CHAPTER 24

RELATIONAL DATABASES

© Jason Allen/iStockphoto.

CHAPTER GOALS

To understand how relational databases store information

To learn to query a database with the Structured Query Language (SQL)

To connect to a database with Java Database Connectivity (JDBC)

To write database programs that insert, update, and query data in a relational database

CHAPTER CONTENTS

© Jason Allen/iStockphoto.

When you store data in a file, you want to be able to add and remove data, change data items, and find items that match certain criteria. However, if you have a lot of data, it can be difficult to carry out these operations quickly and efficiently. Because data storage is such a common task, special *database management systems* have been invented that let the programmer think in terms of the data rather than how it is stored. In this chapter, you will learn how to use SQL, the Structured Query Language, to query and update information in a relational database, and how to access database information from Java programs.

24.1 Organizing Database Information

24.1.1 Database Tables

A relational database stores information in tables. Each table column has a name and a data type.

A relational database stores information in *tables*. Figure 1 shows a typical table. As you can see, each *row* in this table corresponds to a product. The *column headers* correspond to attributes of the product: the product code, description, and unit price. Note that all items in a particular column have the same type: product codes and descriptions are strings, unit prices are floating-point numbers. The allowable column types differ somewhat from one database to another. Table 1 shows types that are commonly available in relational databases that follow the SQL (for Structured Query Language; often pronounced "sequel") standard.

SQL (Structured Query Language) is a command language for interacting with a database.

Most relational databases follow the SQL standard. There is no relationship between SQL and Java—they are different languages. However, as you will see later in this chapter, you can use Java to send SQL commands to a database. You will see in the next section how to use SQL commands to carry out queries, but there are other SQL commands.

For example, here is the SQL command to create a table named Product:

```
CREATE TABLE Product
(
   Product_Code CHAR(7),
   Description VARCHAR(40),
   Price DECIMAL(10, 2)
)
```

Product

Product_Code	Description	Price
116-064	Toaster	24.95
257-535	Hair dryer	29.95
643-119	Car vacuum	19.99

Figure 1 A Product Table in a Relational Database

Table 1 Some Standard SQL Types and Their Corresponding Java Types

SQL Data Type	Java Data Type
`INTEGER or INT`	`int`
`REAL`	`float`
`DOUBLE`	`double`
`DECIMAL(m, n)`	Fixed-point decimal numbers with *m* total digits and *n* digits after the decimal point; similar to `BigDecimal`
`BOOLEAN`	`boolean`
`VARCHAR(n)`	Variable-length `String` of length up to *n*
`CHARACTER(n) or CHAR(n)`	Fixed-length `String` of length *n*

Use the SQL commands `CREATE TABLE` and `INSERT INTO` to add data to a database.

Unlike Java, SQL is not case sensitive. For example, you could spell the command `create table` instead of `CREATE TABLE`. However, as a matter of convention, we will use uppercase letters for SQL keywords and mixed case for table and column names.

To insert rows into the table, use the `INSERT INTO` command. Issue one command for each row, such as

```
INSERT INTO Product
   VALUES ('257-535', 'Hair dryer', 29.95)
```

SQL uses single quotes (`'`), not double quotes, to delimit strings. What if you have a string that contains a single quote? Rather than using an escape sequence (such as `\'`) as in Java, you just write the single quote twice, such as

```
'Sam''s Small Appliances'
```

If you create a table and subsequently want to remove it, use the `DROP TABLE` command with the name of the table. For example,

```
DROP TABLE Test
```

24.1.2 Linking Tables

If you have objects whose instance variables are strings, numbers, dates, or other types that are permissible as table column types, then you can easily store them as rows in a database table. For example, consider a Java class `Customer`:

```
public class Customer
{
   private String name;
   private String address;
   private String city;
   private String state;
   private String zip;
   . . .
}
```

Customer

Name	Address	City	State	Zip
VARCHAR(40)	VARCHAR(40)	VARCHAR(30)	CHAR(2)	CHAR(5)
Sam's Small Appliances	100 Main Street	Anytown	CA	98765

Figure 2 A Customer Table

It is simple to come up with a database table structure that allows you to store customers—see Figure 2.

For other objects, it is not so easy. Consider an invoice. Each invoice object contains a reference to a customer object:

```
public class Invoice
{
   private int invoiceNumber;
   private Customer theCustomer;
   . . .
}
```

Because `Customer` isn't a standard SQL type, you might consider simply entering all the customer data into the invoice table—see Figure 3. However, this is not a good idea. If you look at the sample data in Figure 3, you will notice that Sam's Small Appliances had two invoices, numbers 11731 and 11733. Yet all information for the customer was *replicated* in two rows.

This replication has two problems. First, it is wasteful to store the same information multiple times. If the same customer places many orders, then the replicated information can take up a lot of space. More importantly, the replication is *dangerous*. Suppose the customer moves to a new address. Then it would be an easy mistake to update the customer information in some of the invoice records and leave the old address in place in others.

In a Java program, neither of these problems occurs. Multiple `Invoice` objects can contain references to a single shared `Customer` object.

Invoice

Invoice_ Number	Customer_ Name	Customer_ Address	Customer_ City	Customer_ State	Customer_ Zip	. . .
INTEGER	VARCHAR(40)	VARCHAR(40)	VARCHAR(30)	CHAR(2)	CHAR(5)	. . .
11731	Sam's Small Appliances	100 Main Street	Anytown	CA	98765	. . .
11732	Electronics Unlimited	1175 Liberty Ave	Pleasantville	MI	45066	. . .
11733	Sam's Small Appliances	100 Main Street	Anytown	CA	98765	. . .

Figure 3 A Poor Design for an Invoice Table with Replicated Customer Data

Invoice

Invoice_ Number	Customer_ Number	Payment
INTEGER	INTEGER	DECIMAL(10, 2)
11731	3175	0
11732	3176	249.95
11733	3175	0

Customer

Customer_ Number	Name	Address	City	State	Zip
INTEGER	VARCHAR(40)	VARCHAR(40)	VARCHAR(30)	CHAR(2)	CHAR(5)
3175	Sam's Small Appliances	100 Main Street	Anytown	CA	98765
3176	Electronics Unlimited	1175 Liberty Ave	Pleasantville	MI	45066

Figure 4 Two Tables for Invoice and Customer Data

You should avoid rows with replicated data. Instead, distribute the data over multiple tables.

The first step in achieving the same effect in a database is to organize your data into multiple tables as in Figure 4. Dividing the columns into two tables solves the replication problem. The customer data are no longer replicated—the Invoice table contains no customer information, and the Customer table contains a single record for each customer. But how can we refer to the customer to which an invoice is issued? Notice in Figure 4 that there is now a Customer_Number column in *both* the Customer table and the Invoice table. Now all invoices for Sam's Small Appliances share only the customer number. The two tables are *linked* by the Customer_Number field. To find out more details about this customer, you need to use the customer number to look up the customer in the Customer table.

Note that the customer number is a *unique identifier*. We introduced the customer number because the customer name by itself may not be unique. For example, there may well be multiple Electronics Unlimited stores in various locations. Thus, the customer name alone does not uniquely identify a record (a row of data), so we cannot use the name as a link between the two tables.

A primary key is a column (or set of columns) whose value uniquely specifies a table record.

In database terminology, a column (or combination of columns) that uniquely identifies a row in a table is called a **primary key**. In our Customer table, the Customer_Number column is a primary key. You need a primary key if you want to establish a link from another table. For example, the Customer table needs a primary key so that you can link customers to invoices.

A foreign key is a reference to a primary key in a linked table.

When a primary key is linked to another table, the matching column (or combination of columns) in that table is called a **foreign key**. For example, the Customer_Number in the Invoice table is a foreign key, linked to the primary key in the Customer table. Unlike primary keys, foreign keys need not be unique. For example, in our Invoice table we have several records that have the same value for the Customer_Number foreign key.

24.1.3 Implementing Multi-Valued Relationships

Each invoice is linked to exactly one customer. That is called a *single-valued* relationship. On the other hand, each invoice has many line items. (As in Chapter 12, a *line item* identifies the product, quantity, and unit price.) Thus, there is a *multi-valued* relationship between invoices and line items. In the Java class, the `LineItem` objects are stored in an array list:

```
public class Invoice
{
   private int invoiceNumber;
   private Customer theCustomer;
   private ArrayList<LineItem> items;
   private double payment;
   . . .
}
```

However, in a relational database, you need to store the information in tables. Surprisingly many programmers, when faced with this situation, commit a major faux pas and replicate columns, one for each line item, as in Figure 5 below.

Clearly, this design is not satisfactory. What should we do if there are more than three line items on an invoice? Perhaps we should have 10 line items instead? But that is wasteful if the majority of invoices have only a couple of line items, and it still does not solve our problem for the occasional invoice with lots of line items.

Instead, distribute the information into two tables: one for invoices and another for line items. Link each line item back to its invoice with an Invoice_Number foreign key in the LineItem table—see Figure 6.

Invoice

Invoice_ Number	Customer_ Number	Product_ Code1	Quantity1	Product_ Code2	Quantity2	Product_ Code3	Quantity3	Payment
INTEGER	INTEGER	CHAR(7)	INTEGER	CHAR(7)	INTEGER	CHAR(7)	INTEGER	DECIMAL(10, 2)
11731	3175	116-064	3	257-535	1	643-119	2	0

Figure 5 A Poor Design for an Invoice Table with Replicated Columns

Invoice

Invoice_Number	Customer_Number	Payment
INTEGER	INTEGER	DECIMAL(10, 2)
11731	3175	0
11732	3176	249.50
11733	3175	0

LineItem

Invoice_Number	Product_Code	Quantity
INTEGER	CHAR(7)	INTEGER
11731	116-064	3
11731	257-535	1
11731	643-119	2
11732	116-064	10
11733	116-064	2
11733	643-119	1

Figure 6 Linked Invoice and LineItem Tables Implement a Multi-Valued Relationship

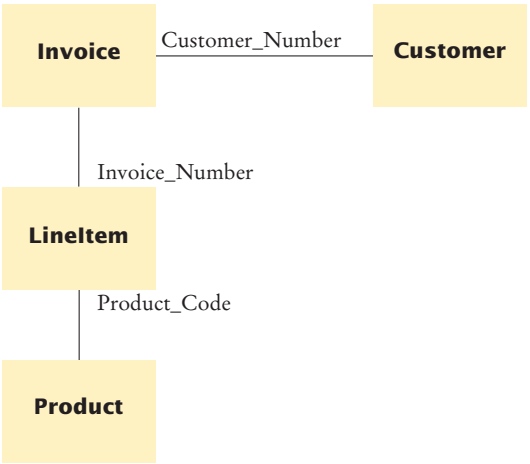

Figure 7 The Links Between the Tables in the Sample Database

Implement one-to-many relationships with linked tables, not replicated columns.

In a similar fashion, the LineItem table links to the Product table via the Product table's Product_Code primary key. Our database now consists of four tables:

- Invoice
- Customer
- LineItem
- Product

Figure 7 shows the links between these tables. In the next section you will see how to query this database for information about invoices, customers, and products. The queries will take advantage of the links between the tables.

