The Complete Reference



Part III

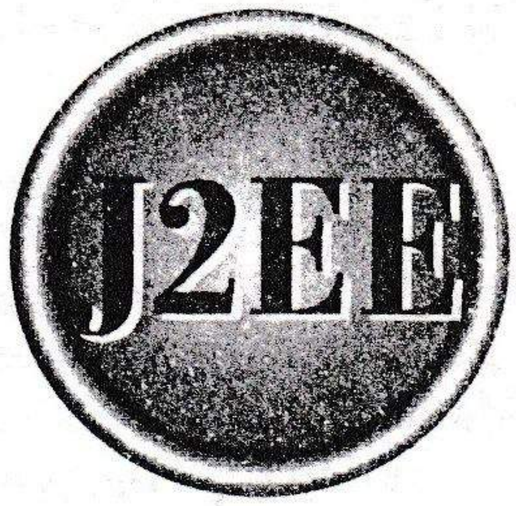
J2EE Databases

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.

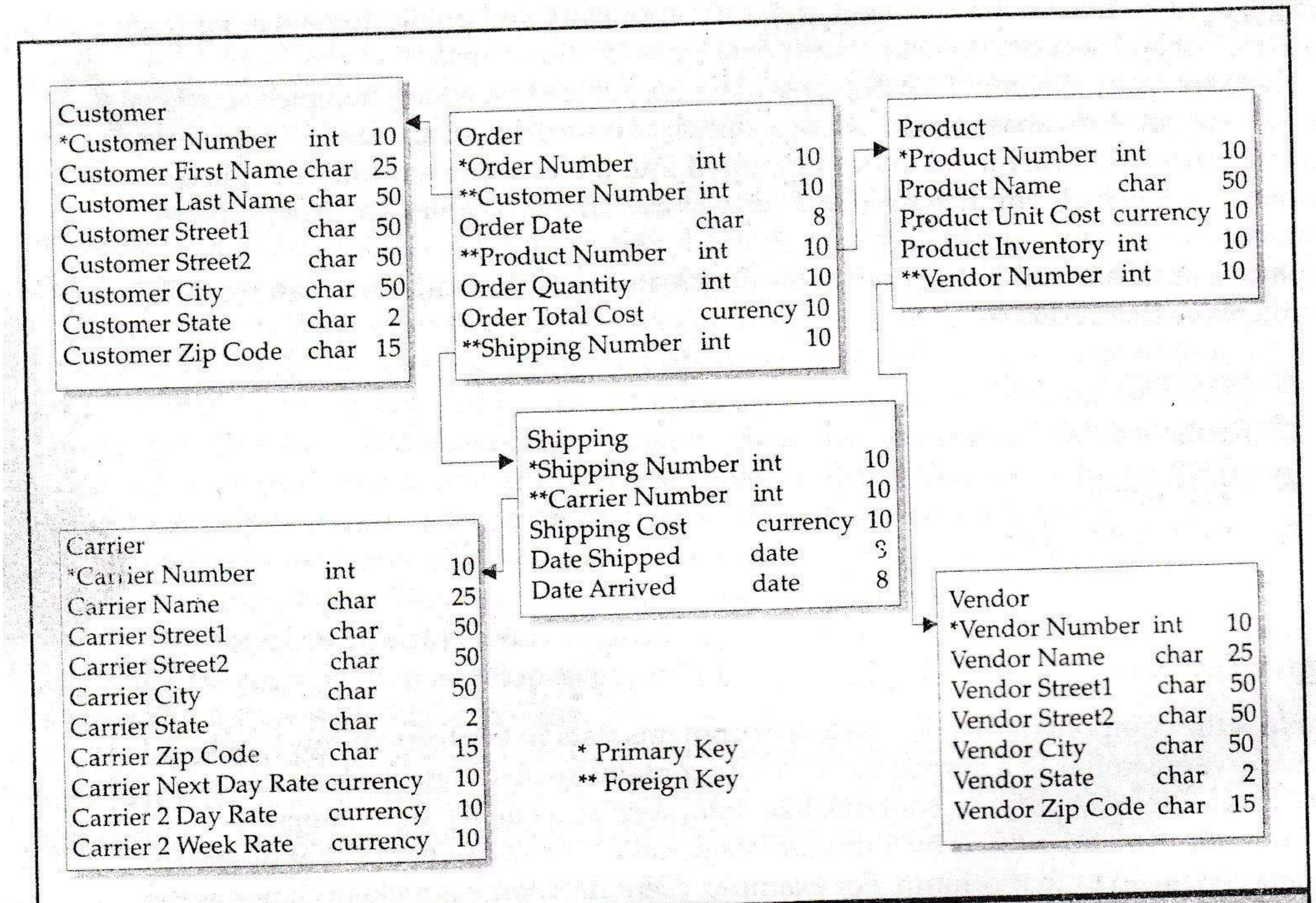


Figure 5-1. *A database schema diagrams the relationships among all components of a database.*

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

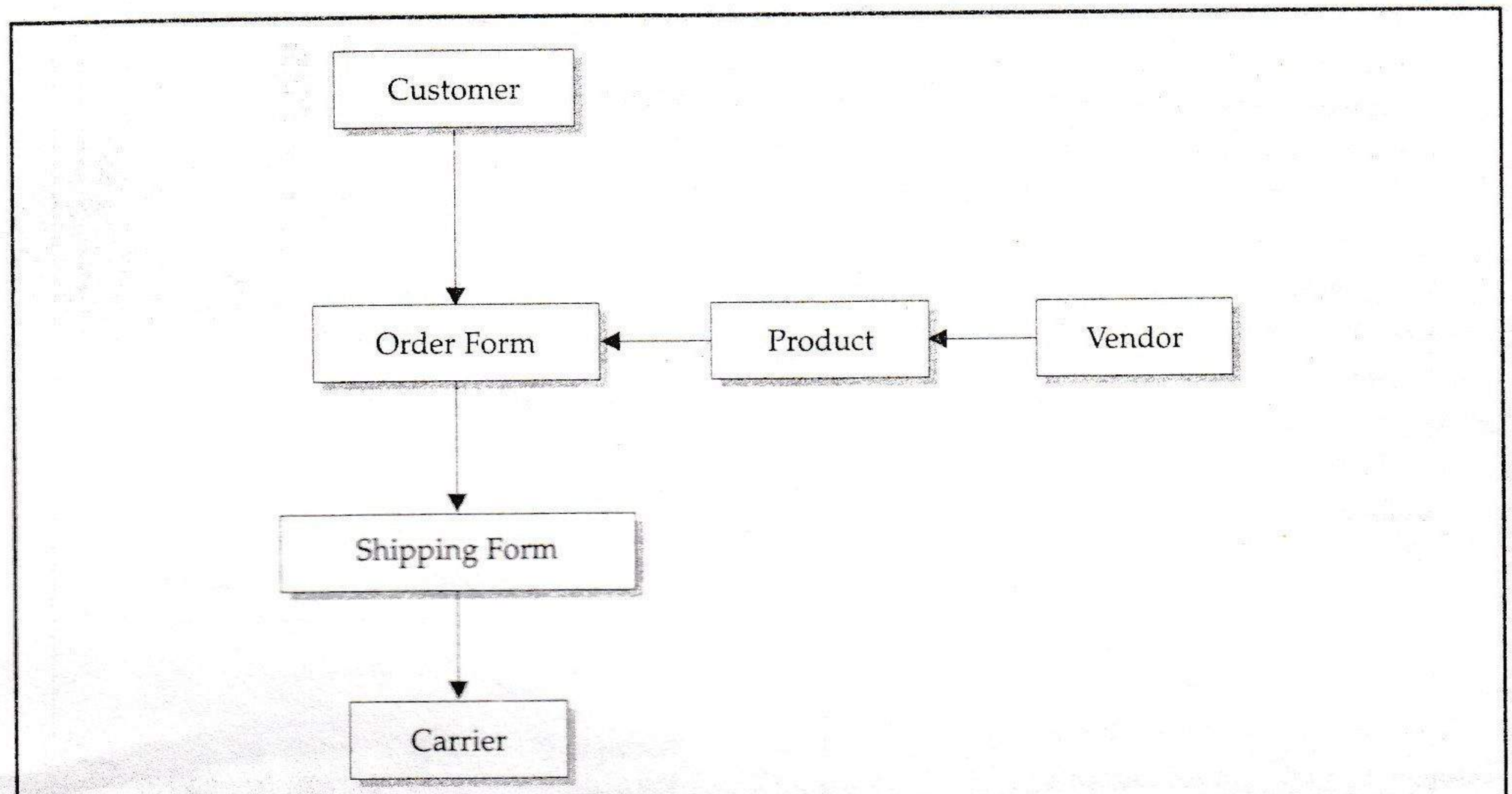


Figure 5-2. *Entities of the order system*

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.

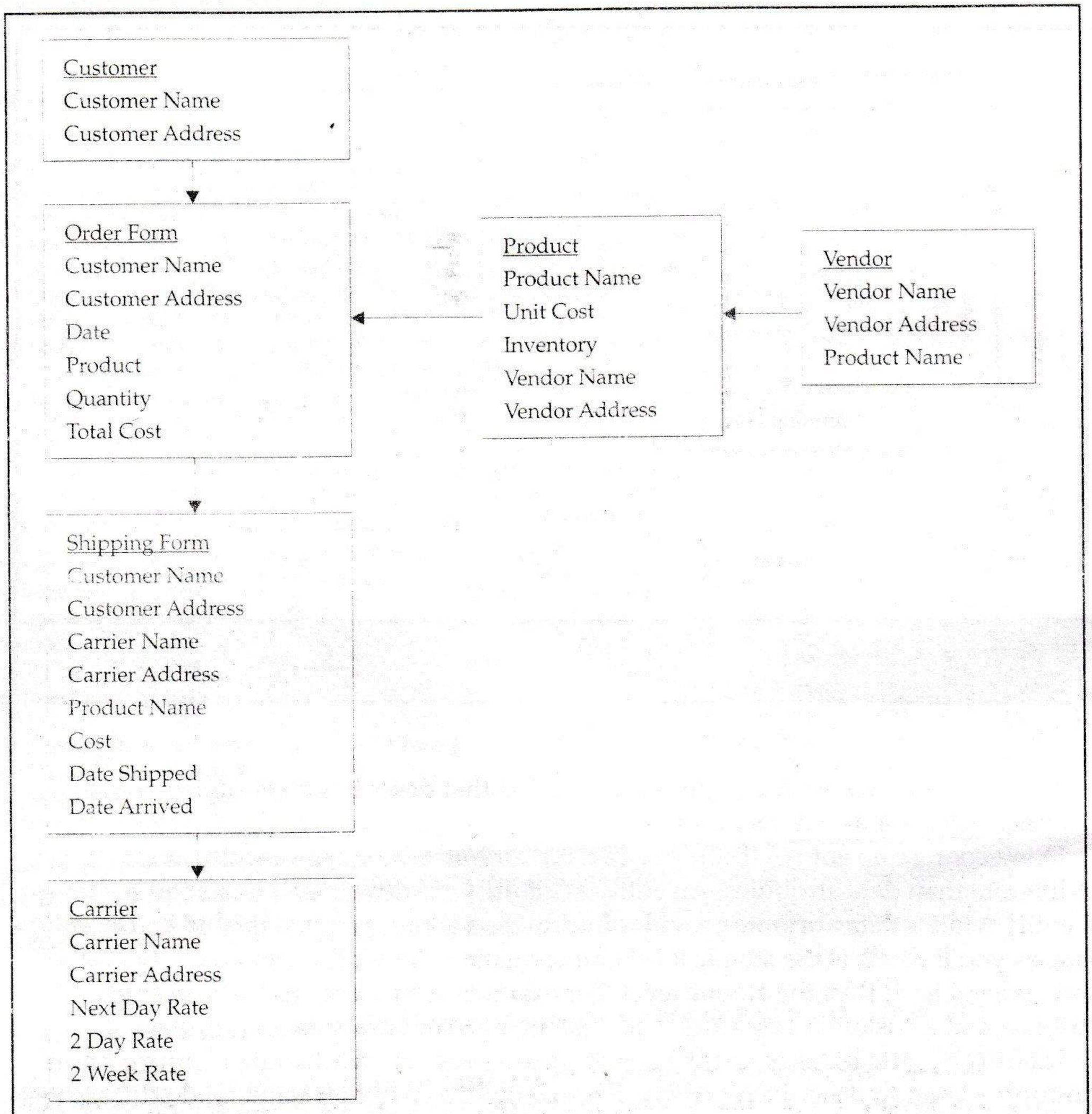


Figure 5-3. *Attributes of entities of the order system*

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.

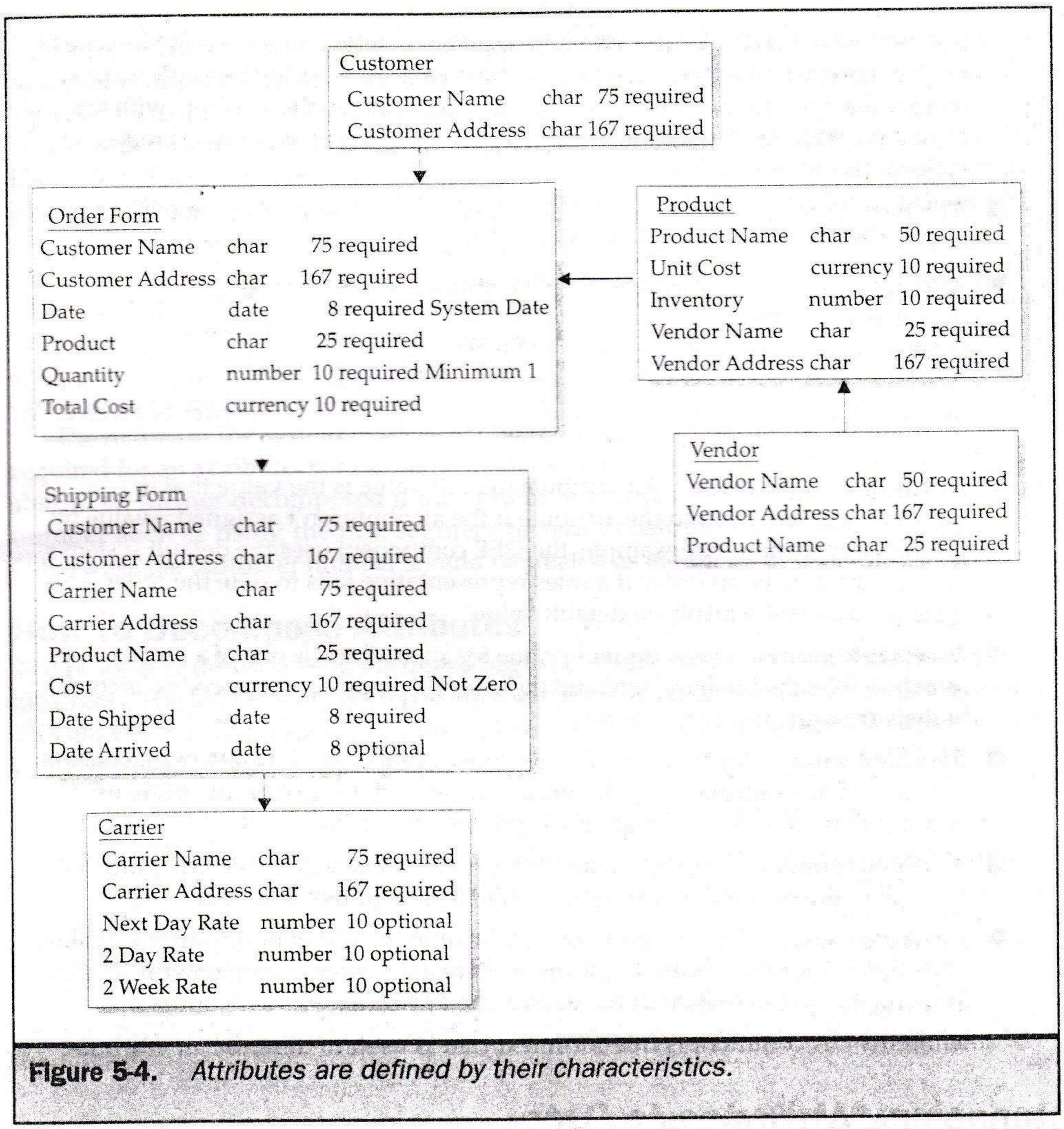


Figure 5-4. *Attributes are defined by their characteristics.*

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.

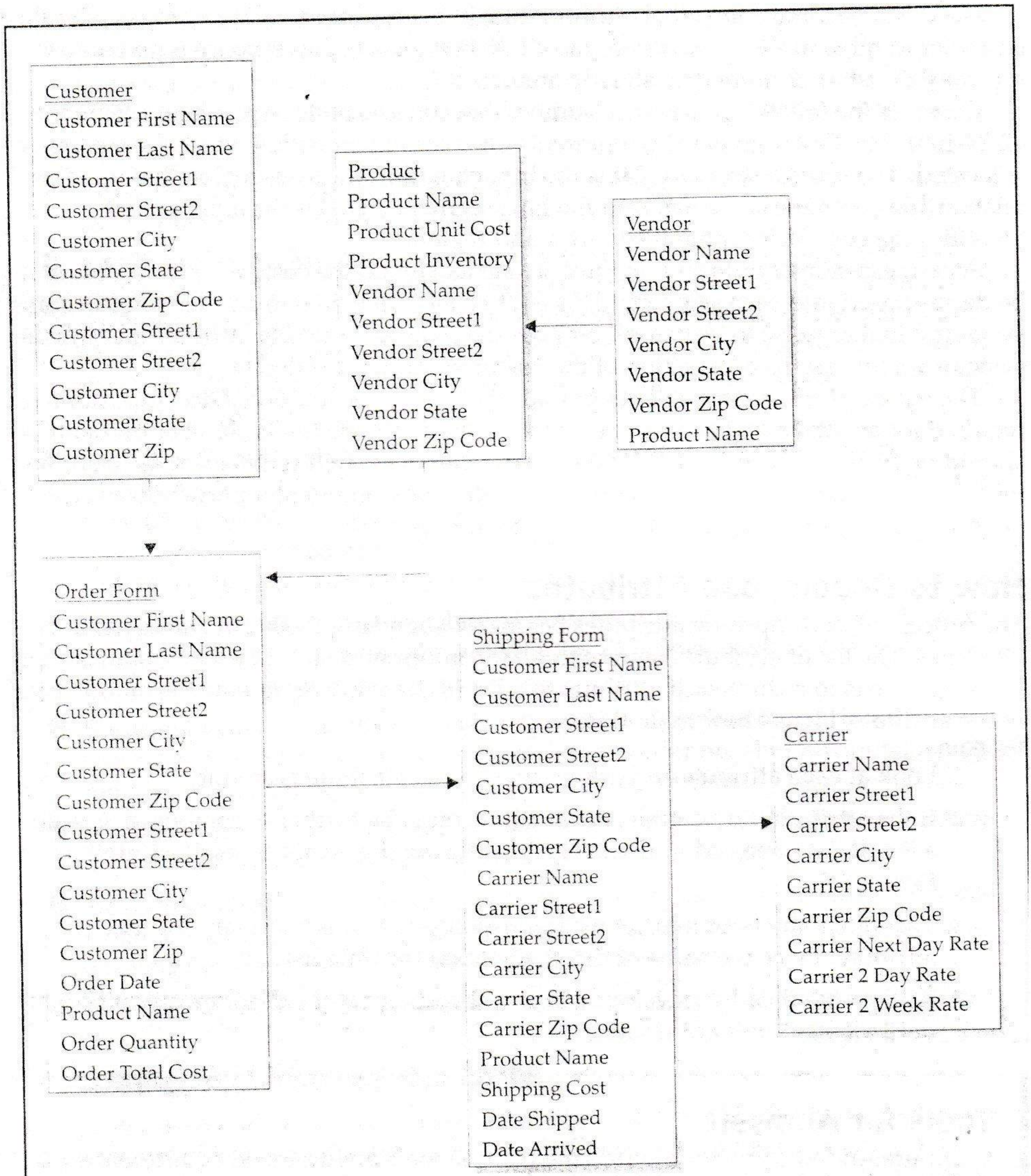


Figure 5-5. *Data elements for the customer name attribute*

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.

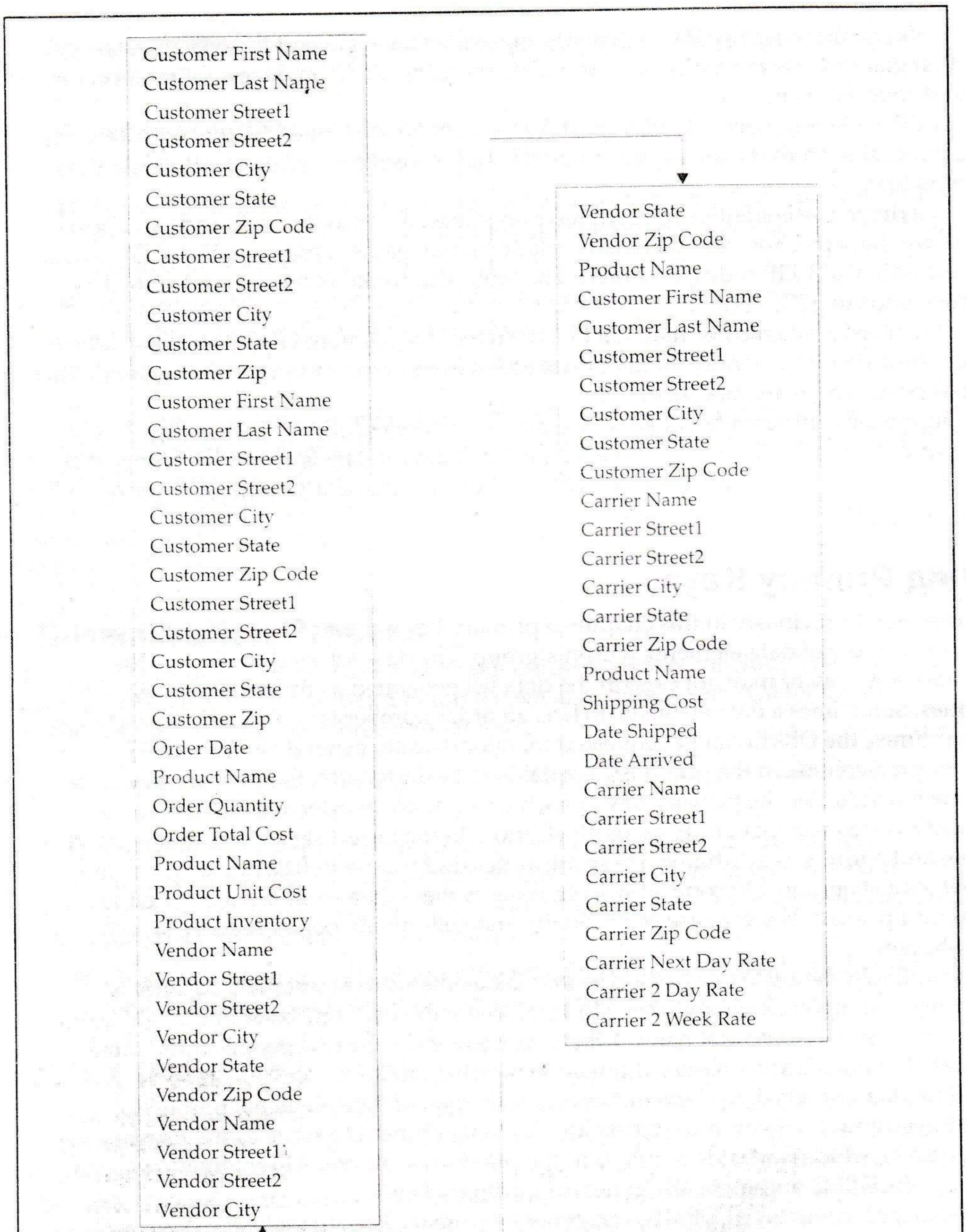


Figure 5-6. *Here is a list of data elements from the entities shown in Figure 5-5.*

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

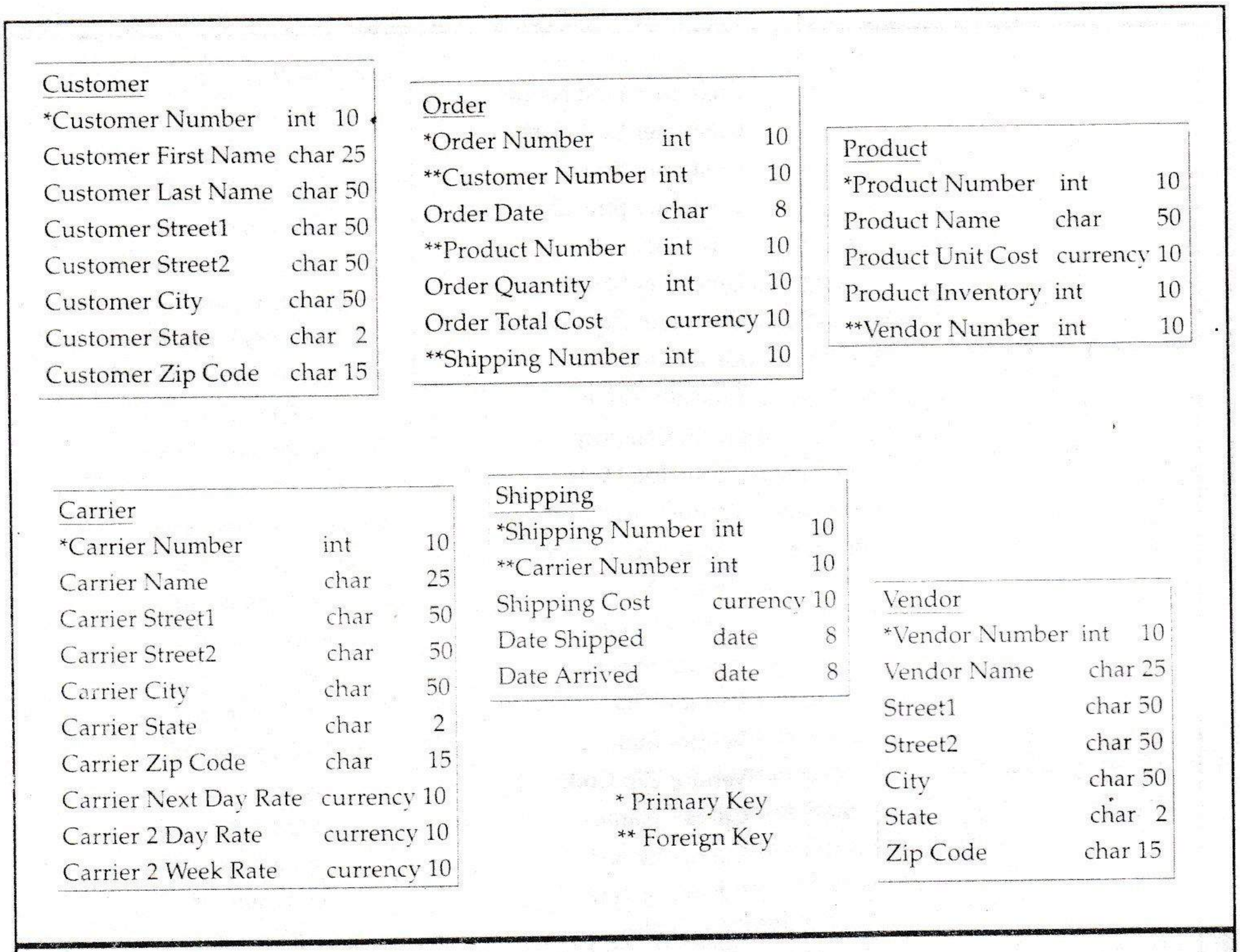


Figure 5-8. *Related data are organized into groups.*

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.

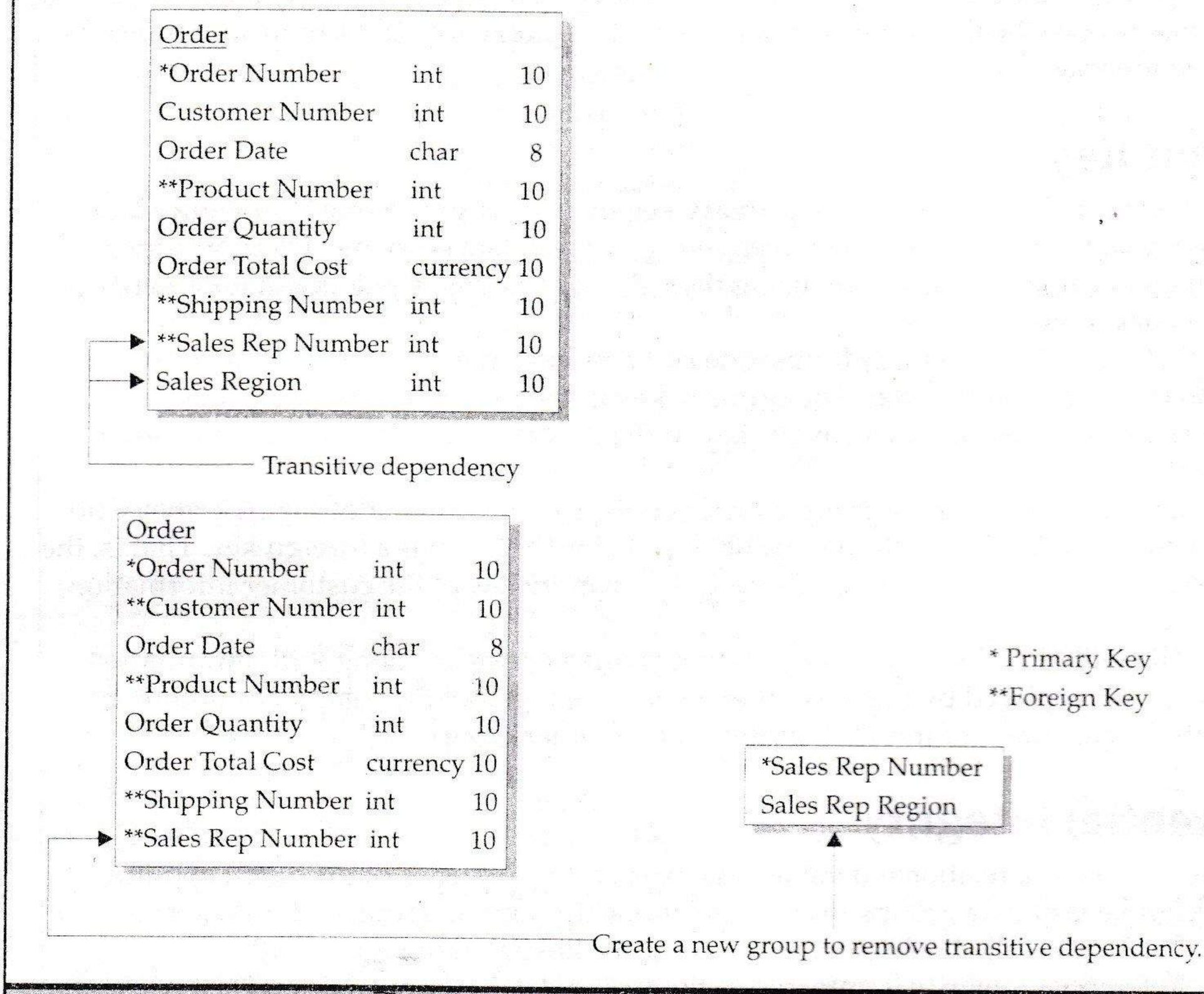


Figure 5-9. *Sales rep number and region (top) form a transitive dependency. Regrouping data elements (see bottom) resolves the transitive dependency problem.*

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.

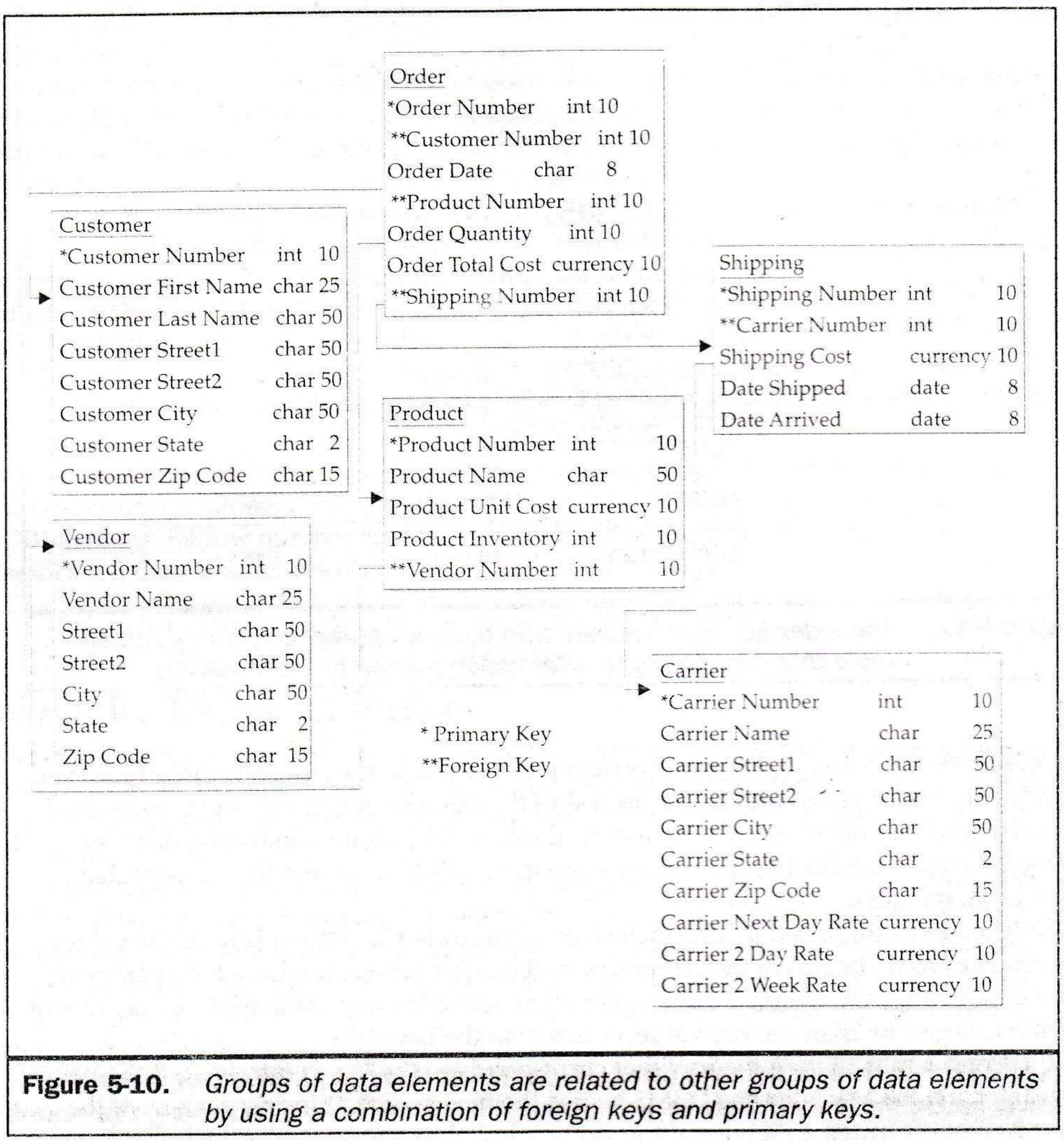


Figure 5-10. *Groups of data elements are related to other groups of data elements by using a combination of foreign keys and primary keys.*

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.

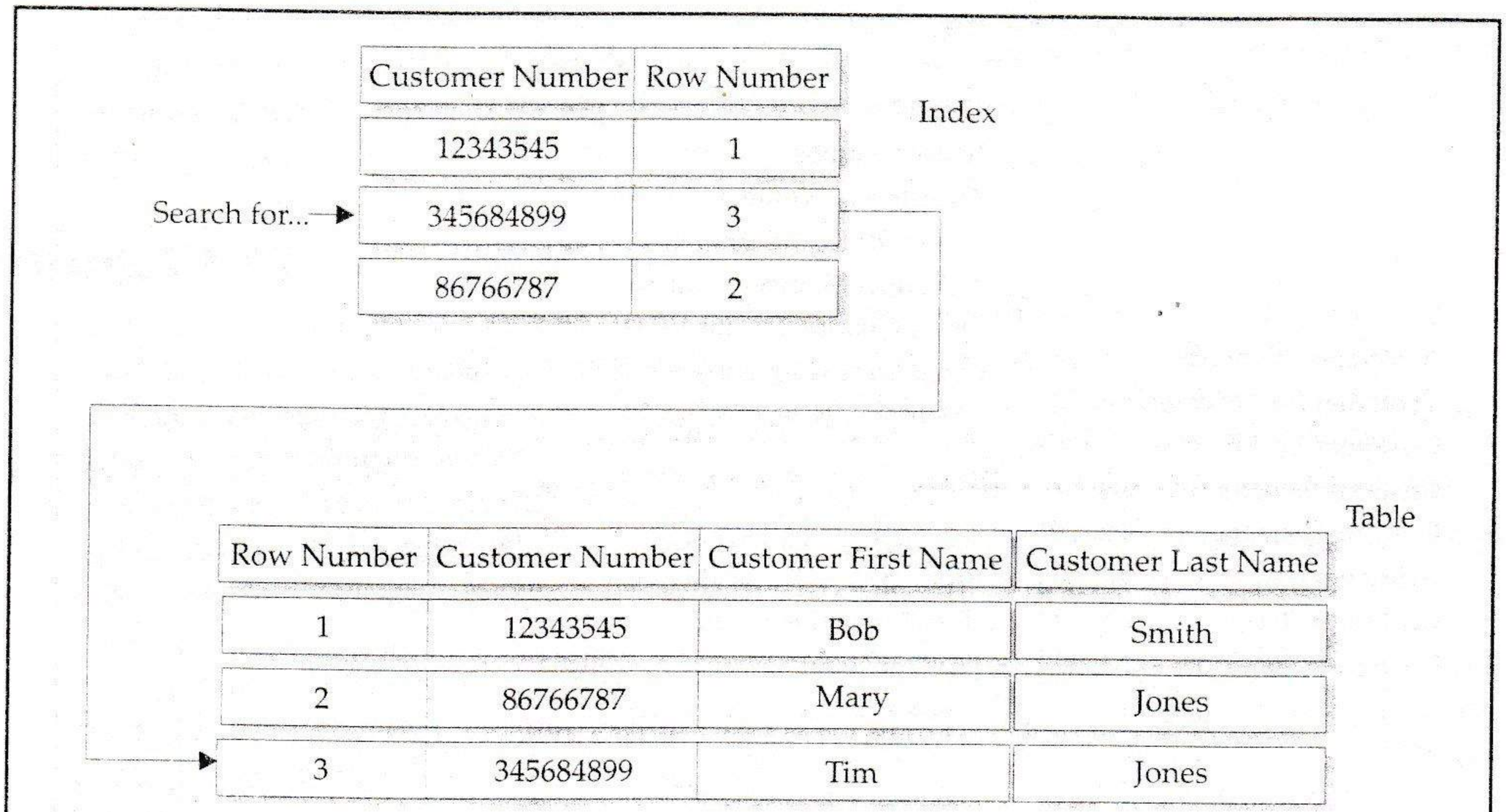


Figure 5-11. *The index key is associated with the row number of the row in the table that contains other information related to the index key.*

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();
}
```

The Complete Reference



Part III

J2EE Foundation

A J2EE application is centered on core Java technologies that provide both client-side and server-side functionality to a J2EE application. In this part you'll learn about these technologies and how you can implement them in your J2EE application.

You begin with HTML, XML, and XHTML, which are markup languages used in a light client application and for transmitting information between web services and other applications. Once you

developed a solid foundation in these markup languages, you are shown how to implement them using Java API for XML Processing and Java API for XML Messaging.

Your focus is then directed to three critical Java technologies. These are Java servlets, Java ServerPages, and Enterprise JavaBeans. Java servlets and Java ServerPages are used to create server-side components that directly interact with a light client application. Enterprise JavaBeans is used to create processing components of a J2EE application and web services.

The Complete Reference



Chapter 8

HTML, XML, and XHTML

This chapter is the first of five chapters that discuss technologies that form the foundation of a J2EE application. The concepts of these technologies were introduced in the first part of this book and include Java servlets, JavaServer Pages (JSP), Enterprise Java Beans (EJB), and the Hypertext Markup Language (HTML), the Extensible Markup Language (XML), and the Extensible Hypertext Markup Language (XHTML).

A user interacts with a J2EE application through a user interface that is created using HTML. The user interface relays a user's request to components built using Java servlets and/or JSP. Sometimes these components need web services built using EJB to fulfill the user's request, which typically results in the Java servlet or JSP dynamically generating another user interface page created using HTML.

XML is a language used to create documents that are shared within systems of a corporation and within an industry. There is an increasing requirement for components built using Java servlets, JSP, and EJB to manipulate data organized into an XML document such as those used to generate an online catalog.

Therefore, we begin the discussion of the J2EE foundation technologies with a look at HTML, XML, and XHTML. XHTML is the reformulation of HTML. This knowledge, along with techniques illustrated in the other chapters of this part, enables you to create J2EE components that dynamically generate a user interface and manipulate the latest form of electronic documents.

HTML

It is a fair assumption that you know what a web page is and how to navigate hyperlinks that appear on the page. And it might also be safe to assume that you already know something about HTML, the language used to create a web page. Knowledge of HTML is important before you create J2EE components using JSP because, as you'll learn in later chapters, the majority of JSP is written using HTML.

Skip this section and proceed to the XML section if you are fluent in HTML. However, continue reading if HTML is new to you or if you need to brush up on your HTML skills. The purpose of this section is not to thoroughly cover HTML, because there just isn't enough room in this book to do so.

The purpose of this section is to introduce you to some general concepts of HTML and focus on two features of the language that are commonly encountered when using HTML in a JSP to dynamically build a web page. These features are HTML forms and HTML tables.

HTML forms are commonly used in a JSP to generate a web page that collects information from a user, which is then submitted to a JSP or Java servlet for processing. For example, an HTML form is used as a data entry form. Data received by the JSP or Java servlet from the form is typically forwarded to a component built using an EBJ that interfaces with a DBMS.

An HTML table is a spreadsheet-like grid that is produced dynamically by the JSP or Java servlet to display data that is retrieved from the DBMS. HTML code in the JSP or Java servlet creates each table component and places labels and data within cells of

the table before transmitting the HTML web page to the browser for display. Tables are also used for other purposes, such as to define a page layout.

The user interface for your J2EE application is created by either writing HTML code yourself or using an HTML development tool to do so, then modifying the HTML code produced by the HTML development tool as needed. This section focuses on HTML topics that you'll find useful when creating dynamically generated web pages. You should read *HTML: The Complete Reference* for a more thorough discussion of HTML.

Basic Concepts of HTML

As the name describes, HTML is a markup language that is used to define how text should be displayed in a browser or other software that is capable of interpreting HTML, such as some word processors. Tags are used to specify how text is to be displayed. Consider a tag as a command to the browser.

Most tags come in pairs, called the opening tag and the closing tag. An opening tag consists of the tag name contained within angled brackets. A closing tag is similar in appearance to the opening tag except the closing tag has a forward slash (/) before the tag name within the angled brackets. Text that appears between the opening and closing tag is displayed according to the definition of the tag. For example, the following tag pair causes text between the tags to appear in bold. The <B> is the HTML tag for bold.

