table and in indexes associated with the order table.

Suppose a J2EE component is used only to search for orders within a specified region. Never will there be an occasion for the component to search other than in the region. Indexes used for searches by the component can be limited to only rows of the table that contain orders placed within the region as identified by the first component of the order number. This means that the index does not contain any reference to rows in the order table that are outside the specified region.

There is a performance benefit that is realized by using an index that contains a subset of all the rows of the associated table. This is because rows that will never be searched are excluded from the index, thereby reducing the number of rows of the index that must be searched by the DBMS.

The boost in performance, however, is only realized in databases that contain huge numbers of rows. Little if any increase in performance is realized by creating a subset of rows in a typical database because many DBMSs have been optimized to search volumes of data without having to use a subset of rows in an index.

Using a conditional statement when creating an index, which you'll learn in the next two chapters, creates a subset. In the order number example, the conditional statement directs the DBMS to include rows in the index where the first component of the order number matches the conditional statement. All other rows are excluded from the index.

Exact Matches and Partial Matches

A DBMS can be instructed to use an index to find an exact or partial match to the search criteria. By default, the DBMS searches for exact matches whenever a query is received from a J2EE component. However, the programmer can construct the query to direct the DBMS to find partial matches, which you'll learn how to do in Chapter 7.

An exact match requires that all the characters of the search criteria match the index key. If a single character is mismatched, then the DBMS does not return data in the corresponding row of the table.

A partial match requires that some—not all—the characters of the index key match the search criteria. That is, if the first character of the search criteria and the index key are the same and the remaining characters are different, the DBMS stills considers it a match and returns data in the corresponding row of the table. Exact matches are used whenever a particular value and only that value is required, such as a specific customer number. Partial matches are used whenever someone is unsure of the exact value.

Let's say that a sales representative is looking for information about a customer, but she isn't sure if the customer last name is Johnson or Johnston. A search can be made using an index of customer names looking for any customer that has the first five letters of Johns and any other characters as their last name.

In this example, the customer name index is a concatenated index. The order in which values are concatenated plays a critical role when a DBMS searches for a partial match. This is because the DBMS begins matches left to right. Therefore, the most significant value must be placed first in the concatenated key.

The most significant value in the customer name is the customer last name because there are more people with the same first name than the same last name. Therefore, the index key is last name then first, such as **JohnstonMike**.

Searching for Phonetic Matches

Some DBMSs feature phonetic searches where the DBMS returns rows containing index keys that sound like the search value. This means that the DBMS stores both exact spelling and phonic spelling of the index key.

Phonetic searches are a valuable feature to look for in a DBMS, especially for use with databases used for customer service. Customer service typically must take bits and pieces of information provided by a customer, then assemble those pieces into meaningful search criteria.

Many times interactions with customers take place over the telephone where the customer service representative does not have access to printed materials (that is, invoices) that the customer has at hand.

This situation typically leads to miscommunication that aggravates a customer relationship. Although a phonetic search won't guarantee better communication between the customer service representative and the customer, it does give the customer service representative a tool with which to locate information necessary to properly respond to a customer.

Phonetic searches are made possible by an algorithm built in to the DBMS. The phonetic algorithm used by the DBMS defines each index key phonetically. Likewise, the DBMS converts the search criteria into its phonetic spelling before comparing the phonetic spelling of the search criteria to the phonetic spelling of the index key.

The Complete Reference



Chapter 6

JDBC Objects

Practically every J2EE application saves, retrieves, and manipulates information stored in a database using web services provided by a J2EE component. A J2EE component supplies database access using Java data objects contained in the JDBC application programming interface (API). Java data objects have methods that open a connection to a database management system (DBMS) and then transmit messages (queries) to insert, retrieve, modify, or delete data stored in a database.

The DBMS uses the same connection to send messages back to the J2EE component. These messages contain rows of data requested by the J2EE component or information indicating the status of the query being processed by the DBMS. Additional Java data objects are used to interact with data that is returned to the J2EE component by the DBMS.

This is the second of three chapters that focus on how to write a J2EE component to interact with a database. The previous chapter showed how to develop a database schema, which is the database design. This chapter shows how to use Java data objects to connect to the DBMS, send queries to the DBMS, and manipulate data returned by the DBMS. The next chapter shows how to use SQL to write queries that interact with DBMSs.

The Concept of JDBC

There are many industrial-strength DBMSs commercially available in the market. These include Oracle, DB2, Sybase, and many other popular brands. The challenge Sun Microsystems, Inc. faced in the late 1990s was to develop a way for Java developers to write high-level code that accesses all popular DBMSs.

One of the major obstacles for Sun Microsystems, Inc. to overcome was a language barrier. Each DBMS defined its own low-level way to interact with programs to access data stored in its databases. This meant low-level code written to communicate with an Oracle database might need to be rewritten to access a DB2 database.

Sun Microsystems, Inc. met the challenge in 1996 with the creation of the JDBC driver and the JDBC API. Both were created out of necessity, because until then Java wasn't considered an industrial-strength programming language since Java was unable to access DBMSs.

The JDBC driver developed by Sun Microsystems, Inc. wasn't a driver at all. It was a specification that described the detail functionality of a JDBC driver. DBMS manufacturers and third-party vendors were encouraged to build JDBC drivers that conformed to Sun Microsystems, Inc.'s specifications. Those firms that built JDBC drivers for their products could tap into the growing Java applications market.

The specifications required a JDBC driver to be a translator that converted low-level proprietary DBMS messages to low-level messages understood by the JDBC API, and vice versa.

This meant Java programmers could use high-level Java data objects defined in the JDBC API to write a routine that interacted with the DBMS. Java data objects convert the routine into low-level messages that conform to the JDBC driver specification and

send them to the JDBC driver. The JDBC driver translates the routine into low-level messages that are understood and processed by the DBMS.

JDBC drivers created by DBMS manufacturers have to

- Open a connection between the DBMS and the J2EE component.
- Translate low-level equivalents of SQL statements sent by the J2EE component into messages that can be processed by the DBMS.
- Return data that conforms to the JDBC specification to the JDBC driver.
- Return information such as error messages that conforms to the JDBC specification to the JDBC driver.
- Provide transaction management routines that conform to the JDBC specification.
- Close the connection between the DBMS and the J2EE component.

The JDBC driver makes J2EE components database independent, which complements Java's philosophy of platform independence. Today there are JDBC drivers for nearly every commercial DBMS, and they are available from the Sun Microsystems, Inc. web site (www.sun.com) or from the DBMS manufacturer's web site.

Java code independence is also extended to implementation of the SQL queries. SQL queries are passed from the JDBC API through the JDBC driver to the DBMS without validation. This means it is the responsibility of the DBMS to implement SQL statements contained in the query.

JDBC Driver Types

JDBC driver specification classifies JDBC drivers into four groups. Each group is referred to as a JDBC driver type and addresses a specific need for communicating with various DBMSs. The JDBC driver types are as follows:

Type 1 JDBC-to-ODBC Driver

Microsoft was the first company to devise a way to create a DBMS-independent database program when they created the Open Database Connection (ODBC). ODBC is the basis from which Sun Microsystems, Inc. created JDBC. Both ODBC and JDBC have similar driver specifications and an API. The JDBC-to-ODBC driver, also called the JDBC/ODBC Bridge, is used to translate DBMS calls between the JDBC specification and the ODBC specification. The JDBC-to-ODBC driver receives messages from a J2EE component that conforms to the JDBC specification as discussed previously in this chapter. Those messages are translated by the JDBC-to-ODBC driver into the ODBC message format, which is then translated into the message format understood by the DBMS. However, avoid using the JDBC/ODBC Bridge in a mission-critical application because the extra translation might negatively impact performance.

Type 2 Java/Native Code Driver

The Java/Native Code driver uses Java classes to generate platform-specific code—that is, code only understood by a specific DBMS. The manufacturer of the DBMS provides both the Java/Native Code driver and API classes so the J2EE component can generate the plattform-specific code. The obvious disadvantage of using a Java/Native Code driver is the loss of some portability of code. The API classes for the Java/Native Code driver probably won't work with another manufacturer's DBMS.

Type 3 JDBC Driver

The Type 3 JDBC driver, also referred to as the Java Protocol, is the most commonly used JDBC driver. The Type 3 JDBC driver converts SQL queries into JDBC-formatted statements. The JDBC-formatted statements are translated into the format required by the DBMS.

Type 4 JDBC Driver

Type 4 JDBC driver is also known as the Type 4 database protocol. This driver is similar to the Type 3 JDBC driver except SQL queries are translated into the format required by the DBMS. SQL queries do not need to be converted to JDBC-formatted systems. This is the fastest way to communicate SQL queries to the DBMS.

JDBC Packages

The JDBC API is contained in two packages. The first package is called `java.sql` and contains core Java data objects of the JDBC API. These include Java data objects that provide the basics for connecting to the DBMS and interacting with data stored in the DBMS. `java.sql` is part of the J2SE.

The other package that contains the JDBC API is `javax.sql`, which extends `java.sql` and is in the J2EE. Included in the `javax.sql` package are Java data objects that interact with Java Naming and Directory Interface (JNDI) and Java data objects that manage connection pooling, among other advanced JDBC features.

A Brief Overview of the JDBC Process

Although each J2EE component is different, J2EE components use a similar process for interacting with a DBMS. This process is divided into five routines. These include:

- Loading the JDBC driver
- Connecting to the DBMS
- Creating and executing a statement
- Processing data returned by the DBMS
- Terminating the connection with the DBMS

It is sometimes better to get a general understanding of how the process works before delving into the details of each routine of the process. Therefore, the next few sections of this chapter provide an overview of the process and each routine. A more detailed discussion of each routine is provided later in this chapter.

Loading the JDBC Driver

The JDBC driver must be loaded before the J2EE component can connect to the DBMS. The Class.forName() method is used to load the JDBC driver. Suppose a developer wants to work offline and write a J2EE component that interacts with Microsoft Access on the developer's PC. The developer must write a routine that loads the JDBC/ODBC Bridge driver called sun.jdbc.odbc.JdbcOdbcDriver. The driver is loaded by calling the Class.forName() method and passing it the name of the driver, as shown in the following code segment:

```
Class.forName( "sun.jdbc.odbc.JdbcOdbcDriver");
```

Connect to the DBMS

Once the driver is loaded, the J2EE component must connect to the DBMS using the DriverManager.getConnection() method. The java.sql.DriverManager class is the highest class in the java.sql hierarchy and is responsible for managing driver information.

The DriverManager.getConnection() method is passed the URL of the database, and the user ID and password if required by the DBMS. The URL is a String object that contains the driver name and the name of the database that is being accessed by the J2EE component.

The DriverManager.getConnection() method returns a Connection interface that is used throughout the process to reference the database. The java.sql.Connection interface is another member of the java.sql package that manages communication between the driver and the J2EE component. It is the java.sql.Connection interface that sends statements to the DBMS for processing. Listing 6-1 illustrates the use of the DriverManager.getConnection() method to load the JDBC/ODBC Bridge and connect to the CustomerInformation database.

Listing 6-1 Open a connection with a database.

```
String url = "jdbc:odbc:CustomerInformation";
String userID = "jim";
String password = "keogh";
Statement DataRequest;
private Connection Db;
try {
   Class.forName( "sun.jdbc.odbc.JdbcOdbcDriver");
   Db = DriverManager.getConnection(url,userID,password);
 }
```

Create and Execute a SQL Statement

The next step, after the JDBC driver is loaded and connection is successfully made with a particular database managed by the DBMS, is to send a SQL query to the DBMS for processing. A SQL query consists of a series of SQL commands that direct the DBMS to do something such as to return rows of data to the J2EE component. You'll learn how to write queries in the next chapter.

The Connect.createStatement() method is used to create a Statement object. The Statement object is then used to execute a query and return a ResultSet object that contains the response from the DBMS, which is usually one or more rows of information requested by the J2EE component.

Typically, the query is assigned to a String object, which is passed to the Statement object's executeQuery() method, which is illustrated in the next code segment. Once the ResultSet is received from the DBMS, the close() method is called to terminate the statement. Listing 6-2 retrieves all the rows and columns from the Customers table.

Listing 6-2 Retrieves all the rows from the Customers table.

```
Statement DataRequest;
ResultSet Results;
try {
      String query = "SELECT * FROM Customers";
      DataRequest = Database.createStatement();
      DataRequest = Db.createStatement();
      Results = DataRequest.executeQuery (query);
      DataRequest.close();
 }
```

Process Data Returned by the DBMS

The java.sql.ResultSet object is assigned the results received from the DBMS after the query is processed. The java.sql.ResultSet object consists of methods used to interact with data that is returned by the DBMS to the J2EE component.

Later in this chapter you'll learn the details of using the java.sql.ResultSet object. However, the following code is an abbreviated example that gives you a preview of a commonly used routine used to extract data returned by the DBMS. Error-catching code is purposely removed from this example in order to minimize code clutter. You'll find the completed version of this routine later in this chapter and throughout Chapter 7.

Assume for Listing 6-3 that a J2EE component requested customers' first names and last names from a table. The result returned by the DBMS is already assigned to the ResultSet object called Results. The first time that the next() method of the ResultSet is called, the ResultSet pointer is positioned at the first row in the ResultSet and returns a boolean value that if false indicates that no rows are present in the ResultSet. The if statement in Listing 6-3 traps this condition and displays the "End of data" message on the screen.

However, a true value returned by the next() method means at least one row of data is present in the ResultSet, which causes the code to enter the do...while loop. The getString() method of the ResultSet object is used to copy the value of a specified column in the current row of the ResultSet to a String object. The getString() method is passed the name of the column in the ResultSet whose content needs to be copied, and the getString() method returns the value from the specified column.

You could also pass the number of the column to the getString() method instead of passing the column number. However, do so only if the columns are specifically named in the SELECT statement; otherwise, you cannot be sure of the order in which the columns appear in the ResultSet, especially because the table might be reorganized since the table was created and therefore the columns might be rearranged.

In Listing 6-3, the first column of the ResultSet contains the customer's first name and the second column contains the customer's last name. Both of these are concatenated in this example and assigned to the printrow String object, which is displayed on the screen. This process continues until the next() method, called as the conditional argument to the while statement, returns a false, which means the pointer is at the end of the ResultSet.

Listing 6-3 Retrieving data from the ResultSet.

```
ResultSet Results;
String FirstName;
String LastName;
String printrow;
boolean Records = Results.next();
if (!Records ) {
   System.out.println( "No data returned");
   return;
}
else
{
 do {
   FirstName = Results.getString (FirstName) ;
   LastName = Results.getString (LastName) ;
   printrow = FirstName + " " + LastName;
   System.out.println(printrow);
 } while ( Results.next() );
}
```

Terminate the Connection to the DBMS

The connection to the DBMS is terminated by using the close() method of the Connection object once the J2EE component is finished accessing the DBMS. The close() method throws an exception if a problem is encountered when disengaging the DBMS. You'll learn how to handle this exception later in this chapter. The following is an example of

calling the close() method. Although closing the database connection automatically closes the ResultSet, it is better to close the ResultSet explicitly before closing the connection.

```
Db..close();
```

Database Connection

A J2EE component does not directly connect to a DBMS. Instead, the J2EE component connects with the JDBC driver that is associated with the DBMS. However, before this connection is made, the JDBC driver must be loaded and registered with the DriverManager as mentioned previously in this chapter.

The purpose of loading and registering the JDBC driver is to bring the JDBC driver into the Java Virtual Machine (JVM). The JDBC driver is automatically registered with the DriverManager once the JDBC driver is loaded and is therefore available to the JVM and can be used by J2EE components.

The Class.forName() method, as illustrated in Listing 6-4, is used to load the JDBC driver. In this example, the JDBC/ODBC Bridge is the driver that is being loaded. You can replace the JDBC/ODBC Bridge with the appropriate JDBC driver for the DBMS being used in your J2EE application.

The Class.forName() method throws a ClassNotFoundException if an error occurs when loading the JDBC driver. Errors are trapped using the catch{} block whenever the JDBC driver is being loaded.

isting 6-4 Load the driver and catch any exceptions t might be wn during e process.

```
try {
   Class.forName( "sun.jdbc.odbc.JdbcOdbcDriver");
}
catch (ClassNotFoundException error) {
   System.err.println("Unable to load the JDBC/ODBC bridge." + error.getMessage());
   System.exit(1);
}
```

The Connection

After the JDBC driver is successfully loaded and registered, the J2EE component must connect to the database. The database must be associated with the JDBC driver, which is usually performed by either the database administrator or the systems administrator. Some students who are learning JDBC programming prefer to use Microsoft Access as the DBMS because the DBMS is usually available on the student's local computer. The "Associating the JDBC/ODBC Bridge with the Database" section shows how to associate the JDBC/ODBC Bridge with a Microsoft Access database.

The data source that the JDBC component will connect to is defined using the URL format. The URL consists of three parts. These are

- **jdbc** which indicates that the JDBC protocol is to be used to read the URL.
- **<subprotocol>** which is the JDBC driver name.
- **<subname>** which is the name of the database.

The connection to the database is established by using one of three getConnection() methods of the DriverManager object. The getConnection() method requests access to the database from the DBMS. It is up to the DBMS to grant or reject access. A Connection object is returned by the getConnection() method if access is granted; otherwise, the getConnection() method throws a SQLException.

Sometimes the DBMS grants access to a database to anyone. In this case, the J2EE component uses the getConnection(String url) method. One parameter is passed to the method because the DBMS only needs the database identified. This is shown in Listing 6-5.

Listing 6-5 Connecting to a database using only the URL.

```
String url = "jdbc:odbc:CustomerInformation";
Statement DataRequest;
Connection Db;
try {
   Class.forName( "sun.jdbc.odbc.JdbcOdbcDriver");
   Db = DriverManager.getConnection(url);
}
catch (ClassNotFoundException error) {
      System.err.println("Unable to load the JDBC/ODBC bridge." + error);
      System.exit(1);
}
catch (SQLException error) {
      System.err.println("Cannot connect to the database." + error);
      System.exit(2);
}
```

Other databases limit access to authorized users and require the J2EE to supply a user ID and password with the request to access the database. In this case, the J2EE component uses the getConnection(String url, String user, String password) method as illustrated in Listing 6-6.

Listing 6-6 Connecting to a database using a user ID and password.

```
String url = "jdbc:odbc:CustomerInformation";
String userID = "jim";
String password = "keogh";
Statement DataRequest;
Connection Db;
try {
   Class.forName( "sun.jdbc.odbc.JdbcOdbcDriver");
```

```
    Db = DriverManager.getConnection(url,userID,password);
 }
catch (ClassNotFoundException error) {
      System.err.println("Unable to load the JDBC/ODBC bridge." + error);
      System.exit(1);
}
catch (SQLException error) {
      System.err.println("Cannot connect to the database." + error);
      System.exit(2);
}
```

There might be occasions when a DBMS requires information besides a user ID and password before the DBMS grants access to the database. This additional information is referred to as properties and must be associated with a Properties object, which is passed to the DBMS as a getConnection() parameter.

Typically, properties used to access a database are stored in a text file, the contents of which are defined by the DBMS manufacturer. The J2EE component uses a FileInputStream object to open the file and then uses the Properties object load() method to copy the properties into a Properties object. This is illustrated in Listing 6-7. Notice that the third version of the getConnection() method passes the Properties object and the URL as parameters to the getConnection() method.

ting 6-7 Using operties connect to the tabase.

```
Connection Db;
Properties props = new Properties ();
try {
FileInputStream propFileStream = new fileInputStream("DBProps.txt";);
props.load(propFileStream);
}
catch(IOException err) {
   System.err.print("Error loading propFile: ");
   System.err.println (err.getMessage());
   System.exit(1);
}
try {
   Class.forName( "sun.jdbc.odbc.JdbcOdbcDriver");
   Db = DriverManager.getConnection(url, props);
}
catch (ClassNotFoundException error) {
      System.err.println("Unable to load the JDBC/ODBC bridge." + error);
      System.exit(2);
}
catch (SQLException error) {
      System.err.println("Cannot connect to the database." + error);
      System.exit(3);
}
```

TimeOut

Competition to use the same database is a common occurrence in the J2EE environment and can lead to performance degradation of a J2EE application. For example, a J2EE application that needs database access requests service from an appropriate J2EE component. In turn, the J2EE component attempts to connect to the database.

However, the DBMS may not respond quickly for a number of reasons, which might include that database connections are not available. Rather than wait for a delayed response from the DBMS, the J2EE component can set a timeout period after which the DriverManager will cease to attempt to connect to the database.

The public static void DriverManager.setLoginTimeout(int seconds) method can be used by the J2EE component to establish the maximum time the DriverManager waits for a response from a DBMS before timing out.

Likewise, the public static int DriverManager.getLoginTimeout() method is used to retrieve from the DriverManager the maximum time the DriverManager is set to wait until it times out. The DriverManager.getLoginTimeout() method returns an int that represents seconds.

Associating the JDBC/ODBC Bridge with the Database

You use the ODBC Data Source Administrator to create the association between the database and the JDBC/ODBC bridge. Here's what you need to do:

1. Select Start | Settings, and then the Control Panel.
2. Select ODBC 32 to display the ODBC Data Source Administrator.
3. Add a new user by selecting the Add button.
4. Select the driver then select Finish. Use the Microsoft Access Driver if you are using Microsoft Access; otherwise, select the driver for the DBMS that you are using. If you don't find the driver for your DBMS on the list, you'll need to install the driver. Contact the manufacturer of the DBMS for more information on how to obtain the driver.
5. Enter the name of the database as the Data Source name in the ODBC Microsoft Access Setup dialog box. This is the name that will be used within your Java database program to connect to the DBMS.
6. Enter a description for the data source. This is optional, but will be a reminder of the kind of data that is stored in the database.
7. Click the Select button. You'll be prompted to browse the directory of each hard drive connected to your computer in order to define the direct path to the database. Click OK once you locate the database, and the directory path and the name of the database will be displayed in the ODBC Microsoft Access Setup dialog box.

8. Since this is your database, you can determine if a login name and password is required to access the database.
9. If so, then click the Advanced button to display the Set Advanced Options dialog box. This dialog box is used to assign a login name, also referred to as a user ID, and a password to the database. Select OK. If not, then skip this step.
10. When the ODBC Microsoft Access Setup dialog box appears, select OK.
11. Select OK to close the ODBC Data Source Administrator dialog box.

Connection Pool

Connecting to a database is performed on a per-client basis. That is, each client must open its own connection to a database and the connection cannot be shared with unrelated clients, which is a serous drawback in the J2EE environment.

For example, a client that needs to frequently interact with a database must either open a connection and leave the connection open during processing, or open or close and reconnect each time the client needs to access the database.

Leaving a connection open might prevent another client from accessing the database should the DBMS have available a limited number of connections. Connecting and reconnecting is simply time-consuming and causes performance degradation.

The release of the JDBC 2.1 Standard Extension API introduced connection pooling to address the problem. A connection pool is a collection of database connections that are opened once and loaded into memory so these connections can be reused without having to reconnect to the DBMS.

Clients use the DataSource interface to interact with the connection pool. The connection pool itself is implemented by the application server, which hides from the client details on how the connection pool is maintained.

There are two types of connections made to the database. The first is the physical connection, which is made by the application server using PooledConnection objects. PooledConnection objects are cached and reused.

The other type of connection is the logical connection. A logical connection is made by a client calling the DataSource.getConnection() method, which connects to a PooledConnection object that already has a physical connection to the database.

Listing 6-8 illustrates how to access a connection from a connection pool. A connection pool is accessible by using the Java Naming and Directory Interface (JNDI), which you'll learn about in Chapter 16. JNDI provides a uniform way to find and access naming and directory services independent of any specific naming or directory service.

First a J2EE component must obtain a handle to the JNDI context, which is illustrated in the first statement in this code segment. Next, the JNDI lookup() method is called and is passed the name of the connection pool, which returns the DataSource object called pool in this example. The getConnection() method of the DataSource object is then called, as illustrated earlier in this chapter. The getConnection() returns the logical connection to the database, which is used by the J2EE component to access the database.

The close() method of the DataSource object is called once when the J2EE component is finished accessing the database. The close() method closes the logical connection to the database and not the physical database connection. This means that the same physical connection can be used by the next J2EE component that needs access to the database.

isting 6-8
onnecting
to a
database
using
a pool
nnection.

```
Context ctext = new InitialContext();
DataSource pool = (DataSource)
ctext.lookup("java:comp/env/jdbc/pool");
Connection db = pool.getConnection();
// Place code to interact with the database here
db.close();
```


Statement Objects

Once a connection to the database is opened, the J2EE component creates and sends a query to access data contained in the database. The query is written using SQL, which you'll learn about in the next chapter.

One of three types of Statement objects is used to execute the query. These objects are Statement, which executes a query immediately; PreparedStatement, which is used to execute a compiled query; and CallableStatement, which is used to execute store procedures.

The Statement Object

The Statement object is used whenever a J2EE component needs to immediately execute a query without first having the query compiled. The Statement object contains the executeQuery() method, which is passed the query as an argument. The query is then transmitted to the DBMS for processing.

The executeQuery() method returns one ResultSet object that contains rows, columns, and metadata that represent data requested by query. The ResultSet object also contains methods that are used to manipulate data in the ResultSet, which you'll learn about later in this chapter.

The execute() method of the Statement object is used when there may be multiple results returned. A third commonly used method of the Statement object is the executeUpdate() method. The executeUpdate() method is used to execute queries that contain UPDATE and DELETE SQL statements, which changes values in a row and removes a row respectively. The executeUpdate() method returns an integer indicating the number of rows that were updated by the query. The executeUpdate() is used to INSERT, UPDATE, DELETE, and DDL statements.

Listing 6-9 is an enhanced version of Listing 6-2, used previously in this chapter to illustrate how to open a database connection. The enhancements are to create a query, execute the query, and return a ResultSet.

Two new objects are declared in Listing 6-9: a Statement object called DataRequest and a ResultSet object called Results. In the second try{ } block, the query is assigned to the String object query. The query requests the DBMS to return all the rows from the Customer table of the CustomerInformation database.

Next, the createStatement() method of the Connection object is called to return a Statement object. The executeQuery() method of the Statement object is passed the query and returns a ResultSet object that contains data returned by the DBMS. Finally, the close() method of the Statement object is called to close the statement.

The close() method closes all instances of the ResultSet object returned by the Statement. Failure to call the close() method might cause resources used by the Statement object to remain unavailable to other J2EE components until the garbage routine is automatically run. Java statements used to manipulate the ResultSet is placed between the call to the executeQuery() method and the close() method.

The executeQuery() method throws an SQLException should an error occur during the processing of the query. For example, the query may contain syntax not understood by the DBMS. In this case, the DBMS returns a SQL error message that is passed along to the J2EE component by the executeQuery() method.

Listing 6-9 Using the Statement object to execute a query.

```
String url = "jdbc:odbc:CustomerInformation";
String userID = "jim";
String password = "keogh";
Statement DataRequest;
ResultSet Results;
Connection Db;
try {
   Class.forName( "sun.jdbc.odbc.JdbcOdbcDriver");
   Db = DriverManager.getConnection(url,userID,password);
 }
catch (ClassNotFoundException error) {
      System.err.println("Unable to load the JDBC/ODBC bridge." + error);
      System.exit(1);
}
catch (SQLException error) {
      System.err.println("Cannot connect to the database." + error);
      System.exit(2);
}
try {
      String query = "SELECT * FROM Customers";
      DataRequest = Db.createStatement();
      Results = DataRequest.executeQuery (query);
      //Place code here to interact with the ResultSet
      DataRequest.close();
 }

catch ( SQLException error ){
     System.err.println("SQL error." + error);
```

```
        System.exit(3);
}
Db.close();
```

Listing 6-10 illustrates how to use the executeUpdate() method of the Statement object. You'll notice that Listing 6-10 is nearly identical to Listing 6-9. However, the query updates a value in the database rather than requesting that data be returned to the J2EE component. You'll learn more about how to write queries to update values in a database in the next chapter.