1. Would a telephone number be a good primary key for a customer table?
2. In the database of Section 24.1.3, what are all the products that customer 3176 ordered?

Practice It Now you can try these exercises at the end of the chapter: R24.2, R24.4, R24.5.

Programming Tip 24.1

Stick with the Standard

The Java language is highly standardized. You will rarely find compilers that allow you to specify Java code that differs from the standard, and if they do, it is always a compiler bug. However, SQL implementations are often much more forgiving. For example, many SQL vendors allow you to use a Java-style escape sequence such as

```
'Sam\'s Small Appliances'
```

in a SQL string. The vendor probably thought that this would be "helpful" to programmers who are familiar with Java or C. (The C language uses the same escape mechanism for denoting special characters.)

However, this is an illusion. Deviating from the standard limits portability. Suppose you later want to move your database code to another vendor, perhaps to improve performance or to lower the cost of the database software. If the other vendor hasn't implemented a particular deviation, then your code will no longer work and you need to spend time fixing it.

To avoid these problems, you should stick with the standard. With SQL, you cannot rely on your database to flag all errors—some of them may be considered "helpful" extensions. That means that you need to *know* the standard and have the discipline to follow it. (See *A Guide to the SQL Standard: A User's Guide to the Standard Database Language*, by Chris J. Date and Hugh Darwen (Addison-Wesley, 1996), for more information.)

Programming Tip 24.2

Avoid Unnecessary Data Replication

It is very common for beginning database designers to replicate data. When replicating data in a table, ask yourself if you can move the replicated data into a separate table and use a key, such as a code or ID number, to link the tables.

Consider this example in an Invoice table:

Invoice

. . .	Product_Code	Description	Price	. . .
. . .	CHAR(7)	VARCHAR(40)	DECIMAL(10, 2)	. . .
. . .	116-064	Toaster	24.95	. . .
. . .	116-064	Toaster	24.95	. . .
. . .	. . .	. . .	. . .	. . .

As you can see, some product information is replicated. Is this replication an error? It depends. The product description for the product with code 116-064 is always going to be "Toaster". Therefore, that correspondence should be stored in an external Product table.

The product price, however, can change over time. When it does, the old invoices don't automatically use the new price. Thus, it makes sense to store the price that the customer was actually charged in an Invoice table. The current list price, however, is best stored in an external Product table.

Programming Tip 24.3

Don't Replicate Columns in a Table

If you find yourself numbering columns in a table with suffixes 1, 2, and so forth (such as Quantity1, Quantity2, Quantity3), then you are probably on the wrong track. How do you know there are exactly three quantities? In that case, it's time for another table.

Add a table to hold the information for which you replicated the columns. In that table, add a column that links back to a key in the first table, such as the invoice number in our example. By using an additional table, you can implement a multi-valued relationship.

Special Topic 24.1

Primary Keys and Indexes

Recall that a **primary key** is a column (or combination of columns) that uniquely identifies a row in a table. When a table has a primary key, then the database can build an *index file:* a file that stores information on how to access a row quickly when the primary key is known. Indexing can greatly increase the speed of database queries.

If the primary key is contained in a single column, then you can tag the column with the `PRIMARY KEY` attribute, like this:

```
CREATE TABLE Product
(
   Product_Code CHAR(7) PRIMARY KEY,
   Description VARCHAR(40),
   Price DECIMAL(10, 2)
)
```

If the primary key is contained in multiple columns, then add a `PRIMARY KEY` clause to the end of the `CREATE TABLE` command, like this:

```
CREATE TABLE LineItem
(
   Invoice_Number INTEGER,
   Product_Code CHAR(7),
   Quantity INTEGER,
   PRIMARY KEY (Invoice_Number, Product_Code)
)
```

Occasionally, one can speed queries up by building *secondary indexes:* index files that index other column sets, which are not necessarily unique. That is an advanced technique that we will not discuss here.

24.2 Queries

Let's assume that the tables in our database have been created and that records have been inserted. Once a database is filled with data, you will want to *query* the database for information, such as

- What are the names and addresses of all customers?
- What are the names and addresses of all customers in California?
- What are the names and addresses of all customers who bought toasters?
- What are the names and addresses of all customers with unpaid invoices?

In this section you will learn how to formulate simple and complex queries in SQL. We will use the data shown in Figure 8 for our examples.

Invoice

Invoice_Number	Customer_Number	Payment
INTEGER	INTEGER	DECIMAL(10, 2)
11731	3175	0
11732	3176	249.50
11733	3175	0

LineItem

Invoice_Number	Product_Code	Quantity
INTEGER	CHAR(7)	INTEGER
11731	116-064	3
11731	257-535	1
11731	643-119	2
11732	116-064	10
11733	116-064	2
11733	643-119	1

Product

Product_Code	Description	Price
CHAR(7)	VARCHAR(40)	DECIMAL(10, 2)
116-064	Toaster	24.95
257-535	Hair dryer	29.95
643-119	Car vacuum	19.99

Customer

Customer_Number	Name	Address	City	State	Zip
INTEGER	VARCHAR(40)	VARCHAR(40)	VARCHAR(30)	CHAR(2)	CHAR(5)
3175	Sam's Small Appliances	100 Main Street	Anytown	CA	98765
3176	Electronics Unlimited	1175 Liberty Ave	Pleasantville	MI	45066

Figure 8 A Sample Database

24.2.1 Simple Queries

Use the SQL SELECT command to query a database.

In SQL, you use the SELECT command to issue queries. For example, the command to select all data from the Customer table is

```
SELECT * FROM Customer
```

The result is

Customer_Number	Name	Address	City	State	Zip
3175	Sam's Small Appliances	100 Main Street	Anytown	CA	98765
3176	Electronics Unlimited	1175 Liberty Ave	Pleasantville	MI	45066

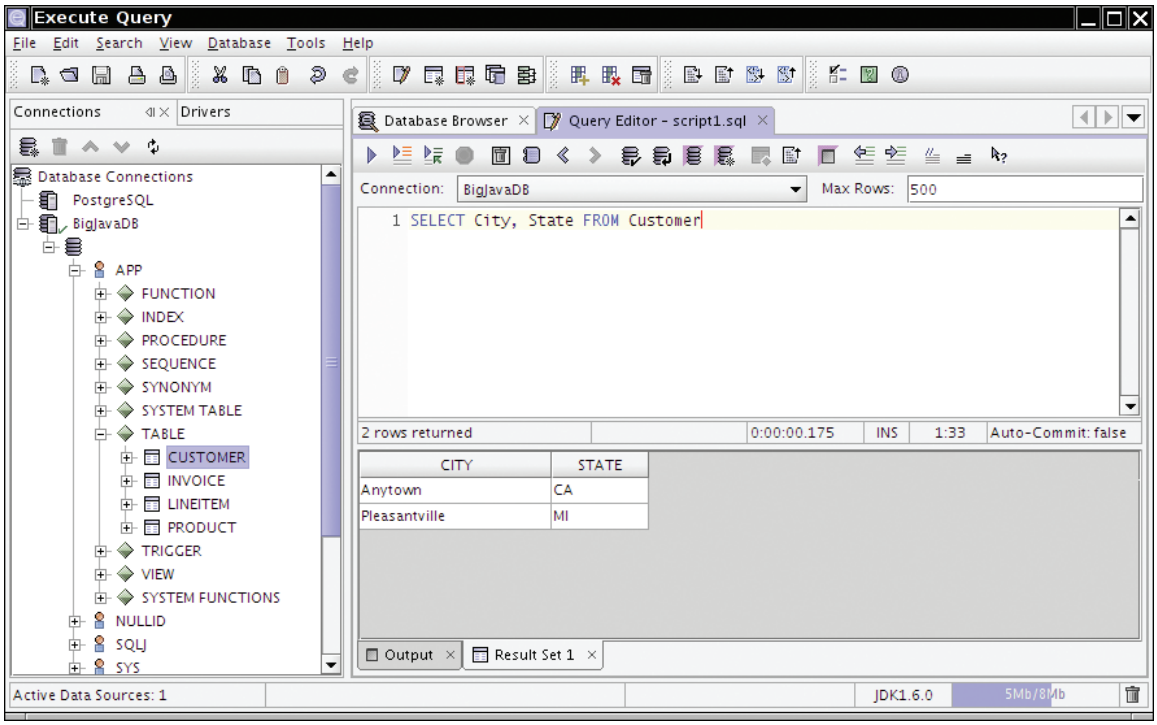

Figure 9 An Interactive SQL Tool

The outcome of the query is a *view*—a set of rows and columns that provides a "window" through which you can see some of the database data. If you select all rows and columns from a single table, of course you get a view into just that table.

Many database systems have tools that let you issue interactive SQL commands—Figure 9 shows a typical example. When you issue a `SELECT` command, the tool displays the resulting view. You may want to skip ahead to Section 24.3 and install a database. Or perhaps your computer lab has a database installed already. Then you can run the interactive SQL tool of your database and try out some queries.

24.2.2 Selecting Columns

Often, you don't care about all columns in a table. Suppose your traveling salesperson is planning a trip to all customers. To plan the route, the salesperson wants to know the cities and states of all customers. Here is the query:

```
SELECT City, State FROM Customer
```

The result is

City	State
Anytown	CA
Pleasantville	MI

As you can see, the syntax for selecting columns is straightforward. Simply specify the names of the columns you want, separated by commas.

24.2.3 Selecting Subsets

The WHERE clause selects data that fulfill a condition.

You just saw how you can restrict a view to show selected columns. Sometimes you want to select certain rows that fit a particular criterion. For example, you may want to find all customers in California. Whenever you want to select a subset, you use the WHERE clause, followed by the condition that describes the subset. Here is an example:

```
SELECT * FROM Customer WHERE State = 'CA'
```

The result is

Customer_ Number	Name	Address	City	State	Zip
3175	Sam's Small Appliances	100 Main Street	Anytown	CA	98765

You have to be a bit careful with expressing the condition in the WHERE clause, because SQL syntax differs from the Java syntax. As you already know, in SQL you use single quotes to delimit strings, such as 'CA'. You also use a single =, not a double ==, to test for equality. To test for inequality, you use the <> operator. For example

```
SELECT * FROM Customer WHERE State <> 'CA'
```

selects all customers that are *not* in California.

You can match patterns with the LIKE operator. The right-hand side must be a string that can contain the special symbols _ (match exactly one character) and % (match any character sequence). For example, the expression

```
Name LIKE '_o%'
```

matches all strings whose second character is an "o". Thus, "Toaster" is a match but "Crowbar" is not.

You can combine expressions with the logical connectives AND, OR, and NOT. (Do not use the Java &&, ||, and ! operators.) For example

```
SELECT *
   FROM Product
   WHERE Price < 100
      AND Description <> 'Toaster'
```

selects all products with a price less than 100 that are not toasters.