```
<B>First Name:</B>
```

Many tags offer a variety of options that refine the display of text that appears between the tags. One or more attributes define these options each of which is assigned a value. Attributes and their values are placed within the opening tag before the closed angle bracket.

The next code segment illustrates how to use attributes and values when setting the FONT tag for text. The opening FONT tag contains the attribute SIZE, which is assigned the value 7. Text that appears between the opening and closing tags is displayed in type size 7. Values can appear within quotation marks, although quotation marks are optional if the value contains only letters, digits, a hyphen, or a period. However, all formatting is typically defined in a style sheet (see *HTML: The Complete Reference*).

```
<FONT SIZE="7">First Name:</FONT>
```

Placing one pair of tags within another pair can nest pairs of tags. However, it is critical that the order of nesting be followed. That is, the inner nested pair must have an open and closed tag within the outer pair of tags. This is shown in the next code segment where the <B> tag pair is placed within the <FONT> tag pair:

```
<FONT SIZE="7"><B>First Name: </B></FONT>
```

HTML consists of many tags. Practically all the tags, attributes, and values that you'll need to construct a user interface for your J2EE application are automatically placed into the HTML page by the development tool. However, you'll still find it necessary to edit HTML code directly to assure that the web page works with all browsers.

The following sections discuss the tags that you'll frequently use for web pages that you'll generate using JSP or Java servlets. Table 8-1 at the end of the HTML section contains a listing of commonly used tags. You can refer to these if you ever need to read HTML code, although typically you'll let the development tool read the code for you.

The Skeleton of a Web Page

All tags on a web page are contained within the HTML tag, although some browsers will interpret a web page even if the HTML tag is omitted. A web page is divided into two sections within the HTML tag. These are the HEAD and the BODY.

The HEAD contains tags that affect the entire web page, such as the TITLE tag, FONT, and background color for the page. Text that appears in the TITLE tag is displayed on the title bar of the browser window or centered at the top of the screen, depending on the browser. The TITLE is also used in search indexes, browser history lists, and bookmarks. The BODY tag contains other tags that define the web page.

Another option used component is a web page is the document type tag that appears above the HTML tag. The document type tag defines the web page as being an HTML document type. Listing 8-1 contains the typical skeleton for a web page. A web page development tool automatically creates the skeleton for web pages created using the tool. You'll have to write the web page skeleton into your JSP or Java servlets that dynamically generate a web page. You'll see how this is done in later chapters of this book.

Listing 8-1 The skeleton of a web page.

```
<!DOCTYPE HTML PUBLC "-//W3C//DTD HTML 4.0 TRANSITIONAL//EN"
<HTML>
<HEAD><TITLE>My J2EE</TITLE></HEAD>
<BODY>
</BODY>
</HTML>
```

Creating a Form

An HTML form is used to collect data from a user, as discussed previously in this chapter. There are two components to an HTML form. These are the shell of the form and the processing of the form. The shell of the form consists of form elements such as fields, labels, and buttons that are used to enter information into the form. The form is placed within the BODY tag of the web page.

Processing is conducted by a server-side component, which is built using JSP or servlet technology in the case of a J2EE application. The next few sections focus on the

shell of the form, while processing is discussed in detail in the Chapter 11 JSP and Chapter 10 servlet chapters.

The FORM tag is used to create an HTML form and is placed within the <BODY> tag of the web page. Tags that create the contents of the form—such as fields, labels, and buttons—are placed within the form tag, as is seen in later sections of this chapter.

The FORM tag has two attributes. These are METHOD and ACTION. METHOD is an optional attribute. If it's not specified, GET is the default value of METHOD. The value of the METHOD attribute describes how data collected in the form is sent to the server. The value of the ACTION attribute defines the server-side component that is called to process the form. This is typically a JSP or servlet.

One of two values can be assigned to the METHOD attribute. These are GET and POST. The GET value tells the browser to append field names and data collected in the form to the URL of the server-side component that will process the form. The field names and data are automatically assigned to the QUERY_STRING environment variable on the server where the processing component breaks down the appended data collection into individual data for processing. The GET method limits the amount of data that can be appended to the URL, which means that some data might be truncated.

The POST value transmits a data file that contains data collected in the form to the server. The processing component receives the file from standard input and proceeds to parse the data. There isn't any size limitation to the data file.

Listing 8-2 contains the shell of an HTML form. In this example, the POST method is used to send data from the form to the processing component. The processing component is called form1.jsp and is assigned to the ACTION attribute. Of course, you can expand the name of the processing component to include the full URL.

You can use any combination of fields, labels, and buttons in a form to collect data from a user. The next several sections discuss each of these in detail.

Listing 8-2 The shell of an HTML form.

```
<FORM METHOD=POST ACTION="form1.jsp">
</FORM
```

Creating Text Boxes

A text box is an element in a form where the user enters text data. Using the INPUT tag creates a text box. The INPUT tag is a single tag and doesn't have a close tag. There are two attributes that can be used to create a text box. These are TYPE and NAME.

The value of the TYPE attribute is "text." This tells the browser that the field is a text box. Other fields are also created using the INPUT tag, but the value of their TYPE attribute defines the field as other than a text box, which you'll see later in this chapter.

The value of the NAME attribute is the field name. The field name is used to identify data that is entered into the field when the data is sent to the processing component. The processing component then extracts the data from the field name.

Let's say that a form contains two text boxes whose NAME attribute values are fname and lname and the user entered **Jim Keogh** into each respective text box.

Furthermore, let's assume that the GET method is used to send data to the form1.jsp component. Here's how the query string, which is the URL and appended data, appears:

```
form1.jsp?fname=Jim&lname=Keogh
```

Notice that a question mark separates the URL from the data. Each data element is paired with the field name and an ampersand separates each field name/data element pair. Just the field name and the data element are used in the file transmitted if the POST method is employed.

Two optional attributes are commonly used with a text box. These are SIZE and MAXLENGTH. The value of the SIZE attribute is the number of characters wide the text box appears on the form. The value of the MAXLENGTH attribute is the maximum number of characters that a user can enter into the text box. Listing 8-3 creates a paragraph on the form (<P>) that contains the text First Name: followed by a text box. The text First Name: is the label for the text box. The closing paragraph tag (</P>) is optional.

Listing 8-3 Creating a text box.

```
<P> First Name: <INPUT TYPE="text" NAME="fname" SIZE="20" MAXLENGTH="20"> </P>
```

Creating a Password Box

A password box is a kind of text box in that a user is able to enter free-form text into the text box. However, characters entered into a password box are replaced on the screen with either bullets or asterisks, although characters that are actually entered by the user are assigned to the field name defined in the NAME attribute. Listing 8-4 illustrates how to construct a password box.

Caution *Characters entered into a password box appear in the query string and file when the data from the form is transmitted to the processing component. That is to say, data that is associated with a password box is not encrypted.*

Listing 8-4 Creating a password box.

```
<P>
Password: <INPUT TYPE="password" NAME="password" SIZE=20 MAXLENGTH="20">
</P>
```

Creating a Larger Text Area

Sometimes a J2EE application requires a user to enter multiple lines of free-formed text, such as when asking the user to make a comment. A TEXTAREA is used to capture large amounts of free-formed text. A TEXTAREA is also used to display a block of text outside of a form.

A TEXTAREA is created by using the TEXTAREA tag, which requires both an opening and closing tag. The TEXTAREA tag may use the NAME attribute, whose value is the field name for the data collected in the TEXTAREA if the TEXTAREA is used in a form.

There are three other attributes that are commonly used with the TEXTAREA tag. These are ROWS, COLS, and WRAP. The WRAP attribute may not work with all browsers. The value of the ROWS attribute defines the height of the TEXTAREA on the form. The default value is four rows. The value of the COLS attribute defines the width of the TEXTAREA in characters. The default value might be 40 characters, depending on the browser. The WRAP attribute doesn't contain a value and tells the browser to wrap text around to the next line as the user enters data into the TEXTAREA. Some browsers automatically wrap text in the TEXTAREA. Scroll bars appear whenever the user enters more text than what can appear within the TEXTAREA.

Listing 8-5 illustrates how to create a TEXTAREA. In this example, the TEXTAREA is five rows high. The user can enter more than five rows, but only five rows are shown in the TEXTAREA. The user needs to use scroll bars to view text that is contained beyond the fifth row.

Listing 8-5 Creating a TEXTAREA.

```
<TEXTAREA NAME="comments" ROWS="5" COLS="20" WRAP> </TEXTAREA>
```

Creating Radio Buttons

A radio button is one of a set of radio buttons that are used by a person to choose one of a set of options. Each radio button is associated with an option. Whenever a radio button within the set is selected by the user, all the other radio buttons within the set are unselected.

An INPUT tag (as discussed earlier to create a text box) is also used to create a radio button. There are three attributes that are used for a radio button. These are TYPE, NAME, and VALUE. The value of the TYPE attribute is "radio" to tell the browser that a radio button is to be displayed. The value of the NAME attribute is the name given to the set of radio buttons. Each radio button in the set must have the same name because it is the value of the NAME attribute that defines the set to which a radio button belongs. The value of the VALUE attribute is the data that is paired with the value of the NAME attribute when the data is sent to the processing component.

Each radio button is defined with its own INPUT tag and related attributes. This is illustrated in Listing 8-6. In this example, there is one set of radio buttons that consists of two radio buttons. You can select one of the radio buttons within the set as the default selection by using the CHECKED attribute, as shown in Listing 8-6. The CHECKED attribute does not have an associated value. Text that appears outside the INPUT tags is displayed on the form.

Caution *The value of the NAME attribute may be treated as case sensitive by some browsers.*

Listing 8-6 Creating radio buttons.

```
<H1>Please choose one:</H1>
<INPUT TYPE="radio" NAME="gender"VALUE="female" CHECKED>Female
<INPUT TYPE="radio" NAME="gender "VALUE="male">Male
```

Creating Check Boxes

A check box is similar in concept to a radio button in that both are used to display and record options that are available to a user. Check boxes are also grouped into sets and are created using the INPUT tag and the TYPE, NAME, VALUE, and CHECKED attributes that are discussed in the previous section.

However, check boxes differ from radio buttons in that the value of the TYPE attribute is "checkbox" and a user can select multiple check boxes within the same group. Likewise, multiple check boxes can be a default selection by using the CHECKED attribute.

Listing 8-7 illustrates how to create a check box. In this example, the user can select one or more ways he or she commutes to work. The default selection is Car. All the values selected by the user are associated with the field name "commute" when the data is sent to the processing component. Let's say that the user selected Car and Train. The browser sends commute=car&commute=train.

Listing 8-7 Creating check boxes.

```
<H2>How do you commute to work?</H2>
<INPUT TYPE="checkbox" NAME="commute"VALUE="car" CHECKED>Car
<INPUT TYPE="checkbox" NAME=" commute "VALUE="train" >Train
<INPUT TYPE="checkbox" NAME=" commute "VALUE="plane" >Plane
<INPUT TYPE="checkbox" NAME=" commute "VALUE="boat" >Plane
```

Creating Menus

A menu is a component of a form that contains a drop-down list of options from which the user can choose. A menu is created using a combination of the SELECT tag, which requires an opening and closing tag, and the OPTION tag, which requires only an opening tag.

The SELECT tag should contain a NAME attribute whose value is the field name. The SELECT tag also has the SIZE attribute, which is the number of visible rows of the menu in characters. The OPTION tag may have one attribute, which is the VALUE. The value of the VALUE attribute is the data that is associated with the option. If the VALUE tag is not specified in the OPTION tag, the default text defined in the OPTION tag becomes the text of the option. Text that appears on the form is placed to the right of the OPTION tag. Any OPTION can be selected as default by using the SELECTED attribute in the OPTION tag. The SELECTED attribute does not have a value.

Listing 8-8 illustrates how to create a menu. In this example, the menu is called "commute" and has four options with the "car" option selected as default. After the user selects an option and submits the form, the value of selected options are associated with the field name "commute" and sent to the processing component.

Listing 8-8 Creating a menu.

```
<H2>How do you commute to work?</H2>
<SELECT NAME="commute" SIZE="5" >
<OPTION SELECTED VALUE="car">Car
```

```
<OPTION VALUE="train">Train
<OPTION VALUE="plane">Plane
<OPTION VALUE="boat">Boat
</SELECT>
```

Options that appear in a menu can be grouped using the OPTGROUP tag. The OPTGROUP tag requires an opening and closing tag and LABEL attributes. The value of the LABEL attribute is the text that appears in the menu to describe the group of options. Listing 8-9 shows how to create groupings within a menu. Groupings are also referred to as submenus and require that the attribute MULTIPLE be placed in the SELECT tag. There isn't a value for the MULTIPLE attribute. The MULTIPLE attribute enables the user to select more than one item using the SHIFT or CTRL key, but may not work with all browsers.

Listing 8-9 Creating menu groupings.

```
<H2>How do you commute to work?</H2>
<SELECT NAME="commute" SIZE="5" MULTIPLE>
  <OPTGROUP LABEL="Traditional">
    <OPTION SELECTED VALUE="car">Car
    <OPTION VALUE="train">Train
  </OPTGROUP>
  <OPTGROUP LABEL="Non-Traditional">
    <OPTION VALUE="plane">Plane
    <OPTION VALUE="boat">Boat
  </OPTGROUP>
</SELECT>
```

Creating Hidden Elements

As discussed in Part I, a common problem with a J2EE application is to retain data during a session. There are a number of techniques that are available to do this, which will be discussed in the Chapter 12 Enterprise Java Beans chapter. One of those techniques is to use a hidden field in a form that is generated dynamically by the processing component, although this method is not as efficient as other methods discussed later in this book.

A hidden field is very similar to a text field, discussed previously in this chapter, except the field doesn't appear on the form. A hidden field is created using the INPUT tag and requires three attributes. These are TYPE, NAME, and VALUE. The value of the TYPE attribute is "hidden" and the value of the NAME attribute is the field name. The value of the VALUE attribute is the data.

Although a hidden field isn't displayed on the form, the field name and value of the VALUE attribute are sent to the processing component along with other fields on the form.

There is no difference between data from a hidden field and data from visible fields on the form. Listing 8-10 illustrates how to create a hidden field.

Listing 8-10 Creating a hidden field.

```
<INPUT TYPE="hidden" NAME="userID" VALUE="123">
```

Creating the Submit Button

A Submit button, when selected by a user, is the button that causes the browser to extract the field name and data from the form and send them to the processing component, as discussed previously in this chapter.

A Submit button is created using the INPUT tag and requires the TYPE attribute whose value is "submit." The VALUE attribute is optional. The value of the VALUE attribute is the text that appears on the Submit button. The default value is Submit Query although the actual default is browser dependent. Listing 8-11 illustrates how to create the Submit button.

Listing 8-11 Creating a Submit button.

```
<INPUT TYPE="submit" VALUE="Submit Form">
```

Using the Reset Button

A form typically has a Reset button, which resets fields to their values when the form was originally displayed on the web page. This means that data entered by the user is removed from all the fields if the Reset button is selected, and fields return to their default values.

A Reset button is created similar to how a Submit button is created, using the INPUT tag as illustrated in Listing 8-12. However, the value of the TYPE attribute is "reset" and the value of the VALUE tag can be any text that is to appear on the button. The default value for the VALUE tag is Reset.

Listing 8-12 Creating a Reset button.

```
<INPUT TYPE="reset" VALUE="Reset Form">
```

Organizing Components on the Form

Components can be organized on the form into groups where the group is identified by a label. This is a clever way to prevent component clutter on the form. The FIELDSET tag is used to create a group on the form. The FIELDSET group requires an open and close tag and a LEGEND open and close tag.

Fields, labels, and buttons that are in the same group are placed within the FIELDSET open and close tag. The LEGEND tag is also placed within the FIELDSET tag and is used to name the group. Listing 8-13 illustrates how to organize components on a form.

Listing 8-13 Creating groups of components on a form.

```
<FIELDSET>
  <LEGEND>Customer Information</LEGEND>
  First Name:<INPUT TYPE="text" NAME="firstname" SIZE=15>
  Last Name:<INPUT TYPE="text" NAME="lastname" SIZE=15>
</FIELDSET>
```

Setting the Tab Order

Typically a person moves from field to field within a form by pressing the TAB key. The order in which fields are selected using the TAB key is determined either by the order in which the fields are placed on the form or by setting the TABINDEX attribute value.

The TABINDEX attribute is used for every field on the form and tells the browser where to move the cursor on the form when the user selects the TAB key. The value of the TABINDEX is a number that corresponds to the order in which the field is to be selected. The first field to be selected is number 1. The second field is 2, and so on.

Listing 8-14 illustrates how to set the TABINDEX value for a field. Each field should also contain an ID attribute whose value uniquely identifies the field on the form. In this example, the value of the NAME attribute is also used as the value of the ID attribute, but these fields can have different values.

Listing 8-14 Setting the TABINDEX value for a field.

```
<INPUT TYPE="text" NAME="firstname" ID="firstname" SIZE=20 TABINDEX=2>
```

Keeping Elements from Being Changed

There are occasions when the processing component places values in a field in a form that it dynamically generates. For example, this is common when a customer is completing a multipage form and each page appears on a different web page. The design might require that information such as the customer name be entered on the first page by the customer, then carried forward to subsequent pages by the processing component.

Typically, information carried forward from the previous page appears as a value assigned to the VALUE attribute of a field by the processing component. While this technique displays previously entered data, it causes a problem. The user might alter the information in the field. For example, the customer might change the value of the customer name fields that appear on subsequent pages of the form.

You can prevent the user from changing values in a field by using the READONLY attribute with the field. The READONLY attribute is illustrated in Listing 8-15.

Caution *A user can save the HTML form locally, modify the value by changing the code, then submit the form.*

Listing 8-15 Preventing a user from changing a value of a field on the form.

```
<INPUT TYPE="text" NAME="firstname" VALUE="Bob" READONLY>
```

Tables

A feature common to many J2EE applications is for a user to recall information that is stored in a database. Typically, a web page is displayed that contains a form, which is used to collect search criteria from a user. The search criteria is sent to the processing component by the browser when the user selects the Submit button, as described in the previous section.

The processing component parses the search criteria from the form and passes this to a web services component such as a EJB, which retrieves the data from the DBMS using techniques described in Part II of this book.

Data is then returned through the web services component to the processing component where the data is properly formatted into a web page using HTML. Many times the data is presented in tabular format and requires the processing component to dynamically create the table using HTML.

Previous sections of this chapter illustrate how to create a form that, among other things, captures search criteria. Part II of this book showed you how to use Java data objects to interact with the DBMS. The next few sections show how to dynamically build a table in HTML. You'll use this method in conjunction with JSP or Java servlets, which are discussed later in this book.

Creating a Table

Using the TABLE tag, the TR tag, the TH tag, and the TD tag creates a table. The TABLE tag requires an open tag and close tag to define the body of the table. Within the TABLE tag are the other tags that create a grid within the table.

The TR tag defines a row and requires an opening and closing tag. The TH tag and the TD tag define a column. Both of these tags also require an opening and closing tag. The TH tag defines a cell header and the TD tag defines a data cell. Headings and data that appear in a cell of the table are placed between the opening and closing TH tag and TD tag. The closing tags for TR, TD, and TH are optional.

Listing 8-16 illustrates how to create a table. In this example, the table contains two rows and three columns. The first row contains column headers that identify the contents of the column. The second row contains data.

Listing 8-16 Creating a table.

```
<TABLE>
 <TR>
   <TH>Sales For Store 1</TH>
   <TH>Sales For Store 2</TH>
   <TH>Sales For Store 3</TH>
 </TR>
 <TR>
   <TD>$500</TD>
   <TD>$300</TD>
   <TD>$200</TD>
 </TR>
</TABLE>
```

Adding a Caption to the Table

The caption of a table describes the kind of information that the table contains. Using the CAPTION tag placed within the TABLE tag above the first row creates a caption. The CAPTION tag requires an open tag and close tag. Text that appears between the open tag and close tag is the caption that appears above the table.

The caption can be positioned above, below, to the right, or to the left of the table by using the ALIGN attribute. The value of the ALIGN attribute is top, bottom, left, or right. The top value is the default value, which places the caption at the top center of the table. However, ALIGN is typically defined in a style sheet.

Keep in mind that the ALIGN attribute conveys to the browser the desired location of the caption. The browser determines the absolute position of the caption on the web page. Listing 8-17 illustrates how to create a caption for a table.

Listing 8-17 Creating a caption for the table.

```
<CAPTION ALIGN=bottom> Store Sales</CAPTION>
```

Adding a Border to the Table

A border can be placed around a table by using the BORDER attribute. The BORDER attribute tells the browser to use a border around the table. You can specify the thickness of the boarder in pixels by assigning the thickness to the BORDER attribute. For example, BORDER="5" creates a border that is 5 pixels thick. The browser uses the default border thickness if you don't specify a pixel value. Listing 8-18 illustrates how to create a border for a table.

Listing 8-18 Creating a border for a table.

```
<TABLE BORDER=6">
```

Choosing Border Style

By default, the BORDER tag causes a frame to appear around the entire table and around each cell within the table. You can control where the border appears by using the FRAME attribute and the RULES attribute.

The FRAME attribute is used to define where the frame appears around the sides of the board. The values of the FRAME attribute are described here:

- **void** No external frame.
- **above** A single frame on top.
- **below** A single frame on bottom.
- **bsides** A frame on both the top and bottom sides.
- **vsides** A frame on both the right and left sides.
- **rbs** A single frame on the right side.
- **lbs** A single frame on the left side.
- **box or border** A frame on all sides (default).

The values of the RULES attribute define the style of frame used within the table to highlight the cells. The values of the RULES attribute are described here. Listing 8-19 illustrates how to create a border and apply a frame style outside and inside the table.

- **none** No internal rules.
- **rows** Horizontal rules between each row.
- **cols** Vertical rules between each column.
- **groups** Rules between column groups and horizontal sections as defined by column groups.
- **all** Rules between each row and columns in the table (default).

Listing 8-19 Creating a border style outside and inside the table.

```
<TABLE BORDER FRAME=vsides RULES=cols >
```

Merging Two Cells

Sometimes the design of a table requires that two cells merge. Although many word processors and spreadsheets refer to this as merging cells, technically one cell spans the space of two cells. This is commonly used when a column header references more than one column.

You can span a cell across multiple columns by using the COLSPAN attribute for either the TH tag or the TD tag. The value of the COLSPAN attribute is a number that represents the number of columns over which the cell spans. Listing 8-20 illustrates this technique.

Listing 8-20 Spanning columns.

```
<TH COLSPAN=2> Store 1 </TH>
```

Spanning a Cell Across Two or More Rows

A cell can span multiple adjacent rows by using the ROWSPAN attribute with either the TH tag or the TD tag. The value of the ROWSPAN attribute is a number that represents the number of rows over which the cell spans. Listing 8-21 illustrates how to span rows.

Listing 8-21 Spanning rows.

```
<TH ROWSPAN=2> Store 1 </TH>
```

Changing a Table's Width and Height

The size of a table on the web page can be altered from the default size by using the WIDTH and HEIGHT attributes of the TABLE tag. The value of both attributes can be a number that reflects the absolute size in pixels or a percentage. A percentage defines

the width and height of the table as a percentage of the full window size so that the table remains proportional to the browser window. Listing 8-22 illustrates how to control the size of the table.

Listing 8-22 Changing the size of a table.

```
<TABLE WIDTH=50% HEIGHT=90%>
```

Changing the Size of a Cell

The WIDTH and HEIGHT attributes are also used to change the size of a cell using the same metric as the attributes used for a table. However, these attributes are used with the TH tag or the TD tag rather than the TABLE tag, as is illustrated in Listing 8-23.

Listing 8-23 Changing the size of a cell.

```
<TD WIDTH=200 HEIGHT=100>
```

Aligning the Contents of Cells

The contents within a cell can be aligned horizontally and vertically by using the ALIGN and VALIGN attributes in either the TH tag or TD tag. The value of the ALIGN attribute can be left, center, right, or justify, which causes the contents to be flush with both sides of the cell. The default value of the ALIGN attribute in the TH tag is center and with the TD tag is left.

The value of the VALIGN can be top, middle, bottom, or baseline. The baseline value aligns the context of the cell with the first line of text that appears in the cell or is the same as value top if no text is in the cell. The default value of the VALIGN tag is middle. Listing 8-24 illustrates how to align the contents of a cell.

Listing 8-24 Aligning the contents of a cell.

```
<TD VALIGN=top ALIGN=left>
```

Changing the Background Color of a Cell

The background color of a cell can be changed from the default color chosen by the browser by using the BGCOLOR attribute with the TH tag or the TD tag. The value of the BGCOLOR attribute is the name of the background color, as is illustrated in Listing 8-25. However, background color is typically defined in a style sheet.

Listing 8-25 Changing the background color of a cell.

```
<TD BGCOLOR=yellow>
```

Spacing and Padding the Cells

The size of a cell can be fine-tuned by adjusting the spacing and padding of the cell. Cell spacing adjusts the space between cells and is set by using the CELLSPACING

attribute for the TABLE tag. The value of the CELLSPACING attribute is a number that defines the number of pixels that appear between each cell. The default value for cell spacing is 2.

Cell padding adjusts the spacing between the contents of the cell and the cell wall. The CELLPADDING attribute is used with the TABLE tag to set the cell padding to a desired number of pixels. The default value for cell padding is 1. The BORDER attribute, as discussed previously in this chapter, can also be used with the TABLE tag. Listing 8-26 illustrates how to set the spacing and padding of cells in a table.

Listing 8-26 Setting the spacing and padding of cells.

```
<TABLE BORDER CELLSPACING=15 CELLPADDING=15>
```

Controlling Line Breaks in a Cell

A browser typically divides lines of text within a cell, depending on the size of the column and row. This means there is always the risk that data within a cell might appear on multiple lines if there isn't room to place the data on one line. The browser can be forced to keep data on one line by using the NOWRAP attribute with either the TH tag or the TD tag. The NOWRAP attribute doesn't require a value. Listing 8-27 illustrates how to use the NOWRAP attribute, and Table 8-1 contains a listing of commonly used tags:

Listing 8-27 Setting the NOWRAP attribute.

```
<TD NOWRAP>
```

Tag	Definition
A	Anchor
APPLET	Java applet
AREA	Client-side image map area
B	Bold text style
BASE	Document base URI
BASEFONT	Base font size
BLOCKQUOTE	Long quotation
BODY	Document body

Table 8-1. *HTML Tags*

Tag	Definition
BR	Forced line break
BUTTON	Push button
CAPTION	Table caption
CENTER	Shorthand for DIV align=center
CITE	Citation
CODE	Computer code fragment
COL	Table column
COLGROUP	Table column group
DD	Definition description
DEL	Deleted text
DFN	Instance definition
DIR	Directory list
DIV	Generic language/style container
DL	Definition list
DT	Definition term
EM	Emphasis
FIELDSET	Form control group
FONT	Local change to font
FORM	Interactive form
FRAME	Subwindow
FRAMESET	Window subdivision
H1	Heading
H2	Heading
H3	Heading
H4	Heading
H5	Heading
H6	Heading
HEAD	Document head
HR	Horizontal rule

Table 8-1. *HTML Tags* (continued)

Tag	Definition
HTML	Document root element
I	Italic text style
IFRAME	Inline subwindow
IMG	Embedded image
INPUT	Form control
ISINDEX	Single line prompt
KBD	Text to be entered by the user
LABEL	Form field label text
LEGEND	Fieldset legend
LI	List item
LINK	A media-independent link
MAP	Client-side image map
MENU	Menu list
OL	Ordered list
OPTGROUP	Option group
OPTION	Selectable choice
P	Paragraph
PARAM	Named property value
PRE	Preformatted text
Q	Short inline quotation
S	Strikethrough text style
SCRIPT	Script statements
SELECT	Option selector
SMALL	Small text style
SPAN	Generic language/style container
STRIKE	Strikethrough text
STRONG	Strong emphasis
SUB	Subscript
SUP	Superscript

Table 8-1. *HTML Tags* (continued)

Tag	Definition
TABLE	Create a table
TBODY	Table body
TD	Table data cell
TEXTAREA	Multiline text field
TFOOT	Table footer
TH	Table header cell
THEAD	Table header
TITLE	Document title
TR	Table row
TT	Teletype or monospaced text style
U	Underlined text style
UL	Unordered list

Table 8-1. *HTML Tags* (continued)

XML

The Extensible Markup Language (XML) is a language used to create other languages that define components of a document. You probably don't realize as you read this book that it is composed of elements that give the text the look and feel of a book.

These elements include the title page, table of contents, the index, and chapters, each of which conforms to a specific style. Each element is considered data, similar to names and addresses that you might save to a table of a database.

In a relational database, data is described using metadata such as a column name, column data type, and column size. The language that you create using XML is used to create metadata that describes elements in a document (that is, book).

XML is similar in concept to the Hypertext Markup Language (HTML) in that an element is defined using tags and attributes. Tags and attributes are comparable to the metadata of a relational database.

Why Use XML?

There are a number of reasons why XML has been adopted as a standard in companies that need to manage documents:

- **Simplifies data exchange** XML provides a standard data definition for information that is commonly shared electronically amongst members of an

industry. For example, a company might want to electronically extract from a vendor's product catalog the product number and unit price of selected items. The selection of product numbers and unit prices is simplified if the vendor uses an industry-accepted set of XML tags to describe product number and unit price elements of the catalog.

- **Organizes a document** XML enables a company to easily identify components of a document that can be recalled by using XML tags. For example, online booksellers typically receive selected information about a book from a publisher, which is displayed on the catalog page of the bookseller's web site. This includes the table of contents, author's biography, back cover copy, the preface, and maybe a sample chapter. The publisher can easily provide an online bookseller with these components because the publisher organizes components of a book using XML tags.
- **Reusable document elements** A company can create a new document from elements of an existing document by referencing the XML tags of those elements. Let's say that a company wants to create a holiday sales catalog. Product information can be easily assembled into the company's special sales catalog by referencing selected XML tags from their general sales catalog.
- **Standardizing display of information** Each element of a document that is created using an XML-based language includes rules for displaying the element. For example, the display format of pricing information of a product is associated with the XML tag that describes pricing information. Therefore, XML provides consistency in display of information.
- **Browser readable** An XML document can be displayed by Internet Explorer 5 (for both Macintosh and Windows) and Netscape 6 browsers.

The XML Flow

There is a two-step process used to create an XML-based document. The first step is to create an XML Schema, which is a document you create that contains the definition of the language that describes your document.

The language consists of a definition of a customized set of tags. These tags are very similar to HTML tags, except you create tag names and attributes of these tags. You'll see how to do this later in this chapter.

You can think of the XML Schema as a dictionary that defines tags used to describe elements of your document. The XML Schema is referenced whenever the document is accessed, such as when a vendor shares catalog information electronically with a client.

The second step is for you to create the document using elements defined in the XML Schema. For example, you might define a tag called RetailPrice in the XML Schema. You'll enclose the retail price element of your document with the retail price tag as shown here:

```
<RetailPrice>$15.95</RetailPrice>
```

There is a growing trend to standardize XML Schemas throughout a company and within an industry, to facilitate an efficient method of exchanging information. This is especially prevalent in industries where documents tend to consist of the same elements.

The book publishing industry is a good example. Practically every book has the same basic set of elements, as described in the previous section. Therefore, the book publishing industry has standardized on a XML Schema that contains tags to describe every possible element that could appear in a book.

In an ideal world, this means that all companies involved in the manufacturing and selling of a book properly interpret elements of an XML-based book by referencing the industry adopted XML Schema. In the real world, it is difficult to create an XML Schema standard that is agreeable to everyone in the industry and is broad enough to encompass all the nuances that could exist in all books, but various industry groups are attempting to do just that.

XML Parsers

An XML parser is a program that transforms an XML document into a structure that allows programmatic access to a document's content. During the transformation process, the XML parser reads the XML document and examines XML syntax, then reports any syntax errors that are encountered. Common syntax errors occur because of unbalanced start and end tags, improper use of quotations for attribute values, and improper use of capitalization. XML is case sensitive.

There are two parses used to access a document's content. These are using the Document Object Model (DOM) and the Simple API for XML (SAX), which is used to by Java, C, and other programming languages to programmatically use the content of an XML document.

XML parsers that use DOM generate a tree structure of the XML document's content, which is stored in memory. In contrast, a SAX parser generates events when components of an XML document are encountered such as a tag and content, which sends the related content from the XML document to the program that is accessing the XML document.

DOM parsers enable the contents of the XML document to be accessed quickly and enable the XML document to be modified by adding and removing nodes. However, SAX parsers, without the memory overhead, provide greater performance than DOM parsers provide. As a general rule, a DOM parser is used whenever an XML document needs to be modified; otherwise, a SAX parser is preferred.

An XML parser is included in Internet Explorer 5 and Netscape 6 browsers. However, these browsers display both tags and content of an XML document, rather than content defined by those tags. This is because, unlike HTML tags, the browser is unable to translate XML tags to display the content. However, a browser can translate the XML document into an HTML document if an Extensible Style Sheet is defined, which I'll show you how to do later in this chapter.

There are many XML parsers available for free on the Internet. These are stand-alone XML parsers (that is, not related to Internet Explorer 5 and Netscape 6). You will find

an XML parser that can be used to access an XML document with Java (JAXP) on the Sun Microsystems's web site (www.sun.com).

Browsers and XML

The reality is that a browser cannot read an XML-based document. This is because a browser reads HTML documents that contain HTML tags and not XML tags. XML tags are literally displayed by a browser as if tags are part of the text that is displayed on the screen.

However, a browser can convert an XML-based document to an HTML document by using an XML style sheet referred to as Extensible Stylesheet Language Transformation (XSLT). An XSLT contains rules a browser uses to translate each element of the XML-based document into tags on an HTML page. You can create an XSLT style sheet. I'll show you how to do this later in this chapter.

Why use XML when a browser must translate the XML document into an HTML document? Why not simply use HTML? Creating a document using HTML makes the document browser readable, but doesn't accurately describe elements of the document. For example, you might place the retail price of an item in an HTML paragraph such as

```
<P>$15.95</P>
```

However, the paragraph tag doesn't describe the text as being the retail price. It simply tells the browser that text appearing between the paragraph tags should be displayed in a paragraph.

In contrast, the RetailPrice tag, as described in the previous section, defines the text as a retail price. This enables other programs that read the document to properly process the text as a retail price. In addition to using the RetailPrice tag, you might create an XSLT for the document that further states that text displayed within the RetailPrice tag should be displayed in a paragraph.

The Design of an XML Document

An XML-based document is defined in a file you created in a text editor and consists of XML tags that are nested within each other, similar to the structure of an HTML web page. The outermost element of the XML document is called the root element, within which other elements are placed that represent components of the document.

Although a book is used in this section to illustrate an XML document, the same design concept applies to any XML document—only the names of the elements are different than the elements of a book.

The root element of a book contains all other elements of the book. Typically, the tag name describes the type of information that is contained in the element, although you are free to use any name for a tag that conforms to the XML naming rule.

Names must begin with a letter, underscore, or colon, and names may contain letters, digits, underscores, hyphens, periods, and colons. XML is case sensitive.

The root element tag in this example is called book, as shown next. Each element must have an open tag and a close tag, which is similar to HTML. That is, the name of an open and close tag is enclosed within an angle bracket and the second character of a closed tag must be a forward slash.

```
<book>
</book>
```

Tags used to identify other elements follow the same naming convention as the root element, except each element has a unique tag name as defined in the XML Schema. Let's assume for now that the XML Schema for the book is already defined.

You can use the same element multiple times within the document, as is illustrated in Listing 8-28 where the <Chapter> tag is used to identify two chapters in the book. The <Chapter> tag is defined once in the XML Schema. This is similar to defining a word in a dictionary, then using the word multiple times in a book.

Listing 8-28 contains other elements of an XML-based book. Text for each element of the book is placed within the appropriate open and close tag set.

Listing 8-28 Elements of an XML-based book.

```
<book>
  <TitlePage>
  </TitlePage>
  <CopyWritePage>
  </CopyWritePage>
  <TableOfContents>
  </TableOfContents>
  <Preface>
  </Preface>
  <Chapter>
  </Chapter>
  <Chapter>
  </Chapter>
</book>
```

Nesting Elements

You've seen in the previous section how the root element contains other elements. This is referred to as *nesting*, where elements within the root element are considered nested elements. You can have any number of nested elements as required by the design of the document.

Any element can have nested elements. There are three new elements in Listing 8-29 that are nested with elements of the book. These new elements are called <Header>, <Footer>, and <PageNumber>.

The <Header> describes the text that appears at the top of the page. The <Footer> describes text that appears at the bottom of the page. <PageNumber> describes the page number element.

Each element in the book contains the <Header> and <Footer> elements as nested elements. Furthermore, each <Footer> element contains the <PageNumber> element as a nested element.

You'll notice that each nested element is indented to make it easier to read. However, newline characters and whitespace characters are treated as text elements. You can write a program to ignore newline characters and whitespace characters as long as the document is properly formed. You'll learn how this is done in the next chapter.

Listing 8-29 How to nest elements.