Three changes are made to Listing 6-9 to illustrate the executeUpdate() method of the Statement object. First, the declaration of the ResultSet object is replaced with the declaration of an int called rowsUpdated.

Next, the query is changed. The SQL UPDATE command directs the DBMS to update the Customers table of the CustomerInformation database. The value of the PAID column of the Customers table is changed to 'Y' if the value of the BALANCE column is zero.

Finally, the executeUpdate() method replaces the executeQuery() method and is passed the query. The number of rows that are updated by the query is returned to the executeUpdate() method by the DBMS and is then assigned to the rowsUpdated int, which can be used for many purposes within the J2EE component such as sending a confirmation notice to the J2EE application that requested database access.

Listing 6-10 Using the executeUpdate() method

```
String url = "jdbc:odbc:CustomerInformation";
String userID = "jim";
String password = "keogh";
Statement DataRequest;
Connection Db;
int rowsUpdated;
try {
   Class.forName( "sun.jdbc.odbc.JdbcOdbcDriver");
   Db = DriverManager.getConnection(url,userID,password);
 }
catch (ClassNotFoundException error) {
      System.err.println("Unable to load the JDBC/ODBC bridge." + error);
      System.exit(1);
}
catch (SQLException error) {
      System.err.println("Cannot connect to the database." + error);
      System.exit(2);
}
try {
      String query = "UPDATE Customers SET PAID='Y' WHERE BALANCE = '0';
      DataRequest = Db.createStatement();
      rowsUpdated = DataRequest.executeUpdate (query);
      DataRequest.close();
```

```
 }
catch ( SQLException error ){
     System.err.println("SQL error." + error);
      System.exit(3);
}
Db.close();
```

PreparedStatement Object

A SQL query must be compiled before the DBMS processes the query. Compiling occurs after one of the Statement object's execution methods is called. Compiling a query is an overhead that is acceptable if the query is called once. However, the compiling process can become an expensive overhead if the query is executed several times by the same instance of the J2EE component during the same session.

A SQL query can be precompiled and executed by using the PreparedStatement object. In this case, the query is constructed similar to queries that were illustrated previously in the chapter. However, a question mark is used as a placeholder for a value that is inserted into the query after the query is compiled. It is this value that changes each time the query is executed.

Listing 6-11 illustrates how to use the PreparedStatement object. Listing 6-11 is very similar to Listing 6-9 in the Statement object that returned information from the Customers table.

However, the query directs the DBMS to return all customer information where the customer number equals the customer number specified in the query. Notice that that query has a question mark as the value of the customer number. The question mark is a placeholder for the value of the customer number that will be inserted into the precompiled query later in the code.

The preparedStatement() method of the Connection object is called to return the PreparedStatement object. The preparedStatement() method is passed the query, which is then precompiled.

The setxxx() method of the PreparedStatement object is used to replace the question mark with the value passed to the setxxx() method. There are a number of setxxx() methods available in the PreparedStatement object, each of which specifies the data type of the value that is being passed to the setxxx() method (see the "Data Types" section later in this chapter). In Listing 6-11, the setString() method is used because the customer number is being passed as a string.

The setxxx() requires two parameters. The first parameter is an integer that identifies the position of the question mark placeholder and the second parameter is the value that replaces the question mark placeholder. In Listing 6-11, the first question mark placeholder is replaced with the value of the second parameter.

Next, the executeQuery() method of the PreparedStatement object is called. The executeQuery() statement doesn't require a parameter because the query that is to be executed is already associated with the PreparedStatement object.

The advantage of using the PreparedStatement object is that the query is precompiled once and the setxxx() method called as needed to change the specified values of the query without having to recompile the query. The PreparedStatement object also has an execute() method and an executeUpdate() method, as described in the previous section.

The precompiling is performed by the DBMS and is referred to as "late binding." When the DBMS receives the request, the DBMS attempts to match the query to a previously compiled query. If found, then parameters passed to the query using the setxxx() methods are bound and the query is executed. If not found, then the query is compiled and retained by the DBMS for later use.

The JDBC driver passes two parameters to the DBMS. One parameter is the query and the other is an array of late binding variables. Both binding and compiling is performed by the DBMS. The late binding is not associated with the specific object or code block where the preparedStatmenet() is declared.

Listing 6-11 Using the PreparedStatement object.

```
String url = "jdbc:odbc:CustomerInformation";
String userID = "jim";
String password = "keogh";
ResultSet Results;
Connection Db;
try {
   Class.forName( "sun.jdbc.odbc.JdbcOdbcDriver");
   Db = DriverManager.getConnection(url,userID,password);
 }
catch (ClassNotFoundException error) {
      System.err.println("Unable to load the JDBC/ODBC bridge." + error);
      System.exit(1);
}
catch (SQLException error) {
      System.err.println("Cannot connect to the database." + error);
      System.exit(2);
}
try {
      String query = "SELECT * FROM Customers WHERE CustNumber = ?";
      PreparedStatement pstatement = Db.preparedStatement(query);
      pstatement.setString(1, "123");
      Results = pstatement.executeQuery ();
      //Place code here to interact with the ResultSet
      pstatement.close();
 }
catch ( SQLException error ){
     System.err.println("SQL error." + error);
      System.exit(3);
}
Db.close();
```

CallableStatement

The CallableStatement object is used to call a stored procedure from within a J2EE object. A stored procedure is a block of code and is identified by a unique name. The type and style of code depends on the DMBS vendor and can be written in PL/SQL, Transact-SQL, C, or another programming language. The stored procedure is executed by invoking the name of the stored procedure.

The CallableStatement object uses three types of parameters when calling a stored procedure. These parameters are IN, OUT, and INOUT. The IN parameter contains any data that needs to be passed to the stored procedure and whose value is assigned using the setxxx() method as described in the previous section.

The OUT parameter contains the value returned by the stored procedures, if any. The OUT parameter must be registered using the registerOutParameter() method and then is later retrieved by the J2EE component using the getxxx() method.

The INOUT parameter is a single parameter that is used to both pass information to the stored procedure and retrieve information from a stored procedure using the techniques described in the previous two paragraphs.

Listing 6-12 illustrates how to call a stored procedure and retrieve a value returned by the stored procedure. Listing 6-12 is similar to other listings used in this chapter, but has been modified slightly to call a stored procedure.

The first statement in the second try{} block creates a query that calls the stored procedure LastOrderNumber, which retrieves the most recently used order number. The stored procedure requires one parameter that is represented by a question mark placeholder. This parameter is an OUT parameter that will contain the last order number following the execution of the stored procedure.

Next, the preparedCall() method of the Connection object is called and is passed the query. This method returns a CallableStatement object, which is called cstatement. Since an OUT parameter is used by the stored procedure, the parameter must be registered using the registerOutParameter() of the CallableStatement object.

The registerOutParameter() method requires two parameters. The first parameter is an integer that represents the number of the parameter, which is 1—meaning the first parameter of the stored procedure. The second parameter to the registerOutParameter() is the data type of the value returned by the stored procedure, which is Types.VARCHAR.

The execute() method of the CallableStatement object is called next to execute the query. The execute() method doesn't require the name of the query because the query is already identified when the CallableStatement object is returned by the prepareCall() query method.

After the stored procedure is executed, the getString() method is called to return the value of the specified parameter of the stored procedure, which in this example is the last order number.

Listing 6-12 Calling a stored procedure.

```
String url = "jdbc:odbc:CustomerInformation";
String userID = "jim";
```

```
String password = "keogh";
String lastOrderNumber;
Connection Db;
try {
   Class.forName( "sun.jdbc.odbc.JdbcOdbcDriver");
   Db = DriverManager.getConnection(url,userID,password);
 }
catch (ClassNotFoundException error) {
      System.err.println("Unable to load the JDBC/ODBC bridge." + error);
      System.exit(1);
}
catch (SQLException error) {
      System.err.println("Cannot connect to the database." + error);
      System.exit(2);
}
try {
      String query = "{ CALL LastOrderNumber (?)}";
      CallableStatement cstatement = Db.prepareCall(query);
      cstatement.registerOutParameter(1, Types. VARCHAR);
      cstatement.execute();
                        lastOrderNumber = cstatement.getString(1);
      cstatement.close();
 }
catch ( SQLException error ){
     System.err.println("SQL error." + error);
      System.exit(3);
}
Db.close();
```

ResultSet

As you'll remember from previous sections in this chapter, a query is used to update, delete, and retrieve information stored in a database. The executeQuery() method is used to send the query to the DBMS for processing and returns a ResultSet object that contains data that was requested by the query.

The ResultSet object contains methods that are used to copy data from the ResultSet into a Java collection object or variable for further processing. Data in a ResultSet object is logically organized into a virtual table consisting of rows and columns. In addition to data, the ResultSet object also contains metadata such as column names, column size, and column data type.

The ResultSet uses a virtual cursor to point to a row of the virtual table. A J2EE component must move the virtual cursor to each row and then use other methods of the ResultSet object to interact with the data stored in columns of that row.

The virtual cursor is positioned above the first row of data when the ResultSet is returned by the executeQuery() method. This means that the virtual cursor must be

moved to the first row using the next() method. The next() method returns a boolean true if the row contains data; otherwise, a boolean false is returned indicating that no more rows exist in the ResultSet.

Once the virtual cursor points to a row, the getxxx() method is used to copy data from the row to a collection, object, or variable. As illustrated previously in this chapter, the getxxx() method is data type specific. For example, the getString() method is used to copy String data from a column of the ResultSet. The data type of the getxxx() method must be the same data type as the column in the ResultSet.

The getxxx() method requires one parameter, which is an integer that represents the number of the column that contains the data. For example, getString(1) copies the data from the first column of the ResultSet.

Columns appear in the ResultSet in the order in which column names appeared in the SELECT statement in the query. Let's say a query contained the following SELECT statement: SELECT CustomerFirstName, CustomerLastName FROM Customer. This query directs the DBMS to return two columns. The first column contains customer first names and the second column contains customer last names. Therefore, getString(1) returns data in the customer first-name column of the current row in the ResultSet.

Reading The ResultSet

Listing 6-13 illustrates a commonly used routine to read values from a ResultSet into variables that can later be further processed by the J2EE component. Listing 6-13 is based on previous code segments in this chapter.

Once a successful connection is made to the database, a query is defined in the second try{} block to retrieve the first name and last name of customers from the Customers table of the CustomerInformation database.

The next() method of the ResultSet is called to move the virtual pointer to the first row in the ResultSet. If there is data in that row, the next() returns a true, which is assigned the boolean variable Records. If there isn't any data in that row, Records is assigned a false value. A false value is trapped by the if statement where the "End of data" message is displayed and the program terminates.

A true value causes the program to enter the do...while in the third try{} block where the getString() method is called to retrieve values in the first and second columns of the ResultSet, which corresponds to the first name and last name. These values are assigned to their corresponding String object, which is then concatenated and assigned the printrow String object and printed on the screen.

The next() method is called in the while statement to move the virtual cursor to the next row in the ResultSet and determine if there is data in that row. If so, statements within the do...while loop are executed again. If not, the program breaks out of the look and executes the close() statement to close the Statement object, as is discussed previously in this chapter.

Listing 6-13 Reading data from the ResultSet.

```
String url = "jdbc:odbc:CustomerInformation";
String userID = "jim";
```

```
String password = "keogh";
String printrow;
String FirstName;
String LastName;
Statement DataRequest;
ResultSet Results;
Connection Db;
try {
   Class.forName( "sun.jdbc.odbc.JdbcOdbcDriver");
   Db = DriverManager.getConnection(url,userID,password);
 }
catch (ClassNotFoundException error) {
      System.err.println("Unable to load the JDBC/ODBC bridge." + error);
      System.exit(1);
}
catch (SQLException error) {
      System.err.println("Cannot connect to the database." + error);
      System.exit(2);
}
try {
      String query = "SELECT FirstName,LastName FROM Customers";
      DataRequest = Db.createStatement();
      Results = DataRequest.executeQuery (query);
}
catch ( SQLException error ){
     System.err.println("SQL error." + error);
      System.exit(3);
}
boolean Records = Results.next();

if (!Records ) {
   System.out.println("No data returned");
   System.exit(4);
}
try {
  do {
     FirstName = Results.getString ( 1 ) ;
      LastName = Results.getString ( 2 ) ;

      printrow = FirstName + " " + LastName;
     System.out.println(printrow);

  } while (Results.next() );
  DataRequest.close();
}
catch (SQLException error ) {
   System.err.println("Data display error." + error);
   System.exit(5);
}
```

Scrollabie ResultSet

Until the release of the JDBC 2.1 API, the virtual cursor could only be moved down the ResultSet object. But today the virtual cursor can be moved backwards or even positioned at a specific row. The JDBC 2.1 API also enables a J2EE component to specify the number of rows to return from the DBMS.

There are six methods of the ResultSet object that are used to position the virtual cursor in addition to the next() method discussed in the previous section. These are first(), last(), previous(), absolute(), relative(), and getRow().

The first() method moves the virtual cursor to the first row in the ResultSet. Likewise, the last() method positions the virtual cursor at the last row in the ResultSet. The previous() method moves the virtual cursor to the previous row. The absolute() method positions the virtual cursor at the row number specified by the integer passed as a parameter to the absolute() method.

The relative() method moves the virtual cursor the specified number of rows contained in the parameter. The parameter is a positive or negative integer where the sign represents the direction the virtual cursor is moved.

For example, a -4 moves the virtual cursor back four rows from the current row. Likewise, a 5 moves the virtual cursor forward five rows from the current row. And the getRow() method returns an integer that represents the number of the current row in the ResultSet.

The Statement object that is created using the createStatement() of the Connection object must be set up to handle a scrollable ResultSet by passing the createStatement() method one of three constants. These constants are TYPE_FORWARD_ONLY, TYPE_SCROLL_INSENSITIVE, and TYPE_SCROLL_SENSITIVE.

The TYPE_FORWARD_ONLY constant restricts the virtual cursor to downward movement, which is the default setting. The TYPE_SCROLL_INSENSITIVE, and TYPE_SCROLL_SENSITIVE constants permit the virtual cursor to move in both directions. The TYPE_SCROLL_INSENSITIVE constant makes the ResultSet insensitive to changes made by another J2EE component to data in the table whose rows are reflected in the ResultSet. The TYPE_SCROLL_SENSITIVE constant makes the ResultSet sensitive to those changes.

Listing 6-14 illustrates how to reposition the virtual cursor in the ResultSet. This listing is a modification of the previous code segments used as examples in this chapter that retrieve customers' first names and last names from the Customers table of the CustomerInformation database.

Since Listing 6-14 moves the virtual cursor in multiple directions, the TYPE_SCROLL_INSENSITIVE constant is passed to the createStatement(). This enables the use of virtual cursor control methods in the third try{} block. Initially, the virtual cursor moves to the first row of the ResultSet and then to the last row before being positioned at the second to last row of the ResultSet.

Next, the virtual cursor is positioned to the tenth row of the ResultSet using the absolute() method. Finally, the relative() method is called twice. The first time

the relative() method is called, the virtual cursor is moved back two rows from the current row, which places the virtual cursor at row 8. The relative() method is again called to return the virtual cursor back to its original row by moving the virtual cursor two rows forward.

If you use any of these methods and end up positioning the cursor before the first record or beyond the last record, there won't be any errors thrown.

Listing 6-14 Using a scrollable virtual cursor.

```
String url = "jdbc:odbc:CustomerInformation";
String userID = "jim";
String password = "keogh";
String printrow;
String FirstName;
String LastName;
Statement DataRequest;
ResultSet Results;
Connection Db;
try {
   Class.forName( "sun.jdbc.odbc.JdbcOdbcDriver");
   Db = DriverManager.getConnection(url,userID,password);
 }
catch (ClassNotFoundException error) {
      System.err.println("Unable to load the JDBC/ODBC bridge." + error);
      System.exit(1);
}
catch (SQLException error) {
      System.err.println("Cannot connect to the database." + error);
      System.exit(2);
}
try {
      String query = "SELECT FirstName,LastName FROM Customers";
      DataRequest = Db.createStatement(TYPE_SCROLL_INSENSITIVE);
      Results = DataRequest.executeQuery (query);
}

catch ( SQLException error ){
     System.err.println("SQL error." + error);
      System.exit(3);
}

boolean Records = Results.next();
if (!Records ) {
   System.out.println("No data returned");
   System.exit(4);
}
try {
  do {
```

```
 Results.first();
Results.last();
Results.previous();
Results.absolute(10);
Results.relative(-2);
Results.relative(2);
 FirstName = Results.getString ( 1 ) ;
 LastName = Results.getString ( 2 ) ;
 printrow = FirstName + " " + LastName;
System.out.println(printrow);
} while (Results.next() );
DataRequest.close();
}
catch (SQLException error ) {
   System.err.println("Data display error." + error);
   System.exit(5);
}
```

Not All JDBC Drivers Are Scrollable

Although the JDBC API contains methods to scroll a ResultSet, some JDBC drivers may not support some or all of these features and therefore are unable to return a scrollable ResultSet. Listing 6-15 can be used to test whether or not the JDBC driver in use supports a scrollable ResultSet.

Listing 6-15 Testing whether a driver supports a scrollable ResultSet.

```
boolean forward, insensitive, sensitive;
DataBaseMetaData meta = Db.getMetaData();
forward = meta.supportsResultsSetType(ResultSet.TYPE_FORWARD_ONLY);
insensitive = meta.supportsResultsSetType(ResultSet. TYPE_SCROLL_INSENSITIVE);
sensitive = meta.supportsResultsSetType(ResultSet. TYPE_SCROLL_SENSITIVE);
System.out.println("forward: " + answer);
System.out.println("insensitive: " + insensitive);
System.out.println("sensitive: " + sensitive);
 (h2) Specify Number Of Rows To Return
```

When the J2EE component requests rows from the ResultSet, some rows are fetched into the driver and returned at one time. Other times, all rows requested may not be retrieved at the same time. In this case, the driver returns to the DBMS and requests another set of rows that are defined by the fetch size, and then discards the current set of rows. This process continues until the J2EE retrieves all rows.

Although the Statement class has a method for setting maximum rows, the method may not be effective since the driver does not implement it. In addition, the maximum row setting is for rows in the ResultSet and not for the number of rows returned by the DBMS. For example, the maximum rows can be set to 100. The DBMS might return

500 rows, but the ResultSet object silently drops 400 of them. This means all 500 rows are still pumped over the network.

The fetch size is set by using the setFetchSize() method, which is illustrated in Listing 6-16. However, all DBMS vendors may not implement the fetch size. Consult the driver documentation to determine if fetch size is supported. If fetch size isn't supported, the methods will compile and execute, but have no effect.

Don't become overly concerned about setting the fetch size, because fetch size is in the area of performance tuning—which is handled by the database administrator or the network engineer.

Listing 6-16 illustrates how to set the maximum number of rows that are returned by the DBMS. The second try{} block in Listing 6-16 calls the createStatement() method of the Connection object and then sets the maximum number of rows to 500 using the setFetchSize() method of the Statement object.

Listing 6-16 Setting the maximum number of rows returned in a ResultSet.

```
String url = "jdbc:odbc:CustomerInformation";
String userID = "jim";
String password = "keogh";
String printrow;
String FirstName;
String LastName;
Statement DataRequest;
ResultSet Results;
Connection Db;
try {
   Class.forName( "sun.jdbc.odbc.JdbcOdbcDriver");
   Db = DriverManager.getConnection(url,userID,password);
 }
catch (ClassNotFoundException error) {
      System.err.println("Unable to load the JDBC/ODBC bridge." + error);
      System.exit(1);
}
catch (SQLException error) {
      System.err.println("Cannot connect to the database." + error);
      System.exit(2);
}
try {
      String query = "SELECT FirstName,LastName FROM Customers";
      DataRequest = Db.createStatement(TYPE_SCROLL_INSENSITIVE);
      DataRequest.setFetchSize(500);
      Results = DataRequest.executeQuery (query);
}

catch ( SQLException error ){
     System.err.println("SQL error." + error);
      System.exit(3);
}
```

Updatable ResultSet

Rows contained in the ResultSet can be updatable similar to how rows in a table can be updated. This is made possible by passing the createStatement() method of the Connection object the CONCUR_UPDATABLE. Alternatively, the CONCUR_READ_ONLY constant can be passed to the createStatement() method to prevent the ResultSet from being updated.

There are three ways in which a ResultSet can be changed. These are updating values in a row, deleting a row, and inserting a new row. All of these changes are accomplished by using methods of the Statement object.

Update ResultSet

Once the executeQuery() method of the Statement object returns a ResultSet, the updatexxx() method is used to change the value of a column in the current row of the ResultSet. The xxx in the updatexxx() method is replaced with the data type of the column that is to be updated.

The updatexxx() method requires two parameters. The first is either the number or name of the column of the ResultSet that is being updated and the second parameter is the value that will replace the value in the column of the ResultSet.

A value in a column of the ResultSet can be replaced with a NULL value by using the updateNull() method. The updateNull() method requires one parameter, which is the number of the column in the current row of the ResultSet. The updateNull() doesn't accept the name of the column as a parameter.

The updateRow() method is called after all the updatexxx() methods are called. The updateRow() method changes values in columns of the current row of the ResultSet based on the values of the updatexxx() methods.

Listing 6-17 illustrates how to update a row in a ResultSet. In this example, customer Mary Jones was recently married and changed her last name to Smith before processing the ResultSet. The updateString() method is used to change the value of the last name column of the ResultSet with 'Jones'. The change takes effect once the updateRow() method is called; however, this change only occurs in the ResultSet. The corresponding row in the table remains unchanged until an update query is run, which is discussed in the next chapter.

ing 6-17 ating the esultSet.

```
String url = "jdbc:odbc:CustomerInformation";
String userID = "jim";
String password = "keogh";
Statement DataRequest;
ResultSet Results;
Connection Db;
try {
   Class.forName( "sun.jdbc.odbc.JdbcOdbcDriver");
   Db = DriverManager.getConnection(url,userID,password);
 }
```

```
catch (ClassNotFoundException error) {
      System.err.println("Unable to load the JDBC/ODBC bridge." + error);
      System.exit(1);
}
catch (SQLException error) {
      System.err.println("Cannot connect to the database." + error);
      System.exit(2);
}
try {
      String query = "SELECT FirstName,LastName FROM Customers WHERE
FirstName = 'Mary' and LastName = 'Smith'";
      DataRequest = Db.createStatement(ResultSet.CONCUR_UPDATABLE);
      Results = DataRequest.executeQuery (query);
}

catch ( SQLException error ){
     System.err.println("SQL error." + error);
      System.exit(3);
}

boolean Records = Results.next();
if (!Records ) {
   System.out.println("No data returned");
   System.exit(4);
}
try {
       Results.updateString ("LastName", "Smith");
     Results.updateRow();
     DataRequest.close();
}
catch (SQLException error ) {
   System.err.println("Data display error." + error);
   System.exit(5);
}
```

Delete Row in the ResultSet

The deleteRow() method is used to remove a row from a ResultSet. Sometimes this is advantageous when processing the ResultSet because this is a way to eliminate rows from future processing. For example, each row of a ResultSet may have to pass three tests. Those that fail to pass the first test could be deleted from the ResultSet, thereby reducing the number of rows in the ResultSet that have to be evaluated for the second test. This also deletes it from the underlying database.

The deleteRow() method is passed an integer that contains the number of the row to be deleted. A good practice is to use the absolute() method described previously in the chapter to move the virtual cursor to the row in the ResultSet that should be deleted. However, the value of that row should be examined by the program to assure it is the

proper row before the deleteRow() method is called. The deleteRow() method is then passed a zero integer indicating that the current row must be deleted, as shown in the following statement:

```
Results.deleteRow(0);
```

Insert Row in the ResultSet

Inserting a row into the ResultSet is accomplished using basically the same technique as is used to update the ResultSet. That is, the updatexxx() method is used to specify the column and value that will be placed into the column of the ResultSet.

The updatexxx() method requires two parameters. The first parameter is either the name of the column or the number of the column of the ResultSet. The second parameter is the new value that will be placed in the column of the ResultSet. Remember that the data type of the column replaces the xxx in the method name.

The insertRow() method is called after the updatexxx() methods, which causes a new row to be inserted into the ResultSet having values that reflect the parameters in the updatexxx() methods. This also updates the underlying database.

Listing 6-18 illustrates how to insert a new row in a ResultSet. In this example, the query returns the first name and last name of all customers. The name Tom Smith is inserted into the ResultSet in the third try{} block using the updateString() method. Remember that columns are numbered based on the order that the column names appear in the SELECT statement of the query. The new row is added to the ResultSet after the insertRow() method is called.

Listing 6-18 Inserting a new row into the ResultSet.

```
String url = "jdbc:odbc:CustomerInformation";
String userID = "jim";
String password = "keogh";
Statement DataRequest;
ResultSet Results;
Connection Db;
try {
   Class.forName( "sun.jdbc.odbc.JdbcOdbcDriver");
   Db = DriverManager.getConnection(url,userID,password);
 }
catch (ClassNotFoundException error) {
      System.err.println("Unable to load the JDBC/ODBC bridge." + error);
      System.exit(1);
}
catch (SQLException error) {
      System.err.println("Cannot connect to the database." + error);
      System.exit(2);
}
try {
      String query = "SELECT FirstName,LastName FROM Customers";
```

```
        DataRequest = Db.createStatement(CONCUR_UPDATABLE);
        Results = DataRequest.executeQuery (query);
}

catch ( SQLException error ){
      System.err.println("SQL error." + error);
       System.exit(3);
}

boolean Records = Results.next();
if (!Records ) {
    System.out.println("No data returned");
   System.exit(4);
}
try {
       Results.updateString (1, "Tom");  // updates the ResultSet
      Results.updateString (2, "Smith"); // updates the ResultSet
      Results.insertRow(); // updates the underlying database
      DataRequest.close();
}
catch (SQLException error ) {
   System.err.println("Data display error." + error);
   System.exit(5);
}
```

Transaction Processing

A transaction may involve several tasks similar to the tasks that are required to complete a transaction at a supermarket. In a supermarket transaction, each item purchased must be registered, the transaction must be totaled, and the customer must tender the amount of the purchase. The transaction is successfully completed only if each task is completed successfully. If one task fails, the entire transaction fails. Previously completed tasks must be reversed if the transaction fails. For example, goods that were registered must be removed from the register and returned to the shelf.

A database transaction consists of a set of SQL statements, each of which must be successfully completed for the transaction to be completed. If one fails, SQL statements that executed successfully up to that point in the transaction must be rolled back.

A database transaction isn't completed until the J2EE component calls the commit() method of the Connection object. All SQL statements executed prior to the call to the commit() method can be rolled back. However, once the commit() method is called, none of the SQL statements can be rolled back.

The commit() method must be called regardless if the SQL statement is part of a transaction or not. This means that the commit() method must be issued in the previous examples used in this chapter. However, the commit() method was automatically called

in these examples because the DBMS has an AutoCommit feature that is by default set to true.

If a J2EE component is processing a transaction, the AutoCommit feature must be deactivated by calling the setAutoCommit() method and passing it a false parameter. Once the transaction is completed, the setAutoCommit() method is called again—this timing passing it a true parameter, reactivating the AutoCommit feature.