Of course, you can select both row and column subsets, such as

```
SELECT Name, City FROM Customer WHERE State = 'CA'
```

24.2.4 Calculations

Suppose you want to find out *how many* customers there are in California. Use the COUNT function:

```
SELECT COUNT(*) FROM Customer WHERE State = 'CA'
```

In addition to the COUNT function, there are four other functions: SUM, AVG (average), MAX, and MIN.

The * means that you want to calculate entire records. That is appropriate only for the COUNT function. For other functions, you have to access a specific column. Put the column name inside the parentheses:

```
SELECT AVG(Price) FROM Product
```

24.2.5 Joins

The queries that you have seen so far all involve a single table. However, the information that you want is usually distributed over multiple tables. For example, suppose you are asked to find all invoices that include a line item for a car vacuum. From the Product table, you can issue a query to find the product code:

```
SELECT Product_Code
   FROM Product
   WHERE Description = 'Car vacuum'
```

You will find out that the car vacuum has product code 643-119. Then you can issue a second query:

```
SELECT Invoice_Number
   FROM LineItem
   WHERE Product_Code = '643-119'
```

But it makes sense to combine these two queries so that you don't have to keep track of the intermediate result. When combining queries, note that the two tables are linked by the Product_Code field. We want to look at matching rows in both tables. In other words, we want to restrict the search to rows where

```
Product.Product_Code = LineItem.Product_Code
```

Here, the syntax

TableName.*ColumnName*

denotes the column in a particular table. Whenever a query involves multiple tables, you should specify both the table name and the column name. Thus, the combined query is

```
SELECT LineItem.Invoice_Number
   FROM Product, LineItem
   WHERE Product.Description = 'Car vacuum'
      AND Product.Product_Code = LineItem.Product_Code
```

The result is

Invoice_Number
11731
11733

A join is a query that involves multiple tables.

In this query, the FROM clause contains the names of multiple tables, separated by commas. (It doesn't matter in which order you list the tables.) Such a query is often called a **join** because it involves joining multiple tables.

You may want to know in what cities hair dryers are popular. Now you need to add the Customer table to the query—it contains the customer addresses. The customers are referenced by invoices, so you need that table as well. Here is the complete query:

```
SELECT Customer.City, Customer.State, Customer.Zip
   FROM Product, LineItem, Invoice, Customer
   WHERE Product.Description = 'Hair dryer'
      AND Product.Product_Code = LineItem.Product_Code
      AND LineItem.Invoice_Number = Invoice.Invoice_Number
      AND Invoice.Customer_Number = Customer.Customer_Number
```

The result is

City	State	Zip
Anytown	CA	98765

Whenever you formulate a query that involves multiple tables, remember to:

- List all tables that are involved in the query in the `FROM` clause.
- Use the *TableName*.*ColumnName* syntax to refer to column names.
- List all join conditions (*TableName1*.*ColumnName1* = *TableName2*.*ColumnName2*) in the `WHERE` clause.

As you can see, these queries can get a bit complex. However, database management systems are very good at answering these queries (see Programming Tip 24.5 on page W1039). One remarkable aspect of SQL is that you describe *what* you want, not *how* to find the answer. It is entirely up to the database management system to come up with a plan for how to find the answer to your query in the shortest number of steps.

Commercial database manufacturers take great pride in coming up with clever ways to speed up queries: query optimization strategies, caching of prior results, and so on. In this regard, SQL is a very different language from Java. SQL statements are descriptive and leave it to the database to determine how to execute them. Java statements are prescriptive—you spell out exactly the steps you want your program to carry out.

24.2.6 Updating and Deleting Data

The `UPDATE` and `DELETE` SQL commands modify the data in a database.

Up to now, you have been shown how to formulate increasingly complex `SELECT` queries. The outcome of a `SELECT` query is a *result set* that you can view and analyze. Two related statement types, `UPDATE` and `DELETE`, don't produce a result set. Instead, they modify the database. The `DELETE` statement is the easier of the two. It simply deletes the rows that you specify. For example, to delete all customers in California, you issue the statement

```
DELETE FROM Customer WHERE State = 'CA'
```

The `UPDATE` query allows you to update columns of all records that fulfill a certain condition. For example, here is how you can add another unit to the quantity of every line item in invoice number 11731:

```
UPDATE LineItem
   SET Quantity = Quantity + 1
   WHERE Invoice_Number = '11731'
```

You can update multiple column values by specifying multiple update expressions in the SET clause, separated by commas.

Both the DELETE and the UPDATE statements return a value, namely the number of rows that are deleted or updated.

3. How do you query the names of all customers that are not from Alaska or Hawaii?
4. How do you query all invoice numbers of all customers in Hawaii?

Practice It Now you can try these exercises at the end of the chapter: R24.7, R24.13, R24.14.

Common Error 24.1

Joining Tables Without Specifying a Link Condition

If you select data from multiple tables without a restriction, the result is somewhat surprising—you get a result set containing *all combinations* of the values, whether or not one of the combinations exists with actual data. For example, the query

```
SELECT Invoice.Invoice_Number, Customer.Name
   FROM Invoice, Customer
```

returns the result set

Invoice.Invoice_Number	Customer.Name
11731	Sam's Small Appliances
11732	Sam's Small Appliances
11733	Sam's Small Appliances
11731	Electronics Unlimited
11732	Electronics Unlimited
11733	Electronics Unlimited

As you can see, the result set contains all six combinations of invoice numbers (11731, 11732, 11733) and customer names (Sam's Small Appliances and Electronics Unlimited), even though three of those combinations don't occur with real invoices. You need to supply a WHERE clause to restrict the set of combinations. For example, the query

```
SELECT Invoice.Invoice_Number, Customer.Name
   FROM Invoice, Customer
   WHERE Invoice.Customer_Number = Customer.Customer_Number
```

yields

Invoice.Invoice_Number	Customer.Name
11731	Sam's Small Appliances
11732	Electronics Unlimited
11733	Sam's Small Appliances

24.3 Installing a Database

A wide variety of database systems are available. Among them are

- Production-quality databases, such as Oracle, IBM DB2, Microsoft SQL Server, PostgreSQL, or MySQL.
- Lightweight Java databases, such as Apache Derby.
- Desktop databases, such as Microsoft Access.

Which one should you choose for learning database programming? That depends greatly on your available budget, computing resources, and experience with installing complex software. In a laboratory environment with a trained administrator, it makes a lot of sense to install a production-quality database. Lightweight Java databases are much easier to install and work on a variety of platforms. This makes them a good choice for the beginner. Desktop databases have limited SQL support and can be difficult to configure for Java programming.

You need a JDBC (Java Database Connectivity) driver to access a database from a Java program.

In addition to a database, you need a *JDBC driver*. The acronym JDBC stands for Java Database Connectivity, the name of the technology that enables Java programs to interact with databases. When your Java program issues SQL commands, the driver forwards them to the database and lets your program analyze the results (see Figure 10).

Different databases require different drivers, which may be supplied by either the database manufacturer or a third party. You need to locate and install the driver that matches your database.

If you work in a computing laboratory, someone will have installed a database for you, and you should ask your lab for instructions on how to use it. If you need to provide your own database, we suggest that you choose Apache Derby. It is included with the Java Development Kit. You can also download it separately from `http://db.apache.org/derby/`.

You should run a test program to check that your database is working correctly. You will find the code for the test program at the end of this section. The following section describes the implementation of the test program in detail.

If you use Apache Derby, then follow these instructions:

1. Locate the JDBC driver file `derby.jar` and copy it into the `ch24/section_3` directory of the companion code for this book.
2. Open a shell window, change to the `ch24/section_3` directory, and run

   ```
   javac TestDB.java
   java -classpath derby.jar;. TestDB database.properties
   ```

   If you run Linux, UNIX, or Mac OS X, use a colon, not a semicolon, as a path separator:

   ```
   java -classpath derby.jar:. TestDB database.properties
   ```

3. If you followed the test instructions precisely, you should see one line of output with the name "Romeo". You may then skip the remainder of this section.

If you install a database other than the one included with Java, you will need to set aside some time to carry out the installation process. Detailed instructions for installing a database vary widely.

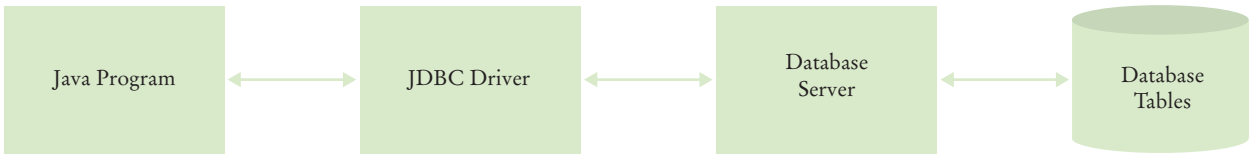

Figure 10 JDBC Architecture

Here we give you a general sequence of steps on how to install a database and test your installation:

1. Install the database program.
2. Start the database. With most database systems (but not some of the lightweight Java database systems), you need to start the database server before you can carry out any database operations. Read the installation instructions for details.
3. Set up user accounts. This typically involves running an administration program, logging in as administrator with a default administration account, and adding user names and passwords. If you are the only user of the database, you may simply be able to use a default account. Again, details vary greatly among databases, and you should consult the documentation.
4. Run a test. Locate the program that allows you to execute interactive SQL instructions. Run the program and issue the following SQL instructions:

   ```
   CREATE TABLE Test (Name VARCHAR(20))
   INSERT INTO Test VALUES ('Romeo')
   SELECT * FROM Test
   DROP TABLE Test
   ```

   At this point, you should get a display that shows a single row and column of the Test database, containing the string "Romeo". If not, carefully read the documentation of your SQL tool to see how you need to enter SQL statements. For example, with some SQL tools, you need a special terminator for each SQL statement.

Next, locate the JDBC driver and run a sample Java program to verify that the installation was successful.

Here are the steps for testing the JDBC driver:

1. Every JDBC driver contains some Java code that your Java programs require to connect to the database. From the JDBC driver documentation, find the *class path* for the driver. Here is a typical example—the class path component for the Apache Derby JDBC driver that is included in the Java Development Kit.

   ```
   c:\jdk1.7.0\db\lib\derby.jar
   ```

   One version of the Oracle database uses a class path

   ```
   /usr/local/oracle/jdbc/classes111b.zip
   ```

   You will find this information in the documentation of your database system.

Make sure the JDBC driver is on the class path when you launch the Java program.

2. If your JDBC driver is not fully compliant with the JDBC4 standard, you need to know the name of the driver class. For example, the Oracle database uses

   ```
   oracle.jdbc.driver.OracleDriver
   ```

   Your database documentation will have this information.
3. Find the name of the *database URL* that your driver expects. All database URLs have the format

   ```
   jdbc:subprotocol:driver-specific data
   ```

   The subprotocol is a code that identifies the driver manufacturer, such as `derby` or `oracle`. The driver-specific data encode the database name and the location of the database. Here are typical examples:

   ```
   jdbc:derby:InvoiceDB;create=true
   jdbc:oracle:thin:@larry.mathcs.sjsu.edu:1521:InvoiceDB
   ```

   Again, consult your JDBC driver information for details on the format of the database URL and how to specify the database that you use.