```
<book>
  <TitlePage>
    <Header>
    </Header>
    <Footer>
      <PageNumber>
      </PageNumber>
    </Footer>
  </TitlePage>
  <CopyWritePage>
    <Header>
    </Header>
    <Footer>
      <PageNumber>
      </PageNumber>
    </Footer>
  </CopyWritePage>
  <TableOfContents>
    <Header>
    </Header>
    <Footer>
      <PageNumber>
      </PageNumber>
    </Footer>
  </TableOfContents>
  <Preface>
    <Header>
    </Header>
```

```
    <Footer>
      <PageNumber>
      </PageNumber>
    </Footer>
  </Preface>
  <Chapter>
    <Header>
    </Header>
    <Footer>
      <PageNumber>
      </PageNumber>
    </Footer>
  </Chapter>
  <Chapter>
    <Header>
    </Header>
    <Footer>
      <PageNumber>
      </PageNumber>
    </Footer>
  </Chapter>
</book>
```

Processing Instructions

Besides tags that identify elements of a document, XML also has a tag called a processing instruction. A processing instruction either declares information that is necessary to process the document or directs the program that processes the document to perform a specific function.

A processing instruction tag begins with a <? and ends with a ?>. All XML documents begin with a processing instruction that declares the version of XML used by the document.

Other processing instructions are illustrated as new features are introduced throughout this chapter. Listing 8-30 shows how a processing instruction is used to declare the XML version used in the document.

Listing 8-30 How to declare the XML version for the document.

```
<?xml version="1.0"?>
<book>
</book>
```

Attributes

An XML attribute is similar to an HTML attribute that you used previously in this chapter to describe an element of a document. An attribute is declared within the opening tag of an element by specifying the attribute name and the attribute's value.

Any name can be used for an attribute as long as the name conforms to the XML naming rules and the name is unique within an element. An attribute is local to an element. This means that two elements can have the same attribute name, but the attribute only affects the element that declared the attribute.

The value of an attribute can be any secondary information that is needed to enhance information contained in the element. However, the value of XML attributes must be enclosed within double or single quotations, which is unlike HTML where quotations are optional.

If the value itself contains single quotations, enclose the value with double quotations. This assures that the single quotations will be considered as text. Likewise, use single quotations to enclose a value that contains double quotations or escape with ' and ".

Let's say that you created a personnel document that, among other elements, contains a photograph of an employee. You might create an XML tag called <Photo> and an attribute called source whose value is the name of the file that contains the employee's picture. Here's how the element is written in an XML-based document:

```
<Photo source="emp12345.jpg"> John Smith </Photo>
```

Some elements may require an attribute value, but not require text within the element. For example, the <Photo> element may not require the text John Smith because John Smith might be identified elsewhere in another element of the document such as <EmployeeFirstName> and <EmployeeLastName>. In this scenario, you'd simply leave the element empty of text. This is referred to as an empty element.

There are characters that have a special meaning to the XML processor that processes an XML document. These are the ampersand, greater than character, and less than character. XML uses abbreviations that can be substituted for those characters. Table 8-2 lists them. Use the abbreviation instead of the character in values and text of elements within the XML-based document.

Create a Document Type Definition (DTD)

A forerunner of an XML Schema is a document type definition (DTD), which is still used with some XML documents. This section provides a brief introduction to DTD because focus has switched from DTD to XML Schema, which is covered in more depth in the next section. You can learn more about DTD from *XML: The Complete Reference*.

A DTD defines tags that can be used with an XML document and like an XML Schema is used to standardize a particular set of tags for a company or industry. A

Character	Abbreviation
Ampersand (&)	&
Greater than (>)	>
Less than (<)	<
Single quotation (')	'
Double quotation (")	"

Table 8-2. *Abbreviations for Special XML Characters*

DTD, as with an XML Schema, is optional and therefore you can create an XML document without referencing either a DTD or XML Schema.

Excluding these from an XML document means that any tag can be used in the document. In contrast, a DTD or XML Schema requires tags in the XML document to be defined in the DTD or XML Schema; otherwise, the XML document is not considered well formed.

A DTD requires that elements be defined using the <!ELEMENT> declaration. The <!ELEMENT> declaration must contain the name of the element and the content type. Here's how a Customer element is defined:

```
<!DOCTYPE CUSTOMER [
  <!ELEMENT CustomerFirstName (#PCDATA)>
]>
```

The previous example is a document type definition file called CUSTOMER. The file extension is .dtd. This DTD defines one element called CustomerFirstName and the element has a content type of #PCDATA, which is called parsed character data because characters are translated using the characters listed in Table 8-2.

The DTD is referenced on the first line of an XML document by using the following directive. This states that the program that processes the XML document will find the customer.dtd file located on the URL mywebsite.com. Of course, you replace mywebsite.com with the URL that contains the DTD:

```
<!DOCTYPE CUSTOMER SYSTEM "http://mywebsite.com/customer.dtd">
```

Create an XML Schema

An XML Schema defines elements and attributes of the language that you create using XML and it replaces a document type definition (DTD), which is another method of

defining tags for use in the XML document. An XML Schema is a kind of dictionary that is used by an XML processor to transform the XML document into another kind of document, which can itself be another XML document. The output of an XML processor is dependent on the nature of the processor.

For example, a common use of an XML processor is to transform an XML document into an HTML document that can be displayed by a browser. You'll see how this is done in the "Create an XSLT" section later in this chapter.

All elements and attributes that you use in an XML document must be defined in the XML Schema; otherwise, the XML processor is unable to process the XML document.

An XML document that conforms to an XML Schema is considered a well-formed document. That is, the XML document does not contain any errors. The XML processor determines if the XML document is well formed by applying the XML Schema rules to the XML document. The XML processor displays an error message if an element or attribute in the XML document is either not defined in the XML Schema or doesn't obey rules defined in the XML Schema.

Let's say that you defined an element in the XML Schema called age with an integer data type. An XML processor will display an error if a string is entered as text for the age element in an XML document. This is because the element requires an integer value. For example, "33" might be entered as the value for age. This is acceptable as long as you didn't narrow the scope of the value using a regular expression, which is discussed later in this chapter. However, an error is thrown if "Joe" is entered.

Defining an XML Schema

An XML Schema is a file that you create using any text editor. The file must have the file extension .xsd and the contents of the file must conform to the XML Schema language.

An XML Schema begins by declaring the XML Schema namespace. A *namespace* is a prefix that identifies your XML Schema from another XML Schema, which is critical whenever your XML document is combined with another XML document.

Both XML documents might use the same name for an element, but have different definitions for the element. Therefore, the namespace prefix is used to distinguish elements of different documents whenever these documents are combined.

Here's how you begin an XML Schema:

```
<?xml version="1.0" ?>
<xsd:schema xmlns:xsd="http://www.w3.org/2000/10/XMLSchema">
</xsd:schema>
```

The <?xml version="1.0" ?> defines the XML version that is used in the XML Schema. The xsd:schema defines the namespace used by the XML Schema and the corresponding XML document, which is xsd. The URL identifies the version of the XML Schema language that is used to create the XML Schema. This is located on the w3 organization's web site, which is the group responsible for defining XML standards.

Defining an Element

There are two types of elements defined in an XML Schema. These are simple elements and complex elements. A simple element is an element that contains only text. A complex element is an element that contains other elements or attributes.

The definition of a simple element consists of three components. First is the XML Schema language syntax that identifies to the XML processor that you are defining an element (xsd:element). Next is the name of the element (name=) and the last component is the data type of the element (type=). Listing 8-31 is a definition of a simple element that defines the FirstName element as being of the string data type. You'll notice that the <xsd:schema> uses the abbreviation closing tag, which is a forward slash immediately preceding the closing angle bracket. The traditional closing tag is not necessary if the abbreviation closing is used.

Listing 8-31 How to define a simple element.

```
<?xml version="1.0" ?>
<xsd:schema xmlns:xsd="http://www.w3.org/2000/10/XMLSchema">
  <xsd:element name="FirstName" type="xsd:string" />
</xsd:schema>
```

Here is how the FirstName element is used in an XML document:

```
<FirstName>Bob</FirstName>
```

Complex Element Types

There are four kinds of complex types of elements. These are as follows:

- Elements that contain other elements and optional attributes
- Elements that contain other elements and text and optional attributes
- Elements that contain only text and optional attributes
- Elements that are empty and optional attributes

Listing 8-32 is a definition of a complex element called Person that contains two elements. These are FirstName and LastName, both of which are of the string data type. Data types are introduced later in this chapter.

Listing 8-32 How to define a complex element.

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2000/10/XMLSchema">
  <xsd:complexType name="Person">
    <xsd:sequence>
      <xsd:element name="FirstName" type="xsd:string"/>
```

```
        <xsd:element name="LastName" type="xsd:string"/>
      </xsd:sequence>
    </xsd:completeType>
</xsd:schema>
```

In this example, the definition of a complex element type begins by using the xsd:complexType syntax followed by the name of the complex element type, which is Person. The xsd:sequence tells the XML processor that elements within the complex element type must appear in the sequence these elements appear within the definition of the complex element type. The next two lines define two simple element types called FirstName and LastName. Listing 8-33 is how the Person element is used in an XML document.

Listing 8-33 How previously defined elements are used in an XML document.

```
<Person>
  <FirstName>Bob< 'FirstName>
  <LastName>Smith< 'LastName>
</Person>
```

Predefine the Content of an Element

There might occasions when you require an element to contain a predefined value such as using the name of your company in an XML document. You can have the XML Schema insert the company name instead of requiring the person who creates the XML document to enter the company name.

A predefined content is created when you define an element in the XML Schema by using either the fixed or default attribute followed by the content that will be included when the element is used in an XML document. This is illustrated here:

- <sxd:element name="Company" type="xsd:string" fixed="My Company"/> is used to assign the value "My Company" to the "Company" element. If the "Company" element isn't included in the XML document, the "Company" element doesn't contain any value.
- <sxd:element name="Company" type="xsd:string" default-="My Company"/> is used to assign the value "My Company" to the "Company" element. If the "Company" element isn't included in the XML document, the "Company" element is automatically assigned the content "My Company".

Setting a Set of Acceptable Values

You can restrict the content of an element to one of a series of values by setting the xsd:enumeration tag when you define the element in the XML Schema. Each xsd:enumeration tag is assigned an acceptable value that can appear as the element's content.

Listing 8-34 illustrates how to create a list of acceptable content for an element. The element is called state and a few abbreviations for states are listed to show how to use the xsd:enumeration tag.

The XML document is considered well formed only if one of the values listed in the definition element is included as content for the element, if the element is used in the XML form.

Listing 8-34 How to create a list of acceptable content for an element.

```
<xsd:element name="state">
  <xsd:simpleType>
    <xsd:restriction base="xsd:string">
      <xsd:enumeration value="CA"/>
      <xsd:enumeration value="NJ"/>
      <xsd:enumeration value="TX"/>
    </xsd:restriction
  </xsd:simpleType>
</xsd:element>
```

Data Types

A data type of an element restricts the kind of data that can be contained by an element to only data that conforms to the specified data type. For example, an element that defines a serial number is an integer data type. This means the content of the element must be an integer. Content other than an integer (for example, a person's name) causes the XML processor to display an error message when the document is processed.

You can specify the data type of an element by assigning one of the several predefined data types to the type attribute of the xsd:element definition. Here are commonly used data types. The entire list of data types is available from http:// www.w3.org/TR/xmlschema-2/#built-in-datatypes.

- **xsd:boolean** An element must contain a Boolean value.
- **xsd:date** An element must contain a date using yyyy-mm-dd with (optionally) time zone indicator—time zone affects sorting rules.
- **xsd:decimal** An element that contains a decimal value, but can also contain an integer. A total of 18 digits is the minimum number of digits available in a processor that conforms to the XML standard.
- **xsd:string** An element must contain character data, including whitespace and punctuation.
- **xsd:time** An element must contain a time value, which is hh:mm:ss.sss with (optionally) time zone indicator—time zone affects sorting.
- **xsd:uri** An element must contain a URI.

In addition to predefined data types, you can create customized data types that are based on predefined data types. This is a way to extend a predefined data type to include restrictions such as a range of dates. You'll see how this is done later in this chapter.

Dates and Time

The XML Schema language contains a variety of predefined date and time data types that enable you to control date and time content that is contained in an element of your XML document.

Table 8-3 contains date and time data types along with the format in which the data and/or time content must appear within an element. The table also illustrates the proper declaration of the data type in the XML Schema and includes an example on how date and/or time are entered as content in an element.

Date Type	Format	Declaration	Example in XML Document
xsd:timeDuration	PnYnMnDTnHnMnS P1Y3M10D = 1 year 3 months 10 days T4H25M30S = 4 hours 25 minutes 30 seconds	<xsd:element name= "LenEmployment" type="xsd:time Duration" />	< LenEmployment > P1Y3M10D </ LenEmployment >
xsd:time	hh:mm:ss.sss 15:30:00.000 = 3:30 p.m.	<xsd:element name="MeetTime" type="xsd:time" />	< MeetTime >15:30 </ MeetTime >
xsd:timeInstance	A moment in time. ccyy-mm-ddThh: mm:ss.sss 2003-10-18T14:43:22.543 = Oct. 18, 2003 2:43:22.543 PM	<xsd:element name= "TradeReceived" type= "xsd:timeInstant" />	< TradeReceived > 2003-10-18T14:43:22.543 </ TradeReceived >
xsd:date	ccyy-mm-dd 2003-10-18 = Oct. 18, 2003	<xsd:element name= "TradeDate" type= "xsd:date" />	< TradeDate > 2003-10-18 </ TradeDate >
xsd:month	ccyy-mm 2003-10 = Oct. 2003	<xsd:element name= "TradeMonth" type= "xsd:month" />	< TradeMonth > 2003-10 </ TradeMonth >
xsd:year	ccyy 2003= 2003	<xsd:element name= "TradeYear" type= "xsd:year" />	< TradeYear >2003 </ TradeYear >
xsd:century	cc 20= 2000	<xsd:element name= "TradeCentury" type= "xsd:century" />	< TradeCentury > 2000 </ TradeCentury >

Table 8-3. *Date and Time Data Types*

Date Type	Format	Declaration	Example in XML Document
xsd:recurringDay	Whenever content of the element should represent a day of the month with regard to year and century. ---15 = the 15th day of the month	<xsd:element name= "RegMeeting" type= "xsd:recurringDate" />	< RegMeeting >---15 </ RegMeeting >

Table 8-3. *Date and Time Data Types* (continued)

Time isn't time zone dependent if the time zone indicator (Z) is excluded from the date/time declaration. However, the time is adjusted for the time zone if the time zone indicator is used as the last character in the content.

The XML Schema language uses the Coordinated (Universal) Time (UTC), which is the local time in London and commonly referred to as Greenwich mean time. The element requires a particular format where a minus or plus sign precedes the time difference between local time and GMT.

For example, eastern standard time is UTC-5, or five hours earlier than it is in London (Greenwich mean time). Pacific standard time is UTC-8. Western Europe is UTC+1, with some exceptions.

You indicate UTC time by using the Z symbol in the data type, or you can use the plus and minus signs to indicate the time difference between the time in the XML document and Greenwich mean time. For example, a -5 indicates eastern standard time or five hours before Greenwich mean time.

Numeric Data Types

You can specify that the content of an element be a numeric data type by using one of the several predefined numeric data types that exist in the XML Schema language. Table 8-4 illustrates each data type.

Keep in mind that defining an element as having a numeric data type does not prevent a person from entering non-numeric values as content in the element.

Content that is of a different data type than the data type you specify in the XML Schema causes the XML processor to consider the XML document not well formed and prevents the XML document from being processed.

Customized Data Types

A customized data type is a data type that you create based on a predefined XML Schema data type. The purpose of creating your own data type is to enhance the data typing capabilities of existing data.

Data Type	Format	Declaration	Example in XML Document
xsd:decimal	Content can contain a decimal value. n.n = 10.95	<xsd:element name= " price" type="xsd: decimal "/>	<price> 10.95</price>
xsd:integer	Content must be an integer. n = 34	<xsd:element name= " age" type="xsd: integer "/>	< age > 34</ age >
xsd:positiveInteger	Content must be a positive integer. n = 34	<xsd:element name= " age" type="xsd: positiveInteger "/>	< age > 34</ age >
xsd:negativeInteger	Content must be a negative integer. -n = -30	<xsd:element name= " countdown" type= "xsd: negativeInteger "/>	< countdown > -30 </ countdown >
xsd:nonPositiveInteger	Content must be a nonpositive integer. -n = -30 nonPositive and nonNegative include 0	<xsd:element name= " countdown" type= "xsd: negativeInteger "/>	< countdown > -30 </ countdown >
xsd:nonNegativeInteger	Content must be a nonnegative integer. n = 34	<xsd:element name= " age" type="xsd: positiveInteger "/>	< age > 34</ age >
xsd:float	Content can be a 32-bit floating-point value. n.nnnnnnn = 1.5647834 Can also use exponential notation, such as 1.234e-3. In addition, the following modifiers can be used: INF—Positive infinity. -INF—Negative infinity. 0—Positive zero. -0—Negative zero. NaN—Not a number (important for parsers).	<xsd:element name= " size" type="xsd: float "/>	<size > 1.5647834 </ size >

Table 8-4. *Numeric Data Types*

Data Type	Format	Declaration	Example in XML Document
xsd:double	Content can be a 64-bit floating-point value. n.nnnnnnnnnnnnnnn =1.564783445673967 Can also use exponential notation, such as 1.234e-3. In addition, the following modifiers can be used: INF—Positive infinity. -INF—Negative infinity. 0—Positive zero. -0—Negative zero. NaN—Not a number (important for parsers).	<xsd:element name= " size" type="xsd: double "/>	<size > 1.564783445673967 </ size >

Table 8-4. *Numeric Data Types* (continued)

Let's say that you define an element whose content is a ZIP code. A ZIP code element is usually declared as a string data type. However, a string data type accepts any combination of characters and numbers, which means an XML document could have a ZIP code element that contains a value other than a ZIP code, such as a 20 digit number.

You can create a custom data type that restricts the content of an element such as a ZIP code element. In that way any extraneous characters entered as content causes the XML document not to be well formed.

There are two kinds of custom data types. These are simpleType and complexType. A simpleType does not contain other elements or attributes. A complexType can contain other elements and attributes, and these are discussed later in this chapter.

SimpleType You define a simpleType by using the xsd:simpleType tag in the XML Schema and include a unique name to identify the data type. Listing 8-35 shows how to define a custom data type that is a simpleType, which is called PostalCodeType.

After the xsd:simpleType tag defines the name of the custom data type, the xsd:restriction tag is used to define the base data type for the custom data type. The base data type is the predefined data type that is enhanced by the custom data type. The PostalCodeType enhances the xsd:string data type.

Next, the xsd:pattern tag is used to describe characters and/or numbers that are acceptable to the custom data type. That is, content entered into an element that is defined as the PostalCodeType is limited to characters and/or numbers that match the value of the xsd:pattern tag.

The pattern is defined using a regular expression language that is similar to the regular expression language used in Perl. The content of an element that is defined as the custom data type must match the pattern to conform to a well-formed XML document. Table 8-5 lists the symbols for a regular expression. You can combine regular expression symbols to create sophisticated limitations to entries made into an element.

In the PostalCodeType example (see Listing 8-35), the xsd:pattern tag is used and the value attribute is assigned a pattern. This means that the content for the PostalCodeType element must have five digits followed by an optional dash and four digits. The [0-9] can be replaced with "\d" to represent a digit.

Symbol	Definition
. (Period)	Any character
\d	Any digit
\D	Any nondigit
\s	Any whitespace
\S	Any nonwhitespace character
x*	Zero or more occurrences of x where x can be any character or pattern. You can evaluate the number of occurrences of the pattern by placing x in parentheses such as (abc)* where the string "abc" appears from zero times to many times in x.
x?	Zero or one occurrence of the character where x can be any character or pattern (see above).
x+	One or more occurrences of the character where x can be any character or pattern (see above). The plus sign means one or more occurrences.

Table 8-5. *Symbols Used to Create a Pattern*

Symbol	Definition
(xyz){3}	A pattern is defined within the parentheses and the value within the French braces indicates the number of times the pattern appears in the string. This means xyz appears three times in the string. Square brackets are used for character sets.
[0-9]	Defines a character set where the value must match one of the values in the character set. The dash is used to define a range of sequential values such as: [0-9A-Za-z] matches any alphanumeric character. [0-9 A-Za-z] is the same as above except it also matches a space character (0x20). [0-9] is equivalent to [0123456789] or [0987654321]. [0-9\-] matches digits and the dash character.
x{4}	Must have exactly four x's in a row where x can be any character or pattern defined using the parentheses (see above).
x{4,}	Must have at least four x's in a row where x can be any character or pattern.
x{4,9}	Must have at least four and at most nine x's in a row where x can be any character or pattern.
'abcd'	This means that the string contains 'abcd'.
'^abcd'	This means that the string starts with 'abcd'.
'abcd$'	This means that the string ends with 'abcd'.
'^abcd$'	This means that the string equals with 'abcd'.
\$	The backslash matches the literal value of any special that follows the backslash character. In this example, it matches a $.

Table 8-5. *Symbols Used to Create a Pattern* (continued)

Once a value is assigned to the xsd:pattern tag, you must insert a close xsd.restriction tag and a close xsd:simpleType tag into the XML Schema.

Listing 8-35 How to create a pattern for a custom data type.

```
<xsd:simpleType name="PostalCodeType">
  <xsd:restriction base = "xsd:string">
    <xsd:pattern value="\[0-9]{5}(-[0-9]{4})?" />
  </xsd:restriction>
</xsd:simpleType>
```

Setting a Range of Values

In addition to creating a pattern for a custom data type, you can also specify a range of values that must be contained in an element. There are four ways to specify a value range. These are by using maximum and minimum and inclusive and exclusive tags:

- **<xsd:maxInclusive value="3500"/>** Requires that the value be less than or equal to the value assigned to value.
- **<xsd:maxExclusive value="3500"/>** Requires that the value be less than the value assigned to value.
- **<xsd:minInclusive value="3500"/>** Requires that the value be equal to or greater than the value assigned to value.
- **<xsd:minExclusive value="3500"/>** Requires that the value be greater than the value assigned to value.

Setting Length of Contents

You can limit the length of characters in a custom data type by using one of three length tags. Each tag is used to set the minimum, maximum, and exact length for the content of an element that is defined using the custom data type.

Here are the length tags:

- **<xsd:length value="15"/>** Specifies that the content of an element must be 15 characters in length for the XML document to be well formed.
- **<xsd:maxLength value="15"/>** Specifies that the content of an element must have no more than 15 characters in length for the XML document to be well formed.
- **<xsd:minLength value="15"/>** Specifies that the content of an element must have at least 15 characters in length for the XML document to be well formed.

Setting Precision and Scale

You can limit the number of digits that are entered into an element and limit the number of digits to the right of the decimal by using the precision and scale tags in a custom data type definition:

- ***<xsd:precision value="6"/>** Sets the maximum number of digits that appear in an element for the XML document to be well formed. The entered number, including the decimal value but excluding the decimal itself, must be no more than six digits.
- ***<xsd:scale value="3"/>** Sets the maximum number of digits that appear to the right of the decimal in an element for the XML document to be well formed. No more than three digits can appear to the right of the decimal in this example.

Complex Data Types

In the previous sections you learned how a simpleType custom data type is defined. A simpleType is used for elements that don't contain other elements or attributes. Many of the XML documents that you create will have elements that contain other elements or attributes, and therefore defining a custom data type that is a simpleType is inappropriate.

Therefore, you need to define a custom data type that is a complexType. A complexType custom data type is designed for elements that contain other elements or attributes.

Listing 8-36 shows how to declare a complexType that contains two elements. You can have as many elements as is required by your document by inserting the definition of elements within the complexType definition as shown in Listing 8-36.

The complexType is called "Person" and the two elements are the person's "FirstName" and "LastName". This means that an element in the XML document that is defined as a "Person" data type must contain two elements that are called "FirstName" and "LastName". An XML document isn't considered well formed if the element of a "Person" data type doesn't contain these two elements.

Furthermore, these elements must appear in the same sequence as defined in the "Person" data type. This is because the xsd:sequence tag is used in the definition. You could enable these elements to appear in any order by replacing the xsd:sequence tag with the <xsd:all> tag.

If you replace the xsd:sequence tag with the <xsd:all> tag , the XML document will be considered well formed if the element defined as a "Person" type contains both elements and those elements could be in reverse order as they are defined in the "Person" data type definition.

Listing 8-36 How to declare a complexType.

```
<xsd:complexType name="Person">
  <xsd:sequence>
    <xsd:element name="FirstName" type="xsd:string"/>
    <xsd:element name="LastName" type="xsd:string"/>
  </xsd:sequence>
</xsd:complexType>
```

Defining Attributes

An attribute is additional information used by an element besides the element's content. You've seen attributes used previously in this chapter with HTML tags. You define an attribute for an XML element by using the xsd:attribute tag in the XML Schema. The attribute requires a name that is unique to the element and a data type that can be a predefined data type or a custom data type.

Listing 8-37 illustrates how to define an attribute. The attribute is called "Gender" and is used by the "Employee" element to identify the gender of an employee.

The element and attribute are used in an XML document by writing <Employee Gender="Female" />. An attribute is optional unless you make the attribute required. That is, a person can use the "Employee" element, exclude the attribute, and the XML document is still considered well formed.

You can require the attribute by assigning the "required" value to the use attribute such as <xsd:attribute name="Gender" type="xsd:string" use="required"/>.

Listing 8-37 How to define an attribute.

```
<xsd:element name="Employee">
  <xsd:complexType>
    <xsd:complexContent>
      <xsd:extension base="xsd:anytype">
        <xsd:attribute name="Gender" type="xsd:string"/>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
</xsd:element>
```

The value of an attribute can be predefined by using fixed or default, similar to how these are used to predefine the content of an element. Declaring the predefined value as fixed means the predefined value is assigned to the attribute only if the element appears in the XML document. The attribute isn't assigned the predefined value if the element isn't used in the XML document.

In contrast, an attribute that has a predefined value as default is always assigned the predefined value even if the element doesn't appear in the XML document.

Listing 8-38 creates a predefined value for the "Gender" attribute as assigned "fixed" and the value attribute is assigned the predefined value "Female". You can make the attribute default by replacing the "fixed" with "default".

Listing 8-38 How to create a predefined value for an attribute.

```
<xsd:element name="Employee">
  <xsd:complexType>
    <xsd:complexContent>
      <xsd:extension base="xsd:anytype">
       <xsd:attribute name="Gender" type="xsd:string" use="fixed" value="Female"/>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
</xsd:element>
```

Referencing an XML Schema

You link an XML document to an XML Schema by referencing the XML Schema file in the root tag of the XML document. Let's say that XML Schema is stored in a file called myXMLSchema.xsd. Listing 8-39 shows how you reference the XML Schema inside the XML document.

Listing 8-39 How to reference the XML Schema inside the XML document.

```
<?xml version="1.0" ?>
<root xmlns :xsi="http://www.w3.org/2000.10.XMLSchema-instance"
xsi:noNamespaceSchemaLocation="http://www.jimkeogh.com/ns/
myXMLSchema/myXMLSchema.xsd">
```

The xsi:schemaLocation attribute of the root tag declares the namespace, which points to the location of the file that contains the XML Schema for the XML document.

Create an XSLT

An XSLT is used by an XML-enabled browser (that is, Internet Explorer 5 and Netscape 6) to transform an XML document into an HTML document so the contents of an XML document can be displayed in a browser.

Here's the process that occurs when an XSLT is used to transform an XML document. First, an XSLT processor analyzes the XML document and translates components (that is, tags, attributes, and content) into a hierarchical representation of the XML document, which is referred to as a node tree. A *node* is a component of an XML document.

Next, the XSLT processor matches each node with a template in the XSLT style sheet. A template contains a two-part instruction. The first part is a label that matches a node in the XML document. The second part is the transformation that takes place when the node is encountered by the XSLT processor.

The transformation part of the instruction is where HTML code is inserted into the XSLT style sheet file that has the extension xsl. This HTML code is displayed or saved to a file whenever that XSLT processor finds a node in the XML document that matches the label of the instruction.

XSLT Defined

An XSLT style sheet consists of one or more template definitions that are processed by the XSLT processor when the template label matches a component in the XML document.

The template definition consists of code that will be output by the XSLT processor. The kind of code that you use in the template definition depends on the program that will use the output.

For example, HTML code is used in the template definition if the output is going to be displayed by a browser. The output is automatically sent to the browser if the browser uses an XSLT processor; otherwise, the XSLT processor sends the output to a file, which can then be read by a browser.

Listing 8-40 shows a short XML document followed by an XSLT style sheet (see Listing 8-41) that contains templates. The XML document has an element called Employee and it contains an attribute called Dept. There are two other elements, which are FirstName and LastName.

The XSLT style sheet begins with the namespace declaration and version identifier. If you're using the Internet Explorer 5 XSLT processor, you'll need to replace the namespace declaration with <xsl:stylesheet xmlns:xsl="http://www.w3.org/TR/WD-xsl">.

The template is defined in the XSLT style sheet and begins with the <xsl:template> tag that contains the match attribute. The value of the match attribute is the name of the XML document component that is transformed by the template.

The content of an element is referenced from within a template definition by using the <xsl:value-of select=" "/> tag. The value of the select is specific to each element and must be enclosed within quotations as illustrated in Listing 8-41. The XSLT processor replaces the tag with the contents of the element.

Make sure that you properly use double quotations and single quotations when referencing the attribute of an XML document element. The value of the match attribute must be enclosed in double quotations. The value of the XML document attribute must be enclosed in single quotes within the square brackets.

The template must have a closed </xsl:template> tag. Listing 8-42 illustrates the output of the HTML document.

Listing 8-40 A sample XML document saved in Employee.xml, in the same directory as Employee.xsl.

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="Employee.xsl" ?>
<MyDocument title="My test page">
  <Employee Dept="52">
    <FirstName>Jane</FirstName>
    <LastName>Smith</LastName>
  </Employee>
  <Employee Dept="59">
    <FirstName>Henry</FirstName>
    <LastName>Hudson</LastName>
  </Employee>
</MyDocument>
```

Listing 8-41 An XSLT style sheet saved in Employee.xsl.

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/
      Transform" version="1.0">
```

```
<xsl:template match="MyDocument">
<html>
  <head>
    <title><xsl:value-of select="@title"/></title>
  </head>
  <body>
    Employee list:<br/>
    <xsl:for-each select="Employee">
      First name: <xsl:value-of select="FirstName"/><br/>
      Last name: <xsl:value-of select="LastName"/><br/>
      Dept: <xsl:value-of select="@Dept"/><br/><br/>
    </xsl:for-each>
  </body>
</html>
</xsl:template>
</xsl:stylesheet>
```

Listing 8-42 Open Employee.xml in IE5 or higher and the following should be displayed and the title bar in the browser should say "My test page".

```
Employee list:
First name: Jane
Last name: Smith
Dept: 52

First name: Henry
Last name: Hudson
Dept: 59
```

Planning an XML Database Program

An XML document has some of the characteristics of a database in that the XML document is a collection of data identified by metadata and stored in a file. In a relational database, data is stored in columns that are identified by a column name. In an XML document, data is stored in elements that are identified by tags.

Data stored in both an XML document and a relational database can be manipulated by a program and exchanged with data from other applications. However, data in a relational database is more efficiently manipulated than data in an XML document because of the way data is organized and managed by a DBMS.

You can blend the simplicity of an XML document with the efficient data management capabilities of a relational database by storing an XML document into a relational database and using a DBMS to manage the XML document.

Concepts of an XML Database Program

Many corporations that need to manage large amounts of information stored in documents have embraced the advantages of XML. XML is intuitive and flexible, and

enables information to be organized in an advantageous way for corporations and industry to exchange information.

An industry group can agree on a standard XML Schema for a particular type of document, such as a book. This standard enables participants in the industry to share and manipulate a document economically. For example, a printer is able to accept XML files of a manuscript to transform the files into a finished book.

However, XML is a fledgling technology that isn't as robust and mature as other data collection and management tools such as relational DBMS. An XML document is stored in a text file, also known as a flat file, that doesn't lend itself to efficient data management.

A text file consists of a long series of characters, some of which represent XML tags while others represent data. There isn't an easy way to distinguish between a tag and data. This means that a program must search each character of the text file looking for data.

Let's say that you wanted an employee's name and you used the employee's number to locate the employee in the following XML document. Listing 8-43 illustrates how to locate the employee information.

Listing 8-43 Locating employee information in an XML document.

```
<?xml version="1.0"?>
<root xmlns:xsi="http?//www.w3.org/2000.10.XMLSchema-instance"
xsi:noNamespaceSchemaLocation="http:/ /www.jimkeogh.com/ns/
myXMLSchema/myXMLSchema.xsd">
<Person EmpNumber="1234">
  <FirstName>Bob</FirstName>
  <LastName>Smith</LastName>
</Person>
```

A program starts by reading the first line of the XML document and compares the contents of the line with elements defined in the XML Schema. That is, the program looks for the line that contains the Person tag.

When the line is found, the program searches the line for the EmpNumber attribute and then compares the value of the attribute with the employee number of the search criteria. If the employee numbers don't match, the program continues to reach subsequent lines until either the Person tag is located or the program reaches the end of the XML document.

If the employee numbers match, the program reads the following lines to retrieve the first name and last name of the employee. However, each of those lines contains an open and close tag (that is, <FirstName> and </FirstName>) and the employee's name. Therefore, the program must parse the employee's name from the line to complete the search process.

In contrast, a DBMS doesn't need to read through all possible metadata and data to locate a particular employee number. Instead, a DBMS typically uses an index (see Part II) that contains the employee number (the key) and a reference to related employee information in a table.

Using XML to organize and manage a database can become unattractive when compared to other forms of data management (for example, a relational database). This is especially true when information is contained in many XML documents.

Let's say that XML is used to organize personnel data. Employees complete a personal form when a company hires them. The personal form could be stored as an XML document. This means that a company that has 1,000 employees has 1,000 XML documents to store and manage, which is a cumbersome task unless XML documents are blended with a relational database.

Objectives of an XML Database

An XML database is a blend of XML technology and DBMS technology that provides companies the best of both technologies. The result is a blend of technologies that enables companies to efficiently manage XML documents.

By blending these technologies, companies are able to

- Store and retrieve XML documents
- Search multiple XML documents for specific data
- Search and replace element tags and/or data stored in an XML document
- Modify an XML document
- Delete components of an XML document or delete the entire XML document

This means information can be organized according to an industry- or enterprise-wide accepted XML Schema and be managed by a DBMS. Also, SQL queries can be used to insert, retrieve, modify, and delete both metadata and data in an XML document.

Types of XML Database Schemas

In essence, an XML database is a relational database that contains one or multiple XML documents. That is, any relational database product (for example, Oracle, mySQL, or DB2) can be used as an XML database.

An XML database uses a different database schema than is commonly used in other relational databases (see Part II). This is because an XML database is used to store components of an XML document rather than simply storing data contained in the document.

For example, the XML document described in the previous section has three elements (Person, FirstName, LastName), one attribute (EmpNumber), and three pieces of information (1234, Bob, Smith).

A traditional database schema could be used to store data contained in the XML document, if only the data was to be stored in the database. A single table of three columns—Employee Number, First Name, and Last Name—could be used. Each row of the table would contain an employee number and employee name.

However, an XML database must store data contained in elements of the XML document and other components of the XML document (that is, tag name, attribute name, open tags, and close tags).

XML database designers use one of three database schemas to store an XML document in a relational database. I'll briefly introduce each here and then explain them later in this chapter:

- **Coarse-grained** The database consists of one table that contains two columns. One column identifies the XML document and the other column is used to store the entire XML document.
- **Medium-grained** The XML document is divided into slices and each slice is placed in a row of a table.
- **Fine-grained** Individual components of the XML document are stored in columns of multiple tables in the database.

Coarse-Grained XML Relational Database

The coarse-grained style requires that the XML document be stored in the database as an entire document, rather than decomposing documents into document elements. At first glance, you might consider the coarse-grained style nothing more than a document stored in a flat file rather than in a database. This is because a program that retrieves the document must parse the data from the document line by line, similar to how data is extracted from a document in a flat file.

As you'll recall from Part II of this book, information in an entity, which a document is, is decomposed into atomic components called data. Data is then organized into rows and columns of a table, and an index is used to locate data.

Therefore, you are probably wondering what the benefits are of storing an XML document in its entirety into a database instead of storing data components of the document in the database. The benefits of using the coarse-grained style are

- XML documents can be referenced quickly using the capabilities of a DBMS.
- DBMS provides an efficient means of protecting documents through the use of security and data recovery features.
- The coarse-grained style provides an intermediate step from moving from a flat file XML document system to an XML document system that is decomposed and normalized into tables.
- XML is often used for transmitting data between partner companies or within a company. The data can be preformatted into XML and stored in the database; then, the query simply selects all the unsent records and sends them to their destination in XML form.

The database schema for the coarse-grained style consists of one table that has three columns. The columns are

- DocumentID, Number, Primary Key
- DocumentName, VARCHAR
- Document, CLOB

An XML document is inserted into the table and retrieved from the table using a query and Java data objects (see Part II). Let's assume that the XML document contains customer information and the XML document name is CustNum123, which is the customer's number. The XML document is recalled from the database by using the query in Listing 8-44. The ResultSet contains one column, which is the XML document, and is retrieved by using the getCLOB() method (see Chapter 6).

Listing 8-44 A query to retrieve an XML document. from a database.

```
SELECT Document
FROM XMLTable
WHERE DocumentName = 'CustNum123'
```

Medium-Grained XML Relational Database

The major disadvantage of using the coarse-grained style as described in the previous section is that the J2EE component that retrieves the XML document from the database must parse data from the XML document before being able to use the data.

The medium-grained style provides finer granularity than that found in the coarse-grained style—but not the finest granularity, as is seen in the fine-grained style, which is discussed later in this chapter.

The medium-grained style requires that the XML document be sliced into subsections. These subsections are then stored using the coarse-grained technique. Let's say that an XML document contains a book. The entire book would be stored in a row of a database if the coarse-grained technique were used.

However, the XML document that contains the book can easily be sliced into subsections where each subsection is a chapter of the book. This means that each chapter is contained in one row of the table.

The coarse-grained style is similar to a book that doesn't have a table of contents or an index. This means you must look line by line through the book trying to locate the information that you need.

The medium-grained style is similar to a book that has a table of contents. The table of contents slices the book into medium chunks that help you identify the general area of the book to look at to locate information that you need. The fine-grained style is similar to a book that has an index. That is, the book is sliced into the finest elements of the book, which are words.

Let's say that you wanted to create an XML database that uses the medium-grained style to store chapters of a book. The following is a likely database schema. The primary key for the table is the ChapterID, which together uniquely identifies each row in the table. Listing 8-45 contains the query used to retrieve a chapter from the database. The chapter is returned in the ResultSet and copied from the ResultSet using the getCLOB() method.

- ChapterID, Number, Primary Key
- DocumentID, Number, Foreign Key
- DocumentName, VARCHAR

- ChapterName, VARCHAR
- Chapter, CLOB

Listing 8-45 A query to retrieve an XML document from a database.

```
SELECT Chapter
FROM XMLTable
WHERE DocumentName = 'ISBN1234' and ChapterName = 'Introduction'
```

Fine-Grained XML Relational Database

The fine-grained style requires that an XML document be decomposed into elements, attributes, and character data regions—each of which are grouped together into a table. There is also another table that is used to store parent/child relationships between elements, subelements, and character data.

The advantage of using a fine-grained style is that each element, attribute, and character data region can be easily accessed, modified, or deleted without affecting other components of the document. For example, data that is associated with an element can be modified by changing the value of the column that contains the data, rather than having to parse the data from the XML document, change the data, then reconstruct the XML document.