Listing 6-19 illustrates how to process a transaction. The transaction in this example consists of two SQL statements, both of which update the Street address of rows in the Customer table. Each SQL statement is executed separately and then the commit() method is called. However, should either SQL statement throw an SQL exception, the catch{} block reacts by rolling back the transaction before displaying the exception on the screen.

ing 6-19
ecuting a
latabase
nsaction.

```
String url = "jdbc:odbc:CustomerInformation";
String userID = "jim";
String password = "keogh";
Statement DataRequest1, DataRequest2 ;
Connection Database;
try {
      Class.forName( "sun.jdbc.odbc.JdbcOdbcDriver");
      Database = DriverManager.getConnection(url,userID,password);
}
catch (ClassNotFoundException error) {
     System.err.println("Unable to load the JDBC/ODBC bridge." + error);
     System.exit(1);
}
catch (SQLException error) {
     System.err.println("Cannot connect to the database." + error);
     System.exit(2);
 }
try {
  Database .setAutoCommit(false)
  String query1 = "UPDATE Customers SET Street = '5 Main Street' " +
           "WHERE FirstName = 'Bob'";
  String query2 = "UPDATE Customers SET Street = '10 Main Street' " +
           "WHERE FirstName = 'Tim'";
DataRequest1= Database.createStatement();
DataRequest2= Database.createStatement();
DataRequest.executeUpdate (query1 );
DataRequest.executeUpdate (query2 );
Database.commit();
DataRequest1.close();
DataRequest2.close();
Database.close();
}
catch(SQLException ex) {
 System.err.println("SQLException: " + ex.getMessage());
```

```
    if (con != null) {
      try {
        System.err.println("Transaction is being rolled back ");
        con.rollback();
      }
     catch(SQLException excep) {
          System.err.print("SQLException: ");
          System.err.println(excep.getMessage());
      }
    }
  }
(h2) Savepoints
```

A transaction may consist of many tasks, some of which don't need to be rolled back should the entire transaction fail. Let's say there are several tasks that occur when a new order is processed. These include updating the customer account table, inserting the order into the pending order table, and sending a customer a confirmation email.

Technically, all three tasks must be completed before the transaction is considered completed. Suppose the email server is down when the transaction is ready to send the customer a confirmation email. Should the entire transaction be rolled back? Probably not since it is more important that the order continue to be processed (i.e., delivered). The confirmation notice can be sent once the email server is back online.

The J2EE component can control the number of tasks that are rolled back by using savepoints. A savepoint, introduced in JDBC 3.0, is a virtual marker that defines the task at which the rollback stops. In the previous example, the task before the email confirmation notice is sent can be designated as a savepoint.

Listing 6-20 illustrates how to create a savepoint. This is the same code segment as Listing 6-19, but a savepoint is created after the execution of the first update SQL statement.

There can be many savepoints used in a transaction. Each savepoint is identified by a unique name. The savepoint name is then passed to the rollback() method to specify the point within the transaction where the rollback is to stop.

In this example, there is one savepoint called sp1. The name "sp1" is the parameter to the rollback() method in the catch{} block. The purpose of this example is to illustrate how to set and release a savepoint and how to use the savepoint name in the rollback() method. Of course, for commercial applications more rigorous code is necessary that identifies the executeUpdate() method that threw the exception among other error-checking routines. The releaseSavepoint() method is called to remove the savepoint from the transaction. The name of the savepoint that is to be removed is passed to the releaseSavepoint() method.

Listing 6-20 Using savepoints in a transaction

```
String url = "jdbc:odbc:CustomerInformation";
String userID = "jim";
String password = "keogh";
```

```
Statement DataRequest1, DataRequest2 ;
Connection Database;
try {
      Class.forName( "sun.jdbc.odbc.JdbcOdbcDriver");
      Database = DriverManager.getConnection(url,userID,password);
}
catch (ClassNotFoundException error) {
     System.err.println("Unable to load the JDBC/ODBC bridge." + error);
     System.exit(1);
}
catch (SQLException error) {
     System.err.println("Cannot connect to the database." + error);
     System.exit(2);
 }
try {
  Database .setAutoCommit(false)
  String query1 = "UPDATE Customers SET Street = '5 Main Street' " +
            "WHERE FirstName = 'Bob'";
  String query2 = "UPDATE Customers SET Street = '10 Main Street' " +
            "WHERE FirstName = 'Tim'";
  DataRequest1= Database.createStatement();
  Savepoint s1 = Database.setSavepoint ("sp1");
  DataRequest2= Database.createStatement();
  DataRequest.executeUpdate (query1);
  DataRequest.executeUpdate (query2);
  Database.commit();
  DataRequest1.close();
  DataRequest2.close();
  Database.releaseSavepoint ("sp1");
  Database.close();
}
catch ( SQLException error ){
   try {
     Database.rollback(sp1);
   }
   catch ( SQLException error ){
      System.err.println("rollback error." + error.getMessage());
      System.exit(3);
   }
    System.err.println("SQL error." + error. getMessage()););
   System.exit(4);
}
 (h2) Batch Statements
```

Another way to combine SQL statements into a transaction is to batch together these statements into a single transaction and then execute the entire transaction. You can do

this by using the addBatch() method of the Statement object. The addBatch() method receives a SQL statement as a parameter and places the SQL statement in the batch.

Once all the SQL statements that comprise the transaction are included in the batch, the executeBatch() method is called to execute the entire batch at the same time. The executeBatch() method returns an int array that contains the number of SQL statements that were executed successfully.

The int array is displayed if a BatchUpdateException error is thrown during the execution of the batch. The batch can be cleared of SQL statements by using the clearBatch() method. The transaction must be committed using the commit() method. Make sure that setAutoCommit() is set to false before executing the batch, as discussed in the previous section.

Listing 6-21 illustrates how to batch SQL statements. In this example, two SQL statements are created as discussed previously in this chapter. Each SQL statement is added to the batch using the addBatch() method.

Once both SQL statements are added to the batch, the executeBatch() method is called to execute each of the SQL statements. The commit() method is then called to commit the changes created by the SQL statement. Until the commit() method is called, the transaction can be rolled back as described in the previous section.

Listing 6-21 Batching SQL statements into a transaction.

```
String url = "jdbc:odbc:CustomerInformation";
String userID = "jim";
String password = "keogh";
Statement DataRequest;
Connection Database;
try {
      Class.forName( "sun.jdbc.odbc.JdbcOdbcDriver");
      Database = DriverManager.getConnection(url,userID,password);
}
catch (ClassNotFoundException error) {
     System.err.println("Unable to load the JDBC/ODBC bridge." + error);
     System.exit(1);
}
catch (SQLException error) {
     System.err.println("Cannot connect to the database." + error);
     System.exit(2);
 }
try {
  Database .setAutoCommit(false)
  String query1 = "UPDATE Customers SET Street = '5 Main Street' " +
            "WHERE FirstName = 'Bob'";
  String query2 = "UPDATE Customers SET Street = '10 Main Street' " +
            "WHERE FirstName = 'Tim'";
  DataRequest= Database.createStatement();
  DataRequest.addBatch(query1);
  DataRequest.addBatch(query2);
```

```
    int [ ] updated = DataRequest.executeBatch ();
    Database.commit();
    DataRequest1.close();
    DataRequest2.close();
    Database.close();
}
catch(BatchUpdateException error) {
    System.out.println("Batch error.");
    System.out.println("SQL State: " + error.getSQLState());
    System.out.println("Message: " + error.getMessage());
    System.out.println(Vendor: " + error.getErrorCode());
    int [ ] updated  = error.getUpdatecount();
    int count = updated.length();
    for int - i = 0; i < count; i++) {
        System.out.print (updated[i]);
    }
   SQLException sql = error;
   While (sql != null)
       {
         System.out.println("SQL error " + sql);
         sql = sql.getnextException();
       }
   try{
      DataRequest.clearBatch();
   }
   catch(BatchUpdateException error) {
       System.out.println("Unable to clear the batch: " + error.getMessage());
   }
}
```

ResultSet Holdability

Whenever the commit() method is called, all ResultSet objects that were created for the transaction are closed. Sometimes a J2EE component needs to keep the ResultSet open even after the commit() method is called.

You can control whether or not ResultSet objects are closed following the call to the commit() method by passing one of two constants to the createStatement() method. These constants are HOLD_CURSORS_OVER_COMMIT and CLOSE_CURSORS_AT_COMMIT.

The HOLD_CURSORS_OVER_COMMIT constant keeps ResultSet objects open following a call to the commit() method and CLOSE_CURSORS_AT_COMMIT closes ResultSet objects when the commit() method is called.

RowSets

The JDBC RowSets object is used to encapsulate a ResultSet for use with Enterprise Java Beans (EJB). A RowSet object contains rows of data from a table(s) that can be used in a disconnected operation.

That is, an EJB can interact with a RowSet object without having to be connected to a DBMS, which is ideal for J2EE components that have PDA clients. You'll learn more about EJB and RowSets in Chapter 12.

Auto-Generated Keys

It is common for a DBMS to automatically generate unique keys for a table as rows are inserted into the table. The getGeneratedKeys() method of the Statement object is called to return keys generated by the DBMS.

The getGeneratedKeys() returns a ResultSet object. You can use the ResultSet.getMetaData() method to retrieve metadata relating to the automatically generated key, such as the type and properties of the automatically generated key. You can learn more about retrieving metadata in the next section of this chapter.

Metadata

Metadata is data about data, as discussed previously in this chapter. A J2EE component can access metadata by using the DatabaseMetaData interface. The DatabaseMetaData interface is used to retrieve information about databases, tables, columns, and indexes among other information about the DBMS.

A J2EE component retrieves metadata about the database by calling the getMetaData() method of the Connection object. The getMetaData() method returns a DatabaseMetaData object that contains information about the database and its components.

Once the DatabaseMetaData object is obtained, an assortment of methods contained in the DatabaseMetaData object are called to retrieve specific metadata. Here are some of the more commonly used DatabaseMetaData object methods:

- **getDatabaseProductName()** Returns the product name of the database.
- **getUserName()** Returns the username.
- **getURL()** Returns the URL of the database
- **getSchemas()** Returns all the schema names available in this database.
- **getPrimaryKeys()** Returns primary keys.
- **getProcedures()** Returns stored procedure names.
- **getTables()** Returns names of tables in the database.

ResultSet Metadata

There are two types of metadata that can be retrieved from the DBMS. These are metadata that describes the database as mentioned in the previous section and metadata that describes the ResultSet. Metadata that describes the ResultSet is retrieved by calling the getMetaData() method of the ResultSet object. This returns a ResultSetMetaData object, as is illustrated in the following code statement:

```
ResultSetMetaData rm = Result.getMetaData()
```

Once the ResultSet metadata is retrieved, the J2EE component can call methods of the ResultSetMetaData object to retrieve specific kinds of metadata. The more commonly called methods are as follows:

- **getColumnCount()** Returns the number of columns contained in the ResultSet.
- **getColumnName(int number)** Returns the name of the column specified by the column number.
- **getColumnType(int number)** Returns the data type of the column specified by the column number.

There are many other methods used to retrieve practically any information you need to know about a database and the ResultSet—these are more methods than can be described in this chapter. You can obtain detailed information about each of these methods by visiting Sun Microsystems, Inc.'s web site at java.sun.com.

Data Types

The setxxx() and getxxx() methods are used throughout this chapter to set a value of a specific data type and to retrieve a value of a specific data type. The xxx in the names of these methods is replaced with the name of the data type.

Table 6-1 contains a list of data types and their Java equivalents. You can use this list to determine the proper data name to use to replace the xxx in the setxxx() and getxxx() methods.

SQL Type	Java Type
CHAR	String
VARCHAR	String
LONGVARCHAR	String
NUMERIC	java.math.BigDecimal
DECIMAL	java.math.BigDecimal
BIT	Boolean
TINYINT	Byte
SMALLINT	Short
INTEGER	Integer
BIGINT	Long
REAL	float
FLOAT	float
DOUBLE	double
BINARY	Byte[]
VARBINARY	Byte[]
LONGVARBINARY	byte[]
BLOB	java.sql.Blob
CLOB	java.sql.Clob
ARRAY	java.sql.Array
STRUCT	java.sql.Struct
REF	java.sql.Ref
DATALINK	java.sql.Types
DATE	java.sql.date
TIME	java.sql.Time
TIMESTAMP	java.sql.Timestamp

Table 6-1. *A List of Data Types for Use with the setxxx() and getxxx() Methods*

Exceptions

There are three kinds of exceptions that are thrown by JDBC methods. These are SQLExceptions, SQLWarnings, and DataTruncation. SQLExceptions commonly reflect a SQL syntax error in the query and are thrown by many of the methods contained in the java.sql package. Hopefully, the syntax errors in your code get resolved quickly. In production, this exception is most commonly caused by connectivity issues with the database. It can also be caused by subtle coding errors like trying to access an object that's been closed. For example, you try to roll back a transaction in a catch clause and don't check first if the database connection is still valid. The getNextException() method of the SQLExceptions object is used to return details about the SQL error or a null if the last exception was retrieved. The getErrorCode() method of the SQLException object is used to retrieve vendor-specific error codes.

The SQLWarning throws warnings received by the Connection from the DBMS. The getWarnings() method of the Connection object retrieves the warning and the getNextWarning() method of the Connection object retrieves subsequent warnings.

Whenever data is lost due to truncation of the data value, a DataTrunction exception is thrown.

Quick Reference Guide

JDBC classes and interface contain many methods. This quick reference guide (encompassed in the following Tables 6-2 through 6-12) provides a brief overview of the more commonly used methods. Full details of these methods and all the JDBC classes and interfaces are available at java.sun.com.

Syntax	Descriptions
public void registerOutParameter(int parameterIndex, int sqlType) throws SQLException	Registers the OUT parameter.
public void registerOutParameter(int parameterIndex, int sqlType, int scale) throws SQLException	Registers the OUT parameter.
public boolean wasNull() throws SQLException	Determines an OUT parameter value is null.

Table 6-2. ***Java.sql Package*** *Public Interface CallableStatement*

Syntax	Descriptions
set by name	
public void setURL(String parameterName, URL val) throws SQLException	Sets a parameter to a java.net.URL object.
Descriptions	Syntax
public void setNull(String parameterName, int sqlType) throws SQLException	Sets a parameter to a null value.
public void setBoolean(String parameterName, boolean x) throws SQLException	Sets a parameter to a Java boolean value.
public void setByte(String parameterName, byte x) throws SQLException	Sets a parameter to a Java byte value.
public void setShort(String parameterName, short x) throws SQLException	Sets a parameter to a Java short value.
public void setInt(String parameterName, int x) throws SQLException	Sets a parameter to a Java int value.
public void setLong(String parameterName, long x) throws SQLException	Sets a parameter to a Java long value.
public void setFloat(String parameterName, float x) throws SQLException	Sets a parameter to a Java float value.
public void setDouble(String parameterName, double x) throws SQLException	Sets a parameter to a Java double value.
public void setBigDecimal(String parameterName, BigDecimal x) throws SQLException	Sets a parameter to a java.math.BigDecimal value.
public void setString(String parameterName, String x) throws SQLException	Sets a parameter to a Java String value.
public void setBytes(String parameterName, byte[] x) throws SQLException	Sets a parameter to a Java array of bytes.
public void setDate(String parameterName, Date x) throws SQLException	Sets a parameter to a java.sql.Date value.

Table 6-2. ***java.sql Package*** *Public Interface CallableStatement* (continued)

Syntax	Descriptions
public void setTime(String parameterName, Time x) throws SQLException	Sets a parameter to a java.sql.Time value.
public void setTimestamp(String parameterName, Timestamp x) throws SQLException	Sets a parameter to a java.sql.Timestamp value.
public void setBinaryStream(String parameterName, InputStream x, int length) throws SQLException	Sets the designated parameter to the given input stream, which will have the specified number of bytes.
public void setObject(String parameterName, Object x, int targetSqlType, int scale) throws SQLException	Sets a parameter with the given object.
public void setObject(String parameterName, Object x, int targetSqlType) throws SQLException	Sets a parameter with the given object.
public void setObject(String parameterName, Object x) throws SQLException	Sets a parameter with the given object.
public void setDate(String parameterName, Date x, Calendar cal) throws SQLException	Sets a parameter to a java.sql.Date value.
public void setTime(String parameterName, Time x, Calendar cal) throws SQLException	Sets a parameter to a java.sql.Time value.
public void setTimestamp(String parameterName, Timestamp x, Calendar cal) throws SQLException	Sets a parameter to a java.sql.Timestamp value.
public void setNull(String parameterName, int sqlType, String typeName) throws SQLException	Sets a parameter to a null value.
get by index	
public String getString(int parameterIndex) throws SQLException	Determines the value of the designated JDBC CHAR, VARCHAR, or LONGVARCHAR parameter.
public boolean getBoolean(int parameterIndex) throws SQLException	Determines the value of the designated JDBC BIT parameter.

Table 6-2. ***Java.sql Package*** *Public Interface CallableStatement* (continued)

Syntax	Descriptions
public byte getByte(int parameterIndex) throws SQLException	Determines the value of the designated JDBC TINYINT parameter.
public short getShort(int parameterIndex) throws SQLException	Determines the value of the designated JDBC SMALLINT parameter.
public int getInt(int parameterIndex) throws SQLException	Determines the value of the designated JDBC INTEGER parameter.
public long getLong(int parameterIndex) throws SQLException	Determines the value of the designated JDBC BIGINT parameter.
public float getFloat(int parameterIndex) throws SQLException	Determines the value of the designated JDBC FLOAT parameter.
public double getDouble(int parameterIndex) throws SQLException	Determines the value of the designated JDBC DOUBLE parameter.
public BigDecimal getBigDecimal(int parameterIndex, int scale) throws SQLException	Determines the value of the designated JDBC NUMERIC parameter.
public byte[] getBytes(int parameterIndex) throws SQLException	Determines the value of the designated JDBC BINARY or VARBINARY parameter.
public Date getDate(int parameterIndex) throws SQLException	Determines the value of the designated JDBC DATE parameter.
public Time getTime(int parameterIndex) throws SQLException	Determines the value of the designated JDBC TIME parameter.
public Timestamp getTimestamp(int parameterIndex) throws SQLException	Determines the value of the designated JDBC TIMESTAMP parameter.

Table 6-2. ***Java.sql Package*** *Public Interface CallableStatement* (continued)

Syntax	Descriptions
public Object getObject(int parameterIndex) throws SQLException	Determines the value of the designated parameter.
public BigDecimal getBigDecimal(int parameterIndex) throws SQLException	Determines the value of the designated JDBC NUMERIC parameter.
public Object getObject(int i, Map map) throws SQLException	Returns an object representing the value of the OUT parameter.
public Ref getRef(int i) throws SQLException	Determines the value of the designated JDBC REF parameter.
public Blob getBlob(int i) throws SQLException	Determines the value of the designated JDBC BLOB parameter.
public Clob getClob(int i) throws SQLException	Determines the value of the designated JDBC CLOB parameter.
public Array getArray(int i) throws SQLException	Determines the value of the designated JDBC ARRAY parameter.
public Date getDate(int parameterIndex, Calendar cal) throws SQLException	Determines the value of the designated JDBC DATE parameter.
public Time getTime(int parameterIndex, Calendar cal) throws SQLException	Determines the value of the designated JDBC TIME parameter.
public Timestamp getTimestamp(int parameterIndex, Calendar cal) throws SQLException	Determines the value of the designated JDBC TIMESTAMP parameter.
public void registerOutParameter(int paramIndex, int sqlType, String typeName) throws SQLException	Registers the designated output parameter.

Table 6-2. ***java.sql Package*** *Public Interface CallableStatement* (continued)

Syntax	Descriptions
public void registerOutParameter(String parameterName, int sqlType) throws SQLException	Registers the OUT parameter named parameterName.
public void registerOutParameter(String parameterName, int sqlType, int scale) throws SQLException	Determines the parameter named parameterName.
public void registerOutParameter(String parameterName, int sqlType, String typeName) throws SQLException	Determines the designated output parameter.
public URL getURL(int parameterIndex) throws SQLException	Determines the value of the designated JDBC DATALINK parameter.
get by name	
public String getString(String parameterName) throws SQLException	Determines the value of a JDBC CHAR, VARCHAR, or LONGVARCHAR parameter.
public boolean getBoolean(String parameterName) throws SQLException	Determines the value of a JDBC BIT parameter.
public byte getByte(String parameterName) throws SQLException	Determines the value of a JDBC TINYINT parameter.
public short getShort(String parameterName) throws SQLException	Determines the value of a JDBC SMALLINT parameter.
public int getInt(String parameterName) throws SQLException	Determines the value of a JDBC INTEGER parameter.
public long getLong(String parameterName) throws SQLException	Determines the value of a JDBC BIGINT parameter.
public float getFloat(String parameterName) throws SQLException	Determines the value of a JDBC FLOAT parameter.
public double getDouble(String parameterName) throws SQLException	Determines the value of a JDBC DOUBLE parameter.
public byte[] getBytes(String parameterName) throws SQLException	Determines the value of a JDBC BINARY or VARBINARY parameter.

Table 6-2. ***java.sql Package*** *Public Interface CallableStatement* (continued)

Syntax	Descriptions
public Date getDate(String parameterName) throws SQLException	Determines the value of a JDBC DATE parameter.
public Time getTime(String parameterName) throws SQLException	Determines the value of a JDBC TIME parameter.
public Timestamp getTimestamp(String parameterName) throws SQLException	Determines the value of a JDBC TIMESTAMP parameter.
public Object getObject(String parameterName) throws SQLException	Determines the value of a parameter as an Object.
public BigDecimal getBigDecimal(String parameterName) throws SQLException	Determines the value of a JDBC NUMERIC parameter.
public Object getObject(String parameterName, Map map) throws SQLException	Returns an object representing the value of the OUT parameter.
public Ref getRef(String parameterName) throws SQLException	Determines the value of a JDBC REF parameter.
public Blob getBlob(String parameterName) throws SQLException	Determines the value of a JDBC BLOB parameter.
public Clob getClob(String parameterName) throws SQLException	Determines the value of a JDBC CLOB parameter.
public Array getArray(String parameterName) throws SQLException	Determines the value of a JDBC ARRAY.
public Date getDate(String parameterName, Calendar cal) throws SQLException	Determines the value of a JDBC DATE parameter.
public Time getTime(String parameterName, Calendar cal) throws SQLException	Determines the value of a JDBC TIME parameter.
public Timestamp getTimestamp(String parameterName, Calendar cal) throws SQLException	Determines the value of a JDBC TIMESTAMP parameter.
public URL getURL(String parameterName) throws SQLException	Determines a URL.

Table 6-2. ***java.sql Package*** *Public Interface CallableStatement* (continued)

Syntax	Descriptions
public void close() throws SQLException	Closes a connection.
public boolean isClosed() throws SQLException	Determines if a Connection object is closed.
public DatabaseMetaData getMetaData() throws SQLException	Retrieves a DatabaseMetaData.
public void setReadOnly(boolean readOnly) throws SQLException	Places a connection in read-only.
public boolean isReadOnly() throws SQLException	Determines if a Connection object is in read-only.
public String getCatalog() throws SQLException	Determines a connection's catalog name.
Warnings	
public SQLWarning getWarnings() throws SQLException	Retrieves the first warning reported.
public void clearWarnings() throws SQLException	Clears all warnings reported.
Holdability	
public void setHoldability(int holdability) throws SQLException	Changes the holdability of ResultSet objects.
public int getHoldability() throws SQLException	Determines the holdability of a ResultSet.
Transactions	
public Savepoint setSavepoint() throws SQLException	Creates an unnamed savepoint.
public Savepoint setSavepoint(String name) throws SQLException	Creates a named savepoint.
public void rollback(Savepoint savepoint) throws SQLException	Reverses all changes made after a savepoint was set.
public void releaseSavepoint(Savepoint savepoint) throws SQLException	Removes the Savepoint object.
public void commit() throws SQLException	Makes all changes permanent.

Table 6-3. ***java.sql Package*** *Public Interface Connection Interface*

Syntax	Descriptions
public void rollback() throws SQLException	Reverses changes made to a transaction and releases any database locks.
public void setAutoCommit(boolean autoCommit) throws SQLException	Sets the auto-commit mode.
public boolean getAutoCommit() throws SQLException	Determines if the auto-commit mode is set.
CallableStatement	
public CallableStatement prepareCall(String sql, int resultSetType, int resultSetConcurrency, int resultSetHoldability) throws SQLException	Creates a CallableStatement object that generates a ResultSet.
public CallableStatement prepareCall(String sql, int resultSetType,int resultSetConcurrency) throws SQLException	Creates a CallableStatement object that generates a ResultSet.
public CallableStatement prepareCall (String sql) throws SQLException	Creates a CallableStatement object.
Statement	
public Statement createStatement(int resultSetType, int resultSetConcurrency, int resultSetHoldability) throws SQLException	Creates a Statement object that will generate a ResultSet.
public Statement createStatement(int resultSetType, int resultSetConcurrency) throws SQLException	Creates a Statement object that generates a ResultSet.
public Statement createStatement() throws SQLException	Creates a Statement object.
PrepareStatement	
public PreparedStatement prepareStatement (String sql) throws SQLException	Creates a PreparedStatement object.
public PreparedStatement prepareStatement (String sql, int resultSetType,int resultSetConcurrency, int resultSetHoldability) throws SQLException	Creates a PreparedStatement object that will generate a ResultSet.

Table 6-3. ***Java.sql Package*** *Public Interface Connection Interface* (continued)

Synatx	Descriptions
public PreparedStatement prepareStatement (String sql, int[] columnIndexes) throws SQLException	Creates a PreparedStatement object that returns auto-generated keys.
public PreparedStatement prepareStatement (String sql, int resultSetType, int resultSetConcurrency) throws SQLException	Creates a PreparedStatement object that generates a ResultSet.
public PreparedStatement prepareStatement (String sql, String[] columnNames) throws SQLException	Creates a PreparedStatement object that returns auto-generated keys.

Table 6-3. ***java.sql Package*** *Public Interface Connection Interface* (continued)

Syntax	Descriptions
public Connection getConnection() throws SQLException	Retrieves the connection that produced metadata.
public boolean supportsSavepoints() throws SQLException	Determines if a database supports savepoints.
public boolean supportsNamedParameters() throws SQLException	Determines if a database supports named parameters to callable statements.
public boolean supportsMultipleOpenResults() throws SQLException	Determines if it is possible to have multiple ResultSet objects returned from a CallableStatement simultaneously.
public int getMaxUserNameLength() throws SQLException	Determines the maximum number of characters in a username.
public boolean supportsTransactions() throws SQLException	Determines if a database supports transactions.
public int getJDBCMajorVersion() throws SQLException	Determines the major JDBC version number of a driver.

Table 6-4. ***java.sql Package*** *Public Interface DatabaseMetaData*

Synatx	Descriptions
public int getJDBCMinorVersion() throws SQLException	Determines the minor JDBC version number of a driver.
Database	
public String getURL() throws SQLException	Retrieves the URL for a DBMS.
public String getDatabaseProductName() throws SQLException	Determines the name of a DBMS.
public String getDatabaseProductVersion() throws SQLException	Determines the version number of a DBMS.
public ResultSet getSchemas() throws SQLException	Determines the schema names.
public ResultSet getCatalogs() throws SQLException	Determines the catalog names.
public ResultSet getTypeInfo() throws SQLException	Retrieves a description of standard SQL types supported by a database.
public int getDatabaseMajorVersion() throws SQLException	Determines the major version number of a database.
public int getDatabaseMinorVersion() throws SQLException	Determines the minor version number of a database.
Columns and Rows	
public int getMaxColumnNameLength() throws SQLException	Determines the maximum number of characters allowed in a column name.
public int getMaxColumnsInGroupBy() throws SQLException	Determines the maximum number of columns allowed in a GROUP BY clause.
public int getMaxColumnsInIndex() throws SQLException	Determines the maximum number of columns allowed in an index.
public int getMaxColumnsInOrderBy() throws SQLException	Determines the maximum number of columns allowed in an ORDER BY clause.

Table 6-4. ***java.sql Package*** *Public Interface DatabaseMetaData* (continued)

Synatx	Descriptions
public int getMaxColumnsInSelect() throws SQLException	Determines the maximum number of columns allowed in a SELECT list.
public int getMaxColumnsInTable() throws SQLException	Determines the maximum number of columns allowed in a table.
public int getMaxConnections() throws SQLException	Determines the maximum number of concurrent connections that are possible to a database.
public int getMaxIndexLength() throws SQLException	Determines the maximum number of bytes allowed for an index.
public int getMaxRowSize() throws SQLException	Determines the maximum number of bytes allowed in a row.
public ResultSet getColumns(String catalog, String schemaPattern, String tableNamePattern, String columnNamePattern) throws SQLException	Retrieves a description of table columns.
public ResultSet getColumnPrivileges (String catalog, String schema, String table, String columnNamePattern) throws SQLException	Determines access rights for columns.
Statement	
public int getMaxStatementLength() throws SQLException	Determines the maximum number of characters permitted in a SQL statement.
public int getMaxStatements() throws SQLException	Determines the maximum number of active statements that can be open at the same time.
public boolean supportsBatchUpdates() throws SQLException	Determines if a database supports batch updates.