To connect to the database, you need to specify a database URL, user name, and password.

4. In order to run the `TestDB.java` program at the end of this section, edit the file `database.properties` and supply
   - The driver class name (if required).
   - The database URL.
   - Your database user name.
   - Your database password.

   With lightweight Java databases such as Apache Derby, you usually specify a blank user name and password.
5. Compile the program as

   ```
   javac TestDB.java
   ```

6. Run the program as

   ```
   java -classpath driver_class_path;. TestDB database.properties
   ```

   In UNIX/Linux/Mac OS X, use a : separator in the class path:

   ```
   java -classpath driver_class_path:. TestDB database.properties
   ```

   If everything works correctly, you will get an output that lists all data in the Test table. If you followed the test instructions precisely, you will see one line of output with the name "Romeo".

Here is the test program. We will explain the Java instructions in this program in the following section.

section_3/TestDB.java

```
1  import java.io.File
2  import java.sql.Connection;
3  import java.sql.ResultSet;
4  import java.sql.Statement;
5
6  /**
7     Tests a database installation by creating and querying
8     a sample table. Call this program as
9     java -classpath driver_class_path;. TestDB propertiesFile
10 */
```

```
11 public class TestDB
12 {
13    public static void main(String[] args) throws Exception
14    {
15       if (args.length == 0)
16       {
17          System.out.println(
18                "Usage: java -classpath driver_class_path"
19                + File.pathSeparator
20                + ". TestDB propertiesFile");
21          return;
22       }
23
24       SimpleDataSource.init(args[0]);
25
26       try (Connection conn = SimpleDataSource.getConnection())
27       {
28          Statement stat = conn.createStatement();
29
30          stat.execute("CREATE TABLE Test (Name VARCHAR(20))");
31          stat.execute("INSERT INTO Test VALUES ('Romeo')");
32
33          ResultSet result = stat.executeQuery("SELECT * FROM Test");
34          result.next();
35          System.out.println(result.getString("Name"));
36
37          stat.execute("DROP TABLE Test");
38       }
39    }
40 }
```

section_3/SimpleDataSource.java

```
1  import java.sql.Connection;
2  import java.sql.DriverManager;
3  import java.sql.SQLException;
4  import java.io.FileInputStream;
5  import java.io.IOException;
6  import java.util.Properties;
7
8  /**
9     A simple data source for getting database connections.
10 */
11 public class SimpleDataSource
12 {
13    private static String url;
14    private static String username;
15    private static String password;
16
17    /**
18       Initializes the data source.
19       @param fileName the name of the property file that
20       contains the database driver, URL, username, and password
21    */
22    public static void init(String fileName)
23          throws IOException, ClassNotFoundException
24    {
25       Properties props = new Properties();
26       FileInputStream in = new FileInputStream(fileName);
27       props.load(in);
```

```
28
29          String driver = props.getProperty("jdbc.driver");
30          url = props.getProperty("jdbc.url");
31          username = props.getProperty("jdbc.username");
32          if (username == null) { username = ""; }
33          password = props.getProperty("jdbc.password");
34          if (password == null) { password = ""; }
35          if (driver != null) { Class.forName(driver); }
36       }
37
38       /**
39          Gets a connection to the database.
40          @return the database connection
41       */
42       public static Connection getConnection() throws SQLException
43       {
44          return DriverManager.getConnection(url, username, password);
45       }
46    }
```

section_3/database.properties (for Apache Derby)

```
jdbc.url=jdbc:derby:BigJavaDB;create=true
# With other databases, you may need to add entries such as these
# jdbc.username=admin
# jdbc.password=secret
# jdbc.driver=org.apache.derby.jdbc.EmbeddedDriver
```

5. After installing a database system, how can you test that it is properly installed?
6. You are starting a Java database program to use the Apache Derby database and get the following error message:

   ```
   Exception in thread "main" java.sql.SQLException: No suitable driver found for
   jdbc:derby:BigJavaDB;create=true
   ```

   What is the most likely cause of this error?

Practice It Now you can try these exercises at the end of the chapter: R24.18, R24.19, R24.20.

24.4 Database Programming in Java

24.4.1 Connecting to the Database

Use a Connection object to access a database from a Java program.

To connect to a database, you need an object of the `Connection` class. The following shows you how to obtain such a connection. With older versions of the JDBC standard, you first need to manually load the database driver class. Starting with JDBC4 (which is a part of Java 6), the driver is loaded automatically. If you use Java 6 or later and a fully JDBC4 compatible driver, you can skip the loading step. Otherwise, use the following code:

```
String driver = . . .;
Class.forName(driver); // Load driver
```

Next, you ask the `DriverManager` for a connection. You need to initialize the `url`, `username`, and `password` strings with the values that apply to your database:

```
String url = . . .;
String username = . . .;
String password = . . .;
Connection conn = DriverManager.getConnection(url, username, password);
```

When you are done issuing your database commands, the database connection needs to be closed. Use the `try`-with-resources statement:

```
try (Connection conn = . . .)
{
   Work with conn.
}
```

Larger programs (such as the bank example in Worked Example 24.1) need to connect to the database from many classes. You don't want to propagate the database login information to a large number of classes. Also, it is usually not feasible to use a single connection for all database requests. In particular, as you will see in Chapter 26, a container for web applications can run many simultaneous web page requests from different browsers. Each page request needs its own database connection. But because opening a database connection is quite slow and page requests come so frequently, database connections need to be pooled rather than closed and reopened.

In order to decouple connection management from the other database code, we supply a `SimpleDataSource` class for this purpose. The implementation is at the end of Section 24.3. This class is a very simple tool for connection management. At the beginning of your program, call the static `init` method with the name of the database configuration file, for example

```
SimpleDataSource.init("database.properties");
```

The configuration file is a text file that may contain the following lines:

```
jdbc.driver= . . .
jdbc.url= . . .
jdbc.username= . . .
jdbc.password= . . .
```

The `init` method uses the `Properties` class, which is designed to make it easy to read such a file. The `Properties` class has a `load` method to read a file of key/value pairs from a stream:

```
Properties props = new Properties();
FileInputStream in = new FileInputStream(fileName);
props.load(in);
```

The `getProperty` method returns the value of a given key:

```
String driver = props.getProperty("jdbc.driver");
```

You don't actually have to think about this—the `init` method takes care of the details.

Whenever you need a connection, call

```
try (Connection conn = . . .)
{
   Work with conn.
}
```

Real-world connection managers have slightly different methods, but the basic principle is the same.

24.4.2 Executing SQL Statements

A Connection object can create Statement objects that are used to execute SQL commands.

Once you have a connection, you can use it to create Statement objects. You need Statement objects to execute SQL statements.

```
Statement stat = conn.createStatement();
```

The execute method of the Statement class executes a SQL statement. For example,

```
stat.execute("CREATE TABLE Test (Name CHAR(20))");
stat.execute("INSERT INTO Test VALUES ('Romeo')");
```

The result of a SQL query is returned in a ResultSet object.

To issue a query, use the executeQuery method of the Statement class. The query result is returned as a ResultSet object. For example,

```
String query = "SELECT * FROM Test";
ResultSet result = stat.executeQuery(query);
```

You will see in the next section how to use the ResultSet object to analyze the result of the query.

For UPDATE statements, you can use the executeUpdate method. It returns the number of rows affected by the statement:

```
String command = "UPDATE LineItem"
      + " SET Quantity = Quantity + 1"
      + " WHERE Invoice_Number = '11731'";
int count = stat.executeUpdate(command);
```

If your statement has variable parts, then you should use a PreparedStatement instead:

```
String query = "SELECT * WHERE Account_Number = ?";
PreparedStatement stat = conn.prepareStatement(query);
```

The ? symbols in the query string denote variables that you fill in when you make an actual query. You call one of several set methods for that purpose, for example

```
stat.setString(1, accountNumber);
```

The first parameter of the set methods denotes the variable position: 1 is the first ?, 2 the second, and so on. There are also methods setInt and setDouble for setting numerical variables. After you set all variables, you call executeQuery or executeUpdate.

Finally, you can use the generic execute method to execute arbitrary SQL statements. It returns a boolean value to indicate whether the SQL command yields a result set. If so, you can obtain it with the getResultSet method. Otherwise, you can get the update count with the getUpdateCount method.

```
String command = . . .;
boolean hasResultSet = stat.execute(command);
if (hasResultSet)
{
   ResultSet result = stat.getResultSet();
   . . .
}
else
{
   int count = stat.getUpdateCount();
   . . .
}
```

You can reuse a Statement or PreparedStatement object to execute as many SQL commands as you like. However, for each statement, you should only have one active

ResultSet. If your program needs to look at several result sets at the same time, then you need to create multiple Statement objects.

When you are done with a Statement object, be sure that it is closed by declaring it in a try-with-resources statement:

```
try (PreparedStatement stat = conn.prepareStatement(query))
{
   Configure stat.
   ResultSet result = stat.getResultSet();
   Analyze result.
}
```

You do not need to close the result set—it is automatically closed when the statement is closed. However, if you make multiple queries with the same statement, close each result set before making a new query.

When you close a connection, it automatically closes all statements and result sets.

24.4.3 Analyzing Query Results

A ResultSet lets you fetch the query result, one row at a time. You iterate through the rows, and for each row, you can inspect the column values. Like the collection iterators that you saw in Chapter 15, the ResultSet class has a next method to visit the next row. However, the behavior of the next method is somewhat different. The next method does not return any data; it returns a boolean value that indicates whether more data are available. Moreover, when you first get a result set from the executeQuery method, no row data are available. You need to call next to move to the first row. This appears curious, but it makes the iteration loop simple:

```
while (result.next())
{
   Inspect column data from the current row.
}
```

If the result set is completely empty, then the first call to result.next() returns false, and the loop is never entered. Otherwise, the first call to result.next() fetches the data for the first row from the database. As you can see, the loop ends when the next method returns false, which indicates that all rows have been fetched.

Once the result set object has fetched a particular row, you can inspect its columns. Various get methods return the column value formatted as a number, string, date, and so on. In fact, for each data type, there are two get methods. One of them has an integer argument that indicates the column position. The other has a string argument for the column name. For example, you can fetch the product code as

```
String productCode = result.getString(1);
```

or

```
String productCode = result.getString("Product_Code");
```

Note that the integer index starts at one, not at zero; that is, getString(1) inspects the first column. Database column indexes are different from array subscripts.

Accessing a column by an integer index is marginally faster and perfectly acceptable if you explicitly named the desired columns in the SELECT statement, such as

```
SELECT Invoice_Number FROM Invoice WHERE Payment = 0
```

However, if you make a SELECT * query, it is a good idea to use a column name instead of a column index. It makes your code easier to read, and you don't have to update the code when the column layout changes.

In this example, you saw the getString method in action. To fetch a number, use the getInt and getDouble methods instead, for example

```
int quantity = result.getInt("Quantity");
double unitPrice = result.getDouble("Price");
```

24.4.4 Result Set Metadata

Metadata are data about an object. Result set metadata describe the properties of a result set.