The major disadvantage of using the fine-grained style is the time necessary to decompose and regenerate the entire XML document. Each time an XML document is stored in a database, a J2EE component must parse the XML document into elements, attributes, and character data region as well as recording parent/child relationships between elements, subelements, and character data. You would probably perform parsing and a series of inserts using a stored procedure, rather than in the J2EE component. Many developers avoid using the fine-grained style and instead prefer the medium-grained style.

A J2EE component must reconstruct the XML document from elements, attributes, and character data regions retrieved from the database, which is time-consuming. The following is the database schema for the fine-grained style.

Listing 8-46 illustrates how to retrieve data stored from a fine-grained style database. In this example, the query retrieves the document name, attribute value, and the element value from the XML document. An entire XML document can be reconstructed by using the ParentID in the Elements table to determine the parent/child relationships between elements and subelements.

- Documents Table
  - DocumentID, Number, 8 digits, Primary Key
  - DocumentName, VARCHAR, 128 characters, Not Null
- Elements Table

- DocumentID, Number, 8 digits, Primary Key, Foreign Key references documents.documentID
- ElementID, Number, 8 digits, Primary Key
- ParentID, Number, 8 digits, Foreign Key references elements.elementID (nullable because of the root node)
- ElementName, VARCHAR, 32 characters, Not Null
- ElementValue, VARCHAR, Not Null
- Attributes Table (the DocumentID and AttributeID combined create the Primary Key)
  - AttributeID, Number, 8 digits, Primary Key
  - ElementID, Number, 8 digits, Not Null, Foreign Key references elements.elementID
  - AttributeName, VARCHAR, 32 characters, Not Null
  - AttributeValue, VARCHAR, 255 characters, Not Null

Listing 8-46 Retrieving the document name, customer number, and the customer last name from an XML document.

```
Select D.DocumentName, A.AttributeValue, E.ElementValue
FROM Documents D, Attributes A, Elements E
WHERE D.DocumentName = 'Customers' and D.DocumentID = A.DocumentID and
A.AttributeName = 'CustomerNumber' and A.AttributeValue = '123467' and
E.DocumentID = D.DocumentID and E.Element = 'FirstName' and A.ElementID = E.
ElementID
```

XHTML

XHTML is the next major generation of HTML and incorporates many aspects of XML. This section is not designed to teach you XHTML. For that, you should pick up a copy of *HTML: The Complete Reference,* which contains all you need to know about HTML and XHTML.

XHTML uses practically all of the HTML elements. However, XHTML imposes new rules that require you to tweak your HTML syntax to make your HTML documents XHTML compliant. This section illustrates the tweaking:

- XHTML is case sensitive. This means all HTML syntax must be in lowercase. For example, <FORM> must be changed to <form>.
- Every open tag must have a closing tag. For example, a paragraph is inserted into a document using the <P> tag. Although the document should also contain a </P> tag, most browsers don't require it. This is no longer the situation with XHTML-compliant browsers.

- A shortcut can be used for tags such as <br> that usually don't have a closing tag.
 is a line break. The shortcut is to use <br />. Notice there is a space between the last letter of the tag and the forward slash. Attributes are placed between the last letter of the tag name and the close angle bracket, such as . If this is used as the opening tag, you don't need a closing tag to be XHTML compliant.
- Values of all attributes must be enclosed with double quotations. This is optional in HTML.



Chapter 9

Java and XML

Information management is key to the success of most businesses. For many companies information is managed at the atomic level in the form of data by using a database management system, as discussed in Part II.

However, information used by some industries such as publishing and pharmaceuticals need to manage information at a coarser level in the form of a document. Documents are composed of components such as forward, preface, and chapters. XML technology enables a company to identify each document component using tags and attributes to create an XML document. XML works well because XML is platform independent and is extendable with little or no impact to existing systems.

An XML document can be created, read, and manipulated by using a J2EE component that is built using Java application program interfaces that are especially designed for use with an XML document. This chapter shows you how to build these J2EE components.

Generating an XML Document

A J2EE application can generate an XML document based on business rules that are encoded into one or more J2EE components. Typically, these J2EE components are either a Java servlet, a Java ServerPage (JSP), Enterprise JavaBeans (EJB), or a Java Message Service component.

Sending strings that contain XML tags and directives as output from either a JSP or Java servlet generates an XML document. Output from a Java servlet and a JSP is directed to the HTTP connection that invoked the Java servlet or a JSP, which is typically associated with a browser.

Java servlets and a JSP are discussed in detail later in this book; however, the following sections provide a brief overview of each and show how they can be used to produce an XML document.

Java Servlet

A Java servlet is a web server extension that provides a web service to a J2EE application. A Java servlet is similar to a Java program in that a Java servlet is a self-contained program that can perform a specific task and is called from another J2EE component such as the J2EE application user interface.

There are two ways in which a Java servlet is executed, which are briefly mentioned here and discussed in detail later in the book:

- Specifying the URL of the Java servlet as the value of the ACTION attribute of an HTML form (see Chapter 8) such as "http://www.myURL.com/servlets/myservlet." You can also place the URL path directly in the location bar of the browser.
- Chaining is another way to call a Java servlet. This enables a Java servlet to automatically be called based on the MIME type of a data stream and is configured by the web server administrator.

Listing 9-1 illustrates a Java servlet that generates an XML document. This document contains elements of a book and is the same document as is discussed in Listing 8-28 in Chapter 8. This is a simplified XML document that illustrates the technique used to create the document using a Java servlet. A real-world XML document contains many more elements than are illustrated here.

The web server calls the doGet() method or the doPost() of a Java servlet to invoke the Java servlet depending on whether the GET or POST method is used to transmit the request (see Chapter 8). Two parameters are used in these methods. These are the HttpServletRequest object (called input in this example) and the HttpServletResponse object (called output in this example). The HttpServletResponse object is used to return the output from the Java servlet to the web server.

The initial task of the Java servlet is to set the content type of information that is generated by the Java servlet by calling the setContentType() method of the HttpServletResponse object and passing the method "text/xml" string.

Next, a PrintWriter object is created using the getWriter() method of the HttpServletResponse object. In this example, the PrintWriter object is called out. The println() method of the PrintWriter object is then used to send strings that contain XML tags and directives to the web server. The tags and directives are processed according to the rules built into the requesting J2EE component—or in this example, rendered by the browser.

In this example, the XML document contains a template for a book. Only the <TitlePage> tag contains text. There are, of course, many more tags and directives used in a real-world XML document used for a book, and all those tags contain text.

Listing 9-1 Creating an XML document using a Java servlet.

```
import java.io.*;
import javax.servlet.http.*;
import javax.servlet.*;
public class BookTemplate extends HttpServlet
{
  public void doGet(HttpServletRequest input, HttpServletResponse output)
    throws ServletException, IOException
  {
  {
    output.setContentType("text/xml");
    PrintWriter out = output.getWriter();
    out.println("<?xml version=\"1.0\" ?>");
    out.println("<book>");
    out.println("<TitlePage>");
    out.println("J2EE, The Complete Reference");
    out.println("</TitlePage>");
    out.println("<CopyWritePage>");
    out.println("</CopyWritePage>");
    out.println("<TableOfContents>");
    out.println("</TableOfContents>");
    out.println("<Preface>");
    out.println("</Preface>");
    out.println("<Chapter>");
```

```
            out.println("</Chapter>");
            out.println("<Chapter>");
            out.println("</Chapter>");
            out.println("</book>");
        }
    }
```

JavaServer Pages

As you'll learn in the JSP chapter later in this book, JSP was developed to make it easier for developers to create programs that generate dynamic HTML pages. Prior to the introduction of JSP, developers used Java servlets to generate dynamic web pages.

Java servlets are written using the Java programming language. HTML code is placed in output strings that are passed to methods such as println(). This meant that developers needed to be proficient Java programmers as well as HTML programmers.

In contrast, JSP is written mainly in HTML code with directives and other JSP constructs interspersed in the HTML. This means that an HTML developer could write a JSP program without being proficient in Java programming. JSP directives and other JSP constructs provide flow control within the JSP and are used to insert data into HTML code, among other things. JSP is translated into a Java servlet before the code is executed. You'll learn more on how this works later in the book.

Although JSP was originally designed to generate dynamic HTML pages, JSP is also used to dynamically generate an XML document. This is illustrated in Listing 9-2. In this example, the JSP is generating the same book template XML document that is created by the Java servlet in Listing 9-1.

Notice that Listing 9-2 doesn't contain any output methods such as println(), which is used in the Java servlet. This is because the entire content of the JSP is output except for JSP directives and other JSP constructs.

The first line of the JSP is the only directive used in this example. The directive <%@ page contentType="text/xml" %> sends an HTTP header that tells the browser what type of content to expect. The content type identifies the markup language used in the document. The default content type is "text/html." This means that a directive isn't necessary to change the content type if the JSP is generating an HTML page. However, the content type must be explicitly set since this JSP is generating an XML document.

Listing 9-2 Creating an XML document sing a JSP.

```
<%@ page contentType="text/xml" %>
<?xml version="1.0"?>
<book>
  <TitlePage>
  J2EE, The Complete Reference
  </TitlePage>
  <CopyWritePage>
  </CopyWritePage>
  <TableOfContents>
```

```
    </TableOfContents>
    <Preface>
    </Preface>
    <Chapter>
    </Chapter>
    <Chapter>
    </Chapter>
</book>
```

Parsing XML

An XML document contains information and tags that identify information, as you learned in the previous chapter. Any application that reads an XML document must parse information from the document in order to access information contained in the document.

Writing an application that parses a document is challenging and time-consuming. However, there are parsers available that are designed to parse an XML document and are accessible through an application programming interface (API). The parser API consists of Java classes that provide easy access to elements of an XML document.

There are two types of XML parsers. These are the Document Object Model (DOM) and the Simple API for XML (SAX). The DOM parser reads the entire XML document, which makes it easy to navigate the document. Although the DOM parser can be used with any size XML documents, the DOM parser is ideal for working with smaller XML documents. Larger documents can be slow to process using a DOM parser.

The SAX parser uses an event strategy. A J2EE component that interacts with a SAX parser to manipulate an XML document must contain event-handler classes that listen for specific events sent by the SAX parser as the SAX parser reads the XML document. Events are triggered whenever the SAX parser encounters an XML tag. The SAX parser is ideal for large XML documents, although the J2EE component might experience difficulties navigating a complex XML document because the SAX parser is event driven. The J2EE component doesn't have control of the parsing once the SAX parser is invoked. That is, the J2EE component must wait until the SAX parser has completed processing the XML document before the J2EE component can search the document.

In the real world, DOM is generally used to build and alter XML documents and SAX is used to read XML documents.

XML parsers can be used to parse any valid XML document, but are inappropriate for parsing non-XML documents such as a DTD (see Chapter 8). In this chapter, you'll learn how to parse an XML document using both DOM and SAX. Listing 9-4 contains the XML document that will be parsed in this chapter. Listing 9-3 is the DTD used for the XML document.

Listing 9-3 The book.dtd used for the book.htm.

```
<!ELEMENT book (chapter+)>
<!ELEMENT chapter (chapNum,chapTitle)>
```

```
<!ELEMENT chapNum (#PCDATA)>
<!ELEMENT chapTitle (#PCDATA)>
```

Listing 9-4 The book.xml that is being parsed.

```
<?xml version="1.0"?>
<!DOCTYPE book SYSTEM "book.dtd">
<book>
<chapter>
<chapNum>One</chapNum>
<chapTitle>Introduction to J2EE</chapTitle>
</chapter>
</book>
```

Document Object Model (DOM)

The DOM parser uses five classes that are used to navigate and parse elements of an XML document. These classes are the Node class, Document class, Element class, CharacterData class, and Attr class. These classes are introduced in this section and are illustrated in Listing 9-5.

A node is an object of the XML document and is not necessarily associated with an XML tag, such as text nodes. The Node class is a superclass so objects can share some common functionality. The values of a node name, node value, and attributes vary according to the node type. Table 9-1 lists the description of node names, node values, and attributes.

Interface	Name	Value	Attributes
Attr	Name of attribute	Value of attribute	null
CDATASection	"#cdata-section"	Content of the CDATASection	null
Comment	"#comment"	Content of the comment	null
Document	"#document"	null	null
DocumentFragment	"#document-fragment"	null	null
DocumentType	Document type name	null	null

Table 9-1. *Node Names, Node Values, and Attributes*

Interface	Name	Value	Attributes
Element	Tag name	null	NamedNode Map
Entity	Entity name	null	null
EntityReference	Name of entity referenced	null	null
Notation	Notation name	null	null
ProcessingInstruction	Target	Entire content excluding the target	null
Text	"#text"	Content of the text node	null

Table 9-1. *Node Names, Node Values, and Attributes* (continued)

A node can have a value that is either the text of the node or the value of an attribute, depending on the nature of the node. The node text is the characters that appear between the open and close XML tags.

There are different kinds of objects within an XML document as you learned in Chapter 8. Each object has an associated note type. There are a number of standard node types that are identified by constants. These are shown in Table 9-2.

Constant	Node
static short ATTRIBUTE_NODE	The node is an Attr.
static short CDATA_SELECTION_NODE	The node is a CDATASection.
static short COMMENT_NODE	The node is a Comment.
static short DOCUMENT_FRAGMENT_NODE	The node is a DocumentFragment.

Table 9-2. *The Commonly Used Node Constants*

Constant	Node
static short DOCUMENT_NODE	The node is a Document.
static short DOCUMENT_TYPE_NODE	The node is a DocumentType.
static short ELEMENT_NODE	The node is an Element.
static short ENTITY_NODE	The node is an Entity.
static short ENTITY_REFERENCE_NODE	The node is an EntityReference.
static short NOTATION_NODE	The node is a Notation.
static short PROCESSING_INSTRUCTION_NODE	The node is a ProcessingInstruction.
static short TEXT_NODE	The node is a Text node.

Table 9-2. *The Commonly Used Node Constants* (continued)

There are three commonly used methods of the Node class that are used to parse information about a node. These are getNodeName(), getNodeValue(), and getNodeType(). Refer to Table 9-1 to determine the return value from each of these methods. The node type is returned as a short integer from the getNodeType() method. The short integer is equal to a node constant (see Table 9-2). The getNodeValue() method throws DOMExceptions.

Navigating The XML Document

Nodes in an XML document appear in a parent/child relationship. For example, a <Chapter> node contains <chapNum> and <chapTitle> nodes, which are child nodes of the <Chapter> node.

A NodeList can be retrieved by using the getChildNodes() method of the NodeList class. The list of child nodes is returned in a construct that is similar to an iterator. The getLength() method is called to determine the number of child nodes on the list. The getLength() method returns an int. The item() method is called to return a specific node, which is identified by the node's int position on the node list. For example, item(4) returns the fifth node on the list.

You can return a node without knowing the item number of the node by using one of a variety of methods. The getFirstChild() and getLastChild() methods return the first and last child node, respectively. The getNextSibling() and getPreviousSibling() methods return the next and previous nodes that have the same parent node as the current node. And you can retrieve the parent node of the current node by using the getParentNode() method, and retrieve the attribute of a node by using the getAttributes() method.

Modifying the Node List

The node list can be modified by using methods that insert, remove, and replace a child node. There are two methods used to insert a child node. You can also use importNode() from the Document class. These are the appendChild() and insertBefore() methods. The appendChild() method requires one argument, which is the node that is appended to the parent nodes.

The insertBefore() method requires two arguments, both of which are nodes. The first argument is the node that is being inserted into the XML document and the other argument is the node ahead of which the new node will be inserted. If the second argument is null, the node will be appended.

Let's say that you wanted to insert the <ChapNum> node above the <ChapTitle> node. The first argument in the insertBefore() method is the <ChapNum> node and the second argument is the <ChapTitle> node.

A node can be removed or replaced by using the removeChild() method and the replaceChild() method. The removeChild() method requires one argument, which is the node that is to be removed. The replaceChild() method requires two arguments. The first argument is the new child node that will replace the existing child node. The second argument is the existing child node that is being replaced.

Caution *The underlying NodeList is modified if you ran a method that returns a NodeList and then you started adding and removing nodes. Let's say that NodeList nl contains elements that are children of Element called ele. You could try to remove the children by using the following code segment, but only half the children will be removed because the getLength() method returns a different value in each iteration:*

```
for(int i=0; i<nl.getLength(); i++)
{
        e.removeChild(nl.item(i));
}
```

You can reference a node by name using the getNamedItem() and setNamedItem() methods. The getNamedItem() method has one argument, which is a String that contains the name of the item, and returns the node. The setNamedItem() is used to append a new node. The setNamedItem() requires one argument, which is the new item. A node can be removed by using the removeNamedItem() where the node that is being removed is the argument to the removeNamedItem() method.

Document-Level Manipulations

A J2EE component can create and retrieve components of an XML document by using the Document class, which is derived from the Node class. You create a new element in the XML document by using the createElement() method. The createElement() method requires one argument, which is a String that contains the name of the element's tag.

You can create an attribute, comment, text node, and CDATA section using methods in the Document class. These are createAttribute(), createComment(), createTextNode(),

and createCDATASection(). Each of these methods requires one argument, which is a String that contains the attribute name, the comment, text for the text node, and text for the CDATASection. Furthermore, createAttribute() and createCDATASection() methods throw a DOMException.

Two of the more commonly used methods to retrieve information from an XML document are getElementsByTagName() and getDocumentElement(). The getElementsByTagName() method requires one argument, which is a String that contains the name of the element. The getDocumentElement() method returns an Element that is the root element of the document.

Element- and Attribute-Level Manipulations

Element-level manipulations involve interacting with attributes of an element. There are three tasks that are commonly performed using an attribute. These are to retrieve the value of an attribute, to change the value of an attribute, and to remove the value of an attribute.

The value of an attribute is retrieved by using the getAttribute() method. The getAttribute() method requires one argument, which is a String that contains the name of the attribute. The value of the attribute is returned as String from the getAttribute() method.

The setAttribute() method is used to change the value of an attribute. Two arguments are required by the setAttribute() method. The first argument is a String that contains the name of the attribute whose value is being changed. The second argument is a String that contains the new value of the attribute. If an attribute with that name already exists, the attribute's value is changed to the new value.

The value of an attribute can be removed by using the removeAttribute() method. The removeAttribute() method requires one argument, which is a String that contains the name of the attribute whose value is to be removed. If the attribute has a default value, an attribute immediately appears with the default value. The setAttribute() and removeAttribute() methods throw a DOMException.

Attributes can also be manipulated by using the Attr class. The Attr class contains the getValue() method to retrieve the value of an attribute and a setValue() method to replace the value of an attribute. The setValue() method has one argument, which is a String that contains the new value of the attribute. You don't need to specify the attribute in these methods because the attribute is identified when the instance of the Attr class is created.

Two other commonly used methods of the Attr class are the getName() method and the getSpecified() method. The getName() method returns a String that contains the name of the attribute. The getSpecified() method returns a Boolean value that indicates if the attribute value is the default value specified in the DTD (see Chapter 8) or a value that is explicitly entered into the attribute. A Boolean true indicates the attribute is explicitly set.

Data-Level Manipulations

Data contained in elements of an XML document are treated as character sequences that are accessed by using the CharacterData class. The CharacterData class contains methods that enable you to determine the length of data and to retrieve data, append data, insert data, replace data, and remove data from the element. All these methods throw a DOMException.

The length of the data is determined by calling the getLength() method. The getLength() method returns an int that represents the number of characters contained in the data. Data can be retrieved from the XML document by using the getData() method for the substringData() method.

The getData() method returns a String that contains the entire sequence of characters in the data. The substringData() method is used to retrieve a portion of data. Two arguments are required by the substringData() method. The first argument is an int that represents the character offset of the first character that is to be retrieved from the data. The second argument is an int that represents the number of characters from the character offset that are to be retrieved. The substringData() method returns a String that contains the substring of the data.

There are four ways in which to place new data into an element. These are to append data, insert data, replace some data, and replace all the data. The appendData() method is used to append data to data that is currently in the element. The appendData() method requires one argument, which is a String that contains the data that will be appended to the existing data.

The insertData() method is used to place data within the existing data. Two arguments are required by the insertData() method. The first argument is an int that represents the character offset where the new data is to be inserted. The second argument is a String that contains that data to be inserted. Let's say that the new data is to be inserted after the fifth character of the current data. The first argument of the insertData() method is 5 and, of course, the second argument is the new data.

The replaceData() method is used to replace some data in an element. The replaceData() method requires three arguments. The first argument is an int that represents the character offset where the new data is to be inserted. The second argument is an int that represents the number of characters that are being replaced. And the third argument is a String that contains data that replaces a portion of the existing data. If you want to replace the entire data of an element, then you'll need to use the setData() method. The setData() method requires one argument, which is a String that contains the new data.

All or some of the data can be removed from an element by using the deleteData() method. The deleteData() method requires two arguments. The first argument is an int that represents the character offset of the first character that is to be removed from the data. The second argument is an int that represents the number of characters from the character offset that are to be removed.

DOM Parsing in Action

Listing 9-5 illustrates the technique used to parse an XML document using DOM. In this example, the book.xml document shown in Listing 9-4 is being parsed. The program begins by creating an instance of the DocumentBuilderFactory, which is called fact1 and uses the setValidating() method to have the DocumentBuilderFactory object create a parse that validates the XML document. The setIgnoringElementContentWhitespace() method is called and passed a Boolean true so whitespace characters won't cause a validation error.

Next, an instance of a DocumentBuilder is created called build1. This is basically the parser. A String object is declared and assigned the name of the XML document that will be parsed, which is book.xml. The String is passed to the File() method that returns the handle to the book.xml file to the parse() method. The parse() method retrieves the XML document and returns an instance of the Document object, which is called bookDoc. The bookDoc is the reference to the book.xml document and is used with methods that manipulate the document.

The first step after the book.xml is accessed is to retrieve the root element of the document. This is accomplished by using the getDocumentElement() method that returns an instance of the Element object called bookEle.

All other elements of the XML document are child nodes to the root element. This means the remaining elements can be retrieved by using the getChildNodes() method that returns an instance of the NodeList object called bookNodes.

The program steps through the node list by using a for loop. The total number of items in the node list, as returned by the getLength() method, is used to terminate the for loop. Each item is retrieved using the item() method that returns an instance of the Element object, which is called chapter. The program then displays the name of the node.

Next, the program calls the getElementsByTagName() method, passing it the chapNum tag name. This method returns a NodeList called numberList. The getFirstChild() method is then called to return the first child. The getData() method is then called to return the value of the first child, which is displayed by the program. This same process is used with the chapTitle tag. Any errors that occur during the execution of these methods are trapped by the catch{ } block.

Listing 9-5 Parsing the book.xml using DOM.

```
import java.io.*;
import javax.xml.parsers.*;
import org.w3c.dom.*;
public class DOMBookParser
{
  public static void main(String[] args)
  {
   try
   {
     DocumentBuilderFactory fact1= DocumentBuilderFactory.newInstance();
     fact1.setValidating(true);
     fact1.setIgnoringElementContentWhitespace(true);
     DocumentBuilder build1 = fact1.newDocumentBuilder();
```

```
String book1 = "book.xml";
Document bookDoc = build1.parse(new File(book1));
Element bookEle = bookDoc.getDocumentElement();
NodeList chapterNodes = bookEle.getChildNodes();
for (int i=0; i < chapterNodes.getLength(); i++)
{
  Element chapter = (Element) chapterNodes.item(i);
  System.out.print("Value: " + chapter.getNodeName() + " ");
  NodeList numberList = chapter.getElementsByTagName("chapNum");
  Text number = (Text) numberList.item(0).getFirstChild();
  System.out.print(number.getData() + " ");
  NodeList titleList = chapter.getElementsByTagName("chapTitle");
  Text title = (Text) titleList.item(0).getFirstChild();
  System.out.println(title.getData());
  }
 }
 catch (Exception error)
 {
  System.err.println("Error parsing:" + error.getMessage());
  System.exit(1);
 }
 }
}
```

Simple API for XML (SAX)

The SAX parser is event driven, as mentioned previously in this chapter. This means that a J2EE component that uses the SAX parser must contain an event handler that listens to events generated by the SAX parser. An event is generated each time the SAX parser encounters a component of the XML document. The event handler must be able to react to events that you want to handle.

Listing 9-6 illustrates how to parse the book.xml (see Listing 9-4). This example begins by creating Boolean and StringBuffer data members of the SAXBookParser class. Each pair is used for elements that are to be parsed from the book.xml document.

The try{ } block in main() begins similar to the try{ } block in the DOM parser example in that a factory is created, the factory is designated as a validating parser, and then an instance of the SAXParser is created.

The factory is established by creating an instance of the SAXParserFactory class, which is called fact1. Then an instance of the SAXParser called build1 is declared, which is the SAXParser. A String object is declared called book1 and is assigned the name of the XML document that will be parsed, which is book.xml.

The similarities between the DOM parser and the SAX parser differ at this point in the program. In the SAX parser example, the program creates an event listener called event, which is an instance of the DefaultHandler. The event listener is associated with the SAXBookParser class.

Next, the program starts the parser by calling the parse() method. The parse() method requires two arguments. The first argument is the handle to the XML document, which is returned by the File() method. The second argument is the event listener that will respond to events generated by the SAX parser.

The program reacts to three events. These are the start of an Element, characters that are associated with anything that inherits from CharacterData, and the end of an element. You must define methods only for events that you want to handle. The SAX parser also generates an event at the beginning and end of the document. These are handled by the startDocument() and endDocument() methods, respectively. These methods are defined in a program if there is a need to implement procedures at the beginning and end of a document, which is not the case in Listing 9-6.

The startElement() method is automatically called when the parser encounters the beginning of an element in the XML document. Four String arguments are passed to the startElement() method. The first argument is the URI, the second argument is the local name, the third argument is the qName, and the last argument contains the attributes.

The startElement() method must contain logic to address specific elements. Listing 9-6 addresses the chapter, chapTitle, and chapNum elements that are found in the book.xml document.

If the SAX parser encounters the beginning of a parent element such as a chapter element, the StringBuffers for the parent and children are set to null, simply removing the content stored from the previous chapter element.

If the SAX parser encounters the beginning of a chapTitle element, the Boolean value of pChapterTitle is set to true indicating that a new chapTitle element is detected and a new StringBuffer is created.

Likewise, a similar process occurs if the SAX parser encounters the beginning of a chapNum element. In this case, the boolean value of pChapNum is set to true and a new StringBuffer is created.

The characters() method is called by the parser whenever text of an element is encountered. There are three arguments passed to the characters() method. The first argument is a character array that contains the text of the element. The second argument is an integer that represents the starting position of the text within the character array. And the third argument is an integer that represents the number of characters of the text.

The characters() method in Listing 9-6 appends the text to the StringBuffer of the appropriate element. The characters() method evaluates the value of Boolean data members (that is, pChapTitle, pChapNum) to determine the proper StringBuffer to append. Arguments passed to the characters() method are then passed to the append() method of the respective StringBuffer. Data is appended because the characters() method might be called multiple times for the same data stream. You shouldn't assume that the entire character stream is passed by one method call.

The endElement() method is called whenever the parser generates the end of element event, which occurs when a closing tag is read from the XML document. The endElement() method in Listing 9-6 is passed three String objects. The first String object is the namespace URI. The second String object is the local name, and the last String object is the qName.

The endElement() method converts the StringBuffer to a String and then displays the String if the name of the element passed to the endElement is the parent element, which is chapter in this example. However, if the element name isn't the parent, then the parse encountered the end of one of the parent's children. Therefore, the value of the Boolean data member for this child is reset to false.

Listing 9-6 Parsing the book.xml using SAX

```
import java.io.*;
import javax.xml.parsers.*;
import org.xml.sax.*;
import org.xml.sax.helpers.*;
public class SAXBookParser extends DefaultHandler
{
  protected boolean pChapNum;
  protected boolean pChapTitle;
  protected StringBuffer cChapNum;
  protected StringBuffer cChapTitle;
  public static void main(String[] args)
  {
    try
    {
      SAXParserFactory fact1 = SAXParserFactory.newInstance();
      fact1.setValidating(true);
      SAXParser build1 = fact1.newSAXParser();
      String book1= "book.xml";
      SAXBookParser event = new SAXBookParser();
      build1.parse(new File(book1), event);
    }
    catch (Exception error)
    {
      System.err.println("Error parsing: " + error.getMessage());
      System.exit(1);
    }
}
public void startElement(String uri, String localName, String qName, Attributes attributes)
{
  if (qName.compareTo("chapter") == 0)
  {
    cChapTitle = null;
    cChapNum = null;
  }
  else if (qName.compareTo("chapNum") == 0)
  {
    pChapNum = true;
    cChapNum = new StringBuffer();
  }
  else if (qName.compareTo("chapTitle") == 0)
  {
    pChapTitle = true;
    cChapTitle = new StringBuffer();
```

```
    }
  }
  public void characters(char[] cha, int start, int length)
  {
    if (pChapTitle)
    {
      cChapTitle.append(cha, start, length);
    }
    else if (pChapNum)
    {
      cChapNum.append(cha, start, length);
    }
  }
  public void endElement(String namespaceURI, String localName, String qName)
  {
    if (qName.compareTo("chapter") == 0)
    {
      System.out.print("Chapter ");
      if (cChapNum != null)
      {
        System.out.print(cChapNum.toString());
        System.out.print(" ");
      }
      if (cChapTitle != null)
      {
        System.out.print(cChapTitle.toString());
      }
      System.out.println();
    }
    else if (qName.compareTo("chapTitle") == 0)
    {
      pChapTitle = false;
    }
    else if (qName.compareTo("chapNum") == 0)
    {
      pChapNum = false;
    }
  }
}
```

Quick Reference Guide

Java classes and interfaces used to interact with an XML document have many methods, the most common of which are discussed in this chapter. This quick reference guide provides a brief overview of other methods that can be used. Full details of these methods and all Java classes and interfaces designed to be used with an XML document are available at java.sun.com. DOM and SAX are both contained in JDK 1.4.

Note *The HandlerBase has been depreciated.*

Syntax	Descriptions
public void parse(InputStream is, HandlerBase hb) throws SAXException, IOException	Parse the content of the given InputStream instance as XML.
public void parse(InputStream is, HandlerBase hb, String systemId) throws SAXException, IOException	Parse the content of the given InputStream instance as XML using the specified HandlerBase.
public void parse(InputStream is, DefaultHandler dh) throws SAXException, IOException	Parse the content of the given InputStream instance as XML using the specified DefaultHandler.
public void parse(DefaultHandler InputStream is, DefaultHandler dh, DefaultHandler String systemId) throws SAXException, IOException	Parse the content of the given InputStream instance as XML using the specified DefaultHandler.
public void parse(String uri, DefaultHandler dh) throws SAXException, IOException	Parse the content described by the giving Uniform Resource Identifier (URI) as XML using the specified DefaultHandler.
public void parse(File f, DefaultHandler dh) throws SAXException, IOException	Parse the content of the file specified as XML using the specified DefaultHandler.
public void parse(InputSource is, DefaultHandler dh) throws SAXException, IOException	Parse the content given InputSource as XML using the specified DefaultHandler.
public abstract Parser getParser() throws SAXException	Returns the SAX parser that is encapsulated by the implementation of this class.
public abstract XMLReader getXMLReader() throws SAXException	Returns the XMLReader that is encapsulated by the implementation of this class.
public abstract boolean isNamespaceAware()	Indicates whether or not this parser is configured to understand namespaces.
public abstract boolean isValidating()	Indicates whether or not this parser is configured to validate XML documents.

Table 9-3. *javax.xml.parsers Public Abstract Class SAXParser*

Syntax	Descriptions
public abstract void setProperty(String name, Object value) throws SAXNotRecoginzedException, SAXNotSupportedException	Sets the particular property in the underlying implementation of XMLReader.
public abstract Object getProperty(String name) throws SAXNotRecoginzedException, SAXNotSupportedException	Returns the particular property requested in the underlying implementation of XMLReader.

Table 9-3. *javax.xml.parsers Public Abstract Class SAXParser* (continued)

Syntax	Descriptions
public String getName()	Returns the name of this attribute.
public boolean getSpecified()	If this attribute was explicitly given a value in the original document, this is true; otherwise, it is false.
public String getValue()	On retrieval, the value of the attribute is returned as a string.
public void setValue(String value) throws DOMException	On retrieval, the value of the attribute is returned as a string. Character and general entity references are replaced with their values.
public Element getOwnerElement()	The Element node this attribute is attached to, or null if this attribute is not in use.

Table 9-4. *org.w3c.dom Public Interface Attr*

Syntax	Descriptions
public Documenttype getDoctype()	The document type declaration associated with this document.
public DOMImplementation getImplementation()	The DOMImplementation object that handles this document.
public Element getDocumentElement()	This is a convenience attribute that allows direct access to the child node that is the root element of the document.
public Element createElement(String tagName) throws DOMException	Creates an Element with the specified name. Note that the instance returned implements the Element interface, so attributes can be specified directly on the returned object.
public DocumentFragment createDocumentFragment()	Creates an empty DocumentFragment object.
public Text createTextNode (String data)	Creates a Text node given the specified string.
public Comment createComment(String data)	Creates a Comment node given the specified string.
public CDATASection createCDATASection(String data) throws DOMException	Creates a CDATASection node whose value is the specified string.
public ProcessingInstruction createProcessingInstruction (String target, String data) throws DOMException	Creates a ProcessingInstruction node given the specified name and data strings.
public Attr createAttribute (String name) throws DOMException	Creates an Attr of the given name. Note that the Attr instance can then be set on an Element using the setAttributeNode method.

Table 9-5. *org.w3c.dom* ***public Interface Document***

Syntax	Descriptions
public EntityReference createEntityReference(String name) throws DOMException	Creates an EntityReference object. In addition, if the referenced entity is known, the child list of the EntityReference node is made the same as that of the corresponding Entity node.
public NodeList getElementsByTagName (String tagname)	Returns a NodeList of all the Elements with a given tag name in the order in which they are encountered in a preorder traversal of the Document tree.
public Node importNode(Node importedNode, boolean deep) throws DOMException	Imports a node from another document to this document. The returned node has no parent (parentNode is null).
public Element createElementNS(String namespaceURI, String qualifiedName) throws DOMException	Creates an element of the given qualified name and namespace URI.
public Attr createAttributeNS(String namespaceURI, String qualifiedName) throws DOMException	Creates an attribute of the given qualified name and namespace URI.
public NodeList getElementsByTagNameNS (String namespaceURI, String localName)	Returns a NodeList of all the Elements with a given local name and namespace URI in the order in which they are encountered in a preorder traversal of the Document tree.
public ElementgetElementById (String elementId)	Returns the Element whose ID is given by elementId. If no such element exists, returns null.

Table 9-5. *org.w3c.dom* ***public interface Document*** (continued)

Syntax	Descriptions
public String getTagName()	The name of the element.
public String getAttribute(String name)	Retrieves an attribute value by name.
public void setAttribute(String name, String value) throws DOMException	Adds a new attribute.
public void removeAttribute(String name) throws DOMException	Removes an attribute by name.
public Attr getAttributeNode (String name)	Retrieves an attribute node by name.
public Attr setAttributeNode(AttrnewAttr) throws DOMException	Adds a new attribute node.
public Attr removeAttributeNode(AttroldAttr) throws DOMException	Removes the specified attribute node.
public NodeList getElementsByTagName (String name)	Returns a NodeList of all descendant Elements with a given tag name, in the order in which they are encountered in a preorder traversal of this Element tree.
public String getAttributeNS(String namespaceURI, String localName)	Retrieves an attribute value by local name and namespace URI.
public void setAttributeNS(DOMException String namespaceURI, DOMException String qualifiedName, DOMException String value) throws DOMException	Adds a new attribute.
public void removeAttributeNS(String namespaceURI, String localName) throws DOMException	Removes an attribute by local name and namespace URI.
public Attr getAttributeNodeNS(String namespaceURI, String localName)	Retrieves an Attr node by local name and namespace URI.

Table 9-6. *org.w3c.dom Public Interface Element*

Syntax	Descriptions
public Attr setAttributeNodeNS(Attr newAttr) throws DOMException	Adds a new attribute.
public NodeList getElementsByTagNameNS(String namespaceURI, String localName)	Returns a NodeList of the entire descendant Elements with a given local name and namespace URI in the order in which they are encountered in a preorder traversal of this Element tree.
public boolean hasAttribute (String name)	Returns true when an attribute with a given name is specified on this element or has a default value, and false otherwise.
public boolean hasAttributeNS(String namespaceURI, String localName)	Returns true when an attribute with a given local name and namespace URI is specified on this element or has a default value, and false otherwise.

Table 9-6. *org.w3c.dom Public Interface Element* (continued)

Syntax	Descriptions
String getNodeName()	Returns the name of a node.
String getNodeValue() throws DOMException	Returns the value of a node.
public void setNodeValue(String nodeValue) throws DOMException	The value of a node.
public short getNodeType()	Returns an int of the node type.
public Node getParentNode()	Returns the parent node.
public NodeList getChildNodes()	Returns children nodes.
public Node getFirstChild()	Returns the first child of a node.
public Node getLastChild()	Returns the last child of a node.

Table 9-7. *org.w3c.dom Public Interface Node*

Syntax	Descriptions
public Node getPreviousSibling()	Returns the node immediately preceding a node.
public Node getNextSibling()	Returns the node immediately following a node.
public NameNodeMap getAttributes()	Returns the NamedNodeMap containing the attributes of a node (if it is an Element), or null otherwise.
public Document getOwnerDocument()	Returns the Document object associated with a node.
public Node insertBefore(Node newChild, Node refChild) throws DOMException	Inserts the node newChild before the existing child node refChild.
public Node replaceChild(Node newChild, Node oldChild) throws DOMException	Replaces the child node oldChild with newChild in the list of children, and returns the oldChild node.
public Node removeChild(Node oldChild) throws DOMException	Removes the child node indicated by oldChild from the list of children, and returns it.
public Node appendChild(Node newChild) throws DOMException	Adds the node newChild to the end of the list of children of this node.
public boolean hasChildNodes()	Returns whether this node has any children.
public Node cloneNode(boolean deep)	Returns a duplicate of this node and serves as a generic copy constructor for nodes.
public void normalize()	Places Text nodes in the full depth of the subtree underneath this Node, including attribute nodes, into a "normal" form where only structure separates Text nodes.
public String getNamespaceURI()	Returns the namespace URI of this node, or null if it is unspecified.

Table 9-7. *org.w3c.dom Public Interface Node* (continued)

Syntax	Descriptions
public String getPrefix()	Returns the namespace prefix of this node, or null if it is unspecified.
public void setPrefix(String prefix) throws DOMException	Returns the namespace prefix of this node, or null if it is unspecified.
public String getLocalName()	Returns the local part of the qualified name of this node.
public boolean hasAttributes()	Returns whether this node (if it is an element) has any attributes.

Table 9-7. *org.w3c.dom Public Interface Node* (continued)

Syntax	Descriptions
public int getLength()	Return the number of attributes in the list.
public String getURI(int index)	Look up an attribute's namespace URI by index.
public String getLocalName (int index)	Look up an attribute's local name by index.
public String getQName(int index)	Look up an attribute's XML 1.0 qualified name by index.
public String getType(int index)	Look up an attribute's type by index.
public String getValue(int index)	Look up an attribute's value by index.
public int getIndex(String uri, String localPart)	Look up the index of an attribute by namespace name.
public int getIndex(String qName)	Look up the index of an attribute by XML 1.0 qualified name.
public String getType(String uri, String localName)	Look up an attribute's type by namespace name.
public String getType(String Name)	Look up an attribute's type by XML 1.0 qualified name.

Table 9-8. *org.xml.sax Public Interface Attributes*

Syntax	Descriptions
public String getValue(String uri, String localName)	Look up an attribute's value by namespace name.
public String getValue(String Name)	Look up an attribute's value by XML.1.0 qualified name.