Table 6-4. ***Java.sql Package*** *Public Interface DatabaseMetaData* (continued)

Syntax	Descriptions
Table	
public int getMaxTableNameLength() throws SQLException	Determines the maximum number of characters in a table name.
public int getMaxTablesInSelect() throws SQLException	Determines the maximum number of tables allowed in a SELECT statement.
public ResultSet getTables(String catalog, String schemaPattern, String tableNamePattern, String[] types) throws SQLException	Determines tables available in a catalog.
public ResultSet getTablePrivileges(String catalog, String schemaPattern, String tableNamePattern) throws SQLException	Determines access rights for each table.
Keys	
public ResultSet getPrimaryKeys(String catalog, String schema, String table) throws SQLException	Retrieves a description of a table's primary key.
public ResultSet getIndexInfo(String catalog, String schema, String table, boolean unique, boolean approximate) throws SQLException	Retrieves a description of table's indices and statistics.
public boolean supportsGetGeneratedKeys() throws SQLException	Determines if auto-generated keys can be retrieved.
Procedure	
public ResultSet getProcedureColumns (String catalog, String schemaPattern, String procedureNamePattern, String columnNamePattern) throws SQLException	Determines a catalog's stored procedure parameter and result columns.
public ResultSet getProcedures(String catalog, String schemaPattern, String procedureNamePattern) throws SQLException	Determines the stored procedures available in a catalog.

Table 6-4. ***java.sql Package*** *Public Interface DatabaseMetaData* (continued)

Syntax	Descriptions
Holdability	
public boolean supportsResultSetHoldability(int holdability) throws SQLException	Determines whether a database supports holdability.
public int getResultSetHoldability() throws SQLException	Determines the default holdability of a ResultSet.

Table 6-4. ***java.sql Package*** *Public Interface DatabaseMetaData* (continued)

Syntax	Descriptions
public static Connection getConnection(String url, Properties info) throws SQLException	Opens a connection to a database.
public static Connection getConnection (String url, String user, String password) throws SQLException	Opens a connection to a database.
public static Connection getConnection (String url) throws SQLException	Opens a connection to a database.
Timeout	
public static void setLoginTimeout (int seconds)	Sets the maximum time in seconds before the login to the database times out.
public static int getLoginTimeout()	Determines the maximum time in seconds before the login to the database times out.

Table 6-5. ***Java.sql Package*** *Public Class DriverManager*

Syntax	Descriptions
public ResultSetMetaData getMetaData() throws SQLException	Retrieves a ResultSetMetaData object.
public ParameterMetaData getParameterMetaData() throws SQLException	Determines the metadata for a PreparedStatement's parameters.
public void addBatch() throws SQLException	Adds a parameter to a PreparedStatement batch of commands.
Execute	
public ResultSet executeQuery() throws SQLException	Executes a query in a PreparedStatement and returns a ResultSet.
public int executeUpdate() throws SQLException	Executes the SQL statement that contains an INSERT, UPDATE, or DELETE statement in a PreparedStatement.
public boolean execute() throws SQLException	Executes an SQL statement in a PreparedStatement.
Parameters	
public void setNull(int parameterIndex, int sqlType) throws SQLException	Sets the parameter to a null value.
public void setString(int parameterIndex, String x) throws SQLException	Sets the String value.
public void setDate(int parameterIndex, Date x) throws SQLException	Sets the java.sql.Date value.
public void setTime(int parameterIndex, Time x) throws SQLException	Sets the java.sql.Time value.
public void setTimestamp(int parameterIndex, Timestamp x) throws SQLException	Sets the java.sql.Timestamp value.

Table 6-6. ***java.sql Package*** *Public Interface PreparedStatement*

Syntax	Descriptions
public void clearParameters() throws SQLException	Clears the current parameter values.
public void setNull(int paramIndex, int sqlType, String typeName) throws SQLException	Sets a parameter to a null value.

Table 6-6. ***java.sql Package*** *Public Interface PreparedStatement* (continued)

Syntax	Descriptions
public void close() throws SQLException	Closes a ResultSet.
public ResultSetMetaData getMetaData() throws SQLException	Retrieves metadata.
public Statement getStatement() throws SQLException	Retrieves the Statement object.
Warnings	
public SQLWarning getWarnings() throws SQLException	Retrieves the first warning reported on a ResultSet.
public void clearWarnings() throws SQLException	Clears all warnings reported on a ResultSet.
Virtual Cursor Movement	
public boolean next() throws SQLException	Moves the virtual cursor down one row.
public boolean isFirst() throws SQLException	Determines if the virtual cursor is on the first row.
public boolean isLast() throws SQLException	Determines if the virtual cursor is on the last row.

Table 6-7. ***Java.sql Package*** *Public Interface ResultSet*

Syntax	Descriptions
public boolean first() throws SQLException	Moves the virtual cursor to the first row.
public boolean last() throws SQLException	Moves the virtual cursor to the last row.
public int getRow() throws SQLException	Determines the current row number.
public boolean absolute(int row) throws SQLException	Moves the virtual cursor to the row.
public boolean relative(int rows) throws SQLException	Moves the virtual cursor a relative number of rows.
public boolean previous() throws SQLException	Moves the virtual cursor to the previous row.
Fetch Size	
public void setFetchSize(int rows) throws SQLException	Gives the database driver a hint as to how many rows should be fetched when more rows are needed for this ResultSet.
public int getFetchSize() throws SQLException	Determines the fetch size for this ResultSet.
Rows	
public boolean rowUpdated() throws SQLException	Determines if a row has been updated.
public boolean rowInserted() throws SQLException	Determines if a row has had an insertion.
public boolean rowDeleted() throws SQLException	Determines if a row is deleted.
public void updateNull(int columnIndex) throws SQLException	Inserts a null into a column.

Table 6-7. ***java.sql Package*** *Public Interface ResultSet* (continued)

Syntax	Descriptions
public void insertRow() throws SQLException	Inserts a row into a ResultSet and into the database.
public void updateRow() throws SQLException	Updates the underlying database with the row of this ResultSet.
public void deleteRow() throws SQLException	Deletes a row from a ResultSet and from the database.
public void refreshRow() throws SQLException	Refreshes the current row with the current value in the database.
public void cancelRowUpdates() throws SQLException	Cancels updates.

Table 6-7. ***java.sql Package*** *Public Interface ResultSet* (continued)

Syntax	Descriptions
public int getColumnCount() throws SQLException	Retrieves the number of columns in a ResultSet object.
public boolean isCaseSensitive(int column) throws SQLException	Determines if a column is case sensitive.
public boolean isSearchable(int column) throws SQLException	Determines if a column can be used in a WHERE clause.
public boolean isCurrency(int column) throws SQLException	Determines if a column is a cash value.
public int isNullable(int column) throws SQLException	Determines if a null value can be placed in the column.
public boolean isSigned(int column) throws SQLException	Determines if values are signed numbers.

Table 6-8. ***java.sql Package*** *Public Interface ResultSetMetaData*

Syntax	Descriptions
Columns	
public int getColumnDisplaySize(int column) throws SQLException	Retrieves the column's maximum width in characters.
public String getColumnLabel(int column) throws SQLException	Retrieves the column's label.
public String getColumnName(int column) throws SQLException	Retrieves the column's name.
public int getPrecision(int column) throws SQLException	Retrieves the column's precision.
public int getColumnType(int column) throws SQLException	Retrieves the data type of the column.
public String getColumnTypeName (int column) throws SQLException	Retrieves the name of the column data type.
public boolean isReadOnly(int column) throws SQLException	Indicates if a column is read-only.
public boolean isWritable(int column) throws SQLException	Indicates if a column is writable.

Table 6-8. ***java.sql Package*** *Public Interface ResultSetMetaData* (continued)

Syntax	Descriptions
public int getSavepointId() throws SQLException	Retrieves a savepoint ID.
public String getSavepointName() throws SQLException	Retrieves a savepoint name.

Table 6-9. ***java.sql Package*** *Public Interface Savepoint*

Syntax	Descriptions
public void close() throws SQLException	Releases a Statement object.
public ResultSet getResultSet() throws SQLException	Retrieves the current ResultSet object.
public int getUpdateCount() throws SQLException	Determines the number of rows that were updated.
public Connection getConnection() throws SQLException	Retrieves a Connection object.
public ResultSet getGeneratedKeys() throws SQLException	Retrieves auto-generated keys.
Timeout	
public int getQueryTimeout() throws SQLException	Determines the maximum number of seconds for a Statement object to execute.
public void setQueryTimeout(int seconds) throws SQLException	Sets the maximum number of seconds for a Statement object to execute.
Rows	
public int getMaxRows() throws SQLException	Retrieves the maximum number of rows for a ResultSet object.
public void setMaxRows(int max) throws SQLException	Sets the maximum number of rows for a ResultSet object.
public void setFetchSize(int rows) throws SQLException	Gives database a hint for how many rows to retrieve when more rows are needed for the ResultSet.
public int getFetchSize() throws SQLException	Determines the number of rows for the fetch size.
Warnings	
public SQLWarning getWarnings() throws SQLException	Retrieves the first warning on a Statement object.
public void clearWarnings() throws SQLException	Clears all the warnings reported on a Statement object.

Table 6-10. ***java.sql Package*** *Public Interface Statement*

Syntax	Descriptions
Execute	
public boolean execute(String sql) throws SQLException	Executes a SQL statement.
public ResultSet executeQuery (String sql) throws SQLException	Executes a SQL statement and returns a ResultSet object.
public int executeUpdate(String sql) throws SQLException	Executes a SQL statement that contains INSERT, UPDATE, or DELETE statements or DDL statements.
Batch	
public void addBatch(String sql) throws SQLException	Adds a SQL statement to a batch.
public void clearBatch() throws SQLException	Empties a batch of SQL statements.
public int[] executeBatch() throws SQLException	Executes a batch of SQL statements.

Table 6-10. ***java.sql Package*** *Public Interface Statement* (continued)

Syntax	Descriptions
public Connection getConnection() throws SQLException	Opens a connection to a data source.
public Connection getConnection (String username, String password) throws SQLException	Opens a connection to a data source.

Table 6-11. ***javaxsql Package*** *Public Interface DataSource.*

Syntax	Descriptions
public void setUrl(String url) throws SQLException	Sets the URL a RowSet.
public boolean isReadOnly()	Determines if a RowSet object is read-only.
public void setReadOnly(boolean value) throws SQLException	Sets a RowSet object to read-only.
public int getMaxFieldSize() throws SQLException	Determines the maximum number of bytes that can be placed in a column.
public int getMaxRows() throws SQLException	Determines the maximum number of rows that can be in a RowSet.
DataSource	
public String getDataSourceName()	Determines the logical name of the data source for a RowSet.
public void setDataSourceName(String name) throws SQLException	Sets the data source name for a RowSet.
Username and Password	
public String getUsername()	Determines the username used to create a database connection for a RowSet.
public void setUsername(String name) throws SQLException	Sets the username for a RowSet.
public String getPassword()	Determines the password used to create a database connection.
public void setPassword(String password) throws SQLException	Sets the database password for a RowSet.

Table 6-12. ***javaxsql Package*** *Public Interface RowSet*

Syntax	Descriptions
Timeout	
public int getQueryTimeout() throws SQLException	Determines the maximum time the driver will wait for a statement to execute.
public void setQueryTimeout(int seconds) throws SQLException	Sets the maximum time the driver will wait for a statement to execute.

Table 6-12. ***Javaxsql Package*** *Public Interface RowSet* (continued)

The Complete Reference



Chapter 7

JDBC and Embedded SQL

A J2EE component interacts with a database management system (DBMS) using Java data objects and SQL statements that are embedded into the J2EE component and are executed by the Java data objects.

Java data objects are discussed in the previous chapter. This chapter continues with a detailed look at how to create a SQL statement, execute the SQL statement, and interact with rows of data returned to the J2EE component by the DBMS in response to the request for data.

SQL is presented in this chapter using a practical approach. Each section of the chapter focuses on common tasks that are used in a J2EE component, rather than on SQL keywords. This means you can quickly locate a routine in the chapter and then incorporate the routine directly into your J2EE component.

Model Programs

There are many programming styles that can be used to write the data access portion of a J2EE component. Two programming styles are used in this chapter to illustrate how a J2EE component can interact with a DBMS. These are referred to as "Model A program" and "Model B program."

The Model A program style is used to execute SQL requests that use the execute() or executeUpdate() methods (see Chapter 6) and don't return a ResultSet. The Model B program style is used to execute SQL requests that use the executeQuery() method, which returns a ResultSet. Each of these is described in detail in the next two sections of this chapter.

Both program styles are designed to minimize the code clutter in this chapter. Code clutter occurs whenever many lines of code are used in code examples, but only a few lines of code change between examples.

In an effort to reduce code clutter, each complete program model is presented at the beginning of the chapter. One or two comments are used to indicate where within the program model to place code segments that are used to illustrate SQL routines throughout the rest of the chapter.

These code segments are contained either within a try { } block or within the DownRow() method. The try { } block is where the SQL statement is created and executed. The DownRow() method is where the J2EE component interacts with the ResultSet.

Let's say that you want to update a row in a database. As you'll remember from the last chapter, the executeUpdate() method is used to execute the query and no ResultSet is returned. Therefore, you can replace the second try { } block in the Model A program with the code segment contained in the section of this chapter that shows how to update a row in the database. As you'll see in the next section, the Model A program does not contain a DownRow() method because no ResultSet is returned.

Suppose that you want to retrieve data from the database. The executeQuery() method is used in this scenario. The Model B program is used for this purpose because a ResultSet

is returned. In this case, the section of this chapter that shows how to retrieve data from a database contains two code segments. The first code segment contains the query and replaces the second try { } block in the Model B program. The other code segment replaces the DownRow() method of the Model B program and interacts with the ResultSet.

The next two sections of this chapter explain how each model program works. Other sections of this chapter discuss commonly used tasks and show how to incorporate those tasks into a J2EE component. Each of these sections also indicates which program model to use with the task.

Model A Program

The Model A program shown in Listing 7-1 is designed to execute queries that do not return a ResultSet. The Model A program is organized into the ModelA() constructor and the main(). The main() creates an instance of the ModelA object called sql1, which causes the execution of the constructor.

The constructor begins by creating three String objects. These are url, userID, and password. The url is assigned the URL of the database. In this example, the JDBC/ODBC bridge is used to connect to the CustomerInformation database. Of course, the URL used in your J2EE component replaces the URL in Model A and will represent the JDBC driver and database that is specific to your J2EE component. Likewise, the database used with your J2EE component may or may not require a user ID and password. If these are required, they will undoubtedly be different than the user ID and password used in Model A.

The first try { } block in Model A uses the Class.forName() method to load the JDBC driver into the Java Virtual Machine (JVM). You will probably need to replace the driver used in Model A with the appropriate JDBC driver for your J2EE component.

Once the JDBC driver is loaded, the getConnection() method is called to open a connection to the database using the url, userID, and password String objects. The getConnection() method returns a Connection object called database, which is declared above the definition of the constructor.

Two catch { } blocks follow the first try { } block and are used to trap exceptions that occur while the driver is loaded and a connection is established with the database. You can change the text of messages shown in Model A, which are displayed when an exception is caught, to a message that is more conducive to your programming style and J2EE requirements.

The second try { } block in Model A contains a comment. You should replace this try { } block with the try { } that appears in the section of this chapter that describes the task that you want performed by the J2EE component. You'll notice as you read through this chapter that the try { } block within each section references a Statement object called DataRequest, which is declared as a private member of the ModelA class.

The second try { } block is followed by a catch { } block that traps SQL exceptions that occur during the execution of the query in the second try { } block.

Anytime a database connection is closed, you should first check for a null value on the Connection object. If the connection is no longer valid, the close() method will throw a NullPointerException.

sting 7-1 A model J2EE mponent that doesn't retrieve ormation from a atabase.

```
import java.sql.*;
public class ModelA
{
  private Connection Database;
  private Statement DataRequest;
  public ModelA ()
  {
    String url = "jdbc:odbc:CustomerInformation";
    String userID = "jim";
    String password = "keogh";
    try {
      Class.forName( "sun.jdbc.odbc.JdbcOdbcDriver");
      Database = DriverManager.getConnection(url,userID,password);
    }
    catch (ClassNotFoundException error) {
      System.err.println("Unable to load the JDBC/ODBC bridge." + error);
      System.exit(1);
   }
   catch (SQLException error) {
     System.err.println("Cannot connect to the database. "+ error);
     if(Database != null) {
       try {
         Database.close();
       }
       catch(SQLException er){}
       }
     System.exit(2);
}
try {
  // insert example code here
}
catch ( SQLException error ) {
  System.err.println("SQL error." + error);
  if(Database != null) {
    try {
      Database.close();
    }
    catch(SQLException er){}
    }
  System.exit(3);
  }
  if(Database != null) {
    try {
      Database.close();
    }
    catch(SQLException er){}
```

```
        }
      }
  public static void main ( String args [] )
  {
    final ModelA sql1 = new ModelA ();
    System.exit ( 0 ) ;
  }
}
```

Model B Program

The Model B program shown is in Listing 7-2 is designed for use by J2EE components that retrieve information from a database. Notice that the Model B program is similar to the Model A program in that the constructor definition (except for the name of the constructor) and the main() are identical. Therefore, refer to the previous section for a description of the constructor and main().

The second try { } block in Model B is where you place the try { } block contained in sections of this chapter that discuss how to retrieve information from a database. This try { } executes a query and returns a ResultSet object called Results, which is declared as a private member of the ModelB class.

Model B also contains DisplayResults() and DownRow(). DisplayResults() is passed the ResultSet returned in the second try { } block, which is used to move the virtual cursor to the first row of the ResultSet using the next() method (see Chapter 6). The next() returns a boolean value that is assigned to the Records variable. The if statement evaluates the value of Records and displays a "No data returned" message if Records contains a false value indicating that there isn't any data in the row.

However, a true value causes the do...while loop to execute, which is where a call to DownRow() is made, passing it the ResultSet. Then, DownRow() retrieves data stored in columns of the ResultSet and assigns those values to variables. Each section of this chapter that discusses a task that retrieves data from a database has its own DownRow(). Therefore, you need to replace the DownRow() in Model B with the DownRow() that is associated with the task that you want performed by the J2EE component.

After DownRow() extracts data from the current row of the ResultSet, control returns to DisplayResults() where the next() method is called and the results are evaluated by the while. If the next() method returns a true value, then DownRow() is recalled; otherwise, the do...while loop is exited and control returns to the constructor.

DisplayResults() also contains a catch { } block that traps errors thrown by statements within the try { } block.

Listing 7-2 A model J2EE component that retrieves information from a database.

```
import java.sql.*;
public class ModelB
{
  private Connection Database;
  private Statement DataRequest;
  private ResultSet Results;
  public ModelB ()
```

```
{
  String url = "jdbc:odbc:CustomerInformation";
  String userID = "jim";
  String password = "keogh";
  try {
    Class.forName( "sun.jdbc.odbc.JdbcOdbcDriver");
    Database = DriverManager.getConnection(url,userID,password);
  }
  catch (ClassNotFoundException error) {
    System.err.println("Unable to load the JDBC/ODBC bridge." + error);
    System.exit(1);
  }
  catch (SQLException error) {
    System.err.println("Cannot connect to the database." + error);
    System.exit(2);
  }
  try {
    // Enter example code here
  }
  catch ( SQLException error ){
    System.err.println("SQL error." + error);
    if(Database != null) {
      try {
        Database.close();
      }
      catch(SQLException er){}
      }
    System.exit(3);
  }
  if(Database != null) {
    try {
      Database.close();
    }
    catch(SQLException er){}
    }
  }
  private void DisplayResults (ResultSet DisplayResults) throws SQLException
  {
    boolean Records = DisplayResults.next();
    if (!Records ) {
      System.out.println( "No data returned");
      return;
    }
    try {
      do {
        DownRow( DisplayResults) ;
      } while ( DisplayResults.next() );
    }
    catch (SQLException error ) {
```

```
        System.err.println("Data display error." + error);
        if(Database != null) {
          try {
            Database.close();
          }
          catch(SQLException er){}
          }
        System.exit(4);
        }
      }
      private  void DownRow ( ResultSet DisplayResults ) throws SQLException
      {
        //Enter new DownRow() code here
      }
      public static void main ( String args [] )
      {
        final ModelB sql1 = new ModelB ();
        System.exit ( 0 ) ;
      }
}
```

Tables

Chapter 5 showed how to design a database. A database design is used as the basis for building tables and indexes that comprise a database. Tables and indexes are created using a query written using SQL. Typically, the database administrator writes and executes the query that creates tables and indexes, although it is possible to have a J2EE component or a Java application execute the same query to create tables and indexes

The following sections demonstrate various techniques that are used in a query to create a table. Later in this chapter, techniques are presented to create indexes. These queries can be executed from within the Model A program, as discussed previously in this chapter, or executed outside of a Java program using an interface such as ISQL that is supplied with a DBMS.

The user ID that is used to log onto the DBMS and execute the query must have rights to create and drop tables. Dropping a table removes the table from the database, which is discussed later in this chapter.

Depending on the policies of your IT department, developers are commonly given rights to create and drop tables in the development environment. Only the database administrator has these rights to do the same in the production environment.

Creating a Table

You create a table by formatting a SQL query. The query contains the CREATE TABLE SQL statement that contains the name of the table, which is called CustomerAddress in Listing 7-3 because the table will contain address information on customers.

Following the table name are the column definitions, which are enclosed within parentheses. The first column is called CustomerNumber. The data type of the column and the column size follows the column name. A comma separates each column.

The CustomerAddress table contains four columns. These are as follows:

- **CustomerNumber** The customer number is a string of a maximum of 30 characters.
- **CustomerStreet** The customer street is a string of a maximum of 30 characters.
- **CustomerCity** The customer city is a string of a maximum of 30 characters.
- **CustomerZip** The customer ZIP code is a string of a maximum of 30 characters.

All the SQL commands are enclosed in double quotations and assigned to a String object called query. Next, the program creates a SQL statement using the createStatement() method of the Connection object (see Chapter 6). This method returns the handle to the DBMS to the DataRequest object, which is an object of the Statement class.

The query is sent to the DBMS using the executeQuery() method. If successful, the DBMS creates the table. If unsuccessful, the table isn't created and the DBMS returns an error message. The SQL statement is then closed using DataRequest.close().

ting 7-3 Creating a table.

```
try {
  String query = "CREATE TABLE CustomerAddress (" +
    " CustomerNumber CHAR(30), " +
    " CustomerStreet CHAR(30), " +
    " CustomerCity CHAR(30), " +
    " CustomerZip CHAR(30))";
  DataRequest = Database.createStatement();
  DataRequest.execute(query);
  DataRequest.close();
}
```

Requiring Data in a Column

There are business rules that require a value in specific columns of a row such as the columns that contain a customer's name. You can require the DBMS to reject rows that are missing a value in specific columns by using the NOT NULL clause in the query when the table is created. Listing 7-4 illustrates how the NOT NULL clause is used in a query.

In this example, a CustomerAddress table is being created that contains the same four columns that are used in Listing 7-3 except all of these columns require a value. That is, a row that is missing a value in any of these columns is rejected by the DBMS and will not be inserted into a table.

Listing 7-4 Requiring a value be placed in a column of a table.

```
try {
  String query = "CREATE TABLE CustomerAddress ( " +
    "CustomerNumber CHAR(30) NOT NULL," +
    "CustomerStreet CHAR(30) NOT NULL," +
    "CustomerCity CHAR(30) NOT NULL," +
    "CustomerZip CHAR(30) NOT NULL)";
  DataRequest = Database.createStatement();
  DataRequest.execute (query );
  DataRequest.close();
}
```


Setting a Default Value for a Column

The DBMS can enter a default value into a column automatically if the column is left empty whenever a new row is inserted into the table. You determine the value entered by the DBMS by creating a default value when you create the table. Any value can be used as the default value as long as the value conforms to the data type and size of the column.

Let's say most of your customers reside within the 07660 ZIP code. Some customers living in the area fail to give their sales representative their ZIP code when placing their first order. They assume that since the ZIP code is the same as your business, you already know the ZIP code.

A default value is assigned to a column when you create the table by using the DEFAULT clause. The DEFAULT clause is followed in the query by the value the DBMS will place in the column if the column is empty in the incoming row. Listing 7-5 illustrates how to set a default value for a column when you create a table.

Listing 7-5 Creating a default value for a column.

```
try {
  String query = "CREATE TABLE CustomerAddress ( " +
    "CustomerNumber CHAR(30) NOT NULL," +
    "CustomerStreet CHAR(30) NOT NULL," +
    "CustomerCity CHAR(30) NOT NULL," +
    "CustomerZip CHAR(30) NOT NULL DEFAULT '07660')";
  DataRequest = Database.createStatement();
  DataRequest.executeUpdate (query );
  DataRequest.close();
}
```

Dropping a Table

A developer may have the right to remove a table, but this is usually reserved for the development environment only. The decision to drop a table shouldn't be made lightly

because once a table is dropped you are unable to recover the table. Instead, the table must be re-created and the data must be reinserted into the table.

In addition to losing data elements stored in the table, dropping a table may also affect the integrity of the database and tables that relate to values in the dropped table.

Using the Drop Table statement in the query drops a table. As illustrated in Listing 7-6, the Drop Table statement contains the name of the table that is to be dropped.

sting 7-6 Dropping a table.

```
try {
  String query = new String ("DROP TABLE CustomerAddress");
  DataRequest = Database.createStatement();
  DataRequest. execute(query);
  DataRequest.close();
}
```

Indexing

The database schema describes indexes along with tables that are used in the database. Writing a query, as is illustrated in the next section, creates an index. Likewise, a query is used to remove an existing index, which is referred to as "dropping an index."

Developers are typically permitted to create or drop tables within the development environment and only the database administrator has rights to do the same in the production environment, although these rights may vary according to IT department policy.

Executing the query to create or drop an index follows similar procedures as used to create and drop a table. That is, the query can be executed within or outside a J2EE component or Java application as is described in the earlier "Tables" section. Listings in the following sections should be used with the Model A program since no ResultSet is returned when the query is executed.

Creating an Index

An index is created by using the CREATE INDEX statement in a query, as is illustrated in Listing 7-7. The CREATE INDEX statement contains the name of the index and any modifier that describes to the DBMS the type of index that is to be created. In addition, the CREATE INDEX statement uses the ON clauses to identify the name of the table and the name of the column whose value is used for the index.

Listing 7-7 creates an index called CustNum that contains values in the CustomerNumber column of the CustomerAddress table. The CustNum index doesn't have duplicate key values and therefore the UNIQUE modifier is used in the CREATE INDEX statement to prohibit duplicate key values from being included in the index.

sting 7-7 reating a unique index.

```
try {
  String query = "CREATE UNIQUE INDEX CustNum " +
```

```
        "ON CustomerAddress (CustomerNumber)";
    DataRequest = Database.createStatement();
    DataRequest.execute (query);
    DataRequest.close();
}
```

Designating a Primary Key

A primary key (see Chapter 5) can be designated when a table is created by using the Primary Key modifier as is illustrated in Listing 7-8. This example contains a query that creates a table, as described previously in this chapter, and one column within the table is designated the primary key.

You'll remember that the primary key uniquely identifies each row in a table. Therefore, the column or columns chosen as the primary key must have unique values. That is, the column cannot contain duplicate values.

As shown in Listing 7-8, the OrderNumber column is used as the primary key and is identified in the PRIMARY KEY modifier. Once a column is designated as the primary key, the DBMS prevents duplicate and null values from being placed in the column.