When you have a result set from an unknown table, you may want to know the names of the columns. You can use the ResultSetMetaData class to find out about properties of a result set. Start by requesting the metadata object from the result set:

```
ResultSetMetaData metaData = result.getMetaData();
```

Then you can get the number of columns with the getColumnCount method. The getColumnLabel method gives you the column name for each column. Finally, the getColumnDisplaySize method returns the column width, which is useful if you want to print table rows and have the columns line up. Note that the indexes for these methods start with 1. For example,

```
for (int i = 1; i <= metaData.getColumnCount(); i++)
{
   String columnName = metaData.getColumnLabel(i);
   int columnSize = metaData.getColumnDisplaySize(i);
   . . .
}
```

ExecSQL.java is a useful sample program that puts these concepts to work. The program reads a file containing SQL statements and executes them all. When a statement has a result set, the result set is printed, using the result set metadata to determine the column count and column labels.

For example, suppose you have the following file:

section_4/Product.sql

```
CREATE TABLE Product
      (Product_Code CHAR(7), Description VARCHAR(40), Price DECIMAL(10, 2))
INSERT INTO Product VALUES ('116-064', 'Toaster', 24.95)
INSERT INTO Product VALUES ('257-535', 'Hair dryer', 29.95)
INSERT INTO Product VALUES ('643-119', 'Car vacuum', 19.95)
SELECT * FROM Product
```

Run the Exec.SQL program as

```
java -classpath derby.jar;. ExecSQL database.properties Product.sql
```

The program executes the statements in the Product.sql file and prints out the result of the SELECT query.

You can also use the Exec.SQL program as an interactive testing tool. Run

```
java -classpath derby.jar;. ExecSQL database.properties
```

Then type in SQL commands at the command line. Every time you press the Enter key, the command is executed.

section_4/ExecSQL.java

```
1  import java.sql.Connection;
2  import java.sql.ResultSet;
3  import java.sql.ResultSetMetaData;
4  import java.sql.Statement;
5  import java.sql.SQLException;
6  import java.io.File;
7  import java.io.IOException;
8  import java.util.Scanner;
9
10 /**
11    Executes all SQL statements from a file or the console.
12 */
13 public class ExecSQL
14 {
15    public static void main(String[] args)
16          throws SQLException, IOException, ClassNotFoundException
17    {
18       if (args.length == 0)
19       {
20          System.out.println(
21                "Usage: java -classpath driver_class_path"
22                + File.pathSeparator
23                + ". ExecSQL propertiesFile [SQLcommandFile]");
24          return;
25       }
26
27       SimpleDataSource.init(args[0]);
28
29       Scanner in;
30       if (args.length > 1)
31       {
32          in = new Scanner(new File(args[1]));
33       }
34       else
35       {
36          in = new Scanner(System.in);
37       }
38
39       try (Connection conn = SimpleDataSource.getConnection(),
40             Statement stat = conn.createStatement())
41       {
42          while (in.hasNextLine())
43          {
44             String line = in.nextLine();
45             try
46             {
47                boolean hasResultSet = stat.execute(line);
48                if (hasResultSet)
49                {
50                   try (ResultSet result = stat.getResultSet())
51                   {
52                      showResultSet(result);
53                   }
54                }
55             }
56             catch (SQLException ex)
57             {
58                System.out.println(ex);
```

```
59            }
60         }
61      }
62   }
63
64   /**
65      Prints a result set.
66      @param result the result set
67   */
68   public static void showResultSet(ResultSet result)
69         throws SQLException
70   {
71      ResultSetMetaData metaData = result.getMetaData();
72      int columnCount = metaData.getColumnCount();
73
74      for (int i = 1; i <= columnCount; i++)
75      {
76         if (i > 1) { System.out.print(", "); }
77         System.out.print(metaData.getColumnLabel(i));
78      }
79      System.out.println();
80
81      while (result.next())
82      {
83         for (int i = 1; i <= columnCount; i++)
84         {
85            if (i > 1) { System.out.print(", "); }
86            System.out.print(result.getString(i));
87         }
88         System.out.println();
89      }
90   }
91 }
```

7. Suppose you want to test whether there are any customers in Hawaii. Issue the statement

   ```
   ResultSet result = stat.executeQuery("SELECT * FROM Customer WHERE State = 'HI'");
   ```

   Which Boolean expression answers your question?

8. Suppose you want to know how many customers are in Hawaii. What is an efficient way to get this answer?

Practice It Now you can try these exercises at the end of the chapter: R24.22, E24.3, E24.5.

Common Error 24.2

Constructing Queries from Arbitrary Strings

Suppose you need to issue the following query with different names.

```
SELECT * FROM Customer WHERE Name = customerName
```

Many students try to construct a SELECT statement manually, like this:

```
String customerName = . . . ;
String query = "SELECT * FROM Customer WHERE Name = '" + customerName + "'";
ResultSet result = stat.executeQuery(query);
```

However, this code will fail if the name contains single quotes, such as `"Sam's Small Appliances"`. The query string has a syntax error: a mismatched quote. More serious failures can be introduced by hackers who deliberately enter names or addresses with SQL control characters, changing the meanings of queries. These "SQL injection attacks" have been responsible for many cases of data theft. Never add a string to a query that you didn't type yourself.

The remedy is to use a `PreparedStatement` instead:

```
String query = "SELECT * FROM Customer WHERE Name = ?";
PreparedStatement stat = conn.prepareStatement(query);
stat.setString(1, aName);
ResultSet result = stat.executeQuery(query);
```

The `setString` method of the `PreparedStatement` class will properly handle quotes and other special characters in the string.

Programming Tip 24.4

Don't Hardwire Database Connection Parameters into Your Program

It is considered inelegant to hardwire the database parameters into a program:

```
public class MyProg
{
   public static void main(String[] args)
   {
      // Don't do this:
      String driver = "oracle.jdbc.driver.OracleDriver";
      String url = "jdbc:oracle:thin:@larry.mathcs.sjsu.edu:1521:InvoiceDB";
      String username = "admin";
      String password = "secret";
      . . .
   }
}
```

If you want to change to a different database, you must locate these strings, update them, and recompile.

Instead, place the strings into a separate configuration file (such as `database.properties` in our sample program). The `SimpleDataSource.java` file reads in the configuration file with the database connection parameters. To connect to a different database, you simply supply a different configuration file name on the command line.

Programming Tip 24.5

Let the Database Do the Work

You now know how to issue a SQL query from a Java program and iterate through the result set. A common error that students make is to iterate through one table at a time to find a result. For example, suppose you want to find all invoices that contain car vacuums. You could use the following plan:

1. Issue the query `SELECT * FROM Product` and iterate through the result set to find the product code for a car vacuum.
2. Issue the query `SELECT * FROM LineItem` and iterate through the result set to find the line items with that product code.

However, that plan is *extremely inefficient*. Such a program does in very slow motion what a database has been designed to do quickly.

Instead, you should let the database do all the work. Give the complete query to the database:

```
SELECT LineItem.Invoice_Number
   FROM Product, LineItem
   WHERE Product.Description = 'Car vacuum'
      AND Product.Product_Code = LineItem.Product_Code
```

Then iterate through the result set to read all invoice numbers.

Beginners are often afraid of issuing complex SQL queries. However, you are throwing away a major benefit of a relational database if you don't take advantage of SQL.

24.5 Application: Entering an Invoice

In this section, we develop a program for entering an invoice into the database shown in Figure 8. Here is a sample program run:

```
Name: Robert Lee
Street address: 833 Lyon Street
City: San Francisco
State: CA
Zip: 94155
Product code (D=Done, L=List): L
116-064 Toaster
257-535 Hair dryer
643-119 Car vacuum
Product code (D=Done, L=List): 116-064
Quantity: 2
Product code (D=Done, L=List): 257-535
Quantity: 3
Product code (D=Done, L=List): D
Robert Lee
833 Lyon Street
San Francisco, CA 94155
2 x 116-064 Toaster
3 x 257-535 Hair dryer
```

This program puts the concepts of the preceding sections to work.

Before running the program, we assume that the Customer, Product, Invoice, and LineItem tables have been created. To do so, you can run the `ExecSQL` program of Section 24.4 with the following files (provided with the book's companion code):

```
Customer.sql
Product.sql
Invoice.sql
LineItem.sql
```

As in the previous programs, we use our `SimpleDataSource` helper class to get a database connection. Then we call `addInvoice` so it can prompt for the invoice information.

```
SimpleDataSource.init(args[0]);

try (Connection conn = SimpleDataSource.getConnection(),
   Scanner in = new Scanner(System.in))
{
   addInvoice(in, conn);
}
```

The `newCustomer` method prompts for the new customer information and adds it to the database. The `nextLine` method is a convenience method for prompting the user and reading a string from a `Scanner`—see the code at the end of this section.

```
private static int newCustomer(Connection conn, Scanner in)
   throws SQLException
{
   String name = nextLine(in, "Name");
   String address = nextLine(in, "Street address");
   String city = nextLine(in, "City");
   String state = nextLine(in, "State");
   String zip = nextLine(in, "Zip");
   int id = . . .;
   try (PreparedStatement stat = conn.prepareStatement(
      "INSERT INTO Customer VALUES (?, ?, ?, ?, ?, ?)"))
   {
      stat.setInt(1, id);
      stat.setString(2, name);
      stat.setString(3, address);
      stat.setString(4, city);
      stat.setString(5, state);
      stat.setString(6, zip);
      stat.executeUpdate();
   }
   return id;
}
```

The method gathers the new customer data and issues an `INSERT INTO` statement to store them in the database.

But how do we provide the ID? We don't want to ask the program user to come up with IDs. They should be automatically assigned. We will query the largest ID that has been used so far, and use the next larger value as the new ID:

```
try (Statement stat = conn.createStatement())
{
   ResultSet result = stat.executeQuery(
      "SELECT max(Customer_Number) FROM Customer");
   result.next();
   int id = result.getInt(1) + 1;
}
```

There is just one potential problem. If two users access the database simultaneously, it is possible that both of them create a customer with the same ID at the same time. The remedy is to place the code for adding a customer inside a transaction—see Special Topic 24.2. This is an important requirement in a database with simultaneous users. However, we will skip this step to keep the program simple.

Later, we also need to generate new IDs for invoices. We provide a method `getNewId` that works for both tables.

This completes the customer portion of the invoice entry. We now add a row for the invoice, calling the `getNewId` method to get a new invoice number.

```
int id = getNewId(conn, "Invoice");
try (PreparedStatement stat = conn.prepareStatement(
   "INSERT INTO Invoice VALUES (?, ?, 0)"))
{
   stat.setInt(1, id);
   stat.setInt(2, customerNumber);
   stat.executeUpdate();
}
```

Next, the user needs to enter the product codes. When a user provides a code, we will check that it is valid. That is a very simple SELECT query. We don't even look at the result set—if it has a row, we have found the product.

```
try (PreparedStatement stat = conn.prepareStatement(
   "SELECT * FROM Product WHERE Product_Code = ?"))
{
   stat.setString(1, code);
   ResultSet result = stat.executeQuery();
   boolean found = result.next();
}
```

You will find this code in the findProduct method.