Table 9-8. *org.xml.sax Public Interface Attributes* (continued)

Syntax	Descriptions
public void setDocumentLocator(Locator locator)	Receive an object for locating the origin of SAX document events.
public void startDocument() throws SAXException	Receive notification of the beginning of a document.
public void endDocument() throws SAXException	Receive notification of the end of a document.
public void startPrefixMapping(String prefix, String uri) throws SAXException	Begin the scope of a prefix-URI namespace mapping.
public void endPrefixMapping(String prefix) throws SAXException	End the scope of a prefix-URI mapping.
public void startElement(String namespaceURI, String localName, String Name, Attributes atts) throws SAXException	Receive notification of the beginning of an element.
public void endElement(String namespaceURI, String localName, String Name) throws SAXException	Receive notification of the end of an element.
public void characters(char[] ch, int start, int length) throws SAXException	Receive notification of character data.

Table 9-9. *org.xml.sax Public Interface ContentHandler*

Syntax	Descriptions
public void ignorableWhitespace(char[] ch, int start, int length) throws SAXException	Receive notification of ignorable whitespace in element content.
public void processingInstruction(String target, String data) throws SAXException	Receive notification of a processing instruction.
public void skippedEntity(java.lang.String name) throws SAXException	Receive notification of a skipped entity.

Table 9-9. *org.xml.sax Public Interface ContentHandler* (continued)

Syntax	Descriptions
public boolean getFeature(String name) throws SAXNotRecognizedException, SAXNotSupportedException	Look up the value of a feature.
public void setFeature(String name, boolean value) throws SAXNotRecognizedException, SAXNotSupportedException	Set the state of a feature.
Object getProperty(String name) throws SAXNotRecognizedException, SAXNotSupportedException	Look up the value of a property.
public void setProperty (String name, Object value) throws SAXNotRecognizedException, SAXNotSupportedException	Set the value of a property.
public void setEntityResolver(EntityResolver resolver)	Allow an application to register an entity resolver.

Table 9-10. *org.xml.sax Public Interface XMLReader*

Syntax	Descriptions
public EntityResolver getEntityResolver()	Return the current entity resolver.
public void setDTDHandler (DTDHandler handler)	Allow an application to register a DTD event handler.
public DTDHandler getDTDHandler()	Return the current DTD handler.
public void setContentHandler (ContentHandler handler)	Allow an application to register a content event handler.
public ContentHandler getContentHandler()	Return the current content handler.
public void setErrorHandler (ErrorHandler handler)	Allow an application to register an error event handler.
public ErrorHandler getErrorHandler()	Return the current error handler.
public void parse(InputSource input) throws java.io.IOException, SAXException	Parse an XML document.
public void parse(String systemId) throws java.io.IOException, SAXException	Parse an XML document from a system identifier (URI).

Table 9-10. *org.xml.sax Public Interface XMLReader* (continued)

Syntax	Descriptions
public DefaultHandler()	constructor
public InputSource resolveEntity(String publicId, String systemId) throws SAXException	Resolve an external entity.
public void notationDecl(String name, String publicId, String systemId) throws SAXException	Receive notification of a notation declaration.
public void unparsedEntityDecl(String name, String publicId, String systemId, String notationName) throws SAXException	Receive notification of an unparsed entity declaration.

Table 9-11. *org.xml.sax.helpers Public Class DefaultHandler*

Syntax	Descriptions
public void setDocumentLocator (Locator locator)	Receive a Locator object for document events.
public void startDocument() throws SAXException	Receive notification of the beginning of the document.
public void endDocument() throws SAXException	Receive notification of the end of the document.
public void startPrefixMapping(String prefix, String uri) throws SAXException	Receive notification of the start of a namespace mapping.
public void endPrefixMapping(String prefix) throws SAXException	Receive notification of the end of a namespace mapping.
public void startElement(String uri, String localName, String qName, Attributes attributes) throws SAXException	Receive notification of the start of an element.
public void endElement(String uri, String localName, String qName) throws SAXException	Receive notification of the end of an element.
public void characters(char[] ch, int start, int length) throws SAXException	Receive notification of character data inside an element.
public void ignorableWhitespace(char[] ch, int start, int length) throws SAXException	Receive notification of ignorable whitespace in element content.
public void processingInstruction(String target, String data) throws SAXException	Receive notification of a processing instruction.
public void skippedEntity(String name) throws SAXException	Receive notification of a skipped entity.
public void warning(SAXParseException e) throws SAXException	Receive notification of a parser warning.
public void error(SAXParseException e) throws SAXException	Receive notification of a recoverable parser error.
public void fatalError(SAXParseException e) throws SAXException	Report a fatal XML parsing error.

Table 9-11. *org.xml.sax.helpers Public Class DefaultHandler* (continued)

Syntax	Descriptions
public ParserAdapter() throws SAXException	Create a new parser adapter.
public ParserAdapter(Parser parser)	Create a new parser adapter.
public void setFeature(String name, boolean state) throws SAXNotRecognizedException, SAXNotSupportedException	Set a feature for the parser.
public boolean getFeature(java.lang.String name) throws SAXNotRecognizedException, SAXNotSupportedException	Check a parser feature.
public void setProperty(java.lang.String name, java.lang.Object value) throws SAXNotRecognizedException, SAXNotSupportedException	Set a parser property.
public java.lang.Object getProperty(String name) throws SAXNotRecognizedException, SAXNotSupportedException	Returns a parser property.
public void setEntityResolver(EntityResolver resolver)	Set the entity resolver.
public EntityResolver getEntityResolver()	Returns the current entity resolver.
public void setDTDHandler(DTDHandler handler)	Set the DTD handler.
public DTDHandler getDTDHandler()	Returns the current DTD handler.
public void setContentHandler(ContentHandler handler)	Set the content handler.
public ContentHandler getContentHandler()	Returns the current content handler.

Table 9-12. *org.xml.sax.helpers Public Class ParserAdapter*

Syntax	Descriptions
public void setErrorHandler(ErrorHandler handler)	Set the error handler.
public ErrorHandler getErrorHandler()	Return the current error handler.
public void parse(String systemId) IOException, SAXException	Parse an XML document.
public void parse(InputSource input) throws IOException, SAXException	Parse an XML document.
public void setDocumentLocator(Locator locator)	Adapt a SAX1 document locator event.
public void startDocument() throws SAXException	Adapt a SAX1 start document event.
public void endDocument() throws SAXException	Adapt a SAX1 end document event.
public void startElement(java.lang.String qName, AttributeList Atts) throws SAXException	Adapt a SAX1 start element event.
public void endElement(java.lang.String qName) throws SAXException	Adapt a SAX1 end element event.
public void characters(char[] ch, int start, int length) throws SAXException	Adapt a SAX1 characters event.
public void ignorableWhitespace (char[] ch, int start, int length) throws SAXException	Adapt a SAX1 ignorable whitespace event.
public void processingInstruction (String target, String data) throws SAXException	Adapt a SAX1 processing instruction event.

Table 9-12. *org.xml.sax.helpers Public Class ParserAdapter* (continued)

Syntax	Descriptions
public XMLReaderAdapter() throws SAXException	Create a new adapter.
public XMLReaderAdapter(XMLReader xmlReader)	Create a new adapter.
public void setLocale(java.util.Locale locale) throws SAXException	Set the locale for error reporting.
public void setEntityResolver(EntityResolver resolver)	Register the entity resolver.
public void setDTDHandler(DTDHandler handler)	Register the DTD event handler.
public void setDocumentHandler(DocumentHandler handler)	Register the SAX1 document event handler.
public void setErrorHandler(ErrorHandler handler)	Register the error event handler.
public void parse(String systemId) throws IOException, SAXException	Parse the document.
public void parse(InputSource input) throws IOException, SAXException	Parse the document.
public void setDocumentLocator(Locator locator)	Set a document locator.
public void startDocument() throws SAXException	Start document event.
public void endDocument() throws SAXException	End document event.
public void startPrefixMapping(String prefix, String uri)	Adapt a SAX2 start prefix-mapping event.
public void endPrefixMapping (String prefix)	Adapt a SAX2 end prefix-mapping event.

Table 9-13. *org.xml.sax.helpers Public Class XMLReaderAdapter*

Syntax	Descriptions
public void startElement(String uri, String localName, String qName, Attributes atts) throws SAXException	Adapt a SAX2 start element event.
public void endElement(String uri, String localName, String qName) throws SAXException	Adapt a SAX2 end element event.
public void characters(char[] ch, int start, int length) throws SAXException	Adapt a SAX2 characters event.
public void ignorableWhitespace(char[] ch, int start, int length) throws SAXException	Adapt a SAX2 ignorable whitespace event.
public void processingInstruction(java.lang.String target, java.lang.String data) throws SAXException	Adapt a SAX2 processing instruction event.
public void skippedEntity(java.lang.String name) throws SAXException	Adapt a SAX2 skipped entity event.

Table 9-13. *org.xml.sax.helpers Public Class XMLReaderAdapter* (continued)

The Complete Reference



Chapter 10

Java servlets

A J2EE application is a compilation of components, each of which provides a particular service to the application, as discussed in Part I. The user interface is the highest level and most visible component, and is used to provide a person with an intuitive means for calling services.

The user interface gathers a request from a person who uses the J2EE application and forwards the request to a component for processing. The component that processes a request is built using either a Java servlet or a JavaServer Page.

A Java servlet is a server-side program that is called by the user interface or another J2EE component and contains the business logic to process a request. A Java ServerPage is also a server-side program that performs practically the same duties as a Java servlet using a different technique. Both a Java servlet and a Java ServerPage call other components that handle processing details.

This chapter focuses on Java servlets and how to build a J2EE component using Java servlet technology. The next chapter shows you how to create a J2EE component using Java server Pages.

Java servlets and Common Gateway Interface Programming

J2EE applications use a lightweight client, typically a browser, that contains little or no processing logic. A middle-layer program based on the web server handles all processing. Originally a server-side program handled client requests by using the Common Gateway Interface (CGI), which was programmed using Perl or other programming languages. Java servlets is Sun Microsystems, Inc.'s initial efforts to replace CGI programming.

Although Java servlet technology replaces CGI, Java servlets technology and CGI both provide the same basic functionality. That is, explicit and implicit data is sent from a client to a server-side program in the form of a request that is processed and another set of explicit implicit data is returned.

Explicit data is information received from the client that is typically either entered by the user into the user interface or generated by the user interface itself (see Chapter 8). For example, user ID and password are explicit data that can be generated by a browser, applet, or custom client.

Implicit data is Hypertext Transmission Protocol (HTTP) information that is generated by the client (that is, browser) rather than the user. HTTP information contains data about the request such as the compression scheme, cookies, and media types. HTTP information is used by the Java servlet (and by the client when the Java servlet returns the results) to learn about the request itself. A complete discussion of HTTP and HTTP information is presented later in this chapter.

Processing a request takes on many different forms that include calling a Java servlet, an Enterprise JavaBean, and possibly a legacy system. Let's say that a request is

made to display product information (see Figure 10-1). The user enters the product number, such as an ISBN used to uniquely identify a book into the user interface. The user interface calls a Java servlet, passing the Java servlet the product number. The Java servlet might call an Enterprise JavaBean to retrieve product information from a database, and returns this information to the Java servlet.

The result of processing a request is returned to the client as explicit data. Typically, this information is returned in the form of an HTML page that is dynamically generated by the Java servlet. The Java servlet also generates implicit HTTP information such as the type of document that is being returned to the client.

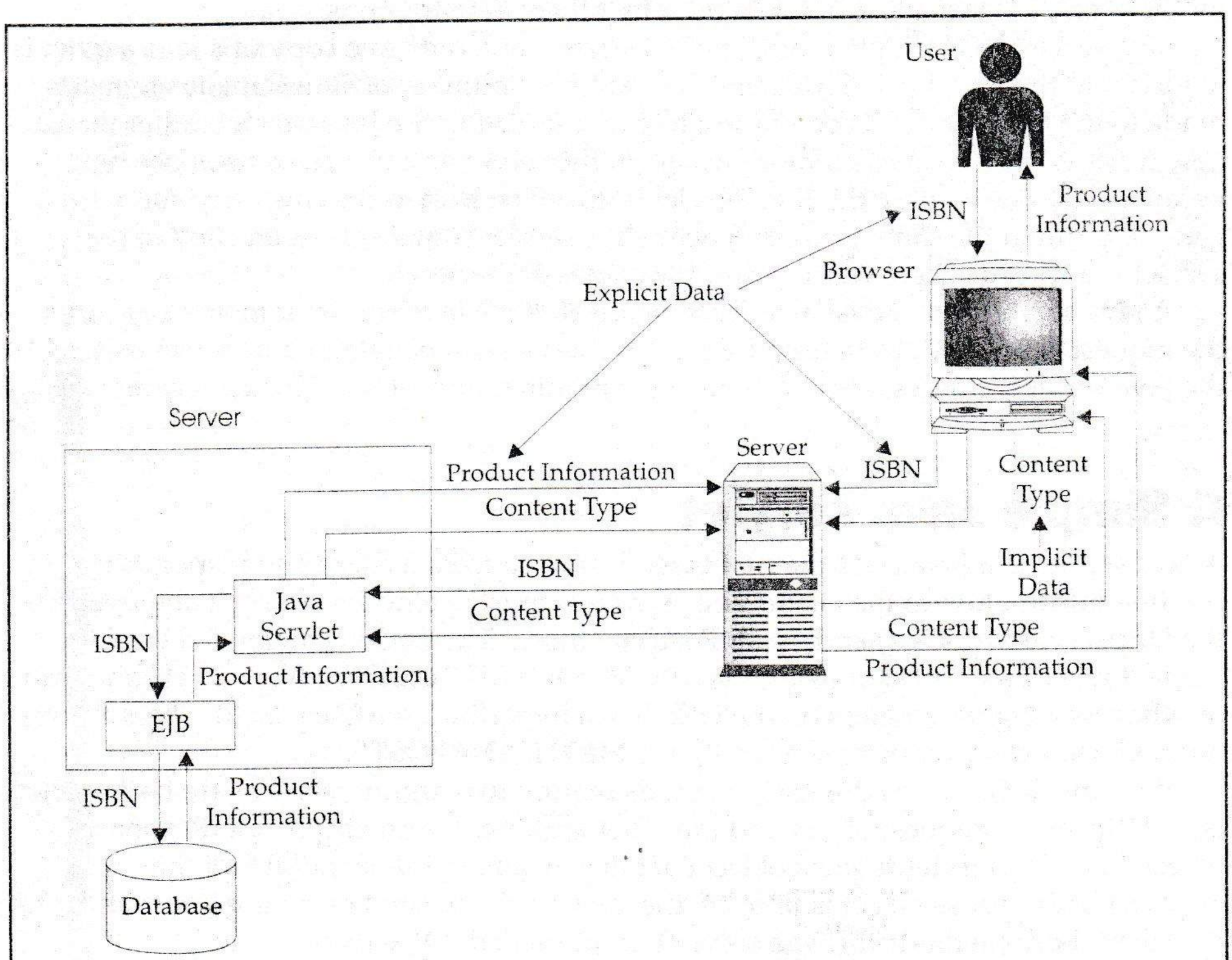


Figure 10-1. *Requesting product information*

Benefits of Using a Java servlet

Sun Microsystems, Inc. designed Java servlet technology to be more efficient and use less environment overhead than CGI. A new process begins each time a request is made to a CGI program. Starting a new process is time-consuming, but acceptable as long as the execution time of the CGI program is longer than the time that is necessary to begin the new process. However, executing a CGI program becomes unacceptable if the CGI program has a shorter execution time than beginning the new process.

Let's say that 100 instances of an application require a CGI program to process their requests simultaneously. The CGI program must be loaded into memory 100 times. That is to say, there are 100 copies of the CGI program in memory. Each instance must begin a new process, meaning 100 new processes are started—which is inefficient.

Also a CGI program isn't persistent. Once a CGI program terminates, all data used by the process is lost and cannot be used by other CGI programs.

Java servlet technology avoids inefficiencies. First, only one copy of a Java servlet is loaded into the Java Virtual Machine no matter the number of simultaneous requests made to the Java servlet. Each request begins a thread to the Java servlet rather than a new process, which saves memory usage on the server and increases response time because only one copy of the Java servlet needs to be loaded into memory. servlets generally run in the same process space as the servlet container—each copy of the servlet is a thread. There will be multiple copies of the servlet.

A Java servlet has persistence. This means that the Java servlet remains alive after the request is fulfilled. Data used by the Java servlet can be retained between calls to the Java servlet, if necessary to the business requirements of the J2EE application.

A Simple Java servlet

A Java servlet is a Java class that reads requests sent from a client and responds by sending information to the client. The Java class must extend Httpservlet and override the Httpservlet's doGet() and/or doPost() methods. The doGet() method is used to interact with a request sent using the METHOD="GET" attribute from an HTML form or when passing parameters on the URL like a hyperlink (see Chapter 8). The doPost() method is used for requests sent using the METHOD="POST".

Both the doGet() and doPost() methods require two arguments. The first argument is an HttpservletRequest object and the other argument is an HttpservletResponse object. The HttpservletRequest object contains incoming information and the HttpservletResponse object is used by the Java servlet to send outgoing information to the client. Both methods throw a servletException and IOException.

Incoming data includes explicit data and implicit data. Explicit data is supplied by a user and is contained in a query string that contains data from a form. Implicit data is HTTP information such as request headers, which are discussed later in this chapter.

Outgoing data is explicit data and implicit data that are generated by the Java servlet. Outgoing data is sent by using a PrintWriter object using the println() method and is forwarded to the client that made the request. The format of the data sent to the client is client dependent. For example, data takes the format of an HTML or XML page if the client is a browser.

The println() method is used in conjunction with a PrintWriter to send outgoing explicit data such as text that appears on a web page. Methods of the HttpservletResponse object are used to send implicit data.

Listing 10-1 illustrates a basic Java servlet that outputs both explicit and implicit data. The println() method is used to send explicit data in the format of a simple web page. The setContentType() method is used to set the value for the ContentType HTTP header information. ContentType identifies the type of document that is being sent explicitly. In this example, text/html is the ContentType passed to the setContentType() method. This identifies the explicit data as being in an HTML format.

You compile a Java servlet using the same process as you compile a Java program. Make sure that the CLASSPATH in your environment contains these entries: ".", javax.servlet (the JAR files containing the Java servlet classes), and the directory that contains the source code for the Java servlet. Once the Java servlet is compiled into a .class file, you must move the .class file into the directory where Java servlets are installed in your environment. Typically, this directory is install_dir/.../WEB-INF/classes. You call the Java servlet by using the URL http://host/servlet/myservlet where myservlet is the name of the Java servlet. The URL can be placed as the value of the ACTION attribute of a form or as the URL of a web page link.

Listing 10-1 A basic Java servlet.

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class js1 extends Httpservlet {
public void doGet(HttpservletRequest request, HttpservletResponse response)
     throws servletException, IOException {
   response.setContentType("text/html");
   PrintWriter out = response.getWriter();
   out.println("<!DOCTYPE HTML PUBLIC \"~//W3C//DTD HTML 4.0 " +
     "Transitional//EN\">\n" +
     "<HTML>\n" +
     "<HEAD><TITLE>Java servlet</TITLE></HEAD>\n" +
     "<BODY>\n" +
     "  <P>My A basic Java servlet </P>\n" +
     "</BODY>\n" +
     "</HTML>");
  }
}
```

Anatomy of a Java servlet

A Java servlet consists of at least four methods, each of which is called during the life cycle of the Java servlet. These methods are init(), service(), the appropriate request method, and destroy().

The init() method is called automatically when the Java servlet is created. You can override the init() method to include statements that are executed once during the life of the Java servlet and not for each request to use the Java servlet. The init() method is similar in concept to the init() method of an applet and is used to initialize variables and objects that are used throughout the Java servlet. The init() method doesn't require an argument, returns a void, and throws a servletException.

In the real world, the init() method is used to read server-specific initialization parameters such as performance parameters and database settings. The J2EE programmer customizes the functionality of a Java servlet based on the values of server-specific initialization parameters.

Listing 10-2 illustrates how to override the init() method to read server-specific initialization parameters. The servletConfig object contains the methods necessary to interact with server-specific initialization parameters. In this example, an instance of the servletConfig object is created and is called con. Next, the getInitParameter() method is called to retrieve the first initialization parameters, which is returned as a String object called p1. Server-specific initialization parameters are used with the deployment descriptor to control the behavior of the Java servlet and Java Server Pages, which is discussed later in this chapter.

sting 10-2 Using the it() method to read erver-specifi nitialization parameters

```
public void init() throw servletException {
  servletConfig con = getservletConfig();
  String p1 = con.getInitParameter("parameter1");
}
```

The service() method is called whenever a request for the Java servlet is made to the web server. A request spawns a new thread or an unused thread from the thread pool. There can be simultaneous calls to the service() method. The service() method examines the HTTP request type and then calls the appropriate request method such as doGet() and doPost().

Simultaneous calls to the service() method result in multiple concurrent threads of either doGet(), doPost(), or other request methods. This means shared data must be synchronized to prevent inadvertent changes to the data. Each thread owns its own object—you need to be careful if you're referencing static data.

Conflicts arising from shared data can be eliminated by preventing multiple threads of the Java servlet. This is accomplished by implementing the SingleThreadModel interface when the Java servlet is created. This results in the creation of one thread for the instance of the Java servlet. Other requests for the Java servlet are queued and invoked only when the active thread terminates.

Caution *Implementing the SingleThreadModel interface prevents instance variables of the Java servlet from being accessed simultaneously, but class variables and shared data outside the Java servlet must still be synchronized. Minimize the use of the SingleThreadModel interface; otherwise, you might experience performance latency, as the server becomes idle while the Java servlet pauses for I/O.*

The destroy() method is the last method to be called right before the Java servlet terminates, such as when an instance of a Java servlet is removed from memory. You can override the destroy() method with statements that release resources, such as closing a database connection.

Caution *The destroy() method is not called when an abnormal occurrence such as a system malfunction causes the Java servlet to abruptly terminate.*

Deployment Descriptor

The deployment descriptor is a file located in the WEB-INF directory that controls the behavior of a Java servlet and Java Server Pages. The file is called the web.xml file and contains the XML header, DOCTYPE, and a web-app element. The web-app element should contain a servlet element with three subelements. These are servlet-name, servlet-class, and init-param.

The servlet-name element contains the name used to access the Java servlet. The servlet-class is the class name of the Java servlet and the init-param is the name of an initialization parameter that is used whenever a request is made to the Java servlet.

Listing 10-3 illustrates a simple web.xml deployment descriptor file. In this example, the name of the Java servlet within the web.xml file is MyJavaservlet. The class name of the Java servlet is myPackage.MyJavaservletClass. The name of the initialization parameter is parameter1 whose value is 735.

Listing 10-3 An example of the deployment descriptor file.

```
<?xml version="1.0" encoding-"ISO-8859=1"?>

<!DOCTYPE web-app PUBLIC "~//Sun Microsystems, Inc.//DTD Web Application2.2//EN"
"http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">
<web-app>
  <servlet>
    <servlet-name>MyJavaservlet</servlet-name>
    <servlet-class>myPackage.MyJavaservletClass</servlet-class>
    <init-param>
      <param-name>parameter1</param-name>
      <param-value>735</param-value>
    </init-param>
```

```
    <servlet-mapping>
      <servlet-name> MyJavaservlet</servlet-name>
      <url-pattern>MyJavaservlet</url-pattern>
    </servlet-mapping>
    </servlet>
<web-app>
```

Reading Data from a Client

A client uses either the GET or POST Method to pass information to a Java servlet as described in Chapter 8. The doGet() or doPost() method is called in the Java servlet depending on the method used by the client.

Data sent by a client is read into a Java servlet by calling the getParameter() method of the HttpservletRequest object that instantiated in the argument list of the doGet() and doPost() methods. The getParameter() method requires one argument, which is the name of the parameter that contains the data sent by the client. The getParameter() method returns a String object. The String object contains the value of the parameter, if the client assigns a value to the parameter. An empty String object is returned if the client didn't assign a value to the parameter. Also, a null is returned if the parameter isn't received from the client.

As you learned in Chapter 8, an HTML form can contain a set of check boxes or other form objects that have the same data name but different values. This means that data received from a client might have multiple occurrences of the same parameter name.

You can read a set of parameters that have the same name by calling the getParameterValues() method. The getParameterValues() method has one argument, which is the name of the parameter, and returns an array of String objects. Each element of the array contains a value of the set of parameters. The getParameterValues() method returns a null if data received from the client doesn't contain the parameter named in the argument.

Caution *Parameter names are case sensitive.*

You can retrieve all the parameters by calling the getParameterNames() method. The getParameterNames() method does not require an argument and returns an Enumeration. Parameter names appear in any order and can be cast to a String object and used with the getParameter() and getParameterValues() methods.

Listing 10-4 contains an HTML form that prompts a user to enter an email address. When the user selects the Submit button, the browser calls the js2.class Java servlet and

sends the email address as data. Listing 10-5 illustrates the js2.class Java servlet that reads data sent by this form. Listing 10-5 is a modification of Listing 10-1.

In this example, the getParameter() method is called and passed the parameter name email as an argument. The getParameter() method returns a string that is assigned to the email String object called email. The value of the email String object is then returned to the browser in the form of an HTML page.

Listing 10-4 An HTML form that calls a Java servlet.

```
<FORM ACTION="/servlet/myservlets.js2">
Enter Email Address: <INPUT TYPE="TEXT" NAME="email">
<INPUT TYPE="SUBMIT">
</FORM>
```

Listing 10-5 Reading data sent by a client using a Java servlet.

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class js2 extends Httpservlet {
  public void doGet(HttpservletRequest request, HttpservletResponse response)
      throws servletException, IOException {
    String email;
    email = request.getParameter("email");
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    out.println("<!DOCTYPE HTML PUBLIC \"~//W3C//DTD HTML 4.0 " +
      "Transitional//EN\">\n" +
      "<HTML>\n" +
      "<HEAD><TITLE>Java servlet</TITLE></HEAD>\n" +
      "<BODY>\n" +
      "  <P>My Email Address: " + email + "</P>\n" +
      "</BODY>\n" +
      "</HTML>");
  }
}
```

Reading HTTP Request Headers

A request from a client contains two components. These are data (explicit data) sent by the client, such as the email address in Listing 10-5. The other component is a set of HTTP request headers sent by the browser that made the request (implicit data).

An HTTP request header (see Listing 10-6) contains information that describes the data component of the request. A Java servlet can read HTTP request headers and use

the value of HTTP request headers to process the data component of the request. The more commonly used HTTP request headers are shown in Table 10-1.

Listing 10-6 A typical set of HTTP request headers.

```
Accept: image/jpg, image/gif, */*
Accept-Encoding: Zip
Connect: Keep-Alive
Cookie: CustNum-12345
Host: www.mywebsite.com
Referer: http://www.mywebsite.com/index.html
```

HTTP Request Header	Description
Accept	Identifies the Multipurpose Internet Mail Extension (MIME) type (see Table 10-2) of data that can be handled by the browser that made the request. The Java servlet can read the Accept HTTP request header and avoid returning data in a data format that cannot be processed by the browser. Table 10-2 contains a list of MIME types. Some browsers such as Explorer 5 send the Accept header correctly for the original request, but incorrectly when the request is reloaded by the browser.
Accept-Charset	Identifies the character sets that can be used by the browser that made the request. The Accept-Encoding HTTP request header indicates the type of encoding handled by the browser. For example, the Java servlet might return data to the browser in a compressed format in an effort to reduce transmission time. The value of the Accept-Encoding HTTP request header tells the Java servlet the type of compression formats that the browser can process.
Accept-Language	Specifies the preferred languages that are used by the browser.
Authorization	Used by a browser to identify the client to the Java servlet whenever a protected web page is being processed.

Table 10-1. *HTTP Request Headers*

HTTP Request Header	Description
Connection	Identifies whether a browser can retrieve multiple files using the same socket, which is referred to as persistence. Persistence reduces transmission time by eliminating the need for the browser to open a connection for each file that is being transmitted. Connection varies between HTTP 1.0 and HTTP 1.1. For more information about HTTP visit www.w3.org/Protocols.
Content-Length	Contains the size of data in bytes that are transmitted using the POST method.
Cookie	Returns cookies to a server. The server originally sent these cookies to the browser. This HTTP request header is rarely read directly in a real-word application.
Host	Contains the host and port of the original URL; however, this is optional.
If-Modified-Since	Signifies that the browser's requests should be fulfilled only if the data has changed since a specified date. For example, a browser may request the Java servlet send a specific catalog page to the browser only after the date specified in the header.
If-Unmodified-Since	Signifies that the browse's requests should be fulfilled only if the data is older than a specified date.
Referer	Contains the URL of the web page that is currently displayed in the browser. This is useful to learn where requests are coming from. The value of the Referer HTTP request header can be spoofed and therefore its value is not reliable. Also note the spelling of this HTTP request header. It is Referer and not Referrer.
User-Agent	Identifies the browser that made the request.

Table 10-1. *HTTP Request Headers* (continued)

MIME Types	Description
application/msword	Microsoft Word file
application/octet-stream	Unrecognized or binary data
application/pdf	Acrobat file
application/postscript	Postscript file
application/vnd.lotus-notes	Lotus Notes file
application/vnd.ms-excel	Microsoft Excel file
application/vnd.ms-powerpoint	Microsoft PowerPoint file
application/x-gzip	Gzip archive
application/x-java-archive	JAR file
application/x-java-serialized-object	Serialized Java object
applciation/x-java-vm	Java bytecode (.class) file
application/zip	Zip file
audio/basic	Sound file in .au or .snd format
audio/mdid	MIDI sound file
audio/x-aiff	AIFF sound file
audio/x-wav	MS Windows sound file
image/gif	GIF image
image/jpeg	JPEG
image/png	PNG image
image/tiff	TIFF image
image/x-xbitmap	X Windows bitmap image
text/css	HTML Cascading Style Sheet
text/html	HTML document
text/plain	Plain text
text/xml	XML
video/mpeg	MPEG video clip
video/quicktime	QuickTime video clip

Table 10-2. *MIME Types*

A Java servlet can read an HTTP request header by calling the getHeader() method of the HttpservletRequest object. The getHeader() requires one argument, which is the name of the HTTP request header that is to be read. The HTTP request header name isn't case sensitive. The getHeader() function returns a String object that contains the value of the specified HTTP request header.

HTTP connections can be told to open or close a connection based on HTTP request. The type of request used depends on the version of HTTP used for the connection. HTTP 1.0 requires you to set the Connection header to Keep-Alive in order to keep the connection open as illustrated here:

```
Connection: Keep-Alive
```

Caution *This heading is meaningless to a client or server that is using HTTP 1.0.*

HTTP 1.1 version also uses the Keep-Alive message to keep a connection open. You can also explicitly set the Connection header to close in order to close the connection after servicing the request. This is illustrated here:

```
Connection: close
```

Caution *There is no guarantee that the connection will remain open. The Connection header is a request, not a command, and therefore only the client or the server can open and close a connection. For more information about HTTP, visit www.w3.org/Protocols.*

Let's say that you wanted to read the Connection HTTP request header in Listing 10-6. You place the following statement in the Java servlet. The String object is null if the set of HTTP request headers doesn't have the specified HTTP request header.

```
String conHeader = request.getHeader("Connection");
```

Besides the getHeader() method, the HttpservletRequest object has a variety of methods that are used to read specific kinds of HTTP request headers. The "HttpservletRequest Class" section of the Quick Reference Guide at the end of this chapter contains a listing of HttpservletRequest methods.

Sending Data to a Client and Writing the HTTP Response Header

A Java servlet responds to a client's request by reading client data and the HTTP request headers, then processing information based on the nature of the request. For example, a client request for information about merchandise in an online product

catalog requires the Java servlet to search the product database to retrieve product information and then format the product information into a web page, which is returned to the client.

There are two ways in which a Java servlet replies to a client request. These are by sending information (explicit data) to the response stream and by sending information in the HTTP response header (implicit data). The HTTP response header is similar to the HTTP request header except the contents of the HTTP response header are generated by the web server that responds to the client's request.

Information is sent to the response stream by creating an instance of the PrintWriter object and then using the println() method to transmit the information to the client. This technique is discussed at the beginning of this chapter and illustrated in Listing 10-1.

An HTTP response header contains a status line, response headers, and a blank line, followed by the document. This is illustrated in Listing 10-7. There are three components to the status line. These are the HTTP version number, a status code, and a brief message associated with the status code.

In the example in Listing 10-7, the HTTP version number is 1.1 and the status code is 200, indicating that everything is fine with the request that was received from the client. OK is the message that is associated with the status code. This example contains one HTTP response header, which is Content-Type that identifies the document MIME (see Table 10-2) type as plain text. The document contains the expression "My response".

Listing 10-7 An HTTP response header.

```
HTTP/1.1 200 OK
Content-Type: text/plain

My response
```

A Java servlet can write to the HTTP response header by calling the setStatus() method of the HttpservletResponse object. The setStatus() method requires one argument, which is an integer that represents the status code. Table 10-3 contains a list of HTTP status codes. The argument can take the form of the int itself or a predefined constant that represents the int. The setStatus() method automatically inserts the status code and the short message associated with the status code into the HTTP response header.

For example, the following statement writes the status code 100 to the HTTP response header informing the client to send a follow-up request:

```
response.setStatus(100);
```

Status Code	Short Message	Constant	Description
100	Continue	SC_CONTINUE	Can the client send an attached document in a follow-up request? Respond with a status 100 if the answer is yes or 417 Expectation Failed if the answer is no.
200	OK	SC_OK	Everything is fine.
202	Accepted	SC_ACCEPTED	The request is being acted upon, but processing is not yet complete.
204	No Content	SC_NO_CONTENT	Continue to display the previous document because no new document is available.
205	Reset Content	SC_RESET_CONTENT	No new document. Reset the document view.
301	Moved Permanently	SC_MOVED_PERMANENTLY	The document is elsewhere. Supply new URL for the document in the Location HTTP response header.
302	Found	SC_FOUND	Interpret the URL given in the Location HTTP response header as a temporary replacement, not a permanent replacement.
303	See Other	SC_SEE_OTHER	Similar to 301 and 302 if the original request is POST; retrieve the new document with GET.
304	Not Modified	SC_NOT_MODIFIED	The client should use the cached version.
307	Temporary Redirect	No constant defined	The client is temporarily redirected to another URL.

Table 10-3. *HTTP 1.1 Status Codes*

Status Code	Short Message	Constant	Description
400	Bad Request	SC_BAD_REQUEST	Bad syntax in the client request.
401	Unauthorized	SC_UNAUTHORIZED	Attempt to access a password-protected page without identifying information in the Authorization HTTP response header.
403	Forbidden	SC_FORBIDDEN	Server refuses access to the resource usually because of bad file or directory permissions on the server.
404	Not Found	SC_NOT_FOUND	Resource not found.
405	Method Not Allowed	SC_METHOD_NOT_ ALLOWED	Request method not allowed for resource.
415	Unsupported Media Type	SC_UNSUPPORTED_ MEDIA_TYPE S	Server doesn't know how to handle attached document type.
500	Internal Server Error	SC_INTERNAL_ SERVER_ERROR	Generic error caused by CGI programs or servlets failure.
501	Not Implemented	SC_NOT_ IMPLEMENTED	The server cannot respond because of maintenance or overloading.
503	Service Unavailable	SC_SERVICE_ UNAVAILABLE	The server is unable to process the request.
505	HTTP Version Not Supported	SC_HTTP_VERSION_ NOT_SUPPROTED HTTP	Version not supported by server.

Table 10-3. *HTTP 1.1 Status Codes* (continued)

There are two other methods that developers use in a Java servlet to write the HTTP response header. These are sendError() and sendRedirect(). Both of these methods throw an IOException exception and return a void.

The sendError() method is used to notify the client that an error has occurred. Two arguments are required by the sendError() method. The first argument is an int that represents the status code, which is typically status code 404. The other argument is a String object that contains the short message. The short message is formatted automatically.

The sendRedirect() method transmits a location header to the browser. The browser uses the location header to connect to the URL specified in the location header. The location header is similar to the content-type: text/html except the location header looks like location:someURL.html. If you write to the response object after sending the location header, the response object fails. The browser terminates the connection as soon as the location header is received, which may cause the server to terminate the Java servlet and a JSP.

An HTTP response header is used to give the client directions on how to process the response document and provides additional information such as a cookie, which is discussed later in this chapter.

There are a number of HTTP response headers that can be set by a Java servlet. Table 10-4 contains a listing of commonly used HTTP response headers. Each HTTP response header is set from within a Java servlet by calling an appropriate method of the HttpservletResponse before the Java servlet writes the response document.

HTTP Response Header	Description
Allow	Specifies the request method (that is, GET, POST) supported by the server.
Cache-Control	Identifies conditions when the response document can safely be cached. One of the following possible values must be used: public Document is cacheable. private Store in private nonshared caches. no-cache Do not cache. no-store Do not cache. Do not store in a temporary location on disk. must-revalidate Revalidate document with original server. proxy-revalidates Same as must-revalidate. Applies only to shared caches. max-age=xxx Document is stale after xxx seconds. s-max-age=xxx Shared caches consider the document stale after xxx seconds.
Connection	Instructs to use or not to use persistence.

Table 10-4. *HTTP 1.1 Response Headers*

HTTP Response Header	Description
Close	This is a parameter for the Connection header. If you don't want persistent connection, you tell the server http1.0.
Content-Encoding	Indicates page encoding.
Content-Language	Indicates the language of the document.
Content-Length	Indicates the number of bytes in the message before any character encoding is applied.
Content-Type	Indicates the MIME type of the response document.
Expires	Specifies the time in milliseconds when document is out of date (long tenMinutes = 18*60*1000).
Last-Modified	Indicates the last time the document was changed.
Location	The location of the document. This gets sent with sendRedirect().
Refresh	Indicates the number of seconds to wait before asking for a page update.
Retry-After	Indicates the number of seconds to wait before requesting service, if the service is unavailable.
Set-Cookie	Identifies the cookie for the page.
WWW-Authenticate	Indicates the authorization type (that is, BASIC, DIGEST).

Table 10-4. *HTTP 1.1 Response Headers* (continued)

The "HttpservletResponse Class" section of the Quick Reference Guide at the end of this chapter contains a list of methods that are used to write an HTTP response header.

Working with Cookies

A cookie is a small amount of data that is saved on the client's machine and can be created by and referenced by a Java servlet using the Java servlet cookie API. A cookie is composed of two pieces. These are the cookie name and the cookie value, both of which are created by the Java servlet. The cookie name is used to identify a particular cookie from among other cookies stored at the client. The cookie value is data associated with

the cookie. Any character can be used except the following: [] () =," / ? @ : ;. However, version 1.0 can use these characters.

A Java servlet writes a cookie by passing the constructor of the Cookie object two arguments. The first argument is a String object that contains the name of the cookie. The other argument is a String object that contains the value of the cookie.

The following statement illustrates how a cookie is created when the instance of the Cookie object called myCookie is declared. In this example, a cookie called userID is being created and assigned the value 123:

```
Cookie myCookie = new Cookie("userID", "123");
```

Once a cookie is created, the Java servlet must insert the cookie into the HTTP response header by calling the addCookie() method of the HttpservletResponse object. The addCookie() method requires one argument, which is the name of the instance of the Cookie object. The addCookie() method must be called before the Java servlet writes the response document.

The following statement shows how to add a cookie to the HTTP response header. In this example, the instance of the Cookie object called myCookie, which is declared in the previous statement, is passed as the argument to the addCookie() method. Listing 10-8 contains a complete Java servlet that writes this cookie and sends a response document to a client.

```
response.addCookie(myCookie);
```

You can read a cookie by calling the getCookies() method of the HttpservletRequest object. The getCookies() method returns an array of Cookie objects. The Java servlet can step through the array of Cookie objects and retrieve attributes of each cookie by calling the getXXX() method where XXX represent the name of the attribute that you want to retrieve. Name and value are two commonly requested attributes. Table 10-5 lists all the cookie attributes and the "Cookie Class" section of the Quick Reference Guide at the end of this chapter contains a list of getXXX() methods.

Typically, a Java servlet steps through the array of Cookie objects comparing the desired cookie name with the name attribute of each cookie in the array. Once a match is found, the Java servlet then retrieves other attributes of that cookie as required by the J2EE application. Listing 10-9 illustrates how to read a cookie.