Listing 7-8 Designating a primary key for a table.

```
try {
  String query = "Create Table Orders ( " +
    "OrderNumber CHAR(30) NOT NULL, " +
    "CustomerNumber CHAR(30),   " +
    "ProductNumber CHAR(30), " +
    "CONSTRAINT ORDERS_PK PRIMARY KEY (OrderNumber))";
  DataRequest = Database.createStatement();
  DataRequest. execute(query);
  DataRequest.close();
}
```

Creating a Secondary Index

A secondary index is created by using the CREATE INDEX statement in a query without the use of the UNIQUE modifier as is described previously in this chapter. This means that a secondary index can have duplicate values.

Listing 7-9 illustrates how to create a secondary index. In this example, the index is called CustZip whose key is the CustomerZip column of the CustomerAddress table.

Listing 7-9 Creating a secondary index.

```
try {
  String query = new String ("CREATE INDEX CustZip " +
    "ON CustomerAddress (CustomerZip) ");
  DataRequest = Database.createStatement();
```

```
    DataRequest. execute(query);
    DataRequest.close();
}
```

Creating a Clustered Index

A clustered index is an index whose key is created from two or more columns of a table as you learned in Chapter 5. For example, combining a customer's last name with a customer first name is a typical key for a clustered index.

Listing 7-10 shows how to create a clustered index. This example creates an index called CustName and uses the LastName and FirstName columns of the Customers table as the key for the index.

It is important to realize that the order in which column names appear in the ON clause plays a critical role in the index. As you remember from Chapter 5, the value of the second column is concatenated to the value of the first column that appears in the query. This means that the key in this key would be KeoghJim, assuming Jim Keogh is a value in the FirstName and LastName columns of the table.

Therefore, the DBMS may not be able to find a search a criterion that is JimKeogh if the key to the clustered index is KeoghJim. Of course, this assumes that the DBMS uses the clustered index for the search.

ng 7-10 eating a ustered index.

```
try {
    String query = "CREATE INDEX CustName " +
      " ON Customers (LastName, FirstName)";
    DataRequest = Database.createStatement();
    DataRequest.execute (query);
    DataRequest.close();
}
```

Dropping an Index

An existing index can be removed from the database by using the DROP INDEX statement. The DROP INDEX statement is similar in construction to the DROP TABLE statement, discussed previously in this chapter.

Listing 7-11 shows how to create a query that removes an index. In this example, the CustName index that is associated with the Customers table is dropped.

ng 7-11 ropping n index.

```
try {
    String query = new String("DROP INDEX CustName ON Customers ");
    DataRequest = Database.createStatement();
    DataRequest. execute(query);
    DataRequest.close();
}
```

Inserting Data into Tables

Once a database, tables, and indexes are created, a J2EE component can insert a new row into a table. In the next several sections of this chapter, you'll learn the technique for writing a query that inserts data into all or some of the columns of a new row.

As you learned in Chapter 6, the executeUpdate() method is used to execute a query that, among other tasks, inserts a row. The executeUpdate() method does not return a ResultSet; therefore, you should use the Model A program with code segments shown in this section that inserts a row into a table.

These code segments insert a row into the Customers table that is defined next. Make sure that you create this table in the CustomerInformation database, using techniques described previously in this chapter, before attempting to insert a new row.

- CustomerNumber, Numeric
- FirstName, VARCHAR, 50
- LastName, VARCHAR, 50
- DateOfFirstOrder, Date

Inserting a Row

The INSERT INTO statement is used to insert a new row into a table. This is illustrated in Listing 7-12. The INSERT INTO statement contains the name of the table into which the row is to be inserted and the name of the columns in which values are inserted. The VALUES clause is used to define the values that are to be placed into the row.

Although the column names are optional, you should always use column names because it is a bad practice to assume that columns appear in a particular order in the table. This is because the table might be restructured by the database administrator without your knowledge.

Each value is placed in the VALUES clause in the same order as the corresponding column's name. In Listing 7-12, a new row is being inserted into the Customers table. The first value in the VALUES clause is the customer number and is followed by the customer's first name, last name, and date of the first order placed by the customer. This order is the same order of the columns names.

You can exclude a value from appearing in a column by excluding the column from the query. For example, you could have removed the LastName column and corresponding value, and the LastName column will be employed for that row.

Be careful that you don't exclude a value that is required by the table; otherwise, the DBMS rejects the row—causing an exception to be thrown.

Listing 7-12 Inserting a row into a table.

```
try {
  String query = "INSERT INTO Customers  " +
    " (CustomerNumber, FirstName, LastName, DateOfFirstOrder) " +
    " VALUES (1,'Mary','Smith','10/10/2001') ";
```

```
    DataRequest = Database.createStatement();
    DataRequest.executeUpdate (query);
    DataRequest.close();
}
```

Inserting the System's Date into a Column

Sometimes business rules require that the current date be placed into a column of a new row. For example, the system's date is usually used as the date of a new order. You can place the system's date into a column by calling the CURRENT_DATE function. This is not implemented in all versions of DBMS. If you're using Microsoft Access for your DBMS, then you use NOW. Oracle uses SYSDATE. DB2 uses CURRENT DATE (no underscore).

The CURRENT_DATE function directs the DBMS to use the system's date of the server as the value for the column. This is illustrated in Listing 7-13. In this example, the CURRENT_DATE function is called to provide the value for the DateOfFirstOrder column in the new row of the Customers table.

ing 7-13 rting the system's date into column.

```
try {
String query = new String ("INSERT INTO Customers (CustomerNumber ,FirstName, "+
  " LastName, DateOfFirstOrder )" +
  " VALUES ( 4,'Mary','Jones', CURRENT_DATE)");
DataRequest = Database.createStatement();
DataRequest.executeUpdate (query );
DataRequest.close();
}
```

Inserting the System Time into a Column

Call the CURRENT_TIME() function whenever a column requires the current time. The CURRENT_TIME() function is called by the DBMS when the query is processed and returns the current time of the server. Listing 7-14 illustrates how to call the CURRENT_TIME() function within a query.

Caution *A word of caution: Implementation of dates and time functions has not been standardized among vendors. This means that some or all date and time functions discussed in this section may not work with all DBMSs. You should consult with your database administrator or DBMS vendor to determine the proper date and time functions to use with your DBMS.*

ting 7-14 erting the current time into a column.

```
try {
  String query = new String ("INSERT INTO Customers (CustomerNumber ,FirstName, "+
    " LastName, TimeOfFirstOrder ) " +
```

```
      " VALUES ( 2,'Bob','Jones', CURRENT_TIME() )") ;
    DataRequest = Database.createStatement();
    DataRequest.executeUpdate(query );
    DataRequest.close();
}
```

Inserting a Timestamp into a Column

A timestamp consists of both the current data and time and is used in applications where both date and time are critical to the operation of the business. For example, Wall Street firms timestamp orders as they arrive, and this is used to correlate with the market price of stocks at precisely the date and time the order was processed.

Caution *A word of caution: Not all DBMSs support timestamp. Therefore, consult with your database administrator or DBMS vendor for comparable functionality if examples in this section do not work with your DBMS.*

You can place the timestamp into a column by calling the TIMESTAMP() function, as is illustrated in Listing 7-15. The server's system date and time is returned by the function and placed into the column.

ing 7-15 serting a mestamp into a column.

```
try {
  String query = new String ("INSERT INTO Customers (CustomerNumber ,FirstName, "+
    " LastName, FirstOrder ) )" +
    " VALUES ( 2,'Bob','Jones', CURRENT_TIMESTAMP() ) ";
  DataRequest = Database.createStatement();
  DataRequest.executeUpdate (query );
  DataRequest.close();
}
```

Selecting Data from a Table

Retrieving information from a database is the most frequently used routine of J2EE components that interact with a database. Some developers feel that retrieving information is one of the most complex routines to write because there are various ways information can be retrieved.

The following sections of this chapter illustrate the various techniques that can be used to retrieve data using the executeQuery() method. As you learned in Chapter 6, the executeQuery() method returns a ResultSet. Therefore, you'll need to use the Model B program with these code segments.

Each section contains a try { } block and a DownRow() method, which replace the second try { } block in the Model B program and the DownRow() method in the Model B program as described previously in this chapter.

Code segments illustrated in these sections retrieve data from the Customers table. Here are the columns that comprise the Customers table, which you should create before executing the code segments:

- FirstName, VARCHAR, 50
- LastName, VARCHAR, 50
- Street, VARCHAR, 50
- City, VARCHAR, 50
- State, VARCHAR, 2
- ZipCode, VARCHAR, 12

Make sure that you insert rows into the Customers table before trying to execute code segments that appear in the following sections. Table 7-1 contains rows of data that you can use in the Customers table.

Selecting All Data from a Table

The SELECT statement is used to retrieve data from a table, as illustrated in Listing 7-16. In this example, all the columns of all the rows of the Customers table are returned in the ResultSet by the DBMS after the query is processed.

As you'll see in the next few sections, names of columns that you want retrieved from the table are included in the SELECT statement. The executeQuery() method is used to execute the query and returns a ResultSet object. The ResultSet object is passed to the DisplayResults() method, which is defined in the Model B program at the beginning of this chapter. The DisplayResults() method is responsible for moving through the ResultSet and displaying the contents of the ResultSet on the screen.

FirstName	LastName	Street	City	State	ZipCode
Bob	Jones	5 First Street	New York City	NY	07555
Mary	Smith	8 Third Street	Dallas	TX	75553
Bob	Jones	5 First Street	New York City	NY	07555
Mark	Russell	3 Sixth Street	Los Angeles	CA	82272
Susan	Allen	18 Fifth Street	Chicago	IL	45003
Mark	Russell	3 Sixth Street	Los Angeles	CA	82272

Table 7-1. *Rows That Need to Be Inserted into the Customers Table*

As you'll recall from the beginning of this chapter, the DisplayResults() method calls the DownRow() method, which extracts data from the current row of the ResultSet and displays the row on the screen.

Listing 7-17 contains the DownRow() method for Listing 7-18. In this example, the DownRow() method is passed the ResultSet object from the DisplayResults() method, as described at the beginning of this chapter.

The DownRow() method then declares String objects that are assigned the value of each column in the ResultSet. There is also a String object called printrow, which is assigned all the other String objects and is used to display the values on the screen.

The getString() method is called to gather the value of a specific column in the ResultSet. Using the number of the column, as described in Chapter 6, specifies the column in the getString() method. The getString() method returns the value of the column, which is assigned to a String object. Remember, the getString() method is really the getxxx() method discussed in Chapter 6. The data type of the column replaces the xxx.

Listing 7-16 Selecting all data from a table.

```
try {
   String query = new String ("SELECT "   +
     " FirstName, LastName, Street, City, State, ZipCode " +
     " FROM Customers");
   DataRequest = Database.createStatement();
   Results = DataRequest.executeQuery (query);
   DisplayResults (Results);
   DataRequest.close();
}
```

Listing 7-17 Copying values from the ResultSet.

```
private  void DownRow ( ResultSet DisplayResults ) throws SQLException
{
   String FirstName= new String();
   String LastName= new String();
   String Street= new String();
   String City = new String();
   String State = new String();
   String ZipCode= new String();
   String printrow;
   FirstName = DisplayResults.getString ( 1 ) ;
   LastName = DisplayResults.getString ( 2 ) ;
   Street = DisplayResults.getString ( 3 ) ;
   City = DisplayResults.getString ( 4 ) ;
   State = DisplayResults.getString ( 5 ) ;
   ZipCode = DisplayResults.getString ( 6 ) ;
   printrow = FirstName + " " + LastName + " " +
     City + " " + State +  " " + ZipCode;
   System.out.println(printrow);
}
```

Requesting One Column

You can specify a column that you want returned from the table by using the column name in the SELECT statement, as illustrated in Listing 7-18. In this example, the LastName column of all the rows in the Customers table is returned in the ResultSet. Listing 7-19 contains the DownRow() method that is used to retrieve data from the ResultSet that is returned by Listing 7-18.

Listing 7-18 Selecting one column to be retrieved from a table.

```
try {
   String query = new String ("SELECT LastName FROM Customers");
   DataRequest = Database.createStatement();
   Results = DataRequest.executeQuery (query);
   DisplayResults (Results);
   DataRequest.close();
}
```

Listing 7-19 Copying values from the ResultSet.

```
private  void DownRow ( ResultSet DisplayResults )throws SQLException
{
   String LastName= new String();
   LastName = DisplayResults.getString ( 1 ) ;
   System.out.println(LastName);
}
```

Requesting Multiple Columns

Multiple columns can be retrieved by specifying the names of the columns in the SELECT statement, similar to the way you select one column in Listing 7-18. This is illustrated in Listing 7-20 where the FirstName and LastName columns for all the rows in the Customers table are returned in the ResultSet. Listing 7-21 shows the DownRow() method that is used to copy and display values from the ResultSet.

Listing 7-20 Copying values from the ResultSet.

```
try {
   String query = new String ("SELECT FirstName, LastName FROM Customers");
   DataRequest = Database.createStatement();
   Results = DataRequest.executeQuery (query);
   DisplayResults (Results);
   DataRequest.close();
}
```

Listing 7-21 Copying values from the ResultSet.

```
private  void DownRow ( ResultSet DisplayResults )throws SQLException
{
   String FirstName= new String();
   String LastName= new String();
```

```
    String printrow;
    FirstName = DisplayResults.getString ( 1 ) ;
    LastName = DisplayResults.getString ( 2 ) ;
    printrow = FirstName + " " + LastName;
    System.out.println(printrow);
}
```

Requesting Rows

Specific rows can be retrieved from a column by using the WHERE clause in conjunction with the SELECT statement. The WHERE clause contains an expression that is used by the DBMS to identify rows that should be returned in the ResultSet. Any logical expression can be used to include or exclude rows based on values in columns of a row, which is discussed later in this chapter.

Listing 7-22 illustrates how to use the WHERE clause to specify rows. In this example, the SELECT statement retrieves all the columns of rows in the Customers table where the LastName column has the value 'Jones'. You can use the DownRow() method in Listing 7-17 to copy and display values in the ResultSet that is returned by Listing 7-22.

Listing 7-22 Selecting particular rows from a table.

```
try {
  String query = new String ("SELECT "  +
    " FirstName, LastName, Street, City, State, ZipCode " +
    " FROM Customers " +
    " WHERE LastName = 'Jones' ");
  DataRequest = Database.createStatement();
  Results = DataRequest.executeQuery (query);
  DisplayResults (Results);
  DataRequest.close();
}
```

Requesting Rows and Columns

A query can select less than all the columns and all the rows of a table by using a combination of techniques shown previously in this chapter, which is illustrated in Listing 7-23. In this example, the FirstName and LastName of rows in the Customers table are returned in the ResultSet as long as the value of the LastName column is 'Jones'. You can use the DownRow() method shown in Listing 7-21 to copy and display values of the ResultSet returned by Listing 7-23.

Listing 7-23 Retrieving specific columns and rows.

```
try {
  String query = new String("SELECT FirstName, LastName " +
    " FROM Customers " +
    " WHERE LastName = 'Jones' ");
```

```
    DataRequest = Database.createStatement();
    Results = DataRequest.executeQuery (query);
    DisplayResults (Results);
    DataRequest.close();
}
```

AND, OR, and NOT Clauses

The WHERE clause in a SELECT statement can evaluate values in more than one column of a row by using the AND, OR, and NOT clauses to combine expressions in the WHERE clause. As you'll learn in the next three sections, compound expressions in the WHERE clause are individually evaluated, then the results of these evaluations are further evaluated based on the AND, OR, and NOT clauses that are used in the WHERE clause.

For example, the AND clause requires that both expressions in the compound expression evaluate to true before the WHERE clause expression evaluates true and includes a specified row in the ResultSet.

The OR clause requires that at least one of the expressions in the compound expression evaluate to true before the WHERE clause expression evaluates true. And the NOT clause is used to reverse the logic, changing an expression that evaluates true to a false. Each of these clauses is illustrated in the next three sections.

AND Clause

The purpose of the AND clause is to join two subexpressions together to form one compound expression. The AND clause tells the DBMS that the boolean value of both subexpressions must be true for the compound expression to be true. If the compound expression is true, then the current row being evaluated by the DBMS is returned to your program.

Listing 7-24 illustrates the use of the AND clause in the WHERE clause. In this example, the FirstName and LastName columns of rows from the Customers table are returned in the ResultSet only if the value of the LastName column is 'Jones' and the value of the FirstName column is 'Bob'. You can use the DownRow() method shown in Listing 7-21 to copy and display values of the ResultSet returned by Listing 7-24.

Listing 7-24 Using the AND clause to form a compound expression in the WHERE clause.

```
try {
    String query = new String ("SELECT FirstName, LastName " +
        " FROM Customers " +
        " WHERE LastName = 'Jones' " +
        " AND FirstName = 'Bob'");
    DataRequest = Database.createStatement();
    Results = DataRequest.executeQuery (query);
    DisplayResults (Results);
    DataRequest.close();
}
```

OR Clause

The OR clause is used to create a compound expression using two subexpressions, nearly identical to the way the AND clause joins subexpressions. However, the OR clause tells the DBMS that the compound expression evaluates to a boolean true if either of the two subexpressions evaluate to a boolean true.

Listing 7-25 illustrates the use of the OR clause in the WHERE clause. In this example, the FirstName and LastName columns of rows from the Customers table are returned in the ResultSet only if the value of the FirstName column is 'Mary' or the value of the FirstName column is 'Bob'. Listing 7-21 contains the DownRow() method that you can use with Listing 7-25.

Listing 7-25 Using the OR clause to form a compound expression in the WHERE clause.

```
try {
  String query = new String ("SELECT FirstName, LastName " +
    " FROM Customers " +
    " WHERE FirstName = 'Mary' " +
    " OR FirstName = 'Bob'");
  DataRequest = Database.createStatement();
  Results = DataRequest.executeQuery (query);
  DisplayResults (Results);
  DataRequest.close();
}
```

NOT Clause

The NOT clause reverses the logic of the subexpression contained in the NOT clause. If the subexpression evaluates to a boolean true value, then the NOT clause reverses the logic to return a boolean false value.

In contrast, if the subexpression evaluates to a boolean false value, then the compound expression evaluates to a boolean true value.

Listing 7-26 illustrates the use of the NOT clause. In this example, only rows that don't contain the value 'Mary' in the FirstName column are returned. Listing 7-21 contains the DownRow() method that can be used with Listing 7-26.

Listing 7-26 Using the NOT clause to form a compound expression in the WHERE clause.

```
try {
  String query = new String( "SELECT FirstName, LastName " +
    "FROM Customers " +
    "WHERE NOT FirstName = 'Mary' " );
  DataRequest = Database.createStatement();
  Results = DataRequest.executeQuery (query);
  DisplayResults (Results);
  DataRequest.close();
}
```

Joining Multiple Compound Expressions

The AND and OR clauses are used to link together two or more subexpressions as discussed previously in this chapter. This results in a compound expression. There can be multiple compound expressions within a WHERE clause expression.

Let's say that you want information about Bob Smith, but you are unsure whether he is in Department 42 or Department 45. You can format a query that uses both the AND clause and the OR clause to retrieve this information. Here's how the WHERE clause might be formatted in this scenario.

Caution *A word of caution: The AND clause has a higher precedence than the OR clause, so the OR clause must be placed within parentheses in this example.*

```
WHERE FirstName = 'Bob' AND LastName = 'Smith AND (Dept = '42' OR Dept = '45')
```

Although the WHERE clause may appear confusing, you can simplify the WHERE clause expression by identifying subexpressions and compound expressions. From reading the previous section, you probably identified the first compound expression and its subexpressions, which is

```
FirstName = 'Bob' AND LastName = 'Smith'
```

The second compound expression is

```
Dept = '42' OR Dept = '45'
```

The second AND clause in the WHERE clause links together these two compound expressions. You can use the AND, OR, and the NOT clauses to create complex selection expressions. The number of compound expressions that you can use in the WHERE clause expressions is practically endless, although some DBMSs establish a limit. This is because of processing time that is necessary to evaluate a complex WHERE clause expression. Contact your database administrator or the manufacturer of your DBMS to determine the limitations of the DBMS that you are using for your system.

Equal and Not Equal Operators

The equal and not equal operators are used to determine if the value in the WHERE clause expression is or isn't in the specify column. You've been using the equal operator throughout examples in this chapter. The next section illustrates the use of the not equal operator.

Before running the query in the next section, modify the Customers table by inserting a Sales column into the table. The Sales column is a long data type. After making the modification, place values in the Sales column of the Customers table as shown in Table 7-2.

FirstName	LastName	Street	City	State	ZipCode	Sales
Bob	Jones	5 First Street	New York City	null	07555	50000
Mary	Smith	8 Third Street	Dallas	TX	75553	20000
Bob	Jones	5 First Street	New York City	null	07555	50000
Mark	Russell	3 Sixth Street	Los Angeles	null	82272	30000
Susan	Allen	18 Fifth Street	Chicago	IL	45003	40000
Mark	Russell	3 Sixth Street	Los Angeles	CA	82272	30000

Table 7-2. *Insert These Values into the Customers Table*

Not Equal Operator

The not equal operator is used in a WHERE clause expression or subexpression to identify rows that should not be returned by the DBMS. Listing 7-27 illustrates the use of the not equal operator.

In this example, all columns of rows in the Customers table are returned where the values in the Sales column are not equal to 50000. Listing 7-28 contains the DownRow() method that is used to copy and display values in the ResultSet.

Listing 7-27 Excluding rows from the ResultSet using the not equal operator.

```
try {
  String query = new String ("SELECT  " +
    "FirstName, LastName, Street, City, State, ZipCode, Sales " +
    "FROM Customers " +
    "WHERE NOT Sales = 50000 " );
  DataRequest = Database.createStatement();
  Results = DataRequest.executeQuery (query);
  DisplayResults (Results);
  DataRequest.close();
  Database.close();
}
```

Listing 7-28 Copying values from the ResultSet.

```
private  void DownRow ( ResultSet DisplayResults )throws SQLException
{
  String FirstName= new String();
  String LastName= new String();
  String Street= new String();
  String City = new String();
```

```
    String State = new String();
    String ZipCode= new String();
    long Sales;
    String printrow;
    FirstName = DisplayResults.getString ( 1 ) ;
    LastName = DisplayResults.getString ( 2 ) ;
    Street = DisplayResults.getString ( 3 ) ;
    City = DisplayResults.getString ( 4 ) ;
    State = DisplayResults.getString ( 5 ) ;
    ZipCode = DisplayResults.getString ( 6 ) ;
    Sales = DisplayResults.getLong ( 7 ) ;
    printrow = FirstName + " " + LastName + " " +
      City + " " + State +  " " + ZipCode + " " + Sales;
    System.out.println(printrow);
}
```

Less Than and Greater Than Operators

The less than and greater than operators direct the DBMS to assess whether or not the value in the specified column of the current row is less than or greater than the value in the WHERE clause expression.

Keep in mind that a value in a column that equals the value in the WHERE clause expression is evaluated as a boolean false. Therefore the row containing that value isn't returned in the ResultSet. The value in the column must be less than or greater than, but not equal to, the value in the WHERE clause expression.

Less Than Operator

Listing 7-29 illustrates the user of the less than operator in the WHERE clause. In this example, all the columns of rows from the Customers table are returned as long as the value of the Sales column is less than 50000.

Listing 7-28 contains the DownRow() method that should be used to copy and display the ResultSet returned by Listing 7-29.

Listing 7-29 Using the less than operator in the WHERE clause expression.

```
try {
  String query = new String ("SELECT  " +
    "FirstName, LastName, Street, City, State, ZipCode, Sales " +
    " FROM Customers " +
    " WHERE Sales < 50000 " );
  DataRequest = Database.createStatement();
  Results = DataRequest.executeQuery (query);
  DisplayResults (Results);
  DataRequest.close();
}
```

Greater Than Operator

The greater than operator is used similar to the less than operator except the value in the column must be greater than the value in the WHERE clause expression for the row to be returned in the ResultSet.

Listing 7-30 illustrates the use of the greater than operator. In this example, all the columns of rows in the Customers table are returned in the ResultSet if the value of the Sales column is greater than 40000. Listing 7-28 contains the DownRow() method that should be used to copy and display the ResultSet returned by Listing 7-30.

Listing 7-30 . Using the greater than operator in the WHERE clause expression.

```
try {
   String query = new String ("SELECT   " +
      "FirstName, LastName, Street, City, State, ZipCode, Sales " +
      "FROM Customers " +
      "WHERE Sales > 40000 " );
   DataRequest = Database.createStatement();
   Results = DataRequest.executeQuery (query);
   DisplayResults (Results);
   DataRequest.close();
}
```


Less Than Equal To and Greater Than Equal To

A drawback of using the less than and greater than operators is that rows containing the value of the WHERE clause expression are not returned in the ResultSet. Alternatively, the less than equal to or the greater than equal to operators include rows that contain the WHERE clause expression. This is illustrated in the next two sections.

Less Than Equal To

Listing 7-31 illustrates how to use the less than operator in the WHERE clause. In this example, all columns of rows in the Customers table are returned if the value of the Sales column of the row is less than or equal to 50000.

Listing 7-28 contains the DownRow() method that should be used to copy and display the ResultSet returned by Listing 7-31.

Listing 7-31 Using the less than equal to operator in the WHERE clause expression.

```
try {
   String query = new String ("SELECT   " +
      "FirstName, LastName, Street, City, State, ZipCode, Sales " +
      "FROM Customers " +
      "WHERE Sales <= 50000 " );
   DataRequest = Database.createStatement();
   Results = DataRequest.executeQuery (query);
```

```
    DisplayResults (Results);
    DataRequest.close();
}
```

Greater Than Equal To

As illustrated in Listing 7-32, the greater than equal to operator is used similar to the less than equal to operator. In this example, all the columns in rows of the Customers table are returned in the ResultSet if the value of the Sales column in the row is greater than or equal to 50000.

Listing 7-28 contains the DownRow() method that should be used to copy and display the ResultSet returned by Listing 7-32.

Listing 7-32 Using the greater than equal to operator in the WHERE clause expression.

```
try {
  String query = new String ("SELECT   " +
    "FirstName, LastName, Street, City, State, ZipCode, Sales " +
    "FROM Customers " +
    "WHERE Sales >= 50000 ");
    DataRequest = Database.createStatement();
    Results = DataRequest.executeQuery (query);
    DisplayResults (Results);
    DataRequest.close();
}
```

BETWEEN

The BETWEEN operator is used to define a range of values that is to be used as the value of the selection expression. The range must consist of a sequential series of values such as from 100 to 200. The BETWEEN operator must follow the name of the column in the WHERE clause. The AND operator is used to join the lower and upper values of the range. All values in the range, including the first and last values, are considered when the DBMS evaluates the value of the column specified in the selection expression.

Listing 7-33 illustrates the use of the BETWEEN operator. In this example, all the columns in rows of the Customers table are returned in the ResultSet if the value of the Sales column in the row is between 20000 and 39999.

Listing 7-28 contains the DownRow() method that should be used to copy and display the ResultSet returned by Listing 7-33.

Listing 7-33 Using the BETWEEN operator in the WHERE clause expression.

```
try {
  String query = new String("SELECT   " +
    "FirstName, LastName, Street, City, State, ZipCode, Sales " +
```

```
      "FROM Customers " +
      "WHERE Sales BETWEEN 20000 AND 39999 " );
    DataRequest = Database.createStatement();
    Results = DataRequest.executeQuery (query);
    DisplayResults (Results);
    DataRequest.close();
}
```

LIKE

The LIKE operator directs the DBMS to return a row in the ResultSet if a value in a specified column partially matches the value of the WHERE clause expression. The WHERE clause expression must include a character that must be an exact match and a wildcard character that is used to match any other character.