When the user chooses to see a list of products, we issue a simple query and list the result, as shown here:

```
try (Statement stat = conn.createStatement())
{
   ResultSet result = stat.executeQuery(
      "SELECT Product_Code, Description FROM Product");
   while (result.next())
   {
      String code = result.getString(1);
      String description = result.getString(2);
      System.out.println(code + " " + description);
   }
}
```

Whenever the user has supplied a product code and a quantity, we add another row to the LineItem table. That is yet another INSERT INTO statement—you will find it in the addLineItem method below.

The following loop keeps asking for product codes and quantities:

```
boolean done = false;
while (!done)
{
   String productCode = nextLine(in, "Product code (D=Done, L=List)");
   if (productCode.equals("D")) { done = true; }
   else if (productCode.equals("L")) { listProducts(conn); }
   else if (findProduct(conn, productCode))
   {
      int quantity = nextInt(in, "Quantity");
      addLineItem(conn, id, productCode, quantity);
   }
   else { System.out.println("Invalid product code."); }
}
showInvoice(conn, id);
```

When the loop ends, we print the invoice. Here, the queries are more interesting. The showInvoice method has the invoice ID as a parameter. It needs to find the matching customer data by joining the Invoice and Customer tables:

```
try (PreparedStatement stat = conn.prepareStatement(
   "SELECT Customer.Name, Customer.Address, "
   + "Customer.City, Customer.State, Customer.Zip "
   + "FROM Customer, Invoice "
   + "WHERE Customer.Customer_Number = Invoice.Customer_Number "
   + "AND Invoice.Invoice_Number = ?"))
{
   stat.setInt(1, id);
   . . .
}
```

The query result contains the customer information which we print. Then we need to get all line items and the product descriptions, again linking two tables in a query:

```
try (PreparedStatement stat = conn.prepareStatement(
   "SELECT Product.Product_Code, Product.Description, LineItem.Quantity "
   + "FROM Product, LineItem "
   + "WHERE Product.Product_Code = LineItem.Product_Code "
   + "AND LineItem.Invoice_Number = ?"))
{
   stat.setInt(1, id);
   . . .
}
```

Our program simply prints the data in a simple form, as you saw at the beginning of this section; Exercise E24.5 asks you to format it better.

Following is the complete invoice entry program.

section_5/InvoiceEntry.java

```
1  import java.sql.Connection;
2  import java.sql.PreparedStatement;
3  import java.sql.ResultSet;
4  import java.sql.SQLException;
5  import java.sql.Statement;
6  import java.io.IOException;
7  import java.io.File;
8  import java.util.Scanner;
9
10 /**
11    Enters an invoice into the database.
12    Be sure to add Customer.sql, Product.sql, Invoice.sql, and LineItem.sql
13    to the database before running this program.
14 */
15 public class InvoiceEntry
16 {
17    public static void main(String args[])
18    {
19       if (args.length == 0)
20       {
21          System.out.println(
22                "Usage: java -classpath driver_class_path"
23                + File.pathSeparator
24                + ". InvoiceEntry propertiesFile");
25          return;
26       }
27
28       try
29       {
30          SimpleDataSource.init(args[0]);
31          try (Connection conn = SimpleDataSource.getConnection(),
32             Scanner in = new Scanner(System.in))
33          {
34             addInvoice(in, conn);
35          }
36       }
37       catch (SQLException ex)
38       {
39          System.out.println("Database error");
40          ex.printStackTrace();
```

```
41          }
42          catch (ClassNotFoundException ex)
43          {
44             System.out.println("Error loading database driver");
45             ex.printStackTrace();
46          }
47          catch (IOException ex)
48          {
49             System.out.println("Error loading database properties");
50             ex.printStackTrace();
51          }
52       }
53
54       public static void addInvoice(Scanner in, Connection conn)
55          throws SQLException
56       {
57          int customerNumber = newCustomer(conn, in);
58
59          int id = getNewId(conn, "Invoice");
60          try (PreparedStatement stat = conn.prepareStatement(
61             "INSERT INTO Invoice VALUES (?, ?, 0)"))
62          {
63             stat.setInt(1, id);
64             stat.setInt(2, customerNumber);
65             stat.executeUpdate();
66          }
67
68          boolean done = false;
69          while (!done)
70          {
71             String productCode = nextLine(in, "Product code (D=Done, L=List)");
72             if (productCode.equals("D")) { done = true; }
73             else if (productCode.equals("L")) { listProducts(conn); }
74             else if (findProduct(conn, productCode))
75             {
76                int quantity = nextInt(in, "Quantity");
77                addLineItem(conn, id, productCode, quantity);
78             }
79             else { System.out.println("Invalid product code."); }
80          }
81          showInvoice(conn, id);
82       }
83
84       /**
85          Prompts the user for the customer information and creates a new customer.
86          @param conn the database connection
87          @param in the scanner
88          @return the ID of the new customer
89       */
90       private static int newCustomer(Connection conn, Scanner in)
91          throws SQLException
92       {
93          String name = nextLine(in, "Name");
94          String address = nextLine(in, "Street address");
95          String city = nextLine(in, "City");
96          String state = nextLine(in, "State");
97          String zip = nextLine(in, "Zip");
98          int id = getNewId(conn, "Customer");
```

```
99          try (PreparedStatement stat = conn.prepareStatement(
100            "INSERT INTO Customer VALUES (?, ?, ?, ?, ?, ?)"))
101         {
102            stat.setInt(1, id);
103            stat.setString(2, name);
104            stat.setString(3, address);
105            stat.setString(4, city);
106            stat.setString(5, state);
107            stat.setString(6, zip);
108            stat.executeUpdate();
109         }
110         return id;
111      }
112
113      /**
114         Finds a product in the database.
115         @param conn the database connection
116         @param code the product code to search
117         @return true if there is a product with the given code
118      */
119      private static boolean findProduct(Connection conn, String code)
120         throws SQLException
121      {
122         try (PreparedStatement stat = conn.prepareStatement(
123            "SELECT * FROM Product WHERE Product_Code = ?"))
124         {
125            stat.setString(1, code);
126            ResultSet result = stat.executeQuery();
127            boolean found = result.next();
128         }
129         return found;
130      }
131
132      /**
133         Adds a line item to the database.
134         @param conn the database connection
135         @param id the invoice ID
136         @param code the product code
137         @param quantity the quantity to order
138      */
139      private static void addLineItem(Connection conn, int id,
140         String code, int quantity) throws SQLException
141      {
142         try (PreparedStatement stat = conn.prepareStatement(
143            "INSERT INTO LineItem VALUES (?, ?, ?)"))
144         {
145            stat.setInt(1, id);
146            stat.setString(2, code);
147            stat.setInt(3, quantity);
148            stat.executeUpdate();
149         }
150      }
151
152      /**
153         Lists all products in the database.
154         @param conn the database connection
155      */
156      private static void listProducts(Connection conn)
157         throws SQLException
158      {
```

```
159         try (Statement stat = conn.createStatement())
160         {
161            ResultSet result = stat.executeQuery(
162               "SELECT Product_Code, Description FROM Product");
163            while (result.next())
164            {
165               String code = result.getString(1);
166               String description = result.getString(2);
167               System.out.println(code + " " + description);
168            }
169         }
170      }
171
172      /**
173         Gets a new ID for a table. This method should be called from
174         inside a transaction that also creates the new row with this ID.
175         The ID field should have name table_Number and type INTEGER.
176         @param table the table name
177         @return a new ID that has not yet been used.
178      */
179      private static int getNewId(Connection conn, String table)
180         throws SQLException
181      {
182         try (Statement stat = conn.createStatement())
183         {
184            ResultSet result = stat.executeQuery(
185               "SELECT max(" + table + "_Number) FROM " + table);
186            result.next();
187            int max = result.getInt(1);
188         }
189         return max + 1;
190      }
191
192      /**
193         Shows an invoice.
194         @param conn the database connection
195         @param id the invoice ID
196      */
197      private static void showInvoice(Connection conn, int id)
198         throws SQLException
199      {
200         try (PreparedStatement stat = conn.prepareStatement(
201            "SELECT Customer.Name, Customer.Address, "
202            + "Customer.City, Customer.State, Customer.Zip "
203            + "FROM Customer, Invoice "
204            + "WHERE Customer.Customer_Number = Invoice.Customer_Number "
205            + "AND Invoice.Invoice_Number = ?"))
206         {
207            stat.setInt(1, id);
208            ResultSet result = stat.executeQuery();
209            result.next();
210            System.out.println(result.getString(1));
211            System.out.println(result.getString(2));
212            System.out.println(result.getString(3).trim() + ", "
213               + result.getString(4) + " " + result.getString(5));
214         }
215
216         try (PreparedStatement stat = conn.prepareStatement(
217            "SELECT Product.Product_Code, Product.Description, LineItem.Quantity "
218            + "FROM Product, LineItem "
```

```
219             + "WHERE Product.Product_Code = LineItem.Product_Code "
220             + "AND LineItem.Invoice_Number = ?"))
221          {
222             stat.setInt(1, id);
223
224             result = stat.executeQuery();
225             while (result.next())
226             {
227                String code = result.getString(1);
228                String description = result.getString(2).trim();
229                int qty = result.getInt(3);
230
231                System.out.println(qty + " x " + code + " " + description);
232             }
233          }
234       }
235
236       /**
237          Prompts the user and reads a line from a scanner.
238          @param in the scanner
239          @param prompt the prompt
240          @return the string that the user entered
241       */
242       private static String nextLine(Scanner in, String prompt)
243       {
244          System.out.print(prompt + ": ");
245          return in.nextLine();
246       }
247
248       /**
249          Prompts the user and reads an integer from a scanner.
250          @param in the scanner
251          @param prompt the prompt
252          @return the integer that the user entered
253       */
254       private static int nextInt(Scanner in, String prompt)
255       {
256          System.out.print(prompt + ": ");
257          int result = in.nextInt();
258          in.nextLine(); // Consume newline
259          return result;
260       }
261    }
```

This example completes this introduction to Java database programming. You have seen how you can use SQL to query and update data in a database and how the JDBC library makes it easy for you to issue SQL commands in a Java program.

9. Why do the `InvoiceEntry` methods throw a `SQLException` instead of catching it?
10. How could one simplify the first query of the `showInvoice` method?
11. This program assumes that the customer does not pay at the time of order entry. How can the program be modified to handle payment?
12. This program does not display the amount due. How could we display it?

Practice It Now you can try these exercises at the end of the chapter: P24.1, P24.2.

Special Topic 24.2

Transactions

An important part of database processing is *transaction handling*. A **transaction** is a set of database updates that should either succeed in its entirety or not happen at all. For example, consider a banking application that transfers money from one account to another. This operation involves two steps: reducing the balance of one account and increasing the balance of another account. No software system is perfect, and there is always the possibility of an error. The banking application, the database program, or the network connection between them could exhibit an error right after the first part—then the money would be withdrawn from the first account but never deposited to the second account. Clearly, this would be very bad. There are many other similar situations. For example, if you change an airline reservation, you don't want to give up your old seat until the new one is confirmed.