The addCookie() method does not affect cookies that already exist on the client. The cookie is limited to 4KB and most browsers limit the total space to 2MB for cookies, so addCookie() may cause others to be deleted. The expiration date is only a suggestion. The browser decides when to delete cookies, depending on storage space considerations. Instead, the addCookie() method appends a new cookie to the existing cookies. A Java servlet can modify the value of an existing cookie by using the setValue() method of the Cookie object, and modify other attributes of the cookie by using the appropriate setXXX() method that corresponds to the getXXX() method. These methods are also listed

Attribute	Description
Comment	The comment about the cookie
Domain	The domain of the cookie
MaxAge	The maximum length of time the cookie remains valid
Name	The name of the cookie
Path	The path of the cookie
Secure	The Boolean value that indicates whether or not the cookie should be transmitted only over an encrypted connection
Value	The value of the cookie
Version	The version of the cookie

Table 10-5. *Cookie Attributes*

in the "Cookie Class" section of the Quick Reference Guide at the end of this chapter. The following statement illustrates how to modify the value attribute of myCookie:

```
myCookie.setValue("456");
```

Listing 10-8 Writing a cookie.

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class SetCookies extends Httpservlet {
  public void doGet(HttpservletRequest request, HttpservletResponse response)
      throws servletException, IOException {
    Cookie myCookie = new Cookie("userID", "123");
    response.addCookie(myCookie);
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    out.println ("<!DOCTYPE HTML PUBLIC \"~//W3C//DTD HTML 4.0 " +
     "Transitional//EN\">\n" +
     "<HTML>\n" +
     "<HEAD><TITLE> My Cookie </TITLE></HEAD>\n" +
     "<BODY>\n" +
     "   <H1">" + My Cookie + "</H1>\n" +
     "   <P> Cookie written</P>\n" +
     "</BODY></HTML>");
  }
}
```

Listing 10-9 Reading a cookie.

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class ShowCookies extends Httpservlet {
  public void doGet(HttpservletRequest request, HttpservletResponse response)
      throws servletException, IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    out.println ("<!DOCTYPE HTML PUBLIC \"~//W3C//DTD HTML 4.0 " +
    "Transitional//EN\">\n" +
     "<HTML>\n" +
    "<HEAD><TITLE> My Cookie </TITLE></HEAD>\n" +
    "<BODY>\n" +
    "<H1">" + My Cookie + "</H1>\n" +
    "<P>" +
    Cookie[] cookies = request.getCookies();
    if (cookies == null)
    {
      out.println("No cookies");
    }
    else
    {
      Cookie MyCookie;
      for(int i=0; i< cookie.length(); i++)
      {
        MyCookie = cookies[i];
        out.println(myCookies.getName() + "=" + MyCookie.getValue());
      }
    }
  }
}
```

Tracking Sessions

A session is created each time a client requests service from a Java servlet. The Java servlet processes the request and responds accordingly, after which the session is terminated. Many times the same client follows with another request to the same Java servlet, and the Java servlet requires information regarding the previous session to process the request. However, HTTP is a stateless protocol, meaning that there isn't any holdover from previous sessions. Therefore, there is no way to track sessions using HTTP.

A Java servlet is capable of tracking sessions by using the HttpSession API. The HttpSession API uses the HttpSession object that is associated with the HttpservletRequest to determine if the request is a continuation from an existing session or is a new session.

A Java servlet calls the getSession() method of the HttpservletRequest object, which returns an HttpSession object if it is a new session. The getSession() method requires one argument, which is a boolean true. The getSession() returns a null if the user doesn't already have a session underway; otherwise, a new HttpSession object is created. The following statement illustrates how this is written: If the getSession() method doesn't use an argument, then, the getSession() method returns either the existing session or the getSession() method creates a new session and returns the session object. If the getSession() method takes a boolean true as an argument, the getSession() method returns the current session as a session object. If the current session doesn't exist, the getSession() method creates a new session and returns a session object.

If the getSession() method takes a boolean false as an argument, the getSession() method returns the current session as a session object. If the current session doesn't exist, the getSession() method returns a null.

```
HttpSession s1 = request.getSession(true);
```

An HtppSession object contains a data structure that is used to store keys and a value associated with each key. A key is an attribute of the session. You can read a value of an attribute by calling the HttpSession object's getAttribute() method, and you can modify the value of an attribute by calling the setAttribute() method.

The getAttribute() method requires one argument, which is a String object that contains the name of the attribute whose value you want to retrieve. The getAttribute() returns the value of the attribute, which is an object. The type of object depends on the nature of the attribute. For example, the value of an attribute can be a String object or an object that is defined in the Java servlet. Listing 10-10 illustrates how to read a session attribute. In this example, the session ID is read and returned to the client, where the session ID is displayed on the screen.

The setAttribute() method requires two arguments. The first argument is a String object that contains the name of the attribute. The other argument is an object similar to an object returned by the getAttribute() method.

The HttpSession API contains an assortment of methods that can be used to retrieve and manipulate information that enables a Java servlet to track a session. The "HttpSession Class" section of the Quick Reference Guide located at the end of the chapter lists these methods.

sting 10-10 Reading a session attribute.

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.net.*;
import java.util.*;
public class ShowSession extends Httpservlet {
  public void doGet(HttpservletRequest request, HttpservletResponse response)
        throws servletException, IOException {
```

```
response.setContentType("text/html");
PrintWriter out = response.getWriter();
HttpSession s1 = request.getSession(true);
String s1ID = s1.getId();
out.println ("<!DOCTYPE HTML PUBLIC \"~//W3C//DTD HTML 4.0 " +
 "Transitional//EN\">\n" +
 "<HTML>\n" +
 "<HEAD><TITLE> Session ID</TITLE></HEAD>\n" +
 "<BODY>\n" +
 "   <H1">Session ID</H1>\n" +
 "   <P>" + s1ID + "</P>\n" +
 "</BODY></HTML>");
    }
}
```

Quick Reference Guide

This Quick Reference Guide provides a brief overview of Java classes used to by a Java servlet to interact with HTTP. Full details of these classes and all Java classes and interfaces are available at java.sun.com.

Syntax	Descriptions
Object clone()	Overrides the standard clone method to return a copy of this cookie.
String getComment()	Returns the comment for the cookie.
String getDomain()	Returns the domain name set for this cookie.
int getMaxAge()	Returns the maximum age of the cookie, specified in seconds. By default, a -1 cookie persists until the browser shuts down.
String getName()	Returns the name of the cookie.
String getPath()	Returns the path on the server to which the browser returns this cookie.

Table 10-6. *javax.servlet.http Cookie Class*

Syntax	Descriptions
boolean getSecure()	Returns true if cookie must be sent by (unspecified) secure means or false if not required to be sent over (unspecified) secure means.
String getValue()	Returns the value of the cookie.
int getVersion()	Returns the version of the protocol of the cookie.
void setComment(String purpose)	Specifies a comment of a cookie.
void setDomain(String pattern)	Specifies the domain of the cookie.
void setMaxAge(int expiry)	Sets the maximum age of the cookie in seconds.
void setPath(String uri)	Specifies a path for the cookie.
void setSecure(boolean flag)	Indicates cookie should only be sent using secure means.
void setValue(String newValue)	Assigns a new value to a cookie.
void setVersion(int v)	Sets the version of the cookie protocol.

Table 10-6. *javax.servlet.http Cookie Class* (continued)

Syntax	Descriptions
String getAuthType()	Returns the name of the authentication scheme.
String getContextPath()	Returns the context of the request.
Cookie[] getCookies()	Returns an array containing all of the cookies sent with this request.
long getDateHeader(String name)	Returns the value of the request header as a long value that represents a Date object.

Table 10-7. *javax.servlet.http HttpservletRequest Class*

Syntax	Descriptions
String getHeader(String name)	Returns the value of the s request header as a String.
Enumeration getHeaderNames()	Returns an enumeration of all the header names.
Enumeration getHeaders(String name)	Returns all the values of a request header as an Enumeration.
int getIntHeader(String name)	Returns the value of the specified request header as an int.
String getMethod()	Returns the HTTP method (GET, POST, or PUT).
String getPathInfo()	Returns any extra path.
String getPathTranslated()	Returns any extra path information after the servlet name.
String getQueryString()	Returns the query string in the request URL after the path.
String getRemoteUser()	Returns the user login if authenticated, or null if it has not been authenticated.
String getRequestedSessionId()	Returns the session ID.
String getRequestURI()	Returns the URI of the URL.
StringBuffer getRequestURL()	Reconstructs the URL.
String getservletPath()	Returns the path of the URL.
HttpSession getSession()	Returns the current session, or creates a session if there isn't a session.
HttpSession getSession(boolean create)	Returns the current HttpSession. If there is no session, a new session is created if the parameter is true—otherwise, returns null.

Table 10-7. *javax.servlet.http HttpservletRequest Class* (continued)

Syntax	Descriptions
Principal getUserPrincipal()	Returns a Principal object containing the current authenticated user.
boolean isRequestedSessionIdFromCookie()	Determines if the session ID is part of a cookie.
boolean isRequestedSessionIdFromURL()	Determines if the session ID is part of the request URL.
boolean isRequestedSessionIdValid()	Determines if session ID is valid.
boolean isUserInRole(String role)	Returns a Boolean indicating whether the authenticated user is specified.

Table 10-7. *javax.servlet.http HttpservletRequest Class* (continued)

Syntax	Descriptions
void addCookie(Cookie cookie)	Adds a cookie.
void addDateHeader(String name, long date)	Adds a response header with the given name and date-value.
void addHeader(String name, String value)	Adds a response header with the given name and value.
void addIntHeader(String name, int value)	Adds a response header with the given name and integer value.
boolean containsHeader(String name)	Returns a Boolean indicating if the named response header is set.
String encodeRedirectURL(String url)	Encodes the specified URL for use in the sendRedirect() method by including the session ID.
String encodeURL(String url)	Encodes the specified URL by including the session ID.

Table 10-8. *javax.servlet.http HttpservletResponse Class*

Syntax	Descriptions
void sendError(int sc)	Sends an error response using the status code.
void sendError(int sc, String msg)	Sends an error response using the status code and message.
void sendRedirect(String location)	Sends a temporary redirect to the client using the redirect location URL.
void setDateHeader(String name, long date)	Sets a response header with the given name and date-value.
void setHeader(String name, String value)	Sets a response header by name and value.
void setIntHeader(String name, int value)	Sets a response header by name and value.
void setStatus(int sc)	Sets the status code.

Table 10-8. *javax.servlet.http HttpservletResponse Class* (continued)

Syntax	Descriptions
SC_ACCEPTED	Status code (202) Request was accepted, but processing was not completed.
SC_BAD_GATEWAY	Status code (502) The HTTP server received an invalid response from a server when acting as a proxy or gateway.
SC_BAD_REQUEST	Status code (400) The request was syntactically incorrect.
SC_CONFLICT	Status code (409) The request was not completed because of a conflict with the resource.

Table 10-9. *HttpservletResponse Class Constants (Fields)*

Syntax	Descriptions
SC_CONTINUE	Status code (100) The client can continue.
SC_CREATED	Status code (201) The request succeeded. New resource created on the server.
SC_EXPECTATION_FAILED	Status code (417) Expectation failed in the Expect request header.
SC_FORBIDDEN	Status code (403) The server refused to fulfill the request.
SC_GATEWAY_TIMEOUT	Status code (504) The upstream server did not send a timely response.
SC_GONE	Status code (410) The resource is no longer available. No forwarding address is known.
SC_HTTP_VERSION_NOT_SUPPORTED	Status code (505) The server does not support the HTTP protocol version.
SC_INTERNAL_SERVER_ERROR	Status code (500) An error inside the HTTP server prevents the request from being fulfilled.
SC_LENGTH_REQUIRED	Status code (411) The request cannot be handled without a *Content-Length*.
SC_METHOD_NOT_ALLOWED	Status code (405) The method specified in the *Request-Line* is not allowed.
SC_MOVED_PERMANENTLY	Status code (301) The resource has permanently moved to a new location. Don't use the original URI to access the resource.
SC_MOVED_TEMPORARILY	Status code (302) The resource has temporarily moved to another location. Still use the original URI to access the resource.

Table 10-9. *HttpservletResponse Class Constants (Fields)* (continued)

Syntax	Descriptions
SC_MULTIPLE_CHOICES	Status code (300) The requested resource corresponds to any one of a set of resources.
SC_NO_CONTENT	Status code (204) The request succeeded, but no new information to return.
SC_NON_AUTHORITATIVE_ INFORMATION	Status code (203) The meta information did not originate from the server.
SC_NOT_ACCEPTABLE	Status code (406) The resource identified by the request is only capable of generating response entities that have content characteristics not acceptable according to the accept headers sent in the request.
SC_NOT_FOUND	Status code (404) The requested resource is not available.
SC_NOT_IMPLEMENTED	Status code (501) The HTTP server does not support the functionality.
SC_NOT_MODIFIED	Status code (304) A conditional GET operation found is available and not modified.
SC_OK	Status code (200) The request succeeded.
SC_PARTIAL_CONTENT	Status code (206) The server has fulfilled the partial GET request.
SC_PAYMENT_REQUIRED	Status code (402) Reserved for future use.
SC_PRECONDITION_FAILED	Status code (412) The precondition of a request-header field evaluated to false when it was tested.

Table 10-9. *HttpservletResponse Class Constants (Fields)* (continued)

Syntax	Descriptions
SC_PROXY_AUTHENTICATION_REQUIRED	Status code (407) The client must authenticate with the proxy.
SC_REQUEST_ENTITY_TOO_LARGE	Status code (413) Process refused due to the request entity size.
SC_REQUEST_TIMEOUT	Status code (408) Request was not produced within the time limit.
SC_REQUEST_URI_TOO_LONG	Status code (414) Refused because of the size of the *Request-URI*.
SC_REQUESTED_RANGE_NOT_SATISFIABLE	Status code (416) The server cannot serve the requested byte range.
SC_RESET_CONTENT	Status code (205) The agent should reset the document view.
SC_SEE_OTHER	Status code (303) The response to the request is under a different URI.
SC_SERVICE_UNAVAILABLE	Status code (503) The HTTP server is temporarily overloaded.
SC_SWITCHING_PROTOCOLS	Status code (101) The server is switching protocols according to Upgrade header.
SC_TEMPORARY_REDIRECT	Status code (307) The requested resource is temporarily under a different URI.
SC_UNAUTHORIZED	Status code (401) The request requires HTTP authentication.
SC_UNSUPPORTED_MEDIA_TYPE	Status code (415) Request refused because the entity format is not supported.
SC_USE_PROXY	Status code (305) The requested resource must be accessed through the proxy identified in the Location.

Table 10-9. *HttpservletResponse Class Constants (Fields)* (continued)

Syntax	Descriptions
Object getAttribute(String name)	Returns the object associated with the attribute, or null if attribute name is not found.
Enumeration getAttributeNames()	Returns an Enumeration of String objects containing the names of all the attributes.
long getCreationTime()	Returns the time when this session was created, measured in milliseconds since midnight January 1, 1970 GMT.
String getId()	Returns a String object that contains the unique identifier that identifies the session.
long getLastAccessedTime()	Returns the last time the client sent a request in the session, as the number of milliseconds since midnight January 1, 1970 GMT, and marked by the time the container received the request.
int getMaxInactiveInterval()	Returns the maximum time, in seconds, that the session remains open between client accesses.
servletContext getservletContext()	Returns the servletContext of the session.
void invalidate()	Invalidates this session.
boolean isNew()	Returns true if the client is unaware of the session or if the client doesn't join the session.
void removeAttribute(String name)	Removes the object associated with the attribute.
void setAttribute(String name, Object value)	Writes an object to the attribute.
void setMaxInactiveInterval(int interval)	Specifies a time in seconds after which the session is invalidated.

Table 10-10. *javax.servlet.http HttpSession*

The Complete Reference



Chapter 11

Java ServerPages

A Java ServerPage (JSP) is a server-side program that is similar in design and functionality to a Java servlet, which is described in the previous chapter. A JSP is called by a client to provide a web service, the nature of which depends on the J2EE application. A JSP processes the request by using logic built into the JSP or by calling other web components built using Java servlet technology or Enterprise JavaBean technology, or created using other technologies. Once the request is processed, the JSP responds by sending the results to the client.

However, a JSP differs from a Java servlet in the way in which the JSP is written. As you'll recall from Chapter 9, a Java servlet is written using the Java programming language and responses are encoded as an output String object that is passed to the println() method. The output String object is formatted in HTML, XML, or whatever formats are required by the client.

In contrast, JSP is written in HTML, XML, or in the client's format that is interspersed with scripting elements, directives, and actions comprised of Java programming language and JSP syntax. In this chapter you'll learn how to create a JSP that can be used as a middle-level program between clients and web services.

JSP

A JSP is simpler to create than a Java servlet because a JSP is written in HTML rather than with the Java programming language. This means that the JSP isn't cluttered with many println() methods as found in a Java servlet. However, a JSP offers basically the same features found in a Java servlet because a JSP is converted to a Java servlet the first time that a client requests the JSP.

There are three methods that are automatically called when a JSP is requested and when the JSP terminates normally. These are the jspInt() method, the jspDestroy() method, and the service() method. These methods can be overridden, although the jspInt() method and jspDestroy() methods are commonly overridden in a JSP to provide customized functionality when the JSP is called and terminates.

The jspInt() method is identical the init() method in a Java servlet and in an applet. The jspInt() method is called first when the JSP is requested and is used to initialize objects and variables that are used throughout the life of the JSP.

The jspDestroy() method is identical to the destroy() method in a Java servlet. The destroy() method is automatically called when the JSP terminates normally. It isn't called if the JSP abruptly terminates such as when the server crashes. The destroy() method is used for cleanup where resources used during the execution of the JSP are released, such as disconnecting from a database.

The service() method is automatically called and retrieves a connection to HTTP.

Installation

Once a JSP is created, you place the JSP in the same directory as HTML pages. This differs from a Java servlet, which must be placed in a particular directory that is included in the CLASSPATH. You don't need to set the CLASSPATH to reference a JSP.

However, there are three factors that you must address when installing a JSP. First, web services called by a JSP must be installed properly. For example, a Java servlet called by a JSP must be placed in the designated directory for Java servlets and referenced on the CLASSPATH. The development environment used to create the J2EE application determines the designated directory.

The second factor to be addressed is to avoid placing the JSP in the WEB-INF or META-INF directories. The development environment prohibits this. The last factor is that the directory name used to store a JSP mustn't have the same name as the prefix of the URL of the web application.

JSP Tags

A JSP program consists of a combination of HTML tags and JSP tags. JSP tags define Java code that is to be executed before the output of the JSP program is sent to the browser.

A JSP tag begins with a <%, which is followed by Java code, and ends with %>. There is also an Extendable Markup Language (XML) version of JSP tags, which are formatted as <jsp:TagID> </JSP:TagID>.

JSP tags are embedded into the HTML component of a JSP program and are processed by a JSP virtual engine such as Tomcat, which is discussed later in this chapter. Tomcat reads the JSP program whenever the program is called by a browser and resolves JSP tags, then sends the HTML tags and related information to the browser.

Java code associated with JSP tags in the JSP program is executed when encountered by Tomcat, and the result of that process is sent to the browser. The browser knows how to display the result because the JSP tag is enclosed within an open and closed HTML tag. You'll see how this works in the next section.

There are five types of JSP tags that you'll use in a JSP program. These are as follows:

- **Comment tag** A comment tag opens with <%-- and closes with --%>, and is followed by a comment that usually describes the functionality of statements that follow the comment tag.
- **Declaration statement tags** A declaration statement tag opens with <%! and is followed by a Java declaration statement(s) that define variables, objects, and methods that are available to other components of the JSP program.
- **Directive tags** A directive tag opens with <%@ and commands the JSP virtual engine to perform a specific task, such as importing a Java package required by objects and methods used in a declaration statement. The directive tag closes with %>. There are three commonly used directives. These are import, include, and taglib. The import tag is used to import Java packages into the JSP program. The include tag inserts a specified file into the JSP program replacing the include tag. The taglib tag specifies a file that contains a tag library. Here are examples of each tag. The first tag imports the java.sql package. The next tag includes

the books.html file located in the keogh directory. And the last tag loads the myTags.tld library.

```
<%@ page import=" import java.sql.*"; %>
<%@ include file="keogh\books.html" %>
<%@ taglib uri="myTags.tld" %>
```

- **Expression tags** An expression tag opens with <%= and is used for an expression statement whose result replaces the expression tag when the JSP virtual engine resolves JSP tags. An expression tags closes with %>.
- **Scriptlet tags** A scriptlet tag opens with <% and contains commonly used Java control statements and loops. A scriptlet tag closes with %>.

Variables and Objects

You can declare Java variables and objects that are used in a JSP program by using the same coding technique as used to declare them in Java. However, the declaration statement must appear as a JSP tag within the JSP program before the variable or object is used in the program.

Listing 11-1 shows a simple JSP program that declares and uses a variable. In this example, the program declares an int called age and initializes the variable with the number 29. The declaration statement is placed within JSP tag.

You'll notice that this JSP tag begins with <%!. This tells the JSP virtual engine to make statements contained in the tag available to other JSP tags in the program. You'll need to do this nearly every time you declare variables or objects in your program unless they are only to be used within the JSP tag where they are declared.

The variable age is used in an expression tag that is embedded within the HTML paragraph tag <P>. A JSP expression tag begins with <%=, which is followed by the expression.

The JSP virtual engine resolves the JSP expression before sending the output of the JSP program to the browser. That is, the JSP tag <%=age%> is replaced with the number 29; afterwards, the HTML paragraph tag and related information is sent to the browser.

Listing 11-1 Declaring and using a variable.

```
<HTML>
  <HEAD>
    <TITLE> JSP Programming </TITLE>
  </HEAD>
  <BODY>
    <%! int age=29; %>
    <P> Your age is: <%=age%> </P>
  </BODY>
</HTML>
```

Any Java declaration statement can be used in a JSP tag similar to how those statements are used in a Java program. You are able to place multiple statements within a JSP tag by extending the close JSP tag to another line in the JSP program. This is illustrated in Listing 11-2 where three variables are declared.

Listing 11-2 Declaring multiple variables within a single JSP tag.

```
<HTML>
  <HEAD>
    <TITLE> JSP Programming </TITLE>
  </HEAD>
  <BODY>
    <%! int age=29;
      float salary;
      int empnumber;
    %>
  </BODY>
</HTML>
```

Besides variables, you are also able to declare objects, arrays, and Java collections within a JSP tag using techniques similar to those used in a Java program. Listing 11-3 shows how to declare an object and an array.

In Listing 11-3, the JSP program creates three String objects, the first two declarations implicitly allocate memory, and the third declaration explicitly allocates memory. In addition, this JSP program creates arrays and a Vector.

Listing 11-3 Declaring objects and arrays within a single JSP tag.

```
<HTML>
  <HEAD>
    <TITLE> JSP Programming </TITLE>
  </HEAD>
  <BODY>
    <%! String Name;
      String [ ] Telephone = {"201-555-1212", "201-555-4433"};
      String Company = new String();
      Vector Assignments = new Vector();
      int[ ]  Grade = {100,82,93};
    %>
  </BODY>
</HTML>
```

Methods

JSP offers the same versatility that you have with Java programs, such as defining methods that are local to the JSP program. A method is defined similar to how a method is defined in a Java program except the method definition is placed within a JSP tag.

You can call the method from within the JSP tag once the method is defined. Listing 11-4 illustrates how this is done. In this example, the method is passed a student's grade and then applies a curve before returning the curved grade.

The method is called from within an HTML paragraph tag in this program, although any appropriate tag can be used to call the method. Technically, the method is called from within the JSP tag that is enclosed within the HTML paragraph tag.

The JSP tag that calls the method must be a JSP expression tag, which begins with <%=. You'll notice that the method call is identical to the way a method is called within a Java program. The JSP virtual engine resolves the JSP tag that calls the method by replacing the JSP tag with the results returned by the method, which is then passed along to the browser that called the JSP program.

Listing 11-4 Defining and calling a method.

```
<HTML>
  <HEAD>
    <TITLE> JSP Programming </TITLE>
  </HEAD>
  <BODY>
    <%! boolean curve (int grade)
        {
         return 10 + grade;
        }
    %>
    <P> Your curved grade is: <%=curve(80)%> </P>
  </BODY>
</HTML>
```

A JSP program is capable of handling practically any kind of method that you normally use in a Java program. For example, Listing 11-5 shows how to define and use an overloaded method.

Both methods are defined in the same JSP tag, although each follows Java syntax structure for defining a method. One method uses a default value for the curve, while the overload method enables the statement that calls the method to provide the value of the curve.

Once again, these methods are called from an embedded JSP tag placed inside two HTML paragraph tags.

Listing 11-5 Defining and calling a method and an overloaded method.

```
<HTML>
  <HEAD>
    <TITLE> JSP Programming </TITLE>
  </HEAD>
  <BODY>
    <%! boolean curve (int grade)
        {
         return 10 + grade;
        }
        boolean curve (int grade,int curveValue)
        {
          return curveValue + grade;
        }
   %>
   <P> Your curved grade is: <%=curve(80, 10)%> </P>
   <P> Your curved grade is: <%=curve(70)%> </P>
  </BODY>
</HTML>
```

Control Statements

One of the most powerful features available in JSP is the ability to change the flow of the program to truly create dynamic content for a web page based on conditions received from the browser.

There are two control statements used to change the flow of a JSP program. These are the if statement and the switch statement, both of which are also used to direct the flow of a Java program.

The if statement evaluates a condition statement to determine if one or more lines of code are to be executed or skipped (as you probably remember from when you learned Java). Similarly, a switch statement compares a value with one or more other values associated with a case statement. The code segment that is associated with the matching case statement is executed. Code segments associated with other case statements are ignored.

The power of these controls comes from the fact that the code segment that is executed or skipped can consist of HTML tags or a combination of HTML tags and JSP tags. That is, these code segments don't need to be only Java statements or Java tags.

Listing 11-6 shows how to intertwine HTML tags and JSP tags to alter the flow of the JSP program. You'll notice that this program is confusing to read because the if statement and the switch statement are broken into several JSP tags. This is necessary because HTML tags are interspersed within these statements.

The if statement requires three JSP tags. The first contains the beginning of the if statement, including the conditional expression. The second contains the else statement, and the third has the closed French brace used to terminate the else block.

Two HTML paragraph tags contain information that the browser displays, depending on the evaluation of the conditional expression in the if statement. Only one of the HTML paragraph tags and related information are sent to the browser.

The switch statement also is divided into several JSP tags because each case statement requires an HTML paragraph tag and related information. And, as with the if statement, only one HTML paragraph tag and related information associated with a case statement that matches the switch value are returned to the browser.

Listing 11-6 contains simple examples of the if statement and switch statement. You can create more complex statements, such as nesting if statements, by using techniques illustrated in this example. Any flow control statements that you use in Java can also be incorporated into a JSP program. However, you must be careful to separate JSP tags from HTML tags and information that will be executed when the program's flow changes.

Listing 11-6 Using an if statement and a switch statement to determine which HTML tags and information are to be sent to the browser.

```
<HTML>
  <HEAD>
    <TITLE> JSP Programming </TITLE>
  </HEAD>
  <BODY>
  <%! int grade=70;%>
  <% if (grade > 69) { %>
       <P> You passed! </P>
  <% } else { %>
  <P> Better luck next time. </P>
  <% } %>
  <% switch (grade) {
       case 90 : %>
         <P> Your final grade is a A </P>
  <%     break; %>
       case 80 : %>
         <P> Your final grade is a B </P>
  <%     break;
       case 70 : %>
         <P> Your final grade is a C </P>
   <%    break;
       case 60 : %>
         <P> Your final grade is an F </P>
   <%    break;
      }
   %>
  </BODY>
</HTML>
```

Loops

JSP loops are nearly identical to loops that you use in your Java program except you can repeat HTML tags and related information multiple times within your JSP program without having to enter the additional HTML tags.

There are three kinds of loops commonly used in a JSP program. These are the for loop, the while loop, and the do...while loop. The for loop repeats, usually a specified number of times, although you can create an endless for loop, which you no doubt learned when you were introduced to Java.

The while loop executes continually as long as a specified condition remains true. However, the while loop may not execute because the condition may never be true. In contrast, the do...while loop executes at least once; afterwards, the conditional expression in the do...while loop is evaluated to determine if the loop should be executed another time.

Loops play an important role in JSP database programs because loops are used to populate HTML tables with data in the result set. However, Listing 11-7 shows a similar routine used to populate three HTML tables with values assigned to an array.

All the tables appear the same, although a different loop is used to create each table. The JSP program initially declares and initializes an array and an integer, and then begins to create the first table.

There are two rows in each table. The first row contains three column headings that are hard coded into the program. The second row also contains three columns each of which is a value of an element of the array.

The first table is created using the for loop. The opening table row tag <TR> is entered into the program before the for loop begins. This is because the for loop is only populating columns and not rows.

A pair of HTML table data cell tags <TD> are placed inside the for loop along with a JSP tag that contains an element of the array. The JSP tag resolves to the value of the array element by the JSP virtual program.

The close table row </TR> tag and the close </TABLE> tag are inserted into the program following the French brace that closes the for loop block. These tags terminate the construction of the table.

A similar process is used to create the other two tables, except the while loop and the do...while loop are used in place of the for loop.

sting 11-7 Using the for loop, while loop, and the do...while op to load HTML tables.

```
<HTML>
  <HEAD>
    <TITLE> JSP Programming </TITLE>
  </HEAD>
  <BODY>
```

```
<%! int[ ]  Grade = {100,82,93};
    int x=0;
%>
<TABLE>
  <TR>
    <TD>First</TD>
    <TD>Second</TD>
    <TD>Third</TD>
  </TR>
  <TR>
    <% for (int 1; i<3; i++) { %>
        <TD><%=Grade[i]%> </TD>
    <%   } %>
  </TR>
</TABLE>
<TABLE>
  <TR>
    <TD>First</TD>
    <TD>Second</TD>
    <TD>Third</TD>
  </TR>
  <TR>
    <% while (x<3){ %>
        <TD><%=Grade[x]%> </TD>
        <%    x++;
      } %>
  </TR>
</TABLE>
<TABLE>
  <TR>
   <TD>First</TD>
   <TD>Second</TD>
   <TD>Third</TD>
  </TR>
<TR>
  <% x=0;
     do{ %>
        <TD><%=Grade[x]%></TD>
     <%x++;
     } while (x<3) %>
```

```
        </TR>
      </TABLE>
    </BODY>
</HTML>
```

Tomcat

JSP programs are executed by a JSP Virtual Machine that runs on a web server. Therefore, you'll need to have access to a JSP Virtual Machine to run your JSP program. Alternatively, you can use an integrated development environment such as JBuilder that has a built-in JSP Virtual Machine or you can download and install a JSP Virtual Machine.

One of the most popular JSP Virtual Machines is Tomcat, and it is downloadable at no charge from the Apache web site. Apache is also a popular web server that you can also download at no cost.

You'll also need to have the Java Development Kit (JDK) installed on your computer, which you probably installed when you learned Java programming. You can download the JDK at no charge from the www.sun.com web site.

Here's what you need to do to download and install Tomcat:

1. Connect to jakarta.apache.org.
2. Select Download.
3. Select Binaries to display the Binary Download page.
4. Create a folder from the root directory called tomcat.
5. Download the latest release of jakarta-tomcat.zip to the tomcat folder.
6. Unzip jakarta-tomcat.zip. You can download a demo copy of WinZip from www.winzip.com if you don't have a zip/unzip program installed on your computer.
7. The extraction process should create the following folders in the tomcat directory: bin, conf, doc, lib src, and webapps.
8. Use a text editor such as Notepad and edit the JAVA_HOME variable in the tomcat.bat file, which is located in the \tomcat\bin folder. Make sure the JAVA_HOME variable is assigned the path where the JDK is installed on your computer.
9. Open a DOS window and type **\tomcat\bin\tomcat** to start Tomcat.
10. Open your browser. Enter **http://localhost:8080**. The Tomcat home page is displayed on the screen verifying that Tomcat is running.

Request String

The browser generates a user request string whenever the Submit button is selected. The user request string consists of the URL and the query string, as you learned at the beginning of this chapter. Here's a typical request string:

```
http://www.jimkeogh.com/jsp/myprogram.jsp?fname="Bob"&lname="Smith"
```

Your program needs to parse the query string to extract values of fields that are to be processed by your program. You can parse the query string by using methods of the JSP request object.

The getParameter(Name) is the method used to parse a value of a specific field. The getParameter() method requires an argument, which is the name of the field whose value you want to retrieve.

Let's say that you want to retrieve the value of the fname field and the value of the lname field in the previous request string. Here are the statements that you'll need in your JSP program:

```
<%! String Firstname = request.getParameter(fname);
    String Lastname = request.getParameter(lname);
%>
```

In the previous example, the first statement used the getParameter() method to copy the value of the fname from the request string and assign that value to the Firstname object. Likewise, the second statement performs a similar function, but using the value of the lname from the request string. You can use request string values throughout your program once the values are assigned to variables in your JSP program.

There are four predefined implicit objects that are in every JSP program. These are request, response, session, and out. The previous example used the request object's getParameter() method to retrieve elements of the request string. The request object is an instance of the HttpServletRequest (see Chapter 9). The response object is an instance of HttpServletResponse, and the session object is an instance of HttpSession. Both of these are described in detail in Chapter 9. The out object is an instance of the JspWriter that is used to send a response to the client.

Copying a value from a multivalued field such as a selection list field can be tricky since there are multiple instances of the field name, each with a different value. However, you can easily handle multivalued fields by using the getParameterValues() method.

The getParameterValues() method is designed to return multiple values from the field specified as the argument to the getParameterValues(). Here is how the getParameterValues() is implemented in a JSP program.

In this example, we're retrieving the selection list field shown in Listing 11-8. The name of the selection list field is EMAILADDRESS, the values of which are copied into an array of String objects called EMAIL. Elements of the array are then displayed in JSP expression tags.

Listing 11-8 Selecting a listing of fields.

```
<%! String [ ] EMAIL = request.getParameterValues("EMAILADDRESS ") ; %>
<P> <%= EMAIL [0]%> </P>
<P> <%= EMAIL [1]%> </P>
```

You can parse field names by using the getParameterNames() method. This method returns an enumeration of String objects that contains the field names in the request string. You can use the enumeration extracting methods that you learned in Java to copy field names to variables within your program.

The Quick Reference Guide in Chapter 9 contains a list of other commonly used HttpServletRequest class methods that you'll find useful when working with parameters.

Parsing Other Information

The request string sent to the JSP by the browser is divided into two general components that are separated by the question mark. The URL component appears to the left of the question mark and the query string is to the right of the question mark.

In the previous section you learned how to parse components of the query string, which are field names and values using request object methods. These are similar to the method used to parse URL information.

The URL is divided into four parts, beginning with the protocol. The protocol defines the rules that are used to transfer the request string from the browser to the JSP program. Three of the more commonly used protocols are HTTP, HTTPS (the secured version of HTTP), and FTP, which is a file transfer protocol.

Next is the host and port combination. The host is the Internet Protocol (IP) address or name of the server that contains the JSP program. The port number is the port that the host monitors. Usually the port is excluded from the request string whenever HTTP is used because the assumption is the host is monitoring port 80. Following the host and port is the virtual path of the JSP program. The server maps the virtual path to the physical path.

Here's a typical URL. The http is the protocol. The host is www.jimkeogh.com. There isn't a port because the browser assumes that the server is monitoring port 80. The virtual path is /jsp/myprogram.jsp.

```
http://www.jimkeogh.com/jsp/myprogram.jsp
```

User Sessions

A JSP program must be able to track a session as a client moves between HTML pages and JSP programs as discussed in Chapter 9. There are three commonly used methods to track a session. These are by using a hidden field, by using a cookie, or by using a JavaBean, which is discussed in the next chapter.

A hidden field is a field in an HTML form whose value isn't displayed on the HTML page, as you learned in Chapter 8. You can assign a value to a hidden field in a JSP program before the program sends the dynamic HTML page to the browser.

Let's say that your JSP database system displays a dynamic login screen. The browser sends the user ID and password to the JSP program when the Submit button is selected where these parameters are parsed and stored into two memory variables (see the "Request String" section).

The JSP program then validates the login information and generates another dynamic HTML page once the user ID and password are approved. The new dynamically built HTML page contains a form that contains a hidden field, among other fields. And the user ID is assigned as the value to the hidden field.

When the person selects the Submit button on the new HTML page, the user ID stored in the hidden field and information in other fields on the form are sent by the browser to another JSP program for processing.

This cycle continues where the JSP program processing the request string receives the user ID as a parameter and then passes the user ID to the next dynamically built HTML page as a hidden field. In this way, each HTML page and subsequent JSP program has access to the user ID and therefore can track the session.

The Quick Reference Guide in Chapter 9 contains a list of other commonly used HttpSession class methods that you'll find useful when working with a session.

Cookies

As you learned in Chapter 9, a cookie is a small piece of information created by a JSP program that is stored on the client's hard disk by the browser. Cookies are used to store various kinds of information, such as user preferences and an ID that tracks a session with a JSP database system.

You can create and read a cookie by using methods of the Cookie class and the response object as illustrated in Listing 11-9 and in Listing 11-10. Listing 11-9 creates and writes a cookie called userID that has a value of JK1234.

The program begins by initializing the cookie name and cookie value and then passes these String objects as arguments to the constructor of a new cookie. This cookie is then passed to the addCookie() method, which causes the cookie to be written to the client's hard disk.

Listing 11-10 retrieves a cookie and sends the cookie name and cookie value to the browser, which displays these on the screen. This program begins by initializing the

MyCookieName String object to the name of the cookie that needs to be retrieved from the client's hard disk. I call the cookie userID.

Two other String objects are created to hold the name and value of the cookie read from the client. Also I created an int called found and initialized it to zero. This variable is used as a flag to indicate whether or not the userID cookie is read.

Next an array of Cookie objects called cookies is created and assigned the results of the request.getCookies() method, which reads all the cookies from the client's hard disk and assigns them to the array of Cookie objects.

The program proceeds to use the getName() and getValue() methods to retrieve the name and value from each object of the array of Cookie objects. Each time a Cookie object is read, the program compares the name of the cookie to the value of the MyCookieName String object, which is userID.

When a match is found, the program assigns the value of the current Cookie object to the MyCookieValue String object and changes the value of the found variable from 0 to 1.

After the program reads all the Cookie objects, the program evaluates the value of the found variable. If the value is 1, the program sends the value of the MyCookieName and MyCookieValue to the browser, which displays these values on the screen.

The Quick Reference Guide in Chapter 9 contains a list of other commonly used Cookie class methods that you'll find useful when working with cookies.

Listing 11-9 How to create a cookie.

```
<HTML>
  <HEAD>
    <TITLE> JSP Programming </TITLE>
  </HEAD>
  <BODY>
    <%! String MyCookieName = "userID";
        String MyCookieValue = "JK1234";
        response.addCookie(new Cookie(MyCookieName, MyCookieValue));
    %>
  </BODY>
</HTML>
```

sting 11-10 How to read a cookie.