Caution *A word of caution: The LIKE operator is very convenient and powerful to use; however, it also uses a lot of database overhead and therefore should be used with discretion.*

Here are the wildcards that are used with the LIKE operator:

- **Underscore (_)** A single character wildcard character. For example, if you are unsure if the customer's last name is Anderson or Andersen, you use the underscore in place of the character that is in question such as Anders_n.
- **Percent (%)** A multicharacter wildcard character used to match any number of characters. For example, Smi% is used to match a value of a column where the first three characters are Smi and any other character that follows Smi.

Listing 7-36 illustrates the use of the LIKE operator. In this example, all the columns in rows of the Customers table are returned in the ResultSet if the value of the LastName column in the row has the first three characters Smi and any other characters.

Listing 7-28 contains the DownRow() method that should be used to copy and display the ResultSet returned by Listing 7-34.

Listing 7-34 Using the LIKE operator in the WHERE clause expression

```
try {
    String query = new String ("SELECT    " +
      "FirstName, LastName, Street, City, State, ZipCode, Sales " +
      "FROM Customers " +
      "WHERE LastName LIKE 'Smi%' ");
    DataRequest = Database.createStatement();
    Results = DataRequest.executeQuery (query);
    DisplayResults (Results);
    DataRequest.close();
}
```

IS NULL Operator

The IS NULL operator is used to determine if a specified column does not contain any value. You'll find it useful to use the IS NULL operator whenever you need to identify rows that are missing information in a column.

It is important to understand that the number zero or a space are not NULL values. For example, a zero value in the sales column of a table is a real value. Likewise, a column that contains a space isn't NULL, because a space is a valid ASCII character. NULL is void of any value and occurs when a row is inserted into a table without having a value or the value is explicitly set to NULL.

Listing 7-35 illustrates the use of the IS NULL operator. In this example, all the columns in rows of the Customers table are returned in the ResultSet if the value of the State column in the row is NULL.

Listing 7-28 contains the DownRow() method that should be used to copy and display the ResultSet returned by Listing 7-35.

Listing 7-35 Using the IS NULL operator in the WHERE clause expression.

```
try {
  String query = new String ("SELECT  " +
    "FirstName, LastName, Street, City, State, ZipCode, Sales " +
    "FROM Customers " +
    "WHERE State IS NULL ");
  DataRequest = Database.createStatement();
  Results = DataRequest.executeQuery (query);
  DisplayResults (Results);
  DataRequest.close();
}
```

DISTINCT Modifier

The SELECT statement returns all rows in a table unless a WHERE clause is used to exclude specific rows. However, the ResultSet includes duplicate rows unless a primary index is created for the table or only unique rows are required in the table, as you learned previously in this chapter.

There will be occasions when you want to exclude all but one copy of a row from the ResultSet. You can do this by using the DISTINCT modifier in the SELECT statement, as illustrated in Listing 7-36. The DISTINCT modifier tells the DBMS not to include duplicate rows in the ResultSet.

You'll find this useful whenever data from multiple sources are combined into one table, as is the case of a mailing list. The DISTINCT modifier filters repeating rows that contain the same name and address.

Caution *A word of caution: This technique requires a lot of system overhead and therefore should be used with discretion. An alternative approach to using this is to write a WHERE clause expression that returns the same results as using the DISTINCT modifier.*

In this example, all the columns in rows of the Customers table are returned in the ResultSet exclusive of duplicate rows. Listing 7-28 contains the DownRow() method that should be used to copy and display the ResultSet returned by Listing 7-36.

Listing 7-36 Using the DISTINCT modifier.

```
try {
   String query = new String ("SELECT DISTINCT  " +
     "FirstName, LastName, Street, City, State, ZipCode, Sales " +
     "FROM Customers ");
   DataRequest = Database.createStatement();
   Results = DataRequest.executeQuery (query);
   DisplayResults (Results);
   DataRequest.close();
}
```

IN Modifier

The IN modifier is used to define a set of values used by the DBMS to match values in a specified column. The set can include any number of values and appear in any order.

Let's say that you wanted a list of customers who purchase three products. You'd place the three product identification numbers in a set, then tell the DBMS to search the order table for orders that contain product numbers that are in the set. If the product numbers match, then customer information associated with the row is returned to your program.

The IN modifier is used in the WHERE clause to define the list of values in the set. This is demonstrated in Listing 7-37 where the DBMS is told to compare values in the Sales column with the three values in the group clause. If any of these three values are found in the Sales column of the current row, the row is returned to the program. Listing 7-28 contains the DownRow() method that should be used to copy and display the ResultSet returned by Listing 7-37.

Listing 7-37 Using the IN modifier in the WHERE clause.

```
try {
   String query = new String ("SELECT  " +
     "FirstName, LastName, Street, City, State, ZipCode, Sales " +
     " FROM Customers " +
     " WHERE Sales IN (20000, 30000, 40000) " );
   DataRequest = Database.createStatement();
   Results = DataRequest.executeQuery (query);
   DisplayResults (Results);
   DataRequest.close();
}
```

NOT IN Modifier

The NOT IN modifier is similar to the IN modifier explained in the previous section except the NOT IN modifier reverses the logic of the previous selection expression. That is, the NOT IN modifier identifies a set of values that shouldn't match rows returned to the program.

Listing 7-38 shows how the NOT IN modifier is used in a program. In this example, all columns of rows in the Customers table where the value of the Sales column isn't 20000, 30000, or 40000 are returned in the ResultSet. Rows with sales of 20000, 30000, or 40000 are not returned. Listing 7-28 contains the DownRow() method that should be used to copy and display the ResultSet returned by Listing 7-38.

Listing 7-38 Using the NOT IN modifier in the WHERE clause.

```
try {
  String query = new String ("SELECT " +
    "FirstName, LastName, Street, City, State, ZipCode, Sales " +
    " FROM Customers " +
    " WHERE Sales NOT IN (20000, 30000, 40000) " );
  DataRequest = Database.createStatement();
  Results = DataRequest.executeQuery (query);
  DisplayResults (Results);
  DataRequest.close();
}
```

Metadata

Metadata is data that describes data, which you learned about in Chapter 6. Metadata is returned with the ResultSet object and can be extracted from the ResultSet object by creating a ResultSetMetaData using techniques that are described in the next several sections of this chapter.

Metadata can be used in a J2EE component for various purposes such as to display the column name of a column and determine the data type of a column among other metadata. The most commonly used metadata are

- Column name
- Column number
- Column data type
- Column width

Number of Columns in ResultSet

Listing 7-39 contains the DownRow() method that illustrates how to determine the number of columns that are in the ResultSet. The getMetaData() method is called to extract metadata from the ResultSet. The getMetaData() method returns a ResultSetMetaData object, which is called metadata in this example.

The ResultSetMetaData object contains several methods that are used to copy specific metadata from the ResultSet. In this example, the getColumnCount() method is called to retrieve the number of columns contained in the ResultSet, which is then assigned to the NumberOfColumns variable and printed on the screen using the println() method. This doesn't work with all drivers.

The DownRow() method in Listing 7-39 can replace the DownRow() method in the Model B program and used in any version of the Model B program that is presented in this chapter. Of course, the DownRow() method must be modified to copy and display data contained in the ResultSet, as shown previously in this chapter.

Listing 7-39 Determining the number of columns in a ResultSet.

```
private  void DownRow ( ResultSet DisplayResults )throws SQLException
{
  ResultSetMetaData metadata = DisplayResults.getMetaData ();
  int NumberOfColumns;
  String printrow;
  NumberOfColumns = metadata.getColumnCount ();
  System.out.println("Number Of Columns: " + NumberOfColumns);
}
```

Data Type of a Column

Listing 7-40 illustrates how to determine the data type of a column. You'll notice that this technique is very similar to the technique described in the previous section that retrieved the number of columns in the ResultSet.

In this example, the getColumnTypeName() method is called to copy the data type from column nine of the ResultSet to the ColumnType object. Any valid column number in the ResultSet can be passed to the getColumnTypeName() method. You can use the data type returned by the getColumnTypeName() method to determine the proper getxxx() method to use to copy data from the ResultSet. The type is returned as the native type, not the SQL type.

Listing 7-40 Determining the date type of columns in a ResultSet.

```
private  void DownRow ( ResultSet DisplayResults ) throws SQLException
{
  ResultSetMetaData metadata = DisplayResults.getMetaData ();
  String ColumnType = new String();
  String printrow;
  ColumnType = metadata.getColumnTypeName ( 9 );
  System.out.println("Column Type: " + ColumnType );
}
```

Name of a Column

Retrieving the column name from the metadata uses a process similar to that used to copy the data type of a column from the metadata and is illustrated in Listing 7-41. In this example, the getColumnLabel() method is used to copy the column name from

a specific column and assign the column name to the String object ColumnName. The number of the column in the ResultSet is passed to the getColumnLabel() method.

Listing 7-41 Determining the names of columns in a ResultSet.

```
private  void DownRow ( ResultSet DisplayResults ) throws SQLException
{
  ResultSetMetaData metadata = DisplayResults.getMetaData ();
  String ColumnName = new String();
  String printrow;
  ColumnName = metadata.getColumnLabel (9) ;
  System.out.println("Column Name: " + ColumnName);
}
```

Column Size

The column size of a column, also referred to as the column width, is called the display size and represents the number of characters that are necessary to display the maximum value that might be stored in the column.

You retrieve the display size by using the getColumnDisplaySize() method as is illustrated in Listing 7-42. This listing is similar to the other listings discussed in metadata sections of this chapter.

The getColumnDisplaySize( ) is called and passed the number of the column whose display size is to be retrieved from the ResultSet. The display size is then shown on the screen.

Listing 7-42 Determining the width of columns in a ResultSet.

```
private  void DownRow ( ResultSet DisplayResults ) throws SQLException
{
  ResultSetMetaData metadata = DisplayResults.getMetaData ();
  int ColumnWidth;
  String printrow;
  ColumnWidth = metadata.getColumnDisplaySize ( 9 ) ;
  System.out.println("Column Width:" + ColumnWidth);
}
```

Updating Tables

Modifying data in a database is one of the most common functionalities included in every J2EE component that provides database interactions. Generally, any information that is retrievable is also changeable depending on access rights and data integrity issues, which are discussed in Chapter 5.

The next several sections illustrate techniques that are used to update rows in a table of a database. Code segments described in these sections use the executeUpdate()

method to process queries. As you recall from Chapter 6, the executeUpdate() method does not return a ResultSet. Therefore, the Model A program described earlier in this chapter should be used with code segments in these sections.

Before executing code segments presented in these sections, you'll need to create a Customers table as defined here, then insert the rows of data that are contained in Table 7-3.

- FirstName, VARCHAR, 50
- LastName, VARCHAR, 50
- Street, VARCHAR, 50
- City, VARCHAR, 50
- State, VARCHAR, 2
- ZipCode, VARCHAR, 12
- Discount, long

Updating a Row and Column

The UPDATE statement is used to change the value of one or more columns in one or multiple rows of a table. The UPDATE statement must contain the name of the table that is to be updated and a SET clause. The SET clause identifies the name of the column and the new value that will be placed into the column, overriding the current value. The UPDATE statement may have a WHERE clause if a specific number of rows are to be updated. If the WHERE clause is omitted, all rows are updated based on the value of the SET clause.

FirstName	LastName	Street	City	State	Zipcode	Discount
Bob	Jones	5 First Street	New York City	NY	07555	10
Mary	Smith	8 Third Street	Dallas	TX	75553	20
Tom	Jones	5 First Street	New York City	NY	07555	11
Mark	Russell	23 Eighth Street	Los Angeles	CA	82272	16
Susan	Allen	18 Fifth Street	Chicago	IL	45003	15
Kelly	Russell	32 Fourth Street	Los Angeles	CA	82272	15

Table 7-3. *Insert These Values into the Customers Table*

Listing 7-43 illustrates how to update one column. In this example, the value of the Street column of the Customers table is changed to '5 Main Street'. However, the change occurs only where the value of the FirstName column is 'Bob'.

Listing 7-43 Updating a row and column.

```
try {
  String query = new String("UPDATE Customers " +
    "SET Street = '5 Main Street' " +
    " WHERE FirstName = 'Bob'");
  DataRequest = Database.createStatement();
  DataRequest.executeUpdate (query);
  DataRequest.close();
}
```

Updating Multiple Rows

Multiple rows of a table can be updated by formatting the WHERE clause expressions to include criteria that qualify multiple rows for the update. There are four common WHERE clause expressions that are used to update multiple rows of a table. These are

- **The IN test** The WHERE clause expression contains multiple values in the IN clause that must match the value in the specified column for the update to occur in the row.
- **The IS NULL test** Rows that don't have a value in the specified column are updated when the IS NULL operator is used in the WHERE clause expression.
- **The comparison test** The WHERE clause expression contains a comparison operator, as described previously in this chapter, that compares the value in the specified column with a value in the WHERE clause expression.
- **All rows** A query can direct the DBMS to update the specified column in all rows of a table by excluding the WHERE clause in the query.

Always be cautious whenever you execute a query that updates multiple rows, because the query has the potential to affect all the information contained in a table. An error in a query is multiplied by the number of rows in a table whenever the query updates more than one row.

IN Test

The IN clause provides two or more values that are compared to the value of the designed column in the IN clause. Rows whose columns contain one of these values are then updated by the UPDATE statement.

Listing 7-44 illustrates how the IN test is used to update rows of the Customers table. In this example, the value of the Discount column is changed to 25 if the current value of the Discount column is either 12 or 15.

Listing 7-44 Updating a row using the IN test.

```
try {
  String query = new String ("UPDATE Customers " +
    "SET Discount = 25 " +
    "WHERE Discount IN (12,15)");
  DataRequest = Database.createStatement();
  DataRequest.executeUpdate (query);
  DataRequest.close();
}
```

IS NULL Test

The IS NULL test evaluates the value of a column designated in the test to determine if the column is NULL. That is, the column is empty of any value. If so, then the IS NULL test returns a true and the UPDATE statement updates the column specified in the SET clause.

Listing 7-45 illustrates how to use the IS NULL test in the UPDATE statement. In this example, the IS NULL test determines if the LastName column is NULL. If so, then a zero is placed in the Discount column. If not, then the value of the Discount column remains unchanged.

Listing 7-45 Updating a row using the IS NULL test.

```
try {
  String query = new String ("UPDATE Customers " +
    "SET Discount = 0 " +
    "WHERE LastName IS NULL  ");
  DataRequest = Database.createStatement();
  DataRequest.executeUpdate (query);
  DataRequest.close();
}
```

Updating Based on Values in a Column

An expression in the WHERE clause can be used to identify rows that are to be updated by the UPDATE statement. All the WHERE clause expressions discussed previously in the SELECT statement also apply to the UPDATE statement. Review the SELECT statement section for details on how to properly formulate the WHERE clause expression.

Listing 7-46 illustrates how to identify a row to update. In this example, rows in the Customers table that have a Discount greater than 20 will have the value of the Discount changed to 20.

Listing 7-46 Updating a row based on a value in a designated column.

```
try {
  String query = new String ("UPDATE Customers " +
```

```
      "SET Discount = 20 " +
      "WHERE Discount > 20 ");
    DataRequest = Database.createStatement();
    DataRequest.executeUpdate (query);
    DataRequest.close();
}
```

Updating Every Row

All rows in a table can be updated by excluding the WHERE clause in the UPDATE statement. This technique is illustrated in Listing 7-47. In this example, the value of the Discount column in all rows in the Customers table is changed to zero.

Listing 7-47 Updating every row in the table.

```
try {
    String query = new String ("UPDATE Customers " +
      "SET Discount = 0 ");
    DataRequest = Database.createStatement();
    DataRequest.executeUpdate (query);
    DataRequest.close();
}
```

Updating Multiple Columns

Multiple columns of rows can be updated simultaneously by specifying the column names and appropriate values in the SET clause of the query. Listing 7-48 illustrates this technique. In this example, rows in the Customers table where the LastName column has the value 'Jones' are updated. The SET clause contains the column names and the new values that override the current values of those columns. In this case, the value of the Discount column is changed to 12 and the value of the Street column is changed to 'Jones Street'.

Listing 7-48 Updating multiple columns of a table.

```
try {
    String query = new String ("UPDATE Customers " +
      "SET Discount = 12, Street = 'Jones Street'" +
      "WHERE LastName = 'Jones'");
    DataRequest = Database.createStatement();
    DataRequest.executeUpdate (query);
    DataRequest.close();
}
```

Updating Using Calculations

The value that replaces the current value in a column does not have to be explicitly defined in the SET clause if the value can be derived from a value in another column of the same row.

Let's say that a row contains the retail price of an item purchased by a customer. The customer is granted a percentage discount based on the how well the customer is valued by the business. The UPDATE statement can calculate the discounted price and place the discount price in the DiscountPrice column of the row. The value of the discount price doesn't need to be included in the SET clause. Instead, the discount price can be calculated.

Listing 7-49 illustrates how to use the results of a calculation to update a value of a column in a table. In this example, the value that is placed in the DiscountPrice column is calculated by using values in the Price column and in the Discount column. The results of this calculation override the current value in the DiscountPrice column.

Before running this query, you'll need to create a new Customers table or modify the existing Customers table to reflect the following columns. You'll also need to insert the data shown in Table 7-4 into the Customers table.

- FirstName, VARCHAR, 50
- LastName, VARCHAR, 50
- Street, VARCHAR, 50
- City, VARCHAR, 50
- State, VARCHAR, 2
- ZipCode, VARCHAR, 12
- Discount, long
- Price, long
- DiscountPrice, long

Listing 7-49 Updating a column using a calculation.

```
try {
  String query = new String ("UPDATE Customers " +
    "SET DiscountPrice = Price * ((100 - Discount) / 100) ");
  DataRequest = Database.createStatement();
  DataRequest.executeUpdate (query);
  DataRequest.close();
 }
```

FirstName	LastName	Street	City	State	Zipcode	Discount	Price
Bob	Jones	5 First Street	New York City	NY	07555	10	100
Mary	Smith	8 Third Street	Dallas	TX	75553	20	200
Tom	Jones	5 First Street	New York City	NY	07555	20	300
Mark	Russell	23 Eighth Street	Los Angeles	CA	82272	10	400
Susan	Allen	18 Fifth Street	Chicago	IL	45003	15	500
Kelly	Russell	32 Fourth Street	Los Angeles	CA	82272	15	600

Table 7-4. *Insert These Values into the Customers Table*

Deleting Data from a Table

Deleting rows is necessary to purge erroneous information from the database and to remove information that is no longer needed. However, you must build safeguards in to assure that critical information isn't inadvertently deleted from the database. Before you delete a row from a table, you must be certain that other tables are not negatively affected.

The next section of this chapter illustrates the technique for removing a row from a table by using the DELETE FROM statement. Multiple rows can be deleted by including a WHERE clause in the DELETE FROM statement as described in detail in the earlier "Updating Tables" section of this chapter.

The query that contains the DELETE FROM statement is executed using the executeQuery() method. As you learned in Chapter 6, the executeQuery() method doesn't return a ResultSet. Therefore, use the Model A program with the code segment described in the next section.

Deleting a Row from a Table

A row is deleted from a table by using the DELETE FROM statement. The DELETE FROM statement includes the name of the table and a WHERE clause that contains an expression that identifies the row or rows to remove from the table.

Listing 7-50 illustrates how to use the DELETE FROM statement. In this example, rows from the Customers table are being deleted where the value of the LastName column is 'Jones' and the value of the FirstName column is 'Tom'. Make sure you have a row that contains these values in the Customers table before running this code segment.

Listing 7-50 Deleting a row.

```
try {
   String query = new String ("DELETE FROM Customers " +
    "WHERE LastName = 'Jones' and FirstName = 'Tom'");
   DataRequest = Database.createStatement();
   DataRequest.executeUpdate (query);
   DataRequest.close();
}
```


Joining Tables

In Chapter 5, you learned about normalization and how rows of data elements that are placed in tables are related to each other by linking rows using a common value in each row of two tables. Linking rows is called joining tables.

Tables are joined in a query using a two-step process. First, both tables that are being joined must be identified in the FROM clause where tables are listed one after the other and are separated by a comma.

Next, an expression is created in the WHERE clause that identifies the columns that are used to create the join. Let's say that an Orders table is joined to the Customers table using the customer number. The customer number in the Orders table is in the CustomerNumber column and the customer number in the Customers table is the CustNum column. The following line contains a WHERE clause expression that joins these tables:

```
WHERE CustomerNumber = CustNum
```

The joined tables create a logical table that has all the columns of both tables. All the tasks performed on a single table previously in this chapter can also be applied to join tables.

Joining too many tables can cause performance degradation and bring response time to a crawl. Typically, five is the maximum number of tables that are joined. However, the actual number may vary depending on the DBMS, so consult with your database administrator or DBMS manufacturer for the maximum number of tables permitted to be joined using your DBMS.

Before you try examples in this chapter, you'll need to create a Customers table, ZipCode table, Store table, Products table, and Orders table. These tables are defined next. Insert the rows described in Tables 7-5 through 7-9 into the appropriate tables before running codes segments described in the following sections of this chapter.

- **Customers Table**
  - CustNumbér, Number, Primary Key
  - FirstName, VARCHAR, 50
  - LastName, VARCHAR, 50
  - Street, VARCHAR, 50
  - Zip, VARCHAR, 12
- **ZipCode Table**
  - ZipCode, VARCHAR, 12, Primary Key
  - City, VARCHAR, 50
  - State, VARCHAR, 2
- **Store Table**
  - StoreNumber, Number, Primary Key
  - ZipCode, VARCHAR, 12
- **Products Table**
  - ProductNumber, Number, Primary Key
  - ProductName, VARCHAR, 50
  - UnitPrice, Long
- **Orders Table**
  - OrderNumber, Number, Primary Key
  - ProdNumber, Number
  - CustomerNumber, Number
  - StoreNumber, Number
  - Quantity, Number
  - SubTotal, Long

CustNumber	FirstName	LastName	Street	Zip
591	Anne	Smith	65 Cutter Street	04735
721	Bart	Adams	15 W. Spruce	05213
845	Tom	Jones	35 Pine Street	07660
901	Mary	Smith	5 Maple Street	08513

Table 7-5. *Insert These Values into the Customers Table*

ZipCode	City	State
04735	Woodridge	TX
05213	River Ville	CA
07660	West Town	NJ
08513	SunnySide	NY

Table 7-6. *Insert These Values into the ZipCode Table*

StoreNumber	ZipCode
278	08513
345	07660
547	05213
825	04735

Table 7-7. *Insert These Values into the Store Table*

ProductNumber	ProductName	Unit Price
1052	CD Player	100
3255	VCR	250
5237	DVD Player	325
7466	50-inch TV	532

Table 7-8. *Insert These Values into the Products Table*

Joining Two Tables

Listing 7-51 illustrates the technique used to join two tables. In this example, the Customers table and the ZipCode table are identified in the FROM clause. These are the tables that are being joined.

The WHERE clause expression identifies the Zip column from the Customers table and the ZipCode column from the ZipCode table. The equal operator directs the DBMS to join rows in both the Customers table and the ZipCode table where values of the Zip column and the ZipCode column have the same value. Rows whose columns don't match are excluded from the join (see the "Inner and Outer Joins" section later in this chapter). If both columns used for the join have the same name, then a column name qualifier is used to associate the column with each table, which is discussed later in this chapter. It's usually considered good practice to always use a table name or alias.

OrderNumber	ProdNumber	CustomerNumber	StoreNumber	Quantity	SubTotal
122	5237	591	345	1	325
334	3255	901	278	1	250
365	3255	901	278	4	1000
534	7466	591	825	3	1596
587	5237	845	345	1	325
717	1052	721	825	2	200
874	7466	721	825	1	532

Table 7-9. *Insert These Values into the Orders Table*

Once the Customers and ZipCode tables are joined, the SELECT statement references column names of both tables to return in the ResultSet. In this example, City, State, and ZipCode columns are in the ZipCode table and the FirstName and LastName columns are in the Customers table.

The query in Listing 7-51 uses the executeQuery() method to execute the query. Therefore, you'll need to use the Model B program to run this example. Listing 7-52 contains the DownRow() method that extracts and displays data from the ResultSet.

Listing 7-51 Joining two tables.

```
try {
  String query = new String ("SELECT FirstName, LastName, City, State, ZipCode " +
    "FROM Customers, ZipCode " +
    "WHERE Zip = ZipCode");
  DataRequest = Database.createStatement();
  Results = DataRequest.executeQuery (query);
  DisplayResults (Results);
  DataRequest.close();
}
```

Listing 7-52 The DownRow() method used when joining two tables.

```
private  void DownRow ( ResultSet DisplayResults )
    throws SQLException
{
  String FirstName= new String();
  String LastName= new String();
  String Street= new String();
  String City = new String();
  String State = new String();
  String ZipCode= new String();
  String printrow;
  FirstName = DisplayResults.getString ( 1 ) ;
  LastName = DisplayResults.getString ( 2 ) ;
  Street = DisplayResults.getString ( 3 ) ;
  City = DisplayResults.getString ( 4 ) ;
  State = DisplayResults.getString ( 5 ) ;
  ZipCode = DisplayResults.getString ( 6 ) ;
  printrow = FirstName + " " + LastName + " " +
    City + " " + State +  " " + ZipCode + " " + Sales + " " + Profit;
  System.out.println(printrow);
}
```

Parent/Child Join

A parent/child join is used to join tables that have a parent/child relationship where rows in the parent table must exist for rows in the child table to exist, which is illustrated in Listing 7-53.

In this example, two tables are joined together using the product number value. These are the Products table and the Orders table. Only rows that have a matching product number in both tables are placed in the virtual table that consists of rows from both tables.

The Products table is the parent table in this relationship because an order cannot be placed without a product appearing in the Products table. Basically, a customer cannot order a product that doesn't exist in the Products table.

Listing 7-53 is executed using the executeQuery() method and therefore returns a ResultSet. This means that you must use the Model B program to run this code segment. Listing 7-54 contains the DownRow() method that retrieves and displays values from the ResultSet.

Listing 7-53 The parent/child join.

```
try {
  String query = new String (" SELECT OrderNumber, ProductName " +
    " FROM Orders, Products " +
    " WHERE ProdNumber = ProductNumber");
  DataRequest = Database.createStatement();
  Results = DataRequest.executeQuery (query);
  DisplayResults (Results);
  DataRequest.close();
}
```

Listing 7-54 The DownRow() method used with a parent/child join.

```
private  void DownRow ( ResultSet DisplayResults )
    throws SQLException
{
  int OrderNum, ProdNum;
  String printrow = new String();
  OrderNum = DisplayResults.getInt ( 1 ) ;
  ProdNum = DisplayResults. getInt ( 2 ) ;
  printrow = OrderNum + " " + ProdNum;
  System.out.println(printrow);
}
```

Multiple Comparison Join

The WHERE clause expression used to join together tables can be a compound expression. A compounded expression consists of two or more subexpressions, each of which is evaluated separately, as is described previously in this chapter.

A compound expression is used to specify more than one selection criteria used to join two tables, which is illustrated in Listing 7-55. In this example, the DBMS is directed to join together rows of the Products table and rows of the Orders table.

There are two components of the subexpression in this example. The first requires the DBMS to match product numbers in both tables. The other subexpression requires that the value in the Quantity column be greater than 2.

The AND operator is used to join together both subexpressions. This means that both subexpressions must evaluate true for the DBMS to join rows of both tables. That is, only a row where the product number matches in both tables and the quantity of the order is greater than 2 is included in the temporary table. If either subexpression evaluates as false, then the row is not included in the virtual table.

Listing 7-55 is executed using the executeQuery() method and therefore returns a ResultSet. This means that you must use the Model B program to run this code segment. Listing 7-56 contains the DownRow() method that retrieves and displays values from the ResultSet.