What all these situations have in common is that there is a set of database operations that are grouped together to carry out the transaction. All operations in the group must be carried out together—a partial completion cannot be tolerated. In SQL, you use the `COMMIT` and `ROLLBACK` commands to manage transactions.

For example, to transfer money from one account to another, you issue the commands

```
UPDATE Account SET Balance = Balance - 1000
   WHERE Account_Number = '95667-2574'
UPDATE Account SET Balance = Balance + 1000
   WHERE Account_Number = '82041-1196'
COMMIT
```

The `COMMIT` command makes the updates permanent. Conversely, the `ROLLBACK` command undoes all changes up to the last `COMMIT`.

When you program with JDBC, by default the JDBC library automatically commits all database updates. That is convenient for simple programs, but it is not what you want for transaction processing. Thus, you should first turn the autocommit mode off:

```
Connection conn = . . .;
conn.setAutoCommit(false);
```

Then issue the updates that form the transaction and call the `commit` method of the `Connection` class:

```
Statement stat = conn.createStatement();
stat.executeUpdate(
      "UPDATE Account SET Balance = Balance - "
      + amount + " WHERE Account_Number = " + fromAccount);
stat.executeUpdate(
      "UPDATE Account SET Balance = Balance + "
      + amount + " WHERE Account_Number = " + toAccount);
conn.commit();
```

Conversely, if you encounter an error, then call the `rollback` method. This typically happens in an exception handler:

```
try
{
   . . .
}
catch (Exception ex)
{
   conn.rollback();
}
```

You may wonder how a database can undo updates when a transaction is rolled back. The database actually stores your changes in a set of temporary tables. If you make queries within a transaction, the information in the temporary tables is merged with the permanent data for the purpose of computing the query result, giving you the illusion that the updates have already

taken place. When you commit the transaction, the temporary data are made permanent. When you execute a rollback, the temporary tables are simply discarded.

Special Topic 24.3

Object-Relational Mapping

Database tables store rows that contain strings, numbers, and other fundamental data types, but not arbitrary objects. In Sections 24.1.2 and 24.1.3, you learned how to translate object references into database relationships. An object-relational mapper automates this process. The Java Enterprise Edition contains such a mapper. You add annotations to the Java classes that describe the relationships. The rules are simple:

- Add `@Entity` to every class that should be stored in the database.
- Each entity class needs an ID that is annotated with `@Id`.
- Relationships between classes are expressed with `@OneToOne`, `@OneToMany`, `@ManyToOne`, and `@ManyToMany`.

Here are the annotations for the invoice classes. A customer can have many invoices, but each invoice has exactly one customer. This is expressed by the `@ManyToOne` annotation. Conversely, each line item is contained in exactly one invoice, but each invoice can have many line items. This is expressed by the `@OneToMany` relationship.

```
@Entity public class Invoice
{
   @Id private int id;
   @ManyToOne private Customer theCustomer;
   @OneToMany private List<LineItem> items;
   private double payment;
   . . .
}

@Entity public class LineItem
{
   @Id private int id;
   @ManyToOne private Product theProduct;
   private int quantity
   . . .
}

@Entity public class Product
{
   @Id private int id;
   private String description;
   private double price;
   . . .
}

@Entity public class Customer
{
   @Id private int id;
   private String name;
   private String address;
   private String city;
   private String state;
   private String zip;
   . . .
}
```

The object-relational mapper processes the annotations and produces a database table layout. You don't have to worry exactly how the data are stored in the database. For example, to store a new invoice, simply build up the Java object and call

```
entityManager.persist(invoice);
```

As a result of this call, the data for the invoice and line items are automatically stored in the various database tables.

To read data from the database, you do not use SQL—after all, you do not know the exact table layout. Instead, you formulate a query in an object-oriented query language. A typical query looks like this:

```
SELECT x FROM Invoice x WHERE x.id = 11731
```

The result is a Java object of type `Invoice`. The references to the customer and line item objects have been automatically populated with the proper data from various tables.

Object-relational mapping technology is powerful and convenient. However, you still need to understand the underlying principles of relational databases in order to specify efficient mappings and queries.

WORKED EXAMPLE 24.1 Programming a Bank Database

In this Worked Example, we will develop a complete database program. We will reimplement the ATM simulation of Chapter 12, storing the customer and account data in a database. Recall that in the simulation, every customer has a customer number, a PIN, and two bank accounts: a checking account and a savings account.

We'll store the information in two tables:

BankCustomer

Customer_Number	PIN	Checking_Account_Number	Savings_Account_Number
INTEGER	INTEGER	INTEGER	INTEGER

Account

Account_Number	Balance
INTEGER	DECIMAL(10, 2)

The `Bank` class now needs to connect to the database whenever it is asked to find a customer. The method that finds a customer makes a query

```
SELECT * FROM BankCustomer WHERE Customer_Number = . . .
```

It then checks that the PIN matches, and it constructs a `Customer` object. This method also turns the row-and-column information of the database into object-oriented data:

```
public Customer findCustomer(int customerNumber, int pin)
      throws SQLException
{
   try (Connection conn = SimpleDataSource.getConnection())
   {
      Customer c = null;
      PreparedStatement stat = conn.prepareStatement(
            "SELECT * FROM BankCustomer WHERE Customer_Number = ?");
```

```
      stat.setInt(1, customerNumber);

      ResultSet result = stat.executeQuery();
      if (result.next() && pin == result.getInt("PIN"))
      {
         c = new Customer(customerNumber,
               result.getInt("Checking_Account_Number"),
               result.getInt("Savings_Account_Number"));
      }
      return c;
   }
}
```

Note that the method throws a `SQLException`. Why don't we catch that exception and return `null` if an exception occurs? There are many potential reasons for a SQL exception, and the `Bank` class doesn't want to hide the exception details. But the `Bank` class also doesn't know anything about the user interface of the application, so it can't display information about the exception to the user. By throwing the exception to the caller, the information can reach the part of the program that interacts with the user.

The `BankAccount` class in this program is quite different from the implementation you have seen throughout the book. Now we do not store the balance of the bank account in the object; instead, we look it up from the database:

```
public double getBalance() throws SQLException
{
   try (Connection conn = SimpleDataSource.getConnection())
   {
      double balance = 0
      PreparedStatement stat = conn.prepareStatement(
            "SELECT Balance FROM Account WHERE Account_Number = ?");
      stat.setInt(1, accountNumber);
      ResultSet result = stat.executeQuery();
      if (result.next())
      {
         balance = result.getDouble(1);
      }
      return balance;
   }
}
```

The `deposit` and `withdraw` operations immediately update the database as well:

```
public void deposit(double amount)
      throws SQLException
{
   try (Connection conn = SimpleDataSource.getConnection())
   {
      PreparedStatement stat = conn.prepareStatement(
            "UPDATE Account"
            + " SET Balance = Balance + ?"
            + " WHERE Account_Number = ?");
      stat.setDouble(1, amount);
      stat.setInt(2, accountNumber);
      stat.executeUpdate();
   }
}
```

It seems somewhat inefficient to connect to the database whenever the bank balance is accessed, but it is much safer than storing it in an object. Suppose you have two instances of the ATM program running at the same time. Then it is possible that both programs modify the same bank account. If each of them copied the bank balances from the database into objects, then the modifications made by one user would not be seen by the other.

You can try out this simultaneous access yourself, simply by running two instances of the ATM simulation. Alternatively, you can modify the `main` method of the `ATMViewer` class to pop up two `ATM` frames.

The source code for the modified ATM application follows. The source code for the `ATM` and `ATMSimulator/ATMFrame` classes is only changed minimally, by adding code to deal with the `SQLException`. The `Customer` class is unchanged. We do not list those classes, but you will find them in the `ch24/worked_example_1` folder of this chapter's companion code.

worked_example_1/Bank.java

```
1  import java.sql.Connection;
2  import java.sql.ResultSet;
3  import java.sql.PreparedStatement;
4  import java.sql.SQLException;
5
6  /**
7     A bank consisting of multiple bank accounts.
8  */
9  public class Bank
10 {
11    /**
12       Finds a customer with a given number and PIN.
13       @param customerNumber the customer number
14       @param pin the personal identification number
15       @return the matching customer, or null if none found
16    */
17    public Customer findCustomer(int customerNumber, int pin)
18          throws SQLException
19    {
20       try (Connection conn = SimpleDataSource.getConnection())
21       {
22          Customer c = null;
23          PreparedStatement stat = conn.prepareStatement(
24                "SELECT * FROM BankCustomer WHERE Customer_Number = ?");
25          stat.setInt(1, customerNumber);
26
27          ResultSet result = stat.executeQuery();
28          if (result.next() && pin == result.getInt("PIN"))
29          {
30             c = new Customer(customerNumber,
31                   result.getInt("Checking_Account_Number"),
32                   result.getInt("Savings_Account_Number"));
33          }
34          return c;
35       }
36    }
37 }
```

worked_example_1/BankAccount.java

```
1  import java.sql.Connection;
2  import java.sql.ResultSet;
3  import java.sql.PreparedStatement;
4  import java.sql.SQLException;
5
6  /**
7     A bank account has a balance that can be changed by
8     deposits and withdrawals.
9  */
```

```
10  public class BankAccount
11  {
12     private int accountNumber;
13
14     /**
15        Constructs a bank account with a given balance.
16        @param anAccountNumber the account number
17     */
18     public BankAccount(int anAccountNumber)
19     {
20        accountNumber = anAccountNumber;
21     }
22
23     /**
24        Deposits money into a bank account.
25        @param amount the amount to deposit
26     */
27     public void deposit(double amount)
28           throws SQLException
29     {
30        try (Connection conn = SimpleDataSource.getConnection())
31        {
32           PreparedStatement stat = conn.prepareStatement(
33                 "UPDATE Account"
34                 + " SET Balance = Balance + ?"
35                 + " WHERE Account_Number = ?");
36           stat.setDouble(1, amount);
37           stat.setInt(2, accountNumber);
38           stat.executeUpdate();
39        }
40     }
41
42     /**
43        Withdraws money from a bank account.
44        @param amount the amount to withdraw
45     */
46     public void withdraw(double amount)
47           throws SQLException
48     {
49        try (Connection conn = SimpleDataSource.getConnection())
50        {
51           PreparedStatement stat = conn.prepareStatement(
52                 "UPDATE Account"
53                 + " SET Balance = Balance - ?"
54                 + " WHERE Account_Number = ?");
55           stat.setDouble(1, amount);
56           stat.setInt(2, accountNumber);
57           stat.executeUpdate();
58        }
59     }
60
61     /**
62        Gets the balance of a bank account.
63        @return the account balance
64     */
65     public double getBalance()
66           throws SQLException
67     {
68        try (Connection conn = SimpleDataSource.getConnection())
69        {
```

```
70          double balance = 0
71          PreparedStatement stat = conn.prepareStatement(
72                "SELECT Balance FROM Account WHERE Account_Number = ?");
73          stat.setInt(1, accountNumber);
74          ResultSet result = stat.executeQuery();
75          if (result.next())
76          {
77             balance = result.getDouble(1);
78          }
79          return balance;
80       }
81    }
82 }
```

CHAPTER SUMMARY

Develop strategies for storing data in a database.