```
<HTML>
  <HEAD>
    <TITLE> JSP Programming </TITLE>
  </HEAD>
  <BODY>
    <%! String MyCookieName = "userID";
        String MyCookieValue;
        String CName, CValue;
```

```
int found=0;
Cookie[] cookies = request.getCookies();
for(int i=0; i<cookies.length; i++) {
  CName  = cookies[i].getName();
  CValue = cookies[i].getValue();
  if(MyCookieName.equals(cookieNames[i])) {
    found = 1;
    MyCookieValue = cookieValue;
  }
 }
 if (found ==1) { %>
  <P> Cookie name = <%= MyCookieName %> </P>
  <P> Cookie value = <%= MyCookieValue %> </P>
<%}%>
</BODY>
</HTML>
```

Session Objects

A JSP database system is able to share information among JSP programs within a session by using a session object. Each time a session is created, a unique ID is assigned to the session and stored as a cookie.

The unique ID enables JSP programs to track multiple sessions simultaneously while maintaining data integrity of each session. The session ID is used to prevent the intermingling of information from clients.

In addition to the session ID, a session object is also used to store other types of information, called *attributes*. An attribute can be login information, preferences, or even purchases placed in an electronic shopping cart.

Let's say that you built a Java database system that enables customers to purchase goods online. A JSP program dynamically generates catalogue pages of available merchandise. A new catalogue page is generated each time the JSP program executes.

The customer selects merchandise from a catalogue page, then jumps to another catalogue page where additional merchandise is available for purchase. Your JSP database system must be able to temporarily store purchases made from each catalogue page; otherwise, the system is unable to execute the checkout process. This means that purchases must be accessible each time the JSP program executes.

There are several ways in which you can share purchases. You might store merchandise temporally in a table, but then you'll need to access the DBMS several times during the session, which might cause performance degradation.

A better approach is to use a session object and store information about purchases as session attributes. Session attributes can be retrieved and modified each time the JSP program runs.

Listing 11-11 illustrates how to assign information to a session attribute. In this example, the program creates and initializes two String objects. One String object is assigned the name of the attribute and the other String object is assigned a value for the attribute. Next, the program calls the setAttribute() method and passes this method the name and value of the attribute.

Listing 11-12 reads attributes. The program begins by calling the getAttributeNames() method that returns names of all the attributes as an Enumeration.

Next, the program tests whether or not the getAttributeNames() method returned any attributes. If so, statements within the while loop execute, which assigns the attribute name of the current element to the AtName String object. The AtName String object is then passed as an argument to the getAttribute() method, which returns the value of the attribute. The value is assigned to the AtValue String object. The program then sends the attribute name and value to the browser.

The Quick Reference Guide in Chapter 9 contains a list of other commonly used HttpSession class methods that you'll find useful when working with a session.

Listing 11-11 How to create a session attribute.

```
<HTML>
  <HEAD>
    <TITLE> JSP Programming </TITLE>
  </HEAD>
  <BODY>
    <%! String AtName = "Product";
        String AtValue = "1234";
        session.setAttribute(AtName, AtValue);
    %>
  </BODY>
</HTML>
```

Listing 11-12 How to read session attributes.

```
<HTML>
  <HEAD>
    <TITLE> JSP Programming </TITLE>
  </HEAD>
  <BODY>
    <%! Enumeration purchases = session.getAttributeNames();
        while(purchases.hasMoreElements()){
          String AtName = (String)attributeNames.nextElement();
          String AtValue = (String)session.getAttribute(AtName); %>
          <P> Attribute Name <%= AtName %> </P>
          <P> Attribute Value <%= AtValue %> </P>
     <% } %>
  </BODY>
</HTML>
```

Quick Reference Guide

Syntax	Descriptions
void _jspService(HttpServletRequest request, HttpServletResponse response)	Corresponds to the body of the JSP page.

Table 11-1. *public interface HttpJspPage extends JspPage*

Sytax	Descriptions
void jspDestroy()	Automatically invoked when a JSP page is to be destroyed.
void jspInit()	Automatically invoked when the JSP page is initialized.

Table 11-2. *public interface JspPage extends Servlet*

Syntax	Descriptions
abstract Object findAttribute(String name)	Returns the value of an attribute.
abstract Object getAttribute(String name)	Returns an object associated with the name.
abstract Object getAttribute(String name, int scope)	Returns an object associated with the name.
abstract Enumeration getAttributeNamesInScope(int scope)	Returns all attributes of a scope.
abstract int getAttributesScope(String name)	Returns the scope based on name.
abstract ExpressionEvaluator getExpressionEvaluator()	Provides programmatic access to the ExpressionEvaluator.
abstract JspWriter getOut()	Determines if out object value of a JspWriter.

Table 11-3. *public abstract class JspContext extends java.lang.Object*

Syntax	Descriptions
abstract Map peekPageScope()	Peeks at the top of the page scope stack.
abstract Map popPageScope()	Removes a page scope from the stack.
abstract void pushPageScope(Map scopeState)	Places a page scope on the stack.
abstract void removeAttribute(String name)	Uses all scopes to delete an object reference associated with a name.
abstract void removeAttribute(String name, int scope)	Delete an object by name.
abstract void setAttribute(String name, Object attribute)	Registers a name associated with an object that has page scope semantics.
abstract void setAttribute(String name, Object o, int scope)	Registers a name associated with an object and scope.

Table 11-3. *public abstract class JspContext* ***extends java.lang.Object*** (continued)

Syntax	Descriptions
abstract String getSpecificationVersion()	Returns the version number of JSP

Table 11-4. *public abstract class JspEngineInfo* ***extends Object***

Syntax	Descriptions
static JspFactory getDefaultFactory()	Returns the default JSP factory.
abstract JspEngineInfo getEngineInfo()	Returns information about the JSP engine.

Table 11-5. *public abstract class JspFactory* ***extends Object***

Syntax	Descriptions
abstract PageContext getPageContext(Servlet servlet, ServletRequest request, ServletResponse response, String errorPageURL, boolean needsSession, int buffer, boolean autoflush)	Returns an instance of a PageContext.
abstract void releasePageContext (PageContext pc)	Releases an allocated PageContext object.
static void setDefaultFactory (JspFactory deflt)	Sets the default factory.

Table 11-5. *public abstract class JspFactory* ***extends Object*** (continued)

Syntax	Descriptions
abstract void clear()	Resets a buffer.
abstract void clearBuffer()	Resets a buffer.
abstract void close()	Closes and flushes a stream.
abstract void flush()	Flushes a stream.
int getBufferSize()	Returns the size of the buffer used by the JspWriter.
abstract int getRemaining()	Returns the number of unused bytes in the buffer.
boolean isAutoFlush()	Determines if a JspWriter is autoFlushing.
abstract void newLine()	Writes a line separator.

Table 11-6. *public abstract class JspWriter* ***extends Writer***

Syntax	Descriptions
abstract void print(boolean b)	Prints a boolean value.
abstract void print(char c)	Prints a character.
abstract void print(char[] s)	Prints an array of characters.
abstract void print(double d)	Prints a double-precision floating-point number.
abstract void print(float f)	Prints a floating-point number.
abstract void print(int i)	Prints an integer.
abstract void print(long l)	Prints a long integer.
abstract void print(Object obj)	Prints an object.
abstract void print(String s)	Prints a string.
abstract void println()	Writes a line separator string to terminate a line.
abstract void println(boolean x)	Prints a boolean value with a terminated line
abstract void println(char x)	Prints a character with a terminated line.
abstract void println(char[] x)	Prints an array of characters with a terminated line
abstract void println(double x)	Prints a double-precision floating-point number with a terminated line.
abstract void println(float x)	Prints a floating-point number with a terminated line.
abstract void println(int x)	Prints an integer with a terminated line.
abstract void println(long x)	Prints a long integer with a terminated line.
abstract void println(Object x)	Prints an Object with a terminated line.
abstract void println(String x)	Prints a String with a terminated line.

Table 11-6. *public abstract class JspWriter* ***extends Writer*** (continued)

Syntax	Descriptions
abstract void forward(String relativeUrlPath)	Redirects the ServletRequest and ServletResponse to another active component in the application.
abstract Exception getException()	Returns the value of the exception object.
abstract Object getPage()	Returns the value of the page object.
abstract ServletRequest getRequest()	Returns the value of the response object.
abstract ServletResponse getResponse()	Returns the value of the response object.
abstract ServletConfig getServletConfig()	Returns the instance of the ServletConfig.
abstract ServletContext getServletContext()	Returns the instance of the ServletContext.
abstract HttpSession getSession()	Returns the value of the session object.
abstract void handlePageException (Exception e)	Redirects an unhandled page level exception to an error page.
abstract void handlePageException (Throwable t)	Makes an unhandled page level exception Throwable.
abstract void **include**(String relativeUrlPath)	Processes the resource as part of the current ServletRequest.
abstract void initialize(Servlet servlet, ServletRequest request, ServletResponse response, String errorPageURL, boolean needsSession, int bufferSize, boolean autoFlush)	Initializes an uninitialized PageContext.
JspWriter popBody()	Updates the page scope "out" attribute of the PageContext; and returns the previous JspWriter "out" saved by the matching pushBody().
BodyContent pushBody()	Saves the current "out" JspWriter; updates the page scope "out" attribute of the PageContext; and returns a new BodyContent object.
abstract void release()	Resets the internal state of a PageContext, for potential reuse.

Table 11-7. *public abstract class PageContext* ***extends JspContext***

Syntax	Descriptions
Object evaluate(String attributeName, String expression, Class expectedType, Tag tag, PageContext pageContext)	Evaluates the expression contained in a request.
String validate(String attributeName, String expression)	Validates an expression at translation time.

Table 11-8. *public interface* ***ExpressionEvaluator***

Syntax	Descriptions
Object resolveVariable(String pName, Object pContext)	Resolves a variable within the given context.

Table 11-9. *public interface* ***VariableResolver***

Syntax	Descriptions
void doInitBody()	Prepares to evaluate the body.
void setBodyContent(BodyContent b)	Sets the bodyContent property.

Table 11-10. *public interface BodyTag extends IterationTag*

Syntax	Descriptions
void setDynamicAttribute(String uri, String localName, Object value)	Sets a dynamic attribute that is not declared in the Tag Library Descriptor.

Table 11-11. *public interface DynamicAttributes*

Syntax	Descriptions
int doAfterBody()	Processes the body content.

Table 11-12. *public interface IterationTag* ***extends Tag***

Syntax	Descriptions
void invoke(Writer out, Map params)	Executes the fragment and directs all output to a Writer.

Table 11-13. *public interface* ***JspFragment***

Syntax	Descriptions
int doTag()	Processes a tag.
Object getParent()	Returns the parent of a tag.
void setJspBody(JspFragment jspBody)	Sets the body of a tag as a JspFragment object.
void setJspContext(JspContext pc)	Set a page context in the protected jspContext field.
void setParent(Object parent)	Sets a parent of a tag.

Table 11-14. *public interface* ***SimpleTag***

Syntax	Descriptions
int doEndTag()	Processes an end tag.
int doStartTag()	Processes a start tag.
Tag getParent()	Returns the parent for a tag.
void release()	Instructs a Tag handler to release state.
void setPageContext(PageContext pc)	Sets a page context.
void setParent(Tag t)	Sets the parent of a tag handler.

Table 11-15. *public interface* ***Tag***

Syntax	Descriptions
void doCatch(Throwable t)	Executes when a Throwable exception happens while the body is being evaluated.
void doFinally()	Executes after doEndTag().

Table 11-16. *public interface* ***TryCatchFinally***

The Complete Reference



Chapter 12

Enterprise JavaBeans

The J2EE architecture consists of components that together enable developers to build a robust, industrial-strength J2EE application that takes advantage of the efficiencies of distributive, web services technology.

Part I of this book introduced you to the J2EE architecture and presented an overview of J2EE components. Three of these components form the nucleus of a J2EE application. These are Java servlets, JavaServer Pages (JSP), and Enterprise JavaBeans (EJB).

Typically, a J2EE application uses a browser-based user interface composed of a web page. An HTML form collects information from a user and formulates a request for web services, which is passed along to a server-side component. The server-side component is written in either Java servlet or JSP technology. A Java servlet and a JSP perform an intermediary function, as you learned in Chapters 10 and 11.

As an intermediary between a client and web services, a Java servlet and JSP receive a request for a web service from a client and fulfill the request by calling other server-side components that contain the business logic needed to comply with part or all of the request. The results are returned to the Java servlet or JSP, which responds to the client.

Many of the server-side components called by a Java servlet and JSP are written using EJB technology. In this chapter, you'll learn about EJB technology and how to create your own EJB. You'll also learn how to call an EJB from with a Java servlet and JSP.

Enterprise JavaBeans

An Enterprise JavaBean (EJB) is a component of the J2EE architecture that primarily provides business logic to a J2EE application and interacts with other server-side J2EE components. The nature of the business logic and the interactions with other server-side J2EE components are dependent on the J2EE application.

Let's say that a user of a J2EE application needs to display a catalog page that contains merchandise information. The J2EE application displays a web page that contains an HTML form where the user can enter the identification number of the merchandise. The browser then calls a JSP program. The JSP program parses the product number of the merchandise from the query string sent by the browser. The product number is then passed to an EJB, which interacts with a database to extract information about the merchandise. This information is returned to the JSP program, which formats the information into a web page and sends the web page to the browser for display.

An EJB is written in the Java programming language. This means that an EJB is operating system and platform independent, provided you're not using any native code. The developer of an EJB class focuses on writing business logic to respond to requests for web services and does not focus on system-level services unless required by the specifications for the EJB. The EJB server handles system-level services such as threading, security, transactions, and persistence if the EJB tells the server how to handle them or the EJB can handle them itself.

The EJB Container

An EJB container is a vendor-provided entity located on the EJB server that manages system-level services for EJB. The EJB container is one of several containers, each of which handles a J2EE component such as Java servlets and JSP.

An EJB container provides a reusable pool of distributed components. Each EJB must be installed in an EJB container. Typically, an EJB container contains many EJBs. There can also be multiple EJB containers. However, an EJB must be installed in only one EJB container.

A client's request for web services is made to a web server, which forwards the client's request to the appropriate server. Typically, either the Java servlet server or the JSP server receives the client's request, depending on which is used for the J2EE application.

The Java servlet or JSP uses JNDI (see Chapter 16) to look up the EJB resource. The EJB object is returned and is used to get the EJB's home class. The home class is used to reference the bean class through the EJB's remote class.

EJB Classes

There are three kinds of EJB types. These are the Entity JavaBean class, the Session Java Bean class and the Message-Driven JavaBean class. These are commonly referred to as an entity bean, a session bean, and a message-driven bean.

An entity bean is used to represent business data. A session bean is used to model a business process. And a message-driven bean is used to receive messages from a JMS resource—either a Queue or Topic, which are discussed later in this chapter.

EJB Interfaces

The session and entity beans must have two interfaces. These are the Home interface and the Remote interface. Both interfaces are declared by the developer of the EJB and are implemented by the EJB container.

Figure 12-1 illustrates the relationships that exist in an EJB environment. At the center of the EJB environment is the EJB that contains the logic necessary to provide service to a client. The EJB is wrapped by the EJB container, which handles the EJB's life-cycle requirements as described previously in this chapter. The EJB container handles communications between the EJB and other components in the EJB environment using the Home interface and the Remote interface.

The local clients that are on the same Java Virtual Machine as the EJB interact with the EJB using the Home interface. The Remote interface is used by remote clients that are capable of accessing the EJB container from an application that is compliant with RMI and IIOP. The Remote and Home interfaces can be used to access the EJB either remotely (outside the container that the EJB resides in) or locally within the same container. Local and LocalHome are part of the EJB 2.0 specification and are used to access the EJB within the same container. Local is used much like the Remote interface

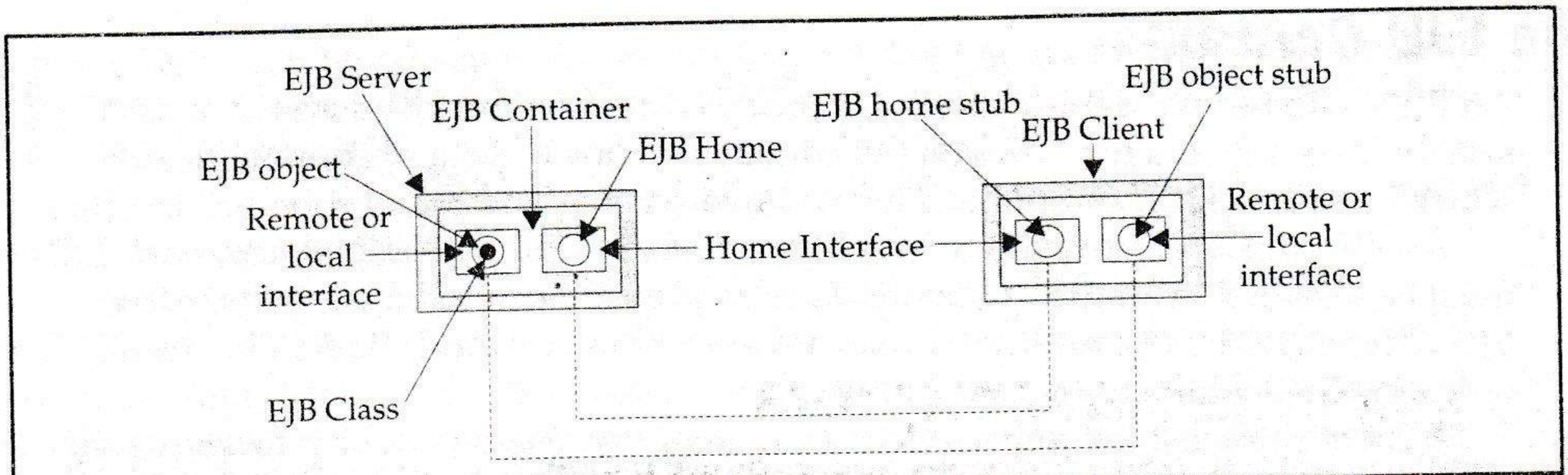


Figure 12-1. *Two interfaces are used to interact with an EJB. These are the Home interface and the Remote interface.*

and LocalHome is used in the same manner as the Home interface. The methods you expose in the Local interface have to match business methods in the bean class; however, the methods exposed through Local don't have to match Remote—you can expose a different set of methods if you choose to do so. This includes CORBA-based applications and non-Java programs. The Remote interface provides the client's view of the EBJ and has the prototype of each business method in the EJB. The EJB implements business methods and all the housekeeping methods.

Listing 12-1 illustrates a Home interface. The Home interface must extend the EJBHome interface. In this example, the interface is called myHome and declares one method, which is create(). The create() method returns myEJB object, which is the EJB that is used with the myHome interface. The create() method is defined in the EJB.

Listing 12-1 A simple Home interface for the myEJB EJB.

```
import javax.ejb.*;
import java.rmi.RemoteException;
public interface MyEJBHome extends EJBHome
{
 public MyEJB create() throws CreateException, RemoteException;
}
```

Listing 12-2 illustrates a Remote interface. The Remote interface is written using Java Remote Method Invocation syntax and declares methods that can be accessed by a remote client. The Remote interface extends the EJBObject. One method is declared in this example, which is called myMethod(). The definition of myMethod() is contained in the EJB bean class. There would be many more methods declared in the Remote interface in a real-world J2EE application. A method's signature is identical in the remote and bean classes except that the Remote interface declares a RemoteException.

sting 12-2
A simple
Remote
terface for
nyEJB EJB.

```
import java.rmi.RemoteException;
import javax.ejb.*;
public interface MyEJB extends EJBObject{
public String myMethod() throws RemoteException;
}
```

Deployment Descriptors

A deployment descriptor describes how EJBs are managed at runtime and enables the customization of EJB behavior without modification to the EJB code such as describing runtime attributes of transactional context. The behavior of an EJB can be modified within the deployment descriptor without having to modify the EJB class or the EJB interfaces.

A deployment descriptor is written in a file using XML syntax. Many times, an integrated development environment (IDE) used to create the EJB has a graphical user interface a developer can use to set deployment descriptor attributes. The IDE then generates the deployment descriptor file. A developer can also create a deployment descriptor file using an editor.

The deployment descriptor file is packaged in the Java Archive (JAR) file along with the other files that are required to deploy the EJB. These include classes and component interfaces that are necessary for each EJB in the package.

An EJB container references the deployment descriptor file to understand how to deploy and manage EJBs contained in the package. The deployment descriptor identifies the types of EJBs that are contained in the package as well as other attributes, such as how transactions are managed. The EJB container continues to reference the deployment descriptor after deployment.

Deployment descriptors are used in EJB 1.1 and EJB 2.0 and are nearly the same except the EJB 2.0 deployment descriptor has two elements that are not found in the EJB 1.1 deployment descriptor (<local> and <local-home>) and a different version is declared in the <!DOCTYPE>.

Listings 12-3 and 12-4 illustrate a skeleton of a deployment descriptor. Listing 12-3 is a deployment descriptor for EJB 1.1 and Listing 12-4 is a deployment descriptor for EJB 2.0. You'll notice that both deployment descriptors use XML elements similar to those you learned about in Chapter 8.

The deployment descriptor begins with the <!DOCTYPE> element that defines the URL for the document type definition (DTD) and the organization that defines the DTD. The DTD defines the structure of the XML document.

Next is the <ejb-jar> element, which is the root element of the deployment descriptor. Every deployment descriptor must contain an <ejb-jar> element and all other EJB elements must be nested with the <ejb-jar> element.

The first element within the <ejb-jar> element is the <enterprise-beans> element. It is here where the EJB is declared. There are three elements that can be contained within the <enterprise-beans> element. These are <entity>, <session>, and <message-driven>, each of which describes a type of EJB. However, the <message-driven> element is used only in EJB 2.0 and is not supported in EJB 1.1.

The <entity> element is used in Listing 12-3 and Listing 12-4 since the deployment descriptor describes an entity EJB, which you'll learn about later in this chapter. The <entity> element contains subelements that describe the entity EJB.

These include the <ejb-name> element that contains a descriptive name for the entity EJB. The <home> element describes the home interface that provides access to the entity EJB to remote clients. The <home> element must contain the fully qualified class name of the remote home interface. The <ejb-name> field is used to tie the two descriptors together and also used within the same descriptor to reference other attributes.

The <remote> element describes the fully qualified class name of the Remote interface, which defines the entity EJB's business methods to remote clients. The <local-home> element is used in EJB 2.0 deployment descriptors to describe the fully qualified class name of the local home interface. The local home interface identifies the entity EJB to co-locate EJB. Likewise, the <local> element is also used in EJB 2.0 deployment descriptors. The <local> element describes the fully qualified class name of the local interface. The local interface defines the entity EJB's business methods to other co-located EJB.

The <ejb-class> element describes the fully qualified class name of the EJB class, which implements the entity EJB's business methods. The <prim-key-class> element describes the fully qualified class name of the entity EJB's primary key that is used to locate data in a database.

The <persistence-type> element defines how the entity EJB manages persistence. As you'll learn later in this chapter, an entity EJB manages persistence through either container-managed persistence or bean-managed persistence. Therefore, the value for the <persistence-type> element must be either Container or Bean.

The <reentrant> element declares whether or not an entity EJB can be looped back without throwing an exception. A true value permits a look back, while a false value causes an exception to be thrown.

Listing 12-3 Deployment descriptor for EJB 1.1.

```
<!DOCTYPE ejb-jar PUBLIC "~//Sun Microsystems, Inc. //DTD EnterpriseJavaBeans
1.1//EN" "http://java/sun/com/dtd/ejb-jar_1_1/dtd">
<ejb-jar>
  <enterprise-beans>
    <entity>
     <ejb-name>myEJB</ejb-name>
     <home>com.jimkeogh.ejb.MyEJBHome</home>
```

```
      <remote>com.jimkeogh.ejb.MyEJBRemote</remote>
      <ejb-class>com.jimkeogh.ejb.MyEJB</ejb-class>
      <peristence-type>Container</persistence-type>
      <prim-key-class>java.lang.String</prim-key-class>
      <reentrant>False</reentrant>
    </entity>
  </enterprise-beans>
</ejb-jar>
```

Listing 12-4 Deployment descriptor for EJB 2.0.

```
<!DOCTYPE ejb-jar PUBLIC "~//Sun Microsystems, Inc. //DTD EnterpriseJavaBeans
2.0//EN" "http://java/sun/com/dtd/ejb-jar_2_0/dtd">
<ejb-jar>
  <enterprise-beans>
    <entity>
      <ejb-name>myEJB</ejb-name>
      <home>com.jimkeogh.ejb.MyEJBHome</home>
      <remote>com.jimkeogh.ejb.MyEJBRemote</remote>
      <local-home>com.jimkeogh.ejb.MyEJBHomeLocal</local-home>
      <local>com.jimkeogh.ejb.MyEJBLocal</local>
      <ejb-class>com.jimkeogh.ejb.MyEJB</ejb-class>
      <peristence-type>Container</persistence-type>
      <prim-key-class>java.lang.String</prim-key-class>
      <reentrant>False</reentrant>
    </entity>
  </enterprise-beans>
</ejb-jar>
```

The Anatomy of a Deployment Descriptor

The deployment descriptor skeleton provides a very brief overview of elements that are used in a typical deployment descriptor. There are many other elements that can be use to specify various deployment requirements and options for an EJB.

The root element of every standard deployment descriptor is the <ejb-jar> element. The <ejb-jar> element contains several subelements—some of which are required, while others are optional. Table 12-1 lists the subelements for the <ejb-jar> element. Some vendors require their own root element for vendor-specific deployment descriptors.

The <enterprise-beans> subelement contains its own set of subelements, three of which are the <session>, <entity>, and <message-driven> subelements. Each of these is used to describe a specific type of EJB, which you'll learn about later in this chapter.

The <session> subelement and the <entity> subelement contain other subelements that specify the deployment configuration for a session EJB and an entity EJB, respectively. These subelements are described in Table 12-2.

Table 12-3 lists subelements for the <message-driven> subelement. The <message-driven> subelement defines how a message-driven bean is to be deployed.

Subelement	Required/ Optional	Description
<description>	Optional	Describes the deployment descriptor.
<display-name>	Optional	Describes the JAR file and individual EJB components.
<small-icon>	Optional	Describes a small icon within the JAR file that is used to represent the JAR file.
<large-icon>	Optional	Describes a large icon within the JAR file that is used to represent the JAR file.
<enterprise-beans>	Required	Describes one or more enterprise beans contained in a JAR file. Only one <enterprise-beans> element is permitted in a deployment descriptor.
<ejb-client-jar>	Optional	Describes the path of the client JAR and is used by the client to access EJBs described in the deployment discriptor.
<assembly-descriptor>	Optional	Describes how EJBs are used in the J2EE application.

Table 12-1. *<ejb-jar> Subelements*

Subelement	Required/ Optional	Description
<description>	Optional	Describes the session or entity EJB.
<display-name>	Optional	Describes the JAR file and individual EJB components.
<small-icon>	Optional	Describes a small icon within the JAR file that is used to represent the session or entity EJB.

Table 12-2. *<session> and <entity> Elements*

Subelement	Required/ Optional	Description
<large-icon>	Optional	Describes a large icon within the JAR file that is used to represent the session or entity EJB.
<ejb-name>	One required	Describes the name of the session or entity EJB.
<home>	EJB 1.1 One required EJB 2.0 Optional	Describes the fully qualified class name of the session or entity EJB remote home interface.
<remote>	EJB 1.1 One required EJB 2.0 Optional	Describes the fully qualified class name of the session or entity EJB remote interface.
<local-home>	EJB 2.0 Optional	Describes the fully qualified class name of the session or entity EJB local home interface.
<local>	EJB 2.0 Optional	Describes the fully qualified class name of the session or entity EJB local interface.
<ejb-class>	One required	Describes the fully qualified class name of the session or entity EJB class.
<primkey-field>	Entity bean only, optional	Describes the primary key field for entity beans that use container-managed persistence.
<prim-key-class>	Entity bean only, one required	Describes the primary key class for entity beans.
<persistence-type>	Entity bean only, one required Values Container or Bean	Specifies either container-managed persistence or bean-managed persistence.

Table 12-2. *<session> and <entity> Elements* (continued)

Subelement	Required/ Optional	Description
<reentrant>	Entity bean only, one required Values true or false	Specifies that back (reentrant invocations) is allowed or not.
<cmp-version>	EJB 2.0 Optional EJB containers must support both EJB 2.0 CMP and EJB 1.1.	Specifies the version of container-managed persistence.
<abstract-schema-name>	EJB 2.0 Optional	Specifies entity beans in a JAR file.
<cmp-field>	Entity bean only, zero or more Must exist for each container-managed field in the entity EJB class. Must include a <field-name> element.	For entity beans with container-managed persistence.
<env-entry>	Optional - zero or more	Specifies an environment entry available through JNDI ENC.
<ejb-ref>	Optional - zero or more	Specifies a remote EJB reference available through the JNDI ENC.
<ejb-local-ref>	Optional - zero or more - EJB 2.0	Specifies a local EJB reference available through the JNDI ENC.
<resource-ref>	Optional - zero or more	Specifies reference to connection factory available through the JNDI ENC.
<resource-env-ref>	Optional - zero or more - EJB 2.0	Specifies required administered objects.
<security-role-ref>	Optional - zero or more	Specifies security roles.

Table 12-2. *<session> and <entity> Elements* (continued)

Subelement	Required/ Optional	Description
<security-identity>	Optional - EJB 2.0	Describes the principal for a method.
<session-type>	One required - session bean Value Stateful or Stateless	Specifies a session bean is either stateful or stateless.
<transaction-type>	One required - session bean Value Bean or Container	Specifies that a session bean manages transactions or the container manages transactions.
<query>	Optional - zero or more - EJB 2.0	Contains an EJB QL statement bound to a find or select method.

Table 12-2. *<session> and <entity> Elements* (continued)

Subelement	Required/ Optional	Description
<description>	Optional	Describes the session or entity EJB.
<display-name>	Optional	Describes the JAR file and individual EJB components.
<small-icon>	Optional	Describes a small icon within the JAR file that is used to represent the message-driven bean EJB.
<large-icon>	Optional	Describes a large icon within the JAR file that is used to represent the message-driven bean EJB.

Table 12-3. *<message-driven> Element*

Subelement	Required/ Optional	Description
<ejb-name>	One required	Describes the name of the message-driven bean EJB.
<ejb-class>	One required	Describes the fully qualified class name of the message-driven bean EJB class.
<transaction-type>	One required - session bean Value Bean or Container	Specifies that a message-driven bean manages transactions or the container manages transactions.
<security-identity>	Optional - EJB 2.0	Describes the principal for a method.
<env-entry>	Optional - zero or more	Specifies an environment entry available through JNDI ENC.
<ejb-ref>	Optional - zero or more	Specifies a remote EJB reference available through the JNDI ENC.
<ejb-local-ref>	Optional - zero or more - EJB 2.0	Specifies a local EJB reference available through the JNDI ENC.
<resource-ref>	Optional - zero or more	Specifies reference to connection factory available through the JNDI ENC.
<resource-env-ref>	Optional - zero or more - EJB 2.0	Specifies required administered objects.
<message-selector>	Optional Caution: Can cause problems with XML process. See CDATA	Specifies a conditional expression using Boolean logic to choose messages that are received from a topic or queue and delivered to a client.

Table 12-3. *<message-driven> Element* (continued)

Subelement	Required/ Optional	Description
<acknowledge-mode>	Required Only if EJB manages transactions. Value Auto-acknowledge or Dups-ok-acknowledge	Specifies the type of acknowledgement used when a message is received.
<message-driven-destination>	Required Values jacax.jms.Queue or javax.jms.Topic	Specifies the type of destination subscribed or listened to by the message-driven bean.

Table 12-3. *<message-driven> Element* (continued)

CDATA

Special characters used in a <message-selector> subelement and the <ejb-ql> subelement can cause parsing errors to occur when the deployment descriptor is processed. The problem is that special characters might be considered XML syntax, such as the less than and greater than signs.

You can avoid this problem by using a CDATA section within the <message-selector> or the <ejb-ql>, as is illustrated in Listing 12-5. The CDATA section tells the XML processor not to translate the special characters as XML syntax. Instead, these characters are treated as literal.

Listing 12-5 Inserting a CDATA section into a <message-selector> subelement.

```
<message-selector>
<![CDATA[
grade > 70 AND grade < 80);]]>
```

Environment Elements

The <env-entry> element is used in the deployment descriptor to define values that an EJB can use to customize the EJB's behavior. There are four subelements that can be used within the <env-entry> element. Some of these are required and others are optional.

The subelements are the <description>, <env-entry-name>, <env-entry-type>, and <env-entry-value>. The <description> subelement is optional and consists of text that describes the <env-entry>. The <env-entry-name> subelement is required and specifies the name of the <env-entity> element.

The <env-entry-type> subelement is also required and specifies the type of <env-entry-value> subelement. Finally, the <env-entry-value> subelement is optional and contains the value of the environment entry.

Listing 12-6 illustrates how to define an <env-entry> element in a deployment descriptor. Keep in mind that you need to place this code segment into a fully defined deployment descriptor, as discussed previously in this chapter.

This example creates an environment entry called PassingGrade that is an Integer type and has a value of 75. Listing 12-7 contains a code segment that illustrates how to access an environment entry from an EJB. In this example, a JNDI lookup is conducted to locate the PassingGrade environment entry.

sting 12-6 Defining an env-entry> ement in a eployment descriptor.

```
<env-entry>
  <env-entry-name>PassingGrade</env-entry-name>
  <env-entry-type>java.lang.Integer</env-entry-type>
  <env-entry-value>75</env-entry-value>
</env-entry>
```

isting 12-7 cessing the <env-entry> ement from deployment descriptor.

```
InitialContext jc = new InitialContext();
Integer pGrade = (Integer) jc.lookup("java:comp/env/PassingGrade");
```

Referencing EJB

The deployment descriptor can specify that an EJB can reference another EJB either locally or remotely by using the <ejb-local-ref> element or <ejb-ref> element. This is possible in EJB 2.0. These references are only remote in EJB 1.1.

The <ejb-local-ref> element is used to declare a local reference to EJBs that are co-located in the same container and should be deployed in the same EJB JAR file, although they could reside in different JAR files. The <ejb-local-ref> element must follow the <ejb-ref> element in the deployment descriptor.

Listing 12-8 illustrates how to define an <ejb-local-ref> element. In this example, the <ejb-local-ref> definition contains the EJB type (Entity) and the names of the local component interfaces. The <ejb-link> subelement is used to explicitly link the EJB to other co-located EJBs. The name specified in the <ejb-link> subelement must be the name of the appropriate EJB in the same JAR file. That is, the <ejb-link> subelement value must be the same as a value for the <ejb-name> subelement in the JAR file. The value of the <ejb-local_ref> elements are mapped to co-located EJBs at deployment

time but using a tool provided in the EJB container. Listing 12-9 illustrates how to access the reference using a JNDI lookup.

Listing 12-8 Defining the <ejb-local-ref> element.

```
<ejb-local-ref>
   <ejb-ref-name>ejb/myEJBHomeLocal</ejb-ref-name>
   <ejb-ref-type>Entity</ejb-ref-type>
   <local-home>com.myejb.myEJBHomeLocal</local-home>
   <local>com.myejb.myEJBLocal</local>
   <ejb-name>myEJB</ejb-name>
</ejb-local-ref>
<ejb-local-ref>
   <ejb-ref-name>ejb/yourEJBHomeLocal</ejb-ref-name>
   <ejb-ref-type>Entity</ejb-ref-type>
   <local--home>com.yourejb.yourEJBHomeLocal</local-home>
   <local>com.yourejb.yourEJBLocal</local>
   <ejb-name>yourEJB</ejb-name
</ejb-local-ref>
```

Listing 12-9 Looking up the local reference.

```
InitialContext jc = new InitialContext();
myEJBHomeLocal myEJBHomeLocal  = (myEJBHomeLocal)
jc.lookup("java:comp/env/ejb/myEJBHomeLocal");
```

Reference to a remote EJB is made by using the <ejb-ref> element. The <ejb-ref> element has several required and optional subelements. The <description> subelement contains a textual description of the <ejb-ref> element and is optional. The <ejb-ref-name> subelement specifies the name of the reference and is a required subelement—as is the <ejb-ref-type> subelement, which specifies the type of the EJB.

Next are the required <remote> subelement and the <home> subelement. These specify the remote and home interfaces, respectively. The last subelement is <ejb-link>, which is optional and is used to explicitly link the EJB to the remote EJB.

Listing 12-10 illustrates how to define a remote reference. In this example, reference is made to myEJBHome, which is a session EJB. Listing 12-11 illustrates how to access the reference.

ting 12-10 Defining a remote reference.

```
<ejb-ref>
   <ejb-ref-name>ejb/myEJBHome</ejb-ref-name>
   <ejb-ref-type>Session</ejb-ref-type>
   <home>com.myEJB.myEJBHome</home>
   <remote>com.myEJB.myEJBHome</remote>
   <ejb-link>myEJB</ejb-link>
</ejb-ref>
```

Listing 12-11 Accessing a remote reference.

```
InitialContext jc = new IntitalContext():
Object obj = jc.lookup("java:comp/env/ejb/myEJBHome");
myEJBHome myEJBhome = (myEJBHome)
PortableRemoteObject.narrow(obj, myEJBHome.class);
```

Reference Other Resources

The deployment descriptor is also used to describe references to external resources such as a database that can be accessed by an EJB. External references are specified by using the <resource-ref> element.

The <resource-ref> element contains several subelements. These are an optional <description> subelement, a required <res-ref-name>, a required <res-type>, and a required <res-auth>.

The <description> is used to describe the external resource and the <res-ref-name> is a name assigned to the external resource for use within the deployment descriptor. The <res-type> subelement specifies the type of resource and the <res-auth> subelement indicates the service responsible for login. This can either be the container or the application.

Listing 12-13 illustrates how to define a reference to an external resource. The resource in this example is a database called myDB. The type of the resources is javax.sql.DataSource. And the container handles all the login chores. Listing 12-14 shows how to access the external resource.

Listing 12-13 Defining an external resource.

```
<resource-ref>
  <Description>myDatabase</description>
  <res-ref-name>jdbc/myDB</res-ref-name>
  <res-type>javax.sql.DataSource</res-type>
  <res-auth>Container</res-auth>
</resource-ref>
```

Listing 12-14 Accessing an external resource.

```
InitialContext jc = new Initialcontext();
DataSource myDB = (DataSource) jc.lookup("java:comp/env/jdbc/myDB");
```

Sharing Resources

In the real world, EJBs work together to complete a transaction. Many times these EJBs interact with the same resource, such as a database. By default, EJB containers share resources. This means that EJBs that work on the same transaction use the same connection to access the common resources. Without sharing, each EJB opens its own connection. However, sometimes sharing resources can have a negative impact

on the operations of the EJB. Therefore, you can explicitly activate or deactivate shared resources within the deployment descriptor.

Resource sharing is specified in the deployment descriptor by using the <res-sharing-scope> subelement within the <resource-ref> element. The <res-sharing-scope> subelement has one of two values. These are Shareable or Unshareable.

Security Elements

The <security-role-ref> element is used to specify an EJB's security role. There are three subelements within the <security-role-ref> element. These are <description>, <role-name>, and <role-link>.

The <description> subelement is optional and is used to provide a textual description for the <security-role-ref>. The <role-name> subelement is required. The value of the <role-name> subelement must be the same as the role name used in the EJBContext.isCallerInRole() method in the EJB.

The <role-link> subelement is optional and is used to map the EJB's role name to a logical role contained in the <security-role> element of the <assembly-descriptor> element in the deployment descriptor. The <security-role-ref> element is matched to the environment security role if the <role-link> subelement is not specified in the <security-role-ref> element.

Listing 12-15 illustrates how to define the <security-role> element in the deployment descriptor. In this example, the <role-name> is Supervisor and the <security-role-ref> subelement is matched to an environment security role. Listing 12-16 shows how the EJBContext.isCallerInRole() method is called within the EJB and matches the <role-name> in the deployment descriptor.

ting 12-15 efining the curity-role> element.

```
<security-role-ref>
  <description>myEBJ security role </description>
   <role-name>Supervisor</role-name>
</security-role-ref>
```

ting 12-16 erifying the role name from within an EJB.

```
boolean supervisor = context.isCallerInRole("Supervisor");
  if (!supervisor) {
    throw new AccessDeniedException();
  }
```

Query Element

A <query> element is used in a deployment descriptor to specify a query method and a QL statement that is used as the criteria for selecting data from a relational database. The <query> element has two subelements. These are <query-method> and <ejb-ql>.

The <query-method> subelement specifies the method that is invoked to execute the QL statement. The <ejb-ql> subelement defines the search criteria.

The <query-method> subelement itself has two subelements. These are the <method-name> subelement and the <method-params> subelement. The <method-name> subelement specifies the name of the method and the <method-params> subelement identifies parameters, if any, that are passed to the method. Each parameter is described by type in the <method-param> subelement.

The <ejb-ql> subelement of the <query> element contains a SQL statement that is used to retrieve information from the database. A question mark is used as a placeholder for a parameter passed to the method, if a parameter is passed to the method.

Listing 12-17 illustrates how to define a <query> element that specifies a method that does not have a parameter. In this example, RetrieveRoster is the name of the method. The <method-params> subelement does not contain a value because the RetrieveRoster method doesn't receive a parameter. The <ejb-ql> subelement contains the QL statement that retrieves the roster from the Rosters table.

Listing 12-18 shows how to define a <query> element that specifies a method that receives one parameter. In this example, the findbyCompany method receives a String as a parameter. The parameter then replaces the question mark in the <ejb-ql> subelement's QL statement. The QL statement retrieves the company object from the vendor table where the company matches the parameter.

Listing 12-17 Defining a <query> element without a parameter.