Listing 7-55 The multiple comparison join.

```
try {
  String query = " SELECT OrderNumber, ProductName, Quantity " +
    " FROM Orders, Products " +
    " WHERE ProdNumber = ProductNumber " +
    " AND Quantity > 2";
  DataRequest = Database.createStatement();
  Results = DataRequest.executeQuery (query);
  DisplayResults (Results);
  DataRequest.close();
}
```

Listing 7-56 The DownRow() method used with a multiple comparison join.

```
private  void DownRow ( ResultSet DisplayResults )
    throws SQLException
{
  int OrderNum, ProdNum, Quantity;
  String printrow = new String();
  OrderNum = DisplayResults.getInt ( 1 ) ;
  ProdNum = DisplayResults. getInt ( 2 ) ;
  Quantity = DisplayResults. getInt ( 3 ) ;
  printrow = OrderNum + " " + ProdNum + " " + Quantity;
  System.out.println(printrow);
}
```

Multitable Join

More than two tables can be joined together by using the name of each table in the join in the FROM clause and by defining the join with the appropriate column names in the WHERE clause expression.

228

Listing 7-57 shows you how to create a multitable join. There are three tables joined in this example. These are the Customers table, the Orders table, and the Products table, as shown in the WHERE clause.

Notice that all three tables don't need to have a common value used to join them. Each pair of tables has a value common to both of them. Customer numbers that are common between the tables join the Customers table and the Orders table. The Orders table and the Products table are joined by the product number.

You'll need to use the Model B program with this code segment because a ResultSet is returned. Listing 7-58 contains the DownRow() method to use to retrieve and display values in the ResultSet.

g 7-57
ultiple
e join.

```
try {
  String query = new String (" SELECT FirstName,
LastName,OrderNumber, ProductName, Quantity " +
    " FROM Customers, Orders, Products " +
    " WHERE ProdNumber = ProductNumber " +
    " AND CustNumber = CustomerNumber");
  DataRequest = Database.createStatement();
  Results = DataRequest.executeQuery (query);
  DisplayResults (Results);
  DataRequest.close();
}
```

g 7-58
The
nRow()
nethod
with a
nultiple
le join.

```
private  void DownRow ( ResultSet DisplayResults )
    throws SQLException
{
  int OrderNum, Quantity;
  String FirstName = new String();
  String LastName = new String();
  String ProductName = new String();
  String printrow = new String();
  FirstName = DisplayResults.getString ( 1 ) ;
  LastName = DisplayResults.getString ( 2 ) ;
  OrderNum = DisplayResults.getInt ( 3 ) ;
  ProductName = DisplayResults.getString ( 4 ) ;
  Quantity = DisplayResults. getInt ( 5 ) ;
  printrow = FirstName + " " + LastName + " " + OrderNum + " " + ProductName + " "
+ Quantity ;
  System.out.println(printrow);
}
```

Creating a Column Name Qualifier

Column names should reflect the kind of data element stored in the column, as you learned in Chapter 5. However, there is likely to be a conflict when two or more tables

use the same column name to store the data element. This is the case where both the Products table and the Orders table contain product numbers.

Conflicts with column names can be resolved in a join by using a column name qualifier. A column name qualifier identifies the table that contains the column name. Listing 7-59 illustrates how to use a column name qualifier. In this example, assume that the name of the Customers table and the name of the Orders table both have a column called CustomerName. Therefore, the name of the CustomerName column must be prefaced with the name of the table.

Listing 7-59 returns a ResultSet. The DownRow() method shown in Listing 7-60 is used along with the Model B program to copy and display values from the ResultSet.

Listing 7-59 Creating a column name qualifier.

```
try {
  String query = new String ("SELECT Customers.CustNumber, " +
    " FirstName, LastName, OrderNumber, " +
    " ProductName, Quantity " +
    " FROM Customers, Orders, Products " +
    " WHERE ProdNumber = ProductNumber " +
    " AND Customers.CustomerNumber = Orders.CustomerNumber");
  DataRequest = Database.createStatement();
  Results = DataRequest.executeQuery (query);
  DisplayResults (Results);
  DataRequest.close();
}
```

Listing 7-60 The DownRow() method used with a column name qualifier.

```
private  void DownRow ( ResultSet DisplayResults )
    throws SQLException
{
  int CustomerNumber, OrderNum, Quantity;
  String FirstName = new String();
  String LastName = new String();
  String ProductName = new String();
  String printrow = new String();
  CustomerNumber = DisplayResults. getInt ( 1 ) ;
  FirstName = DisplayResults.getString ( 2 ) ;
  LastName = DisplayResults.getString ( 3 ) ;
  OrderNum = DisplayResults.getInt ( 4 ) ;
  ProductName = DisplayResults.getString ( 5 ) ;
  Quantity = DisplayResults. getInt ( 6 ) ;
  printrow = CustomerNumber + " " + FirstName + " " + LastName + " " + OrderNum +
" " + ProductName + " " + Quantity ;
  System.out.println(printrow);
}
```

Creating a Table Alias

A query can be made readable by using table aliases. A table alias is an abbreviation for the name of the table, and is used in place of the table name in the join and in the SELECT statement. Listing 7-61 shows you how to create and use a table alias in a query.

Following each table name in the FROM clause is a letter, used as the table alias for the table. The table alias is used in place of the table name in the column name qualifier.

Any letter or combination of letters can be used as a table alias; however, you should make the table alias

- With as few letters as possible to save space in the query string
- Representative of the table name, such as the first letter(s) of the table name
- Unique, and not a duplicate of another table name, table alias, or column name

The DownRow() method shown in Listing 7-60 is used with Listing 7-61 to extract and display values of the ResultSet.

Listing 7-61 Creating a table alias.

```
try {
  String query = new String ("SELECT c.CustNumber , " +
    " c.FirstName, c.LastName, o.OrderNumber, " +
    " p.ProductName, o.Quantity " +
    " FROM Customers c, Orders o, Products  p" +
    " WHERE o.ProdNumber = p.ProductNumber " +
    " AND c.CustomerNumber = o.CustomerNumber");
  DataRequest = Database.createStatement();
  Results = DataRequest.executeQuery (query);
  DisplayResults (Results);
  DataRequest.close();
}
```

Inner and Outer Joins

The joins shown in previous sections of this chapter link together rows that have matching values in the column specified in the join. A row in a table that doesn't match is excluded from the join.

There are two kinds of joins, each of which either excludes or includes rows in both tables of the join that don't match. These are

- **Inner join** An inner join excludes rows of either table that don't have a matching value.
- **Outer join** An outer join includes rows of either table that don't have a matching value.

The following sections in this chapter discuss each type of join in detail. However, some DBMSs may not adhere to the SQL syntax for some or all joins, and use their own syntax. Therefore, consult with your database administrator or DBMS manufacturer for the proper syntax for joins if examples in this book don't work with your DBMS. Code segments used in these sections are executed using the executeQuery() method. Therefore, you'll need to use the Model B program. Remember that joins can also be used with the UPDATE and DELETE statements, in which case the Model A program is used.

Inner Join

Joins discussed previously in this chapter use an inner join to include only rows of both tables that have matching values. Unmatched rows are excluded, and therefore those rows are not returned in the ResultSet.

Listing 7-62 demonstrates the use of an inner join by joining the Orders table and the Products table using product number and joining the Orders table and the Customers table using the customer number. Listing 7-63 contains the DownRow() method used with Listing 7-62 to copy and display the contents of the ResultSet.

Listing 7-62 Creating an inner join.

```
try {
  String query = new String (" SELECT FirstName, LastName, OrderNumber,
ProductName, Quantity " +
    " FROM Customers,Orders, Products " +
    " WHERE ProdNumber = ProductNumber " +
    " AND Customers.CustNumber = Orders.CustomerNumber");
  DataRequest = Database.createStatement();
  Results = DataRequest.executeQuery (query);
  DisplayResults (Results);
  DataRequest.close();
}
```

Listing 7-63 The DownRow() method used with an inner join.

```
private  void DownRow ( ResultSet DisplayResults )
    throws SQLException
{
  int OrderNum, Quantity;
  String FirstName = new String();
  String LastName = new String();
  String ProductName = new String();
  String printrow = new String();
  FirstName = DisplayResults.getString ( 1 ) ;
  LastName = DisplayResults.getString ( 2 ) ;
  OrderNum = DisplayResults.getInt ( 3 ) ;
  ProductName = DisplayResults.getString ( 4 ) ;
  Quantity = DisplayResults. getInt ( 5 ) ;
  printrow = FirstName + " " + LastName + " " + OrderNum + " " + ProductName + " "
+ Quantity ;
  System.out.println(printrow);
}
```

Outer Join—Left, Right, Full

An outer join occurs when matching and no matching rows of either or both tables are contained in the join. There are three kinds of outer joins:

- **Left outer join** All matched and unmatched rows of the first table and matched rows of the second table are included in the join.
- **Right outer join** Matched rows of the first table and matched and unmatched rows of the second table are included in the join.
- **Full outer join** Matched and unmatched rows of both tables are included in the join.

Left Outer Join Listing 7-64 shows you how to create a left outer join. This example creates a join between the Customers table and the Orders table using the customer number. The *= operator is used in the WHERE clause to create a left outer join. Think of the asterisk as a wildcard that tells the DBMS to use any row in the first table.

Before running Listing 7-64, you need to insert the row shown in Table 7-10 into the Orders table; otherwise, there will not be unmatched rows because all the rows in both tables currently have matching values. Listing 7-65 contains the DownRow() method that is used with Listing 7-64.

Listing 7-64 Creating a left outer join.

```
try {
  String query = new String (" SELECT FirstName, LastName,OrderNumber " +
    "  FROM Customers LEFT JOIN Orders " +
    "  ON Customers.CustNumber = Orders.CustomerNumber");
  DataRequest = Database.createStatement();
  Results = DataRequest.executeQuery (query);
  DisplayResults (Results);
  DataRequest.close();
}
```

Listing 7-65 The DownRow() method used with an inner join.

```
private  void DownRow ( ResultSet DisplayResults )
    throws SQLException
{
  int, OrderNum;
  String FirstName = new String();
  String LastName = new String();
  String printrow = new String();
  FirstName = DisplayResults.getString ( 1 ) ;
  LastName = DisplayResults.getString ( 2 ) ;
  OrderNum = DisplayResults.getInt ( 3 ) ;
  printrow = FirstName + " " + LastName + " " + OrderNum;
  System.out.println(printrow);
}
```

OrderNumber	ProdNumber	CustomerNumber	StoreNumber	Quantity	SubTotal
733	7466	999	825	1	532

Table 7-10. *Insert These Values into the Orders Table*

Right Outer Join Listing 7-66 illustrates how to create a right outer join. In this example, the Customers table and the Orders table are joined using the customer number. All rows in the Orders table are used in the join regardless if the customer number in the Orders table has a corresponding customer number in the Customers table.

The WHERE clause contains the right outer join operator (=*), which is similar to the left outer join operator except the asterisk is positioned to the right of the equivalent operator.

Before running Listing 7-66 you need to insert the row shown in Table 7-11 into the Orders table; otherwise, there will not be unmatched rows because all the rows in both tables currently have matching values. Listing 7-65 contains the DownRow() method that is used with Listing 7-66.

Listing 7-66 Creating a right outer join.

```
try {
  String query = new String (" SELECT FirstName, LastName,OrderNumber" +
    " FROM Customers c, Orders o" +
    " WHERE c.CustNumber =* o.CustomerNumber");
  DataRequest = Database.createStatement();
  Results = DataRequest.executeQuery (query);
  DisplayResults (Results);
  DataRequest.close();
}
```

Full Outer Join A full outer join uses all the rows of both tables regardless if they match. Listing 7-67 shows how to create a full outer join. The full outer join operator (*=*) is a combination of the left outer join operator and the right outer join operator, although

OrderNumber	ProdNumber	CustomerNumber	StoreNumber	Quantity	SubTotal
555	1052	999	278	14	1400

Table 7-11. *Insert These Values into the Orders Table*

some DBMSs use FULL JOIN or other syntax to create a full join. Consult with your database administrator or the DBMS manufacturer for the syntax used by your DBMS. Listing 7-65 contains the DownRow() method that is used with Listing 7-67.

Listing 7-67 Creating a full outer join.

```
try {
  String query = new String("SELECT FirstName, LastName,OrderNumber " +
    " FROM Customers c, Orders o" +
    " WHERE c.CustNumber *=* o.CustomerNumber");
  DataRequest = Database.createStatement();
  Results = DataRequest.executeQuery (query);
  DisplayResults (Results);
  DataRequest.close();
}
```

Calculating Data

The DBMS can calculate values in a table and return the result of the calculation in the ResultSet by using one of the five built-in calculation functions. There are five kinds of built-in calculation functions that are discussed in the following sections. These are

- **SUM()** Tallies values in a column that is passed to the built-in function.
- **AVG()** Averages values in a column that is passed to the built-in function.
- **MIN()** Determines the minimum value in a column that is passed to the built-in function.
- **MAX()** Determines the maximum value in a column that is passed to the built-in function.
- **COUNT()** Determines the number of rows in a column that is passed to the built-in function.

Before you try examples in this chapter, you'll need to create an Orders table, Sales table, and Customers table. These tables are defined next. Insert the rows described in Tables 7-12 through 7-14 into the appropriate tables before running the code segments described in the following sections of this chapter.

- Orders Table
  - OrderNumber, Number, Primary Key
  - ProdNumber, Number
  - CustomerNumber, Number
  - StoreNumber, Number
  - Quantity, Number
  - SubTotal, Long

OrderNumber	ProdNumber	CustomerNumber	StoreNumber	Quantity	SubTotal
122	5237	591	345	1	325
334	3255	901	278	1	250
365	3255	901	278	4	1000
534	7466	591	825	3	1596
587	5237	845	345	1	325
717	1052	721	825	2	200
874	7466	721	825	1	532

Table 7-12. *Insert These Values into the Orders Table*

- **Sales Table**
  - StoreNumber, Number, Primary Key
  - Sales, Number
  - Estimate, Number
- **Customers Table**
  - CustNumber, Number, Primary Key
  - FirstName, VARCHAR
  - LastName, VARCHAR
  - Street, VARCHAR
  - Zip, VARCHAR

StoreNumber	Sales	Estimate
6245	500	450
8644	650	700

Table 7-13. *Insert These Values into the Sales Table*

CustNumber	FirstName	LastName	Street	Zip
591	Anne´	Smith	65 Cutter Street	04735
721	Bart	Adams	15 W. Spruce	05213
845	Tom	Jones	35 Pine Street	07660
901	Mary	Smith	5 Maple Street	08513

Table 7-14. *Insert These Values into the Customers Table*

Sum()

The SUM() built-in function calculates the sum of the values in the column that is passed to the function. Listing 7-68 illustrates how to use the SUM() built-in function. In this example, values in the Quantity column in the Orders table are passed to the SUM() built-in function.

Listing 7-68 uses the executeQuery() method to execute the query. Therefore, the Model B program should be used with this listing. Listing 7-69 contains the DownRow() method that is used with Listing 7-68.

Listing 7-68 Determining the sum of values in a column.

```
try {
  String query = new String ("SELECT SUM(Quantity) " +
    "FROM Orders ");
  DataRequest = Database.createStatement();
  Results = DataRequest.executeQuery (query);
  DisplayResults (Results);
  DataRequest.close();
}
```

Listing 7-69 The DownRow() method used with the SUM() built-in function.

```
private  void DownRow ( ResultSet DisplayResults )
    throws SQLException
{
  long ReturnValue;
  ReturnValue = DisplayResults.getLong ( 1 ) ;
  System.out.println(ReturnValue);
}
```

AVG()

The AVG() built-in function calculates the average value in the column that is passed to the function. The average is returned in the ResultSet. Listing 7-70 illustrates how to use the AVG() built-in function to calculate the average of the values in the Quantity

column of the Orders table. This code segment is used with the Model B program and Listing 7-69 contains the DownRow() method to use with Listing 7-70 to retrieve and display the average from the ResultSet.

Listing 7-70 Determining the average of values in a column.

```
try {
   String query = new String ("SELECT AVG(Quantity) " +
     "FROM Orders ");
   DataRequest = Database.createStatement();
   Results = DataRequest.executeQuery (query);
   DisplayResults (Results);
   DataRequest.close();
}
```


MIN()

The MIN() built-in function returns the lowest value contained in the column passed to the MIN() built-in function. Listing 7-71 shows how to use the MIN() built-in function to determine the minimum value in the Quantity column of the Orders table. This code segment is used with the Model B program and Listing 7-69 contains the DownRow() method to use with Listing 7-71 to retrieve and display the minimum value from the ResultSet.

Listing 7-71 Determining the minimum value in a column.

```
try {
   String query = new String ("SELECT MIN(Quantity) " +
     "FROM Orders ");
   DataRequest = Database.createStatement();
   Results = DataRequest.executeQuery (query);
   DisplayResults (Results);
   DataRequest.close();
}
```

MAX()

The MAX() built-in function returns the highest value contained in the column passed to the MAX() built-in function. Listing 7-72 shows how to use the MAX() built-in function to determine the maximum value in the Quantity column of the Orders table. This code segment is used with the Model B program and Listing 7-69 contains the DownRow() method to use with Listing 7-72 to retrieve and display the value from the ResultSet.

Listing 7-72 Determining the maximum value in a column.

```
try {
   String query = new String ("SELECT MAX(Quantity) " +
     "FROM Orders ");
   DataRequest = Database.createStatement();
```

```
    Results = DataRequest.executeQuery (query);
    DisplayResults (Results);
    DataRequest.close();
}
```

COUNT()

Counting the number of rows or values in a table is a very common calculation. The COUNT() built-in function returns the number of rows in a column. Listing 7-73 uses the COUNT() built-in function to determine the number of values that appear in the Quantity column of the Orders table. The result is returned in the ResultSet.

Rows without values in the column are excluded from the count. This means the value returned does not necessarily represent the total number of rows in the table. The return value represents the total number of rows in the table containing values for every row.

This code segment is used with the Model B program and Listing 7-69 contains the DownRow() method to use with Listing 7-73 to retrieve and display the value from the ResultSet.

Listing 7-73 Counting the number of values in a column.

```
try {
    String query = new String ("SELECT COUNT(Quantity) " +
      "FROM Orders ");
    DataRequest = Database.createStatement();
    Results = DataRequest.executeQuery (query);
    DisplayResults (Results);
    DataRequest.close();
}
```

Counting All Rows in a Table

The COUNT() built-in function as described in the previous section is also used to return the number of rows in a table. This is accomplished by passing the COUNT() built-in function an asterisk rather than the name of a column. Listing 7-74 illustrates this technique.

This code segment is used with the Model B program and Listing 7-69 contains the DownRow() method to use with Listing 7-74 to retrieve and display the value from the ResultSet.

Listing 7-74 Counting the number of rows.

```
try {
    String query = new String ("SELECT COUNT(*) " +
      "FROM Orders ");
    DataRequest = Database.createStatement();
```

```
    Results = DataRequest.executeQuery (query);
    DisplayResults (Results);
    DataRequest.close();
}
```

Retrieving Multiple Counts

Multiple counts can be returned in the ResultSet by using more than one COUNT() built-in function in the SELECT statement. This is illustrated in Listing 7-75 where the ResultSet contains the total number of rows in the Orders table and the number of rows from the Orders table that contain values in the Quantity column.

This code segment is used with the Model B program and Listing 7-76 contains the DownRow() method to use with Listing 7-75 to retrieve and display values from the ResultSet.

Listing 7-75 Retrieving multiple counts.

```
try {
    String query = new String ("SELECT COUNT(*), COUNT(Quantity) " +
      "FROM Orders ");
    DataRequest = Database.createStatement();
    Results = DataRequest.executeQuery (query);
    DisplayResults (Results);
    DataRequest.close();
}
```

Listing 7-76 The DownRow() method used with multiple calls to the COUNT() built-in function.

```
private  void DownRow ( ResultSet DisplayResults )
     throws SQLException
{   long TotalRows, TotalValues;
    TotalRows = DisplayResults.getLong ( 1 ) ;
   TotalValues = DisplayResults.getLong ( 2 ) ;
   System.out.println("Rows: " + TotalRows + "\nValues: " + TotalValues);
}
```

Calculating a Subset of Rows

You can restrict the scope of a built-in calculation function by using a WHERE clause expression to specify the criteria for a row to be included in a calculation. Any valid WHERE clause expression can be used to filter rows to be excluded from the calculation. Likewise, the WHERE clause expression is used to include rows in the calculation. WHERE clause expressions are discussed earlier in this chapter.

Listing 7-77 illustrates how to restrict the scope of a built-in calculation function by using a WHERE clause expression. In this example, the DBMS is told to count the number of values in the OrderNumber column of the Orders table and to average and total the

value of the Quantity column of the Orders table. The WHERE clause contains an expression that joins together the Customers table and the Orders table using the customer number as the common value between these tables. This means the calculations are performed only on rows that are in the join. Rows whose customer numbers don't appear in both the Customers table and Orders table are excluded from the calculations.

This code segment is used with the Model B program and Listing 7-78 contains the DownRow() method to use with Listing 7-77 to retrieve and display values from the ResultSet.

Listing 7-77 Calculating a subset of rows.

```
try {
  String query = new String (" SELECT COUNT(OrderNumber), AVG(Quantity),
SUM(Quantity) " +
    " FROM Orders o, customers c " +
    " WHERE o.CustomerNumber = c.CustNumber");
  DataRequest = Database.createStatement();
  Results = DataRequest.executeQuery (query);
  DisplayResults (Results);
  DataRequest.close();
  }
```

Listing 7-78 Here is the DownRow() method to be used when calculating a subset of rows.

```
private  void DownRow ( ResultSet DisplayResults )
    throws SQLException
{
  long Orders;
  long Average;
  long Sum;
  Orders = DisplayResults.getLong ( 1 ) ;
  Average = DisplayResults.getLong ( 2 ) ;
  Sum = DisplayResults.getLong ( 3 ) ;
  System.out.println("Total Orders = " + Orders);
  System.out.println("Average Qty  = " + Average);
  System.out.println("Total Qty  = " + Sum);
}
```

NULLs and Duplicates

Two common problems that occur when using built-in functions are columns that don't contain a value and rows that contain duplicate values in the same column. Many times, you don't want empty columns and duplicate rows included in the calculation.

You can use the DISTINCT modifier to exclude duplicate rows from the calculation. Likewise, problems posed by NULL columns can be avoided by using the IS NULL operator along with the NOT operator in a selection expression. Both of these are discussed earlier in this chapter.

Calculating Without Using Built-In Functions

Although built-in calculation functions are very useful, the DBMS can perform calculations that are defined in the SELECT statement.

Listing 7-79 illustrates how to do this. In this example, the DBMS is directed to return to the program the value of the StoreNumber column and the difference between the value in the Sales column and the value in the Estimate column. Neither the Sales column nor the Estimate column data is returned. Instead, only the difference between these two columns is returned. Any arithmetic expression can be used in the SELECT statement, along with the appropriate names of columns that contain data elements used in the calculation.

This code segment is used with the Model B program and Listing 7-80 contains the DownRow() method to use with Listing 7-79 to retrieve and display values from the ResultSet.

Listing 7-79 Calculating without built-in functions.

```
try {
  String query = new String ("SELECT StoreNumber, Sales - Estimate " +
    " FROM Sales ");
  DataRequest = Database.createStatement();
  Results = DataRequest.executeQuery (query);
  DisplayResults (Results);
  DataRequest.close();
}
```

Listing 7-80 Here is the DownRow() method to be used when calculating without built-in functions.

```
private  void DownRow ( ResultSet DisplayResults )
    throws SQLException
{
  String Store;
  long Difference;
  Store = DisplayResults.getString ( 1 ) ;
  Difference = DisplayResults.getLong ( 2 ) ;
  System.out.println("Store = " + Store);
  System.out.println("Difference = " + Difference);
}
```

Grouping and Ordering Data

Columns are returned in the ResultSet in the order the column names appear in the SELECT statement of the query. The order in which rows appear in the ResultSet can be grouped into similar values or sorted in ascending or descending order by using the GROUP BY clause or the ORDER BY clause, which are discussed in the next few sections of this chapter.

Grouping is the task of organizing rows of data according to similar values within the same column. Let's say that you want to see sales for each store. The ResultSet can be grouped by store number.

Sorting is the task of organizing rows of data in either alphabetical or numerical order according to the value of a column in the ResultSet. A DBMS is capable of creating simple and complex sorting. A simple sort is when the values in a single column are used for the sort. A complex sort is when multiple columns are used for the sort, such as sorting rows by last name—and within last names, by first names.

Code segments in these sections return a ResultSet; therefore, the Model B program should be used to execute these code segments. Before executing these code segments, create a Sales table as defined next and then insert rows of data contained in Table 7-15 into the Sales table.

- Sales Table
  - StoreNumber, Number, Primary Key
  - Sales, Number
  - Estimate, Number
  - SalesRepNumber, Number

Group By

The GROUP BY clause specifies the name of the column whose values are used to group rows in the ResultSet. Listing 7-81 illustrates this technique. In this example, the

StoreNumber	Sales	Estimate	SalesRepNumber
123	300	200	4
223	450	500	3
123	322	200	4
345	56	30	8
345	125	100	5
223	76	50	2
345	200	156	5

Table 7-15. *Insert These Values into the Sales Table*

StoreNumber column and the sum of the values in the Sales column from the Sales table are returned.

The GROUP BY clause organizes the ResultSet by the value in the StoreNumber column. This means that the ResultSet contains the sum of the Sales column for each StoreNumber. Listing 7-82 contains the DownRow() method that is used to retrieve and display values from the ResultSet.

Listing 7-81 Grouping the ResultSet.

```
try {
  String query = new String (" SELECT StoreNumber, SUM(Sales) " +
    " FROM Sales " +
    " Group By StoreNumber");
  DataRequest = Database.createStatement();
  Results = DataRequest.executeQuery (query);
  System.out.println("Store      Sales");
  System.out.println("-----      -----");
  DisplayResults (Results);
  DataRequest.close();
}
```

Listing 7-82 Here is the DownRow() method to be used when grouping the ResultSet.

```
private  void DownRow ( ResultSet DisplayResults )
    throws SQLException
{
  long Store;
  long Sum;
  Store = DisplayResults.getLong ( 1 ) ;
  Sum = DisplayResults.getLong ( 2 ) ;
  System.out.println(Store + "          " + Sum);
}
```

Grouping Multiple Columns

The DBMS can create a subgroup within a group in the ResultSet. A subgroup is a grouping within another grouping. For example, the business unit may want to see orders organized by store—and within each store, orders organized by product.

A subgroup is created by placing the name of the column used for the subgroup as the second column name in the GROUP BY clause of the query. Any number of subgroups can be created, depending on the limitations established by the manufacturer of the DBMS that is used by your program. Column names in the GROUP BY clause must be separated with a comma.

Listing 7-83 illustrates how to create a subgroup. [illegible] ample, the ResultSet is grouped by store number, and within each store [illegible] a subgroup is created

using the sales representative number. Listing 7-84 contains the DownRow() method that is used to retrieve and display values from the ResultSet.

Listing 7-83 Creating groupings and subgroupings in the ResultSet.

```
try {
  String query = new String (" SELECT StoreNumber,SalesRepNumber, SUM(Sales) " +
    " FROM Sales " +
    " Group By StoreNumber, SalesRepNumber");
  DataRequest = Database.createStatement();
  Results = DataRequest.executeQuery (query);
  System.out.println("Store  SalesRep   Sales");
  System.out.println("-----  --------   -----");
  DisplayResults (Results);
  DataRequest.close();
}
```

Listing 7-84 Here is the DownRow() method to be used when grouping and subgrouping the ResultSet.

```
private  void DownRow ( ResultSet DisplayResults )
    throws SQLException
{
  long StoreNumber;
  long SalesRepNumber;
  long Sum;
  StoreNumber = DisplayResults.getLong ( 1 ) ;
  SalesRepNumber = DisplayResults.getLong ( 2 ) ;
  Sum = DisplayResults.getLong ( 3 ) ;
  System.out.println(StoreNumber + "      " + SalesRepNumber + "     " + Sum);
}
```

Conditional Grouping

The number of rows that are included in a group can be limited by including a conditional expression in the query. A conditional expression is similar to the WHERE clause expression discussed previously in this chapter.