- A relational database stores information in tables. Each table column has a name and a data type.
- SQL (Structured Query Language) is a command language for interacting with a database.
- Use the SQL commands `CREATE TABLE` and `INSERT INTO` to add data to a database.
- You should avoid rows with replicated data. Instead, distribute the data over multiple tables.
- A primary key is a column (or set of columns) whose value uniquely specifies a table record.
- A foreign key is a reference to a primary key in a linked table.
- Implement one-to-many relationships with linked tables, not replicated columns.

Use SQL to query and update a database.

- Use the SQL `SELECT` command to query a database.
- The `WHERE` clause selects data that fulfill a condition.
- A join is a query that involves multiple tables.
- The `UPDATE` and `DELETE` SQL commands modify the data in a database.

Install a database system and test that you can connect to it from a Java program.

- You need a JDBC (Java Database Connectivity) driver to access a database from a Java program.
- Make sure the JDBC driver is on the class path when you launch the Java program.
- To connect to the database, you need to specify a database URL, user name, and password.

Write Java programs that access and update database records.

- Use a `Connection` object to access a database from a Java program.
- A `Connection` object can create `Statement` objects that are used to execute SQL commands.

- The result of a SQL query is returned in a `ResultSet` object.
- Metadata are data about an object. Result set metadata describe the properties of a result set.

Develop programs that use the JDBC library for accessing a database.

STANDARD LIBRARY ITEMS INTRODUCED IN THIS CHAPTER

```
java.io.File
   pathSeparator
java.lang.Class
   forName
java.sql.Connection
   close
   commit
   createStatement
   prepareStatement
   rollback
   setAutoCommit
java.sql.DriverManager
   getConnection
java.sql.PreparedStatement
   execute
   executeQuery
   executeUpdate
   setDouble
   setInt
   setString
java.sql.ResultSet
   close
   getDouble
   getInt
   getMetaData
   getString
   next
java.sql.ResultSetMetaData
   getColumnCount
   getColumnDisplaySize
   getColumnLabel
java.sql.SQLException
java.sql.Statement
   close
   execute
   executeQuery
   executeUpdate
   getResultSet
   getUpdateCount
java.util.Properties
   getProperty
   load
```

REVIEW EXERCISES

- **R24.1** Design a set of database tables to store people and cars. A person has a name, a unique driver license number, and an address. Every car has a unique vehicle identification number, manufacturer, type, and year. Every car has one owner, but one person can own multiple cars.
- **R24.2** Design a set of database tables to store library books and patrons. A book has an ISBN (International Standard Book Number), an author, and a title. The library may have multiple copies of each book, each with a different book ID. A patron has a name, a unique ID, and an address. A book may be checked out by at most one patron, but one patron can check out multiple books.
- **R24.3** Design a set of database tables to store sets of coins in purses. Each purse has an owner name and a unique ID. Each coin type has a unique name and a value. Each purse contains some quantity of coins of a given type.
- **R24.4** Design a set of database tables to store students, classes, professors, and classrooms. Each student takes zero or more classes. Each class has one professor, but a professor can teach multiple classes. Each class has one classroom.
- **R24.5** Give SQL commands to create a Book table, with columns for the ISBN, author, and title, and to insert all textbooks that you are using this semester.
- **R24.6** Give SQL commands to create a Car table, with columns for the vehicle identification number, manufacturer, model, and year of each car, and to insert all cars that your family members own.

Exercises R24.7–R24.17 refer to the invoice database of Section 24.2 on page W1022.

■ **R24.7** Give a SQL query that lists all products in the invoice database of Section 24.2.

■ **R24.8** Give a SQL query that lists all customers in California.

■ **R24.9** Give a SQL query that lists all customers in California or Nevada.

■ **R24.10** Give a SQL query that lists all customers not in Hawaii.

■■ **R24.11** Give a SQL query that lists all customers who have an unpaid invoice.

■■ **R24.12** Give a SQL query that lists all products that have been purchased by a customer in California.

■■ **R24.13** Give a SQL query that lists all line items that are part of invoice number 11731.

■■ **R24.14** Give a SQL query that computes the sum of all quantities that are part of invoice number 11731.

■■■ **R24.15** Give a SQL query that computes the total cost of all line items in invoice number 11731.

■■ **R24.16** Give a SQL update statement that raises all prices by ten percent.

■■ **R24.17** Give a SQL statement that deletes all customers in California.

■■ **R24.18** Pick a database system (such as DB2, Oracle, Postgres, or SQL Server) and determine from the web documentation:

- What JDBC driver do you need? Is it automatically discovered?
- What is the database URL?

■ **R24.19** Where is the file `derby.jar` located in your installation of the Java development kit?

■■ **R24.20** Suppose you run the command

```
java -classpath derby.jar;. TestDB database.properties
```

as described in Section 24.3. Match the following five error messages to the error conditions they correspond to.

Error Message	Error Condition
`Usage: java [-options] class [args...]`	The suffix `; create=true` was missing from the `jdbc.url` entry in `database.properties`.
`Exception in thread "main" java.sql.SQLException: No suitable driver found for jdbc:derby:BigJavaDB;create=true`	The `database.properties` file was not present in the current directory.
`Exception in thread "main" java.lang.NoClassDefFoundError: TestDB`	`derby.jar` and `.` should have been separated by a colon on this operating system.
`Exception in thread "main" java.io.FileNotFoundException: database.properties`	The `TestDB.class` file was not present in the current directory.
`Exception in thread "main" java.sql.SQLException: Database 'BigJavaDB' not found.`	The `derby.jar` file was not present in the current directory.

- **R24.21** What is the difference between a `Connection` and a `Statement`?
- **R24.22** Of the SQL commands introduced in this chapter, which yield result sets, which yield an update count, and which yield neither?
- **R24.23** How is a `ResultSet` different from an `Iterator`?

PRACTICE EXERCISES

- **E24.1** Write a Java program that creates a Coin table with coin names and values; inserts coin types penny, nickel, dime, quarter, half dollar, and dollar; and prints out the sum of the coin values. Use SQL commands `CREATE TABLE`, `INSERT`, and `SELECT SUM`.
- **E24.2** Write a Java program that creates a Car table with car manufacturers, models, model years, and fuel efficiency ratings. Insert several cars. Print out the average fuel efficiency. Use SQL commands `CREATE TABLE`, `INSERT`, and `SELECT AVG`.
- ■■ **E24.3** Improve the `ExecSQL` program and make the columns of the output line up. *Hint:* Use the `getColumnDisplaySize` method of the `ResultSetMetaData` class.
- ■■ **E24.4** Modify the program in Section 24.5 so that the user has the choice of selecting an existing customer. Provide an option to search for the customer by name or zip code.
- ■■ **E24.5** Write a Java program that uses the database tables from the invoice database in Section 24.2. Prompt the user for an invoice number and print out the invoice, formatted as in Chapter 12.
- ■■ **E24.6** Write a Java program that uses the database tables from the invoice database in Section 24.2. Produce a report that lists all customers, their invoices, the amounts paid, and the unpaid balances.

PROGRAMMING PROJECTS

- ■■ **P24.1** Write a Java program that uses a library database of books and patron data, as described in Exercise R24.2. Patrons should be able to check out and return books. Supply commands to print the books that a patron has checked out and to find who has checked out a particular book. Create and populate Patron and Book tables before running the program.
- ■■ **P24.2** Write a Java program that creates a grade book for a class. Create and populate Student and Grade tables before running the program. The program should be able to display all grades for a given student. It should allow the instructor to add a new grade (such as "Homework 4: 100") or modify an existing grade.
- ■■■ **P24.3** Write a program that assigns seats on an airplane as described in Exercise P12.6. Keep the seating information in a database.
- ■■■ **P24.4** Write a program that keeps an appointment calendar in a database. An appointment includes a description, a date, the starting time, and the ending time; for example,

```
Dentist 2016/10/3 17:30 18:30
CS1 class 2016/10/4 08:30 10:00
```

Supply a user interface to add appointments, remove canceled appointments, and print out a list of appointments for a particular day.

■ **P24.5** Modify the ATM simulation program of Worked Example 24.1 so that the program pops up two ATM frames. Verify that the database can be accessed simultaneously by two users.

■■ **P24.6** Write a program that uses a database of quizzes. Each quiz has a description and one or more multiple-choice questions. Each question has one or more choices, one of which is the correct one. Use tables `Quiz`, `Question`, and `Choice`. The program should show the descriptions of all quizzes, allow the user to choose one, and present all questions in the chosen quiz. When the user has provided responses to all questions, show the score.

■■■ **P24.7** Enhance the program of Exercise P24.6 so that it stores the user's responses in the database. Add `Student` and `Response` tables. (User is a reserved word in SQL.)

■■ **P24.8** Write a program that uses the database of Exercise P24.7 and prints a report showing how all students performed on all quizzes.

ANSWERS TO SELF-CHECK QUESTIONS

1. The telephone number for each customer may not be unique—the same number might be shared by roommates. Even if the number were unique, however, it can change when a customer moves. In that situation, both the primary and all foreign keys would need to be updated. Therefore, a customer ID is a better choice.
2. Customer 3176 ordered ten toasters.
3.
```
SELECT Name
   FROM Customer
   WHERE State <> 'AK' AND State <> 'HI'
```
4.
```
SELECT Invoice.Invoice_Number
   FROM Invoice, Customer
   WHERE Invoice.Customer_Number
         = Customer.Customer_Number
      AND Customer.State = 'HI'
```
5. Connect to the database with a program that lets you execute SQL instructions. Try creating a small database table, adding a record, and selecting all records. Then drop the table again.
6. You didn't set the class path correctly. The JAR file containing the JDBC driver must be on the class path.
7. `result.next()`. If there is at least one result, then `next` returns `true`.
8.
```
ResultSet result = stat.executeQuery(
   "SELECT COUNT(*) FROM Customer "
   + "WHERE State = 'HI'");
result.next();
int count = result.getInt(1);
```
   Note that the following alternative is significantly slower if there are many such customers.
```
ResultSet result = stat.executeQuery(
   "SELECT * FROM Customer "
   + "WHERE State = 'HI'");
while (result.next()) { count++; } // Inefficient
```
9. In this program, error reporting is the responsibility of the `main` method.
10. By passing the customer number as an argument. Then the query would only involve the customer table.
11. The most convenient approach is to ask the user about the payment before entering the items. Then the payment can be added to the statement in the `addInvoice` method.
12. We can either compute it in the loop that displays the line items, or we can issue a query
```
SELECT SUM(Product.Price * LineItem.Quantity)
   FROM Product, LineItem
   WHERE Product.Product_Code
      = LineItem.Product_Code
   AND LineItem.Invoice_Number = ?
```