```
<query>
  <query-method>
    <method-name>RetrieveRoster</method-name>
    <method-params></method-params>
  </query-method>
  <ejb-ql>
    SELECT OBJECT(r) FROM Rosters As r
  </ejb-ql>
</query>
```

Listing 12-18 Defining a <query> element with a parameter.

```
<query>
  <query-method>
    <method-name>findbyCompany</method-name>
    <method-params>
      <method-param>java.lang.String</method-param>
    </method-params>
  </query-method>
  <ejb-ql>
    SELECT OBJECT (v) FROM vendor v WHERE v.company = ?
  </ejb-ql>
</query>
```

Relationship Elements

An entity bean can model relationships similar to those found in the relational database model (see Part II). Relationships are divided into two groups. These are cardinality and directional. There are four types of cardinality relationships. These are

- One-to-one
- One-to-many
- Many-to-one
- Many-to-many

Each of these cardinality relationships has one of two directions, which are unidirectional or bidirectional. A unidirectional relationship is one where the relationship flows in one direction. For example, a customer places an order. The order doesn't place a customer; therefore, the customer-to-order relationship is unidirectional.

In contrast, a bidirectional relationship is one where the relationship flows in both directions. This is the case between a customer and a sales representative. A customer initiates contact with a sales representative and a sales representative can initiate contact with a customer.

The relationships are defined in the deployment descriptor using the <relationships> element, which is illustrated in Listing 12-19. In this example, the deployment descriptor defines a bidirectional, one-to-one relationship between myEJB and yourEJB. Keep in mind that a real-word deployment descriptor would have more statements than those shown in this example, to specify how one or more EJBs are to be deployed.

This example begins by defining two enterprise beans using the <enterprise-beans> element and <entity> subelement. Once the EJBs are defined, the deployment descriptor uses the <ejb-relationship> element to define the relationship between these EJBs. Two relationship roles are defined, which are myEJB to yourEJB and yourEJB to myEJB.

The first of these relationship roles are defined using the <ejb-relationship-role> element. There are three subelements in the <ejb-relationship-role> element. These are the <ejb-relationship-role-name>, <multiplicity>, the <ejb-relationship-role-source>, and the <cmr-field> subelements.

The <ejb-relationship-role-name> subelement specifies the name for the relationship role that is being defined. The first relationship role is named myEJB-one-yourEJB. The <multiplicity> subelement is used to define the relationship. The value of the <multiplicity> subelement is either One or Many, which reflects one of the cardinality relationships.

The <ejb-relationship-role-source> subelement identifies the EJB that is initiating the relationships, which is myEJB in the first relationship role and, later, yourEJB in the second relationship role. The <cmr-field> subelement contains the <cmr-field-name> element. The <cmr-field-name> element identifies the field name common to both EJBs that is used to create the relationship between the EJBs. In this example, homeID is the common field.

If the relationship is unidirectional, a second relationship role doesn't need to be defined in the deployment descriptor. However, this example is bidirectional and therefore the yourEJB-to-myEJB relationship role is defined.

Listing 12-19 Defining relationships in the deployment descriptor.

```
<ejb-jar>
   <enterprise-beans>
      <entity>
      <ejb-name>myEJB</ejb-name>
      <local-home>com.ejb.myejb.myEJBHomeLocal</local-home>
      <local>com.ejb.myejb.myEJBLocal</local>
     </entity>
     <entity>
      <ejb-name>yourEJB</ejb-name>
      <local-home>com.ejb.yourejb.yourEJBHomeLocal</local-home>
      <local>com.ejb.yourejb.mathGrade.yourEJBLocal</local>
     </entity>
   </enterprise-beans>
   <relationships>
     <ejb-relation>
       <ejb-relation-name>myEJB-yourEJB</ejb-relation-name>
       <ejb-relationship-role>
       <ejb-relationship-role-name>myEJB-one-yourEJB</ejb-relationship-role-name>
        <multiplicity>One</multiplicity>
        <ejb-relationship-role-source>
          <ejb-name>myEJB</ejb-name>
        </ejb-relationship-role-source>
        <cmr-field>
         <cmr-field-name>homeID</cmr-field-name>
        </cmr-field>
       </ejb-relationship-role>
       <ejb-relationship-role>
       <ejb-relationship-role-name>yourEJB-one-myEJB</ejb-relationship-role-name>
        <multiplicity>One>/multiplicity>
        <ejb-relationship-role-source>
          <ejb-name>yourEJB</ejb-name>
        </ejb-relationship-role-source>
       </ejb-relationship-role>
    </relationships>
  </ejb-relation>
</ejb-jar>
```

Assembly Elements

The <assembly-descriptor> element is the second major section of the <ejb-jar> element. The first is the <enterprise-beans> element, which was discussed previously

in this chapter. The <assembly-descriptor> element specifies method permissions, logical security roles, and transaction attributes. The <assembly-descriptor> element contains three subelements. These are <method-permission>, <security-role>, and <container-transactions>.

The <method-permission> is an optional subelement that describes security roles that can call an EJB's method. There are three subelements in the <method-permission> subelement. These are <description>, <role-name>, and <method>. The <description> subelement is the only one of the three subelements that is optional.

The <security-role> subelement is also optional and is used to define the security role that is used when accessing an EJB. There are two subelements within the <security-role> subelement. These are <description> and <role-name>. The <description> subelement is optional and the <role-name> subelement, is used to define the security role.

The <container-transactions> subelement is optional and is used to specify transaction attributes that apply to a method. There are three subelements contained within the <container-transactions> subelement. These are <description>, <method>, and <trans-attribute>. The <description> subelement is optional, and there must be at least one <method> subelement and only one <trans-attribute>. Session EJBs do not require a <container-transactions> subelement if the session EJB manages its own transactions; otherwise, a <container-transactions> subelement is necessary. All remote and home interface methods of an entity EJB must have a <container-transaction> subelement.

Method Permissions

Security roles and method permissions work together to specify that a role can invoke an EJB method. A role defines permissions a group of users have to call a method. As mentioned previously in this chapter, each security role has a <role-name> subelement that identifies the security role. The value associated with the <role-name> subelement is considered a label that has no relationship to security role names used in an environment.

For example, a <role-name> might have the value "Manager," but there isn't a "Manager" security role name in the environment's security mechanism. That is, the environment doesn't have a user group called "Manager" that is assigned specific rights.

However, you can assign values to the <role-name> subelement that correspond to security role names used in the environment's security mechanism, although there isn't a direct connection between them.

The <method-permission> subelement maps security roles to methods in the EJB's home interface and remote interface. Listing 12-20 illustrates how to define security oles and map them to method permissions.

This example specifies two method permissions. The first maps the Director security role to all the methods of the myEJB EJB. The asterisk is used in the <ejb-name> subelement as a wildcard indicating all methods of the EJB. The second maps the Faculty security role to a single method of the myEJB EJB. This is the getRoster method, as indicated in the <method-name> subelement.

This means that anyone having the Director security role can access all the methods of the myEJB EJB, including the getRoster method. However, those with the Faculty security role can access the getRoster method only.

Listing 12-20 Defining security roles and method permissions.

```
<method-permission>
  <role-name>Director</role-name>
  <method>
    <ejb-name>myEJB</ejb-name>
    <method-name>*</method-name>
  </method>
</method-permission>
<method-permission>
  <role-name>Faculty</role-name>
  <method>
    <ejb-name>myEJB</ejb-name>
    <method-name>getRoster</method-name>
</method>
</method-permission>
```

Security permissions are normally validated before access is given to a method. However, EJB 2.0 enables you to activate or deactivate this validation process by placing the <unchecked/> subelement in the <method-permission> subelement. Notice that <unchecked/> is an empty element. That is, the open and close tags are one.

Listing 12-21 illustrates how the <unchecked/> subelement is specified. Only the methods within the <method-permission> subelement are affected by the <unchecked/> subelement.

Listing 12-21 Defining security roles and method permissions.

```
<method-permission>
  <role-name>Director</role-name>
  <method>
    <ejb-name>myEJB</ejb-name>
    <method-name>*</method-name>
  </method>
</method-permission>
<method-permission>
  <unchecked/>
  <role-name>Faculty</role-name>
  <method>
    <ejb-name>myEJB</ejb-name>
    <method-name>getRoster</method-name>
</method>
</method-permission>
```

EJB 2.0 reintroduced the <runas> element that was originally included in EJB 1.0 but abandoned in EJB 1.1 because the vendor had a difficult time implementing it. The <runas> element is a subelement within the <security-identity> element.

The <runas> specifies the security identity that the EJB must run under. This is illustrated in Listing 12-22. In this example, myEJB runs under the Director security role.

Listing 12-22 Specifying <run-as> subelement in the <security-identity> element.

```
<entity>
  <ejb-name>myEJB</ejb-name>
  <security-identity>
    <run-as>
      <role-name>Director</role-name>
    </run-as>
  </security-identity>
</entity>
```

Transaction Elements

Many EJBs are designed to perform a transaction. A transaction is to execute a unit of work that may involve multiple tasks. However, the unit of work isn't completed unless all the tasks are successfully executed. If one task fails, the transaction fails and all successfully completed tasks of the transaction must be rolled back.

An EJB developer can declare a transaction using the deployment descriptor rather than explicitly controlling each task of the transaction within the EJB. This means that the developer can control the behavior of a transaction without modifying the business logic of the EJB.

Tasks of a transaction are defined as EJB methods. More than one EJB can take part in a transaction. Then, every EJB that participates in a transaction is considered to be within the transaction scope. The thread of execution of a transaction follows the transaction scope.

Let's say an order is placed. There is a four-step transaction used to process the order. First, the order message is read from the queue of incoming messages. Next, the J2EE application sends an email confirmation to the JMS queue notifying the customer that the order has been received and is being processed.

The third step in the transaction is to record the order by entering the order into the database. The final step is to send a JMS shipping order message to the warehouse to fulfill the order. If any of these steps fail, completed steps can be rolled back, which leaves the order message on the queue ready to be processed again.

Caution *In the real world, rolling back portions of a transaction that are processed by different vendors might be troublesome. For example, the JMS queue and the database are typically implemented by different vendors and therefore depend on the vendor's implementation of transactions for a rollback to be successful. Always load test a J2EE application that uses transactions to be sure that various vendor implementations don't cause problems with transactions.*

Any EJB method involved in the transaction can throw an exception that may or may not cause a rollback of the transaction. A rollback occurs only if a task needs to be reversed. Let's say that the credit card company doesn't approve the authorization in our example. Should the previous task, which examines the customer credit rating, be rolled back?

Of course not, because the examination didn't change the data in the customer's account. However, the EJB method could have posted the transaction to the customer's account at the same time as the credit check. In this scenario, a rollback is necessary to remove the transaction from the customer's account.

EJB Transaction Attributes An EJB server will manage a transaction based on the EJB's transaction attribute when the EJB is deployed. This is referred to as Container managed. A transaction attribute can apply at the EJB level or the method level. All methods of the EJB are affected by the transaction attribute at the EJB level, while only specified methods are affected at the method level.

An EJB's transaction attribute is defined in the deployment descriptor by assigning one of the following values to the transaction attribute. Table 12-4 lists the transaction context the Container passes to the business method and resource managers based on the function of the transaction.

- **Mandatory** The EJB must be within the transaction scope of the calling EJB (or client). A TransactionRequiredException (remote client) or TransactionRequiredLocalException (local client) is thrown if the calling EJB (or client) is not in the scope of a transaction.
- **Never** The EJB must not be within a transaction scope. The EJB throws a RemoteException (remote client) or an EJBException (local client) if the calling EJB (or client) is involved in a transaction.
- **NotSupported** The EJB and EJBs it calls are not within the transaction scope. The transaction is suspended while the EJB executes, and resumes once the EJB's execution is completed.
- **Required** The EJB must be invoked within a transaction. The EJB can be part of another transaction or create its own transaction when called. The new transaction scope contains the EJB and EJBs it calls.
- **RequiresNew** A new transaction is started when the EJB is called regardless if the calling EJB (or client) is part of an existing transaction. An existing transaction is suspended if it exists until the new transaction is completed.
- **Supports** The EJB is included in the transaction scope. The EJB and EJBs it calls are part of the transaction. The EJB can interact with clients and EJBs that are not part of the transaction.

Caution *The EJB must be able to function with or without a transaction.*

Transaction Attribute	Client	Associated with Business Method	Associated with Resource Managers
NotSupported	None	None	None
NotSupported	Passed with Client Request	None	None
Required	None	Initiated by Container	Initiated by Container
Required	Passed with Client Request	Passed with Client Request	Passed with Client Request
Supports	None	None	None
Supports	Passed with Client Request	Passed with Client Request	Passed with Client Request
RequiresNew	None	Initiated by Container	Initiated by Container
RequiresNew	Passed with Client Request	Initiated by Container	Initiated by Container
Mandatory	None	Error	Not Available
Mandatory	Passed with Client Request	Passed with Client Request	Passed with Client Request
Never	None	None	None
Never	Passed with Client Request	Error	Not Available

Table 12-4. *Transaction Context Passed by the Container to Business Method and Resource Managers*

Listing 12-23 illustrates how to set the transaction attribute in the deployment descriptor. This example contains only an abbreviated deployment descriptor. A typical deployment descriptor contains many more statements than appear here, as you learned in the deployment descriptor section of this chapter.

A transaction is defined in the <container-transaction> element of the deployment descriptor, which is a subelement of the <assembly-descriptor>. There are two general components of the <container-transaction> element. These are defined in the <method> subelement and the <trans-attribute> subelement.

The <method> subelement identifies the EJB and the EJB's method whose transaction attribute is being set. The <ejb-name> subelement identifies the name of the EJB and the <method-name> subelement identifies the method. The <trans-attribute> subelement identifies the value of the transaction attribute.

Two transaction attributes are set in Listing 12-23. The first one is set for all the methods of the myEJB EJB. The transaction attribute is set to Never, which means none of the methods in myEJB can be within a transaction scope.

The other transaction attribute is for the myMethod of the yourEJB. Here, the transaction attribute is set to NotSupported, which means that the myMethod and EJB and methods it calls are not within the transaction scope.

This is all assuming you specified container-managed transactions in the <transaction-type> attribute of the EJB.

Listing 12-23 Setting the transaction attribute.

```
<ejb-jar>
  <assembly-descriptor>
    <container-transaction>
      <method>
        <ejb-name>myEJB</ejb-name>
          <method-name>*</method -name>
      </method>
      <trans-attribute>Never</trans-attribut >
    </container-transaction>
    <container-transaction>
      <method>
        <ejb-name>yourEJB</ejb-name>
          <method-name>myMethod</method -name>
      </method>
      <trans-attribute>NotSupported</trans-attribute>
    </container-transaction>
  <assembly-descriptor>
</ejb-jar>
```

Transaction Limitations Container-managed persistence (CMP) entity beans should only use either the Mandatory, Required, or RequiresNew transaction attributes to assure proper database access during the transaction. Some EJB container/server vendors may offer support for Never, Supports, and NotSupported transaction attribute values for CMP entity beans.

CMP entity beans also restrict access to collections within a transaction. A Collection object of a transaction cannot be used by another transaction. However, the Collection object can be used by another bean if the bean obtains and accesses the same transaction context.

Message-driven beans should only use either NotSupported or Required transaction attribute values since other transaction values apply only to client transactions. As you'll learn later in this chapter, message-driven beans are not called by clients.

Exclude List Element

EJB 2.0 introduced another way to disable access to a method—by using the <exclude-list> element. The <exclude-list> element specifies methods that are uncallable. Attempts to invoke a method that is uncallable requires clients to receive either a java.rmi.RemoteException or javax.ejb.AccessLocalException, depending on whether a remote client or local client attempted to invoke the method.

Listing 12-24 illustrates how to specify the <exclude-list> element. The <exclude-list> element is a subelement within the <assembly-descriptor> element. The <exclude-list> element requires the <method> element and related subelements within the <method> element. One of these is the <ejb-name> subelement, which identifies the EJB that contains the method(s) being excluded. Also required is the <method-name> element, which is used to specify the name of the method that is being excluded.

In this example, myMethod of the myEJB EJB is contained within the <exclude-list> element, which means that the myMethod is not callable. You can place as many methods within the <exclude-list> as is necessary according to the business rules of the J2EE application.

Listing 12-24 Specifying methods that are not to be called.

```
<ejb-jar>
  <enterprise-beans>
    <entity>
      <ejb-name>myEJB</ejb-name>
    </entity>
  </enterprise-beans>
  <assembly-descriptor>
    <exclude-list>
      <method>
       <ejb-name>myEJB</ejb-name>
      <method-name>myMethod</method-name>
      </method>
    </exclude-list>
  </assembly-descriptor>
</ejb-jar>
```

Session Java Bean

A session bean contains business logic used to provide a service to a client and exists for the duration of the client server session. A session bean terminates once the session with the client server terminates.

A session bean can be stateless or stateful. A stateless session bean doesn't retain state between method calls and typically performs business logic that doesn't require data to

be maintained during the session. This means that a client server can call any instances of a stateless session bean. It is common for an EJB container to create a pool of instances of a stateless session bean. The pool size is controlled by the deployment descriptor.

A stateful session bean retains data (state) between method calls with a client during a session. When calls are made to multiple methods, the stateful session bean is able to retain state between calls. This means the second method call between the client and the session bean can access data from its first method call.

The session bean can implement the SessionSynchronization interface, which makes the container notify the session bean of transaction-related events. Multiple instances of a session bean can exist; however, the state of each session bean is exclusive to an instance, and therefore each instance accesses its own state. Implementing SessionSychronization is optional. Simply stated, if you implement SessionSychronization, the container will notify the bean of three transactional events: beforeBegin(), beforeCompletion(), and afterCompletion(boolean committed).

Although a stateful bean retains state during a session, a session bean by its nature is not persistent. That is, transitional state is maintained for the life of the session. Once the session terminates, the EJB container removes the handle to the session bean and in effect removes any record of the session bean's identity. This means there isn't any way for the session bean developer to retrieve the instance of the session bean and its state after the session is completed.

Stateless vs. Stateful

Before creating a session bean, you must determine if the session bean should be stateless or stateful. Stateless session beans are generally more resource efficient than a stateful session bean because a stateless session bean can be used by the EJB container to service another request instead of waiting for the next method call.

A stateless session bean is shared amongst clients. In contrast, a stateful session bean is dedicated to a client. The more clients that use the stateful session bean, the more resources that are necessary to support those clients, and stateful session beans must be moved in and out of memory if there is insufficient memory to hold all instances of the stateful session bean.

Although a stateless session bean utilizes resources more efficiently than a stateful session bean, a stateless session bean can participate in a transaction the same way a stateful one can. If a stateless session bean needs to share data between calls, you should make a stateful session bean. Both deal with transactions the same way.

Creating a Session Java Bean

A session bean is a Java class that implements the SessionBean interface. The SessionBean interface requires that five methods be defined in the session bean class. These are ejbActivate(), ejbPassivate(), ejbRemove(), setSessionContext(SessionContext), ejbCreate(), and methods that contain business logic. The EJB container calls each of these methods at an appropriate time during the life of the session bean.

Listing 12-25 contains a skeleton of a session bean class. Although each method in this listing simply displays a brief message, they would normally execute appropriate statements in a real-world session bean class.

The ejbActivate() method is called whenever the session bean is removed from the pool and is referenced by a client. You should place routines in the ejbActivate() method that reacquire resources that were released when the session bean entered the passive state. A passive state occurs between method calls during a session.

The ejbPassivate() method is called before the instance enters the "passive" state when the session bean is returned to the object pool and should contain routines that release resources. Those resources can be required when the ejbActivate() method is called

The ejbRemove() method is called just before the bean is available for garbage collection. This is when the session bean is removed from the object pool (removed from memory). You should place routines in this method that release resources that were encumbered during the life of the session bean. The setSessionContext(SessionContext ctx) method is called before any business methods are called. This is where transactions are managed and accessed. The SessionContext object contains a variety of methods such as getEnvironment(), which returns a Properties objects. The Quick Reference Guide at the end of this chapter contains a complete listing of SessionContext methods.

The create() in the Home interface obtains a reference to a session bean from the EJB container. The session bean could be from the object pool or a new instance. The ejbCreate() method is used to initialize fields of the EJB instances and must have parameters that match the create() method in the Home interface.

The ejbCreate() method should take the form of ejbCreateSuffix(), where Suffix represents the action of the method. For example, the suffix ByCustomerID can be used in the name of the ejbCreate() method as ejbCreateByCustomerID().

Caution *The ejbCreateSuffix() cannot be used in EJB 1.1. This convention is used in EJB 2.0.*

The myMethod() method is a method that contains business logic that is customized to the service provided by the EJB. You can replace myMethod() with one or multiple methods as required by the J2EE application. However, be sure to declare those methods in the Remote interface. Once the session bean is defined, you must reference the session bean in the deployment descriptor, as discussed previously in this chapter.

Listing 12-25 A code skeleton of a session bean.

```
import javax.ejb.*;
public class MyEJBBean implements SessionBean {
  public void ejbActivate()
  {
    System.out.println("Called ejbActivate()");
  }
  public void ejbRemove()
```

```
{
    System.out.println("Called ejbRemove()");
}
public void ejbPassivate()
{
    System.out.println("Called ejbPassivate()");
}
public void setSessionContext(SessionContext ctx)
{
    System.out.println("Called setSessionContext()");
}
public void ejbCreate ()
{
  System.out.println("Called ejbCreate()");
}
public String myMethod ()
{
  return("Called myMethod()");
}
}
```

Entity Java Bean

An entity bean is considered the powerhouse of a J2EE application because an entity bean is used to manage a collection of data retrieved from a database and stored in memory. An entity bean inserts, updates, and removes data while maintaining the integrity of the data.

Data collected and managed by an entity bean is referred to as *persistent data* and is managed in one of two ways: using Bean-Managed Persistence (BMP) or Container-Managed Persistence (CMP). BMP requires the bean to manage persistence and CMP requires the container to do the same. The EJB container synchronizes data stored in memory by the entity bean with the database. An entity bean is considered light on business logic and heavy on data management.

The entity bean developer uses JDBC API (see Part II) within the entity bean to interact with a relational database or uses other methods to access nonrelational databases. Alternatively, the entity bean developer can call a data access object to access data. Multiple clients can share an entity bean simultaneously.

An entity bean can have a remote interface, a local interface, or both interfaces. A remote interface enables remote clients to access the entity bean while the local interface provides access to clients running within the same environment. Typically, a remote interface is used by a client not located in the EJB container and Local is used by

clients within the same EJB container. Local calls to an entity bean reduce the overhead commonly occurring in remote method invocations.

Container-Managed Persistence

A CMP entity bean is heavily dependent on support from the EJB container. The EJB container synchronizes the state of the entity bean with the database. However, EJB container support varies by vendor. Most vendors support automatic persistence between the entity bean and a relational database. A few vendors support more sophisticated object-to-relational mapping, which is where objects are automatically synchronized to a relational database.

There are three groups of methods that are typically contained in an entity bean. These are creation methods, business methods, and callback methods. Here's how the creation methods operate. The client calls the EJBHome client method, which causes the EJB container to call the ejbCreate() method. The default values are then set by the CMP entity bean, and the EJB container extracts field values from the CMP entity bean. Next, the EJB container creates both a representation in the database and a new Remote object, commonly referred to as a *wrapper*. Once this is completed, the EJB container calls the ejbPostCreate() method and the primary key becomes valid. The CMP entity bean performs any additional initialization that is required and the EJB container creates a new EJB instance before the remote reference is returned to the client.

Business methods are methods that are unique to the J2EE application. There are two general types of business methods in an entity bean. These are getXXX and setXXX, where XXX is the type of object that is either being retrieved from a database or written to a database. For example, the getProduct() method retrieves product information from a database. Likewise, the setProduct(Product prod) method writes a Product object to the database.

Callback methods are invoked in response to events that occur. There are seven commonly used callback methods. These are setEntityContext(EntityContext), unsetEntityContext(), ejbLoad(), ejbStore(), ejbActivate(), ejbPassivate(), and ejbRemote().

The setEntityContext(EntityContext ctx) method is called immediately following the creation of the instance and sets the context that is associated with the entity. The unsetEntityContext() method is called when the instance terminates.

A container invokes the ejbLoad() method to instruct the instance to synchronize its state by loading its state from the underlying database. The ejbStore() method is invoked by a container to instruct the instance to synchronize its state by storing it to the underlying database.

The ejbActivate() method is called whenever the instance of the entity bean is activated from its "passive" state. You should place routines in the ejbActivate() method that reacquire resources that were released when the entity bean entered the passive state. A passive state occurs between calls to the entity bean.

The ejbPassivate() method is called before the instance enters the "passive" state and should contain routines that release resources. Those resources can be required when the ejbActivate() method is called.

The ejbRemove() method is called immediately before the entity terminates by either the client or by the EJB container. You should place routines in this method that release resources that were encumbered during the life of the entity bean.

Listing 12-26 illustrates the skeleton of an entity bean. In this example, there are only two business methods, which are getProduct() and setProduct(Product prod). These retrieve product information and write product information to the underlying database. A real-world entity bean will have more than two business methods, based on the requirements of the J2EE application.

isting 12-26 A skeleton of an entity bean.

```
import javax.ejb.EntityBean;
import javax.ejb.EntityContext;
public class myEJB implements EntityBean {
  int ID;
  private Product myProduct;
  public void ejbPostCreate()
  {
  }
public Product getProduct()
{
  return myProduct;
}
public void setProduct(Product prod)
{
  myProduct = prod;
}
public void setEntityContext(EntityContext cntx)
{
}
public void unsetEntityContext()
{
}
public void ejbLoad()
{
}
public void ejbStore()
{
}
public void ejbActivate()
{
```

```
    }
    public void ejbPassivate()
    {
    }
    public void ejbRemove()
    {
    }
}
```

The CMP entity bean does not provide database access. Database access is made available through an EJB container vendor tool. The tool is used to map instance fields in the CMP entity bean to a database. There are two instance fields in this example. These are int and the dependent object Product. Listing 12-27 defines the Product object. These fields are referred to as *container-managed fields* since the EJB container synchronizes the database with the state of these fields.

A CMP entity bean can contain both instance fields and container-managed fields. However, both fields appear as instance fields in the CMP entity bean because nothing distinguishes a container-managed field from an instance field in the CMP entity bean. Container-managed fields are identified as such in the deployment descriptor.

A container-managed field must have a column in a table of a database that is of a like type. That is, an int container-managed field in a CMP entity bean must have a corresponding int or equivalent column in the database that is associated with the CMP entity bean.

One or more container-managed fields must be identified in the deployment descriptor as the primary key that is used to locate records in the database. Typically, the primary key for the database is also defined as the primary key for the CMP entity bean.

Listing 12-27 The Product object definition.

```
import java.io.Serializable;
public class Product implements Serializable
  {
   public String prodName, prodDescription;
   public Product(String prodName, String prodDescription)
   {
     this.prodName = prodName;
     this.prodDescription = prodDescription;
   }
}
```

CMP Entity Bean Home Interface and Remote Interface

The CMP entity bean requires both home and remote interfaces. As described previously in this chapter, a client located on the same Java Virtual Machine (JVM)

as the EJB uses a Local interface. A client on a different JVM interacts with the EJB by using the remote interface.

Listing 12-28 contains the home interface for the EJB defined in Listing 12-27, and Listing 12-29 contains the remote interface for this EJB. In this example, the home interface has one create() method that accepts the prodID as an argument. You can overload the create() method to have no arguments or more and/or different arguments than that shown in this example, depending on the functionality that you give to the EJB.

The create() method is used to initialize instance variables and objects prior to inserting a record into a database. The create() method is called by the EJB container and returns a null to the EJB container. It is the container's responsibility to insert a record into the database using the value of container-managed fields in the CMP bean. The insertion occurs while the create() method is invoked; afterwards, the CMP bean is said to exist. Other methods can be accessed once the CMP bean exists.

The home interface should contain one or more find methods that are used to retrieve information from a database. The way in which find methods are implemented is unique to each EJB container vendor. Vendors of EJB containers provide a tool that enables the developer to define the behavior of find methods. One of the two more common definitions is to require the developer to enter SQL commands that define the logic of a find method. The other is to require the developer to use an object-relational mapping tool to define the find methods.

The remote interface provides a definition for business methods available in the CMP bean. Each business method defined in the remote interface must have a matching business method defined in the CMP bean.

Once the CMP bean is defined, you must reference the CMP bean in the deployment descriptor as discussed previously in this chapter.

Listing 12-28 Home interface.

```
public interface myEJBHome extends javax.ejb.EJBHome {
public myEJB create (Integer prodID) throws RemoteException, CreateException;
public myEJB findByPrimaryKey(Integer prodID) throws RemoteException,
FinderException;
}
```

Listing 12-29 Remote interface.

```
import javax.ejb.EJBObject;
import java.rmi.RemoteException;

public interface myEJBRemote extends EJBObject {
public getProduct() throws RemoteException;
public setProduct(Product prod()) throws Remote Exception;
}
```

Accessing the CMP Bean

You can use the home interface or the remote interface to access the CMP bean. In both cases, the initial step is to get a reference to the CMP bean. In the home interface, an instance of the CMP bean is created by invoking the create() method, as illustrated in Listing 12-30. You can call methods of the CMP bean once the create() method returns an instance of the CMP Remote interface.

The remote interface directly calls methods of the CMP bean, as is illustrated in Listing 12-31. In this example, the getProduct() method of the CMP bean is called. This method returns an instance of the Product object.

Listing 12-30 Calling the CMP using the home interface.

```
javax.naming.Context jndiContext = new InitialContext();
Object obj = jndiContext.lookup("java:comp/env/ejb/myEJBHome");
myEJB mybean = obj.create(4321);
```

Listing 12-31 Calling the CMP using the remote interface.

```
javax.naming.Context jndiContext = new InitialContext();
Object obj = jndiContext.lookup("java:comp/env/ejb/myEJBHome");
Product prod = obj.getProduct();
```

Bean-Managed Persistence

A bean-managed persistence (BMP) bean uses the JDBC API or another appropriate database API to interface with the database. However, these interactions take place under the direction of the EJB container. That is, the EJB container tells the BMP bean when to insert a new record, retrieve data, modify data, or delete data. The developer of the BMP bean must define methods within the BMP bean that respond to requests received from the EJB container. You should use a BMP bean when you need complex relationship fields instead of simple persistence fields.

There are typically a minimum of five methods defined in a BMP bean, as illustrated in Listing 12-32. These are ejbLoad(), ejbStore(), ejbCreate(), ejbRemote(), and findXXX()

The ejbLoad() method must contain code that reads data from a database. Similarly, the ejbStore() method writes data to a database. The ejbCreate() method must have code that inserts a new record in a database and the ejbRemote() removes a record from the database. The findXXX() method contains code that locates one or more records in the database. The XXX identifies the kind of information the method locates. In this example there is one find method, which is findProduct(), and it requires the product identification number as an argument. You can have any number of findXXX() methods in a BMP bean, as required by the nature of the project.

The BMP bean uses the same home interface and remote interface as is used for the CMP bean. Once the BMP bean is defined, you must reference the BMP bean in the deployment descriptor, as discussed previously in this chapter.

Listing 12-32 A skeleton of a bean-managed persistence bean.

```
public class myBMPBean implements EntityBean
{
  public void ejbLoad()
  {
    //Read data from database
  }
  public void ejbStore()
  {
    //Save data to a database
  }
  public void ejbCreate()
  {
   //Insert a record into the database
  }
  public void ejbRemote()
  {
   //Remove a record from a database
  }
  public Product findProduct(Integer prod)
  {
   //Find the specified product
  }
}
```

Message-Driven Bean

A message-driven bean MDB is designed for clients to invoke server-side business logic using asynchronous communication, which is a special case of a stateless session bean, described previously in this chapter.

There isn't any home interface or remote interface for a MDB. Instead, a MDB monitors Java Message Service communications and reacts to messages sent by clients. Clients don't directly access a MDB. Instead, the MDB intercedes and processes requests anonymously. This is possible because a MDB is stateless. Therefore, the EJB container can begin a new instance of the MDB when the message is received or use an existing instance of the MDB from the instance pool. And unlike a session bean or entity bean, a client does not control the life of an MDB. The EJB container handles the responsibility for creating and removing an MDB.

Requests from clients are sent via JMS. The EJB container listens for messages in the JMS service that MDBs are registered to receive. The developer must provide all the logic to process a message in the onMessage() method. The JMS service enables the client and MDB to work independently and without having to wait until the other is finished processing. The MDB doesn't know anything about the client. You'll learn the details on how JMS works in Chapter 17.

An MDB processes a client request by using business methods defined in the MDB and by invoking business methods defined in other EJBs. This means an MDB is scalable, because an MDB is stateless and efficient by using asynchronous messaging while still maintaining functionality.

Before the introduction of MDBs, developers wrote their own queue processing code, which was not standardized. MDB introduced a standard way to process messages handled by the EJB container and JMS. Developers can focus on coding business logic rather than on queue processing.

Behind the Scenes

An advantage of using an MDB is that the EJB container does most of the work for you. As previously mentioned, clients send requests in the form of a JMS message to the JMS service. The EJB container acts as the intermediary between JMS service and the MDB.

The EJB container makes a connection to the JMS service and registers to receive JMS messages that have a particular JMS topic or that are on a particular JMS message queue (see Chapter 18), depending on if a client uses JMS topics or a JMS message queue to send a request for service.

The EJB container then creates a TopicSubscriber object or QueueReceiver object, depending on how JMS messages are handled by the JMS server. The TopicSubscriber object or QueueReceiver object is then used to receive JMS messages from clients. The EJB container knows which receiver object to create because the receiver object is specified in the MDB deployment descriptor.

The EJB container also registers the message listener and sets the message acknowledge mode to respond to the JMS server when a JMS message is received by the EJB container. All this occurs behind the scenes without the MDB developer needing to write additional code in the MDB.

Creating an MDB

An MDB must define four methods. These are ejbCreate(), ejbRemove(), setMessageDrivenContext(), and onMessage(), as illustrated in Listing 12-33. The ejbCreate() method is called when the MDB is created by the container first invoked by the EJB container, but not when a JMS message is received from a client. The ejbRemove() is called by the EJB container when the container terminates the instance of the MDB.

The setMessageDrivenContext() method creates the context for the MDB, which is similar to the session and entity context classes. And the onMessage() method is called

each time the EJB container receives a JMS message from a client, as discussed previously in this chapter.

Listing 12-33 A skeleton of an MDB.

```
import javax.ejb.MessageDrivenBean;
import javax.ejb.MessageDrivenContext;
import javax.jms.MessageListener;
import javax.jms.Message;
public class myMDB implements MessageListener, MessageDrivenBean {
  public void setMessageDrivenContext(MessageDrivenContext mdc)
  {
  }
  public void ejbCreate()
  {
  }
  public void ejbRemove()
  {
  }
  public void onMessage(Message clientMessage)
  {
  }
}
```

The onMessage() method is where the MDB processes messages received indirectly from a client. Any type of stateless process can be included in this method. The simplest form of processing is to display incoming messages on the screen. Listing 12-34 shows how this is done.

In this example, the message from the client is passed to the onMessage() method in the form of a Message object, which is called clientMessage in this sample code. The onMessage() method determines if the clientMessage is a TextMessage object. If so, the message is assigned to an instance TextMessage object called tmpMesg. The body of the message, if the type is a TextMessage object, is then retrieved using the getText() method and is displayed on the screen. If the clientMessage isn't a TextMessage object, the name of the class is retrieved and displayed on the screen. Any JMSException errors that are thrown are trapped by the catch { } block. Once the MDB bean is defined, you must reference the MDB bean in the deployment descriptor as discussed previously in this chapter.

Listing 12-34 An onMessage() method displays the message received from a client.

```
public void onMessage(Message clientMessage) {
  TextMessage tmpMesg = null;
  try {
    if (clientMessage instanceof TextMessage) {
      tmpMesg = (TextMessage) clientMessage;
```

```
    System.out.println("Incoming Message: " + tmpMesg.getText());
  }
  else
  {
   System.out.println("Incorrect Message: "
     + clientMessage.getClass().getName());
  }
    }
  }
  catch (JMSException error)
  {
   System.out.println("Error: " + error.getMessage());
  }
}
```

The JAR File

EJB classes and related files are packaged together into a Java Archive (JAR) file for deployment. The JAR file is a compressed file format that was originally designed to reduce the size of software so it could be easily be transported.

The JAR file used to package an EJB must contain the following; however, it is customary to keep dependent classes and dependent interfaces in different JAR files. In either case, the dependent classes and dependent interfaces must be in the runtime CLASSPATH.

- EJB classes
- Dependent classes
- Remote interface
- Home interface
- Dependent interfaces
- Primary key class
- Deployment descriptor

In addition, the deployment descriptor must be located in the META-INF/ejb-jar.xml path. This means that you place the deployment descriptor in the META-INF/ejb-jar.xml directory before archiving the EJB package. Classes must retain the directory structure associated with the package names.

You can use the JAR utility by entering **jar cf** followed by name of the JAR file, and then followed by the path and names of files that will be archived.

Quick Reference Guide

This Quick Reference Guide provides a brief overview of Java classes used with EJB. Full details of these classes and all Java classes and interfaces are available at java.sun.com.

Syntax	Descriptions
void ejbActivate()	Called when the instance is activated from its "passive" state.
void ejbPassivate()	Called before the instance enters the "passive" state.
void ejbRemove()	Called before the EJB container ends the session object.
void setSessionContext (SessionContext con)	Sets the associated session context.

Table 12-5. *Public Interface Session Bean Extends Enterprise Bean*

Syntax	Descriptions
void ejbActivate()	Called when an instance is removed from the pool of instances to become associated with a specific EJB object.
void ejbLoad()	Instructs the instance to synchronize its state by loading its state from the underlying database.
void ejbPassivate()	Called by the EJB container before an instance becomes disassociated with an EJB object.
void ejbRemove()	Called before the EJB container removes the EJB object associated with the instance.
void ejbStore()	Instructs the instance to synchronize its state by storing it to the underlying database.
void setEntityContext (EntityContext ctx)	Sets the associated entity context.
void unsetEntityContext()	Unsets the associated entity context.

Table 12-6. *Public Interface Entity Bean Extends Enterprise Bean*

Syntax	Descriptions
void ejbRemove()	This method is called before the EJB container ends the life of the message-driven object.
void setMessageDrivenContext (MessageDrivenContext con)	Sets the associated message-driven context.

Table 12-7. *Public Interface Message-Driven Bean Extends Enterprise Bean*

Syntax	Descriptions
EJBMetaData getEJBMetaData()	Obtains the EJB's EJBMetaData interface.
HomeHandle getHomeHandle()	Obtains a handle for the remote home object.
void remove(Handle handle)	Removes an EJB object identified by its handle.
void remove(Object primaryKey)	Removes an EJB object identified by its primary key.

Table 12-8. *Public Interface EJB Home Extends java.rmi.Remote*

Syntax	Descriptions
EJBHome getEJBHome()	Obtains the EJB remote home interface.
Class getHomeInterfaceClass()	Obtains the Class object for the EJB remote home interface.
Class getPrimaryKeyClass()	Obtains the Class object for the EJB primary key class.
Class getRemoteInterfaceClass()	Obtains the Class object for the EJB remote interface.
boolean isSession()	Tests if the EJB type is "session."
boolean isStatelessSession()	Tests if the EJB type is "stateless session."

Table 12-9. *Public Interface EJB Metadata*