The DBMS uses the conditional expression to qualify whether or not the current row should be included in any group of the ResultSet. Only rows that meet the condition are returned. A row that doesn't meet the condition is excluded. The conditional expression is placed in the HAVING clause of the query. The HAVING clause sets the criteria for a row to be included in a group.

Listing 7-85 demonstrates the use of the HAVING clause in a query. In this example, the value of the StoreNumber column and the total sales for each store from the Sales table are grouped by store number. The HAVING clause excludes from the group stores that have total sales of less than $401. Listing 7-86 contains the DownRow() method used to retrieve and display values in the ResultSet.

Here are the requirements for using a conditional expression in the HAVING clause:

- The expression must result in a single value.
- The result must appear in every column named in the expression.
- The expression can include a built-in calculation function.

Listing 7-85 Conditional grouping the ResultSet.

```
try {
  String query = new String ("SELECT StoreNumber, SUM(Sales) " +
    " FROM Sales " +
    " Group By StoreNumber" +
    " HAVING SUM(Sales) > 400");
  DataRequest = Database.createStatement();
  Results = DataRequest.executeQuery (query);
  System.out.println("Store  Sales");
  System.out.println("-----  -----");
  DisplayResults (Results);
  DataRequest.close();
}
```

Listing 7-86 Here is the DownRow() method to be used when conditionally grouping the ResultSet.

```
private  void DownRow ( ResultSet DisplayResults )
    throws SQLException
{
  long StoreNumber;
  long Sum;
  StoreNumber = DisplayResults.getLong ( 1 ) ;
  Sum = DisplayResults.getLong ( 2 ) ;
  System.out.println(StoreNumber + "          " + Sum);
}
```

Working with NULL Columns

Columns that are empty can create unexpected results when you execute a query. This is because sometimes the empty column is included or excluded from the operation depending on the nature of the query.

As you learned previously in this chapter, the DBMS includes empty columns when calculating average values and counting rows, but excludes empty columns when calculating the minimum or maximum value within a column.

The DBMS may include or exclude a row in a group depending on the conditional expression. Here's how this works:

- A row is included in a group if the empty column isn't used to group rows or used in the conditional expression in the HAVING clause.

- A row is excluded from the group if the empty column is used in the conditional expression and the conditional expression normally excludes empty columns from the calculation (i.e., MIN(), MAX()).
- A row is included in the group if the empty column is used in the conditional expression and the conditional expression normally includes empty columns from the calculation (i.e., AVG(), COUNT()).
- A row is included in the group if the empty column is used to group rows. Rows containing the empty column are placed in their own group.

Sorting Data

The ResultSet can be placed in alphabetical or numerical order by using the ORDER BY clause in the query. Listing 7-87 shows how to use the ORDER BY clause. In this example, values in the StoreNumber column and values in the Sales column from the Sales table are returned in the ResultSet. The ORDER BY clause specifies the StoreNumber column, which sorts the value of the StoreNumber column in ascending numerical order. Listing 7-88 contains the DownRow() method that is used to retrieve and display values from the ResultSet.

Listing 7-87 Sorting the ResultSet.

```
try {
   String query = new String ("SELECT StoreNumber, Sales " +
      " FROM Sales " +
      " ORDER BY StoreNumber");
   DataRequest = Database.createStatement();
   Results = DataRequest.executeQuery (query);
   System.out.println("Store  Sales");
   System.out.println("-----  -----");
   DisplayResults (Results);
   DataRequest.close();
}
```

Listing 7-88 Here is the DownRow() method to be used when sorting the ResultSet.

```
private  void DownRow ( ResultSet DisplayResults )
    throws SQLException
{
   long StoreNumber;
   long Sales;
   StoreNumber = DisplayResults.getLong ( 1 ) ;
   Sales = DisplayResults.getLong ( 2 ) ;
   System.out.println(StoreNumber + "          " + Sales);
}
```

Major and Minor Sort keys

You can create a sort within a sort by specifying more than one column to be sorted, such as sorting rows by customer last name—and then within last name, sorting by customer first name.

The first column specified in the ORDER BY clause is called the major sort key and is the initial value used to sort rows. The second and subsequent columns in the ORDER BY clause are called minor sort keys. A comma must separate each column name.

Listing 7-89 illustrates how to sort the ResultSet using major and minor sort keys. In this example, the StoreNumber column and Sales column from the Sales table are returned in the ResultSet.

The ORDER BY clause sorts the ResultSet by values in the StoreNumber column. Within each StoreNumber value, the value in the Sales column is sorted. Both sorts are ascending numerical sorts. Listing 7-88 contains the DownRow() method that is used to retrieve and display the ResultSet returned by Listing 7-89.

Listing 7-89 Sorting the ResultSet using major and minor sort keys.

```
try {
  String query = new String ("SELECT  StoreNumber, Sales " +
    " FROM Sales " +
    " ORDER BY StoreNumber, Sales");
  DataRequest = Database.createStatement();
  Results = DataRequest.executeQuery (query);
  System.out.println("Store  Sales");
  System.out.println("-----  -----");
  DisplayResults (Results);
  DataRequest.close();
}
```

Descending Sort

In addition to choosing the column to sort, you can also select the direction of the sort by using the ASC or DESC modifier. Listing 7-90 shows how to use the DESC modifier in the ORDER BY clause to specify a descending sort. By default the sort is ascending, although you can use the ASC modifier to explicitly direct that the sort be in ascending order.

The ASC or DESC modifier must appear after the column name in the ORDER BY clause, as illustrated in this example. The ASC or DESC modifier can be used for major and minor sort keys.

Listing 7-88 contains the DownRow() method that is used to retrieve and display the ResultSet returned by Listing 7-90.

Listing 7-90 Sorting the ResultSet in descending order.

```
try {
  String query = new String ("SELECT StoreNumber, Sales "
```

```
    " FROM Sales " +
    " ORDER BY StoreNumber DESC ");
  DataRequest = Database.createStatement();
  Results = DataRequest.executeQuery (query);
  System.out.println("Store  Sales");
  System.out.println("-----  -----");
  DisplayResults (Results);
  DataRequest.close();
}
```

Sorting Using Derived Data

Derived data is data that doesn't exist in a table but can be derived from data in a table, such as the calculation that was discussed previously in "Calculating Data." As illustrated in Listing 7-91, placing the calculation that derives the data in the SELECT statement creates derived data.

Listing 7-91 returns the store number and the difference between the value in the Sales column and the value in the Estimate column from the Sales table. The difference is derived data.

Derived data doesn't have a column name. This means that you are unable to include a column name in the ORDER BY clause to designate the derived data as the major or minor sort key for the sort.

However, you can use the column number of the derived data in place of a column name to sort the derived data. The column number of the derived data corresponds to the position of the derived data in the SELECT statement.

In this example, the calculation expression that produces the derived data is the second data referenced in the SELECT statement. Therefore, the derived data appears in the second column of the ResultSet. Use 2 in place of the column name in the ORDER BY clause. Listing 7-88 contains the DownRow() method that is used to retrieve and display the ResultSet returned by Listing 7-91.

Listing 7-91 Sorting derived data.

```
try {
  String query = new String ("SELECT StoreNumber, (Sales-Estimate) " +
    " FROM Sales " +
    " ORDER BY 2 ");
  DataRequest = Database.createStatement();
  Results = DataRequest.executeQuery (query);
  System.out.println("Store  Sales");
  System.out.println("-----  -----");
  DisplayResults (Results);
  DataRequest.close();
}
```

Subqueries

In the real world, you'll find that you need to create queries that are more complex than simply requesting columns be returned from the database. Instead, you might request columns based on the result of another query.

You can direct the DBMS to query the result of a query by creating a subquery within the query. A subquery joins together two queries to form one complex query, which efficiently identifies data to be included in the ResultSet.

A subquery is a query that is formatted very similar to a query. However, rows that are selected as a result of the subquery are not returned to your program. Instead, selected rows are queried by another query. Rows chosen by the second query are returned in the ResultSet.

A subquery resembles the format of a query. Each has a SELECT statement and a FROM clause, and can also include a WHERE clause and a HAVING clause to qualify rows to return. The WHERE clause and the HAVING clause are used to express a condition that must be met for a row to be included in the ResultSet.

There are also differences between a query and a subquery. The most noticeable difference is with the ResultSet. The ResultSet of a query is returned to the J2EE component. In contrast, the ResultSet of a subquery is returned to a temporary table, which is then used by a query or another subquery to further extract rows.

This means the J2EE component never sees the result of a subquery. Instead, the subquery result is an intermediate step working towards the final selection of data from the database.

There are two rules that you must follow when using a subquery in your program. These rules are

- **Return one column from the subquery** The purpose of a subquery is to derive a list of information from which a query can choose appropriate rows. Only a single column needs to be included in the list.
- **Don't sort or group the result from a subquery** Since the ResultSet of the subquery isn't going to be returned in the ResultSet of the query, there isn't a need to sort or group data in the ResultSet of a subquery.

The next several sections illustrate how to create a subquery. These sections can be run using the Model B program, as described previously in this chapter. Before attempting to run these code segments, create the following tables and insert rows shown in Table 7-16 and Table 7-17 into the appropriate tables.

- Sales Table
  - StoreNumber, Number, Primary Key
  - Estimate, Number
  - SalesRepNumber, Number

StoreNumber	Estimate	SalesRepNumber
345	200	4
278	500	3
825	200	2

Table 7-16. *Insert These Values into the Sales Table*

- Orders Table
  - OrderNumber, Number, Primary Key
  - ProdNumber, Number
  - CustomerNumber, Number
  - StoreNum, Number
  - Amount, Number

Creating a Subquery

A subquery is a query whose results are evaluated by an expression in the WHERE clause of another query, as is illustrated in Listing 7-92. In this example, the subquery is defined below the first WHERE clause. The subquery joins rows of the Sales table and the Orders table by store number value, as is shown in the second WHERE clause. Once joined, the subquery totals the value of the Amount column for each store number and returns the results to a temporary table.

OrderNumber	ProdNumber	CustomerNumber	StoreNum	Amount
122	5237	591	345	200
334	3255	901	278	321
365	3255	901	278	433
534	7466	591	825	523
555	1052	999	278	23
717	1052	721	825	75
874	7466	721	825	354

Table 7-17. *Insert These Values into the Orders Table*

The query then returns a ResultSet that contains store numbers where the value of the Estimate column in the Sales table is equal to the total number of the Amount column returned by the subquery. Listing 7-93 contains the DownRow() method that retrieves and displays the ResultSet from Listing 7-92.

Listing 7-92 Creating a subquery.

```
try {
  String query = new String (" SELECT StoreNumber " +
    " FROM Sales "+
    " WHERE Estimate = (SELECT SUM(Amount) " +
    " FROM Orders, Sales " +
    " WHERE StoreNum = StoreNumber) ");
  DataRequest = Database.createStatement();
  Results = DataRequest.executeQuery (query);
  System.out.println("Store");
  System.out.println("-----");
  DisplayResults (Results);
  DataRequest.close();
}
```

Listing 7-93 The DownRow() method used with a subquery.

```
private  void DownRow ( ResultSet DisplayResults )
      throws SQLException
{
  long Store;
  Store = DisplayResults.getLong ( 1 ) ;
  System.out.println(Store);
}
```

Conditional Testing

Any conditional expression, as discussed previously in this chapter, can be used to evaluate a relationship between a query and the results of the subquery. There are four types of conditional tests that you can use with a subquery. These are

- **Comparison test** This is a test that uses comparison operators to compare values in the temporary table with values in the table used by the query.
- **Existence test** This test determines if a value in the current row of the table used by the query also exists in the temporary table.
- **Set membership test** This test is similar to the existence test in that the DBMS is directed to determine if a value in the current row of the table used by the query also exists in the temporary table.
- **Qualified test** This test consists of either the Any test or the All test and determines if a value in the current row of the table used by the query is in one row of the temporary table or all rows of the temporary table.

The comparison test is illustrated earlier in this chapter. The next several sections show how to perform the other tests. Before executing code segments in these sections, insert rows shown in Table 7-18 in the Sales table.

The Existence Test

The existence test is used whenever you need to return rows in the ResultSet where a value in a column is present in the results of the subquery. The existence test requires that you place the EXISTS modifier between the query and the subquery.

Listing 7-94 gives a good example of how to use an existence test. The DBMS is told to return a single instance of store numbers from the Sales table only if the store number is in the Orders table. The store number isn't returned if the store hasn't placed an order. Listing 7-93 contains the DownRow() method that can be used with Listing 7-94.

Listing 7-94 Performing the existence test.

```
try {
  String query = new String (" SELECT DISTINCT StoreNumber " +
    " FROM Sales  "+
    " WHERE EXISTS " +
    " (SELECT StoreNum " +
    " FROM Orders " +
    " WHERE StoreNum = StoreNumber) ");
  DataRequest = Database.createStatement();
  Results = DataRequest.executeQuery (query);
  System.out.println("Store");
  System.out.println("-----");
  DisplayResults (Results);
  DataRequest.close();
}
```

Membership Test

The IN modifier is used to determine if a value in the table that is being queried is a member of the results produced by the subquery. If the value appears in both tables

StoreNumber	Estimate	SalesRepNumber
345	150	5
278	300	6
825	230	7

Table 7-18. *Insert These Values into the Sales Table*

and results of the subquery, then the conditional test is evaluated as true and the row that contains the value is returned in the ResultSet; otherwise, the row is skipped.

The IN modifier is used in Listing 7-95. In this example, the subquery returns store numbers from the Orders table where the value in the Estimate column of the corresponding row in the Sales table is less than the value in the Amount column of the Orders table.

The IN modifier is used to determine if the store number in the Sales table is returned in the results of the subquery. If so, then the sales representatives number from the Sales table is returned in the ResultSet; otherwise, the sales representative number in the current row of the Sales table is skipped. Listing 7-93 contains the DownRow() method that can be used with Listing 7-95.

Listing 7-95 Performing the membership test.

```
try {
  String query = new String (" SELECT SalesRepNumber " +
    " FROM Sales  "+
    " WHERE StoreNumber IN " +
    " (SELECT StoreNum " +
    " FROM Orders " +
    " WHERE Estimate < Amount) ");
  DataRequest = Database.createStatement();
  Results = DataRequest.executeQuery (query);
  System.out.println("Store");
  System.out.println("-----");
  DisplayResults (Results);
  DataRequest.close();
}
```

ANY Test

The ANY test determines if the value specified in the query is in any of the rows returned by the subquery. If there is at least one match, the row is returned in the ResultSet; otherwise, the row is ignored.

Listing 7-96 shows how to construct an ANY test. The objective of this program is to return the store number from the Sales table for any store where the estimate is greater than the amount of any of the store's orders in the Orders table.

The ANY test tells the DBMS to include the store number in the ResultSet if the value of the Estimate column for the store in the Sales table is greater than any value in the Amount column for that store in the temporary table. Listing 7-93 contains the DownRow() method that can be used with Listing 7-96.

Rules for the ANY Test

- The ANY test fails if the subquery produces an empty result. Therefore, the query doesn't return any rows in the ResultSet.

- A NULL value returned by the subquery causes the query to return a NULL value. This is because the query is unsure if the value of the table in the query is in the results of the subquery.
- No value is returned in the ResultSet if the value in the table being queried is not in the results of the subquery.

Listing 7-96 Performing the ANY test.

```
try {
   String query = new String (" SELECT DISTINCT StoreNumber " +
      " FROM Sales  "+
      " WHERE Estimate  > ANY  " +
      " (SELECT Amount" +
      " FROM Orders " +
      " WHERE StoreNumber = StoreNum) ");
   DataRequest = Database.createStatement();
   Results = DataRequest.executeQuery (query);
   System.out.println("Store");
   System.out.println("-----");
   DisplayResults (Results);
   DataRequest.close();
}
```

ALL Test

The ALL test is similar to the ANY test in that the value of the table specified in the query is compared to values returned by the subquery. However, these tests differ in that the ANY test requires at least one match for the row to be included in the ResultSet. The ALL test requires that the values in the table specified in the query match all the values returned by the subquery.

Listing 7-97 illustrates how to implement the ALL test. In this example, the store number joins together the Sales table and the Orders table. Once the tables are joined, the subquery is processed, which returns the value in the Amount column for each store in the Orders table.

Next, the value in the Estimate column of the Sales table is compared with each Amount value returned by the subquery. If the Amount value for each row returned by the subquery is less than the value in the Estimate column, the row in the Sales table is placed in the ResultSet. Otherwise, the row in the Sales table is skipped. The ALL test follows the same rules that apply to the ANY test (see the previous section). Listing 7-93 contains the DownRow() method that can be used with Listing 7-97.

Listing 7-97 Performing the ALL test.

```
try {
   String query = new String (" SELECT DISTINCT Store " +
      " FROM Sales  "+
      " WHERE Estimate  > ALL  " +
```

```
    " (SELECT Amount" +
    " FROM Orders " +
    " WHERE StoreNumber = StoreNum) ");
DataRequest = Database.createStatement();
Results = DataRequest.executeQuery (query);
System.out.println("Store");
System.out.println("-----");
DisplayResults (Results);
DataRequest.close();
}
```

VIEW

You can reduce the complexity of your J2EE component by creating one or more VIEW of the database for each user ID that is passed to the J2EE component for data access. A VIEW is similar to creating a table that contains only the data the user ID is permitted to access. A VIEW limits columns and rows that can be queried to specific information that pertains to a user ID. A VIEW is like a filter that hides information from a user ID.

Each VIEW is uniquely identified with a name and contains selection criteria for columns and rows that appear in the VIEW when the VIEW is used by a J2EE component. Once a VIEW is created, the J2EE component references a VIEW the same way that a table is referenced in a query.

Rules for Using VIEWs

- Create as many VIEWs as necessary to simplify access to a database.
- Restrict access to a table on a need to know basis.
- Work with the owner of the data to establish reasonable restrictions.
- Classify users into groups that have similar access requirements to information.
- Create a VIEW for each classification of user, rather than for each user.
- More than one column can be used in a VIEW.
- More than one table can be used in a VIEW.
- A VIEW is treated as a table in a query regardless of the number of columns and tables that are used to create the VIEW.
- Use a VIEW whenever your program accesses some columns in many tables. The VIEW simplifies the number of tables that a J2EE component needs to directly access. Beware of data security restrictions that affect the underlying columns and tables used to create the VIEW. A VIEW inherits data security restrictions from tables used to create the VIEW. Therefore, if the user ID doesn't have rights to information in a table, the VIEW also has the same restrictions.

The following sections illustrate how to create and use a VIEW. You can execute code segments within each section using the Model A program. However, before doing so, create the following tables and insert into those tables the rows shown in Table 7-19 and Table 7-20.

- Orders Table
  - OrderNumber, Number, Primary Key
  - ProdNumber, Number
  - CustomerNumber, Number
  - StoreNum, Number
  - Amount, Long
- Products Table
  - ProductNumber, Number, Primary Key
  - ProductName, VARCHAR
  - UnitPrice, Long

Creating a VIEW

A VIEW is created by using the CREATE VIEW statement, as is illustrated in Listing 7-98. In this example, the CREATE VIEW statement contains the name of the VIEW, which is Store278. Any unique name can be used as the name of a VIEW.

The AS modifier contains the query whose results form the rows and column that are contained in the VIEW. In this example, all the columns and rows of the Orders table

OrderNumber	ProdNumber	CustomerNumber	StoreNum	Amount
122	5237	591	345	200
334	3255	901	278	321
365	3255	901	278	433
534	7466	591	825	523
555	1052	999	278	23
717	1052	721	825	75
874	7466	721	825	354

Table 7-19. *Insert These Values into the Orders Table*

ProductNumber	ProductName	UnitPrice
1052	CD Player	100
3255	VCR	250
5237	DVD Player	325
7466	50-inch TV	532

Table 7-20. *Insert These Values into the Products Table*

where the value of the StoreNum column is 278 are placed in the VIEW. Only rows that pertain to store 278 are contained in the VIEW. Data from all other stores are excluded.

Listing 7-98 Creating a VIEW.

```
try {
  String query = new String (" CREATE VIEW Store278 AS " +
    " SELECT  * " +
    " FROM Orders  " +
    " WHERE StoreNum  = 278");
  DataRequest = Database.createStatement();
  DataRequest.execute(query);
  DataRequest.close();
}
```

Selecting Columns to Appear in the VIEW

You can include or exclude any column in a VIEW. Columns excluded from a VIEW remain in underlying tables used to create the VIEW. However, those columns are hidden from the J2EE component.

Listing 7-99 illustrates how to include columns in a VIEW. This example creates a VIEW called StoreProd. The SELECT statement contains column names from the Orders table that are included in the VIEW. All other columns of the Orders table are excluded from the VIEW.

Listing 7-99 Selecting columns to appear in a VIEW.

```
try {
  String query = new String (" CREATE VIEW StoreProd AS " +
    " SELECT StoreNum, ProdNumber " +
    " FROM Orders  ");
  DataRequest = Database.createStatement();
  DataRequest.executeQuery (query);
  DataRequest.close();
}
```

Creating a Horizontal VIEW

There are two kinds of VIEWs. These are a vertical VIEW and a horizontal VIEW. A vertical VIEW includes all rows of the underlying table and includes some, but not all, columns of the table. This is illustrated in Listing 7-99. A horizontal VIEW contains all columns in the underlying table, but only some rows of the table. This means some rows are excluded from the VIEW and cannot be accessed.

Listing 7-100 shows how to create a horizontal VIEW. In this example, a VIEW called cust901 is created that contains all rows in the Orders table where the value of the CustomerNumber column is 901. Horizontal VIEWs are ideal for situations where access to all columns of a table(s) is required, but access to all rows is not required.

isting 7-100 Creating a horizontal VIEW.

```
try {
  String query = new String (" CREATE VIEW cust901 AS " +
    " SELECT  * " +
    " FROM Orders" +
    " WHERE CustomerNumber = 901");
  DataRequest = Database.createStatement();
  Results = DataRequest.execute(query);
  DataRequest.close();
}
```

Creating a Multitable VIEW

A multitable VIEW is created similar to how a single table VIEW is created, as illustrated in previous sections of this chapter. Tables used in the VIEW must be joined. Listing 7-101 shows how this is done. Notice that a vertical VIEW is created and contains three columns from the Orders table and the Products table. The Orders table and the Products table are joined together in the WHERE clause using the product number. Although columns in the VIEW come from two tables, columns are treated as if they are from the same table in a query that uses the VIEW.

sting 7-101 Creating a multitable VIEW.

```
try {
  String query = new String (" CREATE VIEW ProdDesc AS " +
    " SELECT  StoreNum, ProdNumber, ProductName " +
    " FROM Orders; Products  " +
    " WHERE ProdNumber = ProductNumber");
  DataRequest = Database.createStatement();
  Results = DataRequest.execute(query);
  DataRequest.close();
}
```

Grouping and Sorting VIEW

Rows in a VIEW can be grouped and sorted when the VIEW is created, rather than grouping or sorting rows in a query that uses the VIEW. A VIEW can be grouped or sorted by using the GROUP BY clause and/or the ORDER BY clause, as discussed previously in this chapter.

Listing 7-102 shows how to sort rows in the VIEW. In this example, a VIEW called GroupProdDesc is created based on the underlying Orders table and Products table, which are joined together by product number. The GroupProdDesc VIEW consists of the store number, product number, and product name that are sorted by the value in the ProdNumber column.

Listing 7-102 Grouping and sorting VIEWs.

```
try {
  String query = new String (" CREATE VIEW GroupProdDesc AS " +
    " SELECT  StoreNum, ProdNumber, ProductName " +
    " FROM Orders, Products  " +
    " WHERE ProdNumber = ProductNumber" +
    " ORDER BY ProdNumber  ");
  DataRequest = Database.createStatement();
  Results = DataRequest.execute(query);
  DataRequest.close();
}
```

Modifying a VIEW

A VIEW can be modified and those modifications affect the underlying tables that are used to create the VIEW. There are three ways in which a VIEW can be modified. These are

- **Update** Values in one or more columns of a VIEW are changed to values supplied by the query.
- **Insert** A new row is added to the VIEW and indirectly to the underlying table(s) used to create the VIEW.
- **Delete** A row is removed from the VIEW and from the underlying table(s) used to create the VIEW.

Rules for Updating a VIEW

Calculation, expressions, and built-in column functions cannot be used in the SELECT statement of the VIEW. Instead, only names of columns can be used.

- Exclude the GROUP BY clause and HAVING clause from the VIEW.
- Duplicate rows must be included in the modification. You cannot use the DISTINCT modifier in the VIEW if values of the VIEW are going to be modified.

- The user ID used in your program must have rights to modify the underlying tables that comprise the VIEW.
- Subqueries cannot be used in a VIEW if rows of the VIEW are to be modified.

Updating a VIEW

You can replace values in a VIEW by using the UPDATE statement, as discussed previously in this chapter. Listing 7-103 illustrates how this is done. In this example, the Store278 VIEW is referenced in the UPDATE statement. The SET clause specifies the Amount column, which is to be updated, and the value that will override the current value in the Amount column. This means the number 700 replaces whatever value is in the column.

The WHERE clause is used to identify rows in the VIEW that are to be updated. Rows that have 334 as the value in the OrderNumber column will have 700 inserted into the Amount column.

Listing 7-103 Updating a VIEW.

```
try {
  String query = new String (" UPDATE Store278 " +
    " SET Amount = 700 " +
    " WHERE OrderNumber  = 334 ");
  DataRequest = Database.createStatement();
  Results = DataRequest.execute(query);
  DataRequest.close();
}
```

Inserting a Row into a VIEW

A new row can be inserted into the underlying tables that comprise a VIEW by using the INSERT INTO statement and referencing the name of the VIEW, as shown in Listing 7-104. In this example, a new row is inserted in the Store278 VIEW using the same technique as described previously in this chapter to insert a row into a table.

Listing 7-104 Inserting a row into a VIEW.

```
try {
  String query = new String (" INSERT INTO Store278 " +
    " (OrderNumber, ProdNum, CustomerNumber, StoreNum, Amount) " +
    " VALUES (325, 9545 ,301 ,278 ,400) ");
  DataRequest = Database.createStatement();
  Results = DataRequest.execute(query);
  DataRequest.close();
}
```

Deleting a Row from a VIEW

You can remove a row from a VIEW by using the DELETE FROM statement. The DELETE FROM statement requires the name of the VIEW from which the row or rows are to be deleted.

Listing 7-105 illustrates how to delete a row from a VIEW. In this example, one or more rows from Store278 are deleted. Rows designated to be deleted are identified in the WHERE clause. In this case, rows that have the value 555 in the OrderNumber column are removed from the Orders table, which is the underlying table of the Store278 VIEW.

Listing 7-105 Deleting a row from a VIEW.

```
try {
  String query = new String (" DELETE FROM Store278 " +
    " WHERE OrderNumber = 555   ");
  DataRequest = Database.createStatement();
  Results = DataRequest.execute(query);
  DataRequest.close();
}
```

Dropping a VIEW

A VIEW can be removed by using the DROP VIEW statement. The DROP VIEW statement requires the name of the VIEW that is to be dropped. There are two modifiers that are used with the DROP VIEW statement. These are

- **CASCADE** Removes all VIEWs that depend on the VIEW specified in the DROP VIEW statement, and removes the specified VIEW.
- **RESTRICT** Removes only the VIEW specified in the DROP VIEW statement. All dependent VIEW remain intact.

A dependent VIEW is a VIEW that has another VIEW as one of its underlying components, rather than a table. Listing 7-106 shows you how to drop a VIEW. In this example, the Store278 VIEW and all of its dependent VIEWs are dropped, although there aren't any dependent VIEWs. Some DBMSs use DROP TABLE instead of DROP VIEW.

Listing 7-106 Dropping a VIEW.

```
try {
  String query = new String (" DROP VIEW  Store278 CASCADE");
  DataRequest = Database.createStatement();
  Results = DataRequest.DataRequest.execute(query);
  DataRequest.close();
}
```