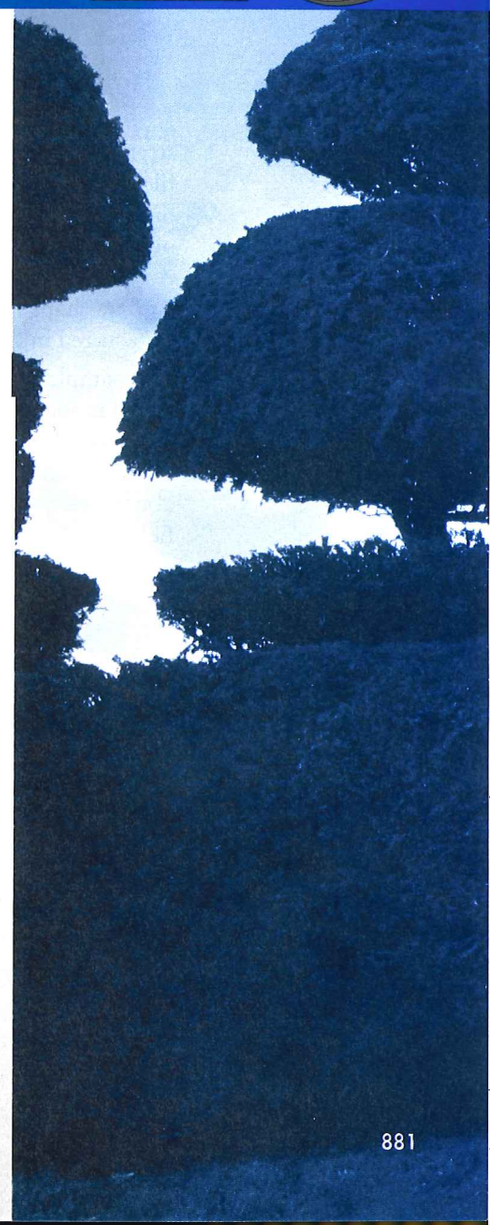
25 Databases



CHAPTER OBJECTIVES

- Understand the basic concept of a database as it relates to data storage.
- Explain the concepts behind relational databases.
- Examine how Java programs can connect to databases for the purposes of creating, reading, updating, and deleting data.
- Briefly introduce the syntax of several different types of SQL statements.

This chapter provides an introduction to databases and interacting with databases using Java. A database is a large repository of data organized for efficient storage and searching. Discussing databases and the way we interact with them through a Java program is a natural extension of the concept of collections that has been a major theme throughout this book.

25.1 Introduction to Databases

A *database* is a potentially large repository of data, organized so that the data can be quickly stored, searched, and organized in various ways. A *database management system* is software that provides the ability to quickly search or query the data contained within the database and generally perform four primary operations on the data: create, read, update, and delete (also known as CRUD). Applications such as a university's class scheduling system and an airline reservation system use a database to organize and manage large amounts of data.

KEY CONCEPT
Databases are software applications used to provide data to other programs.

There are many different types of databases, each with its own strengths and weaknesses. For example, there are object-oriented databases, flat-file databases, and relational databases. A comprehensive discussion of the various types of databases is beyond the scope of this text, but we will focus on the most commonly used type of database in existence today—a relational database. A relational database organizes its basic information into one or more *tables*. And, perhaps more important, the relationships among various data elements are also stored in tables.

A simple example of a relational database (or a database that uses the relational model) appears in the following two tables, which we shall call Person and Location. Let's look at each more closely. The Person table contains a series of rows, which are called records. Each record in the table contains the information about one person in our database. Each person's record provides a number of fields, including that person's first name, her or his last name, and two integer values: `personID` and `locationID`. The `personID` is a unique integer value (that is, unique within the Person table) used to identify a particular record. For example, Peter's `personalID` is 1. John's `personalID` is 2. Each record also contains a `locationID`. This value is used to find (or "look up") a particular matching record in the `Location` table; hence the name `locationID` that contains the table name as part of the field name.

Person

personID	firstName	LastName	locationID
0	Matthew	Williamson	0
1	Peter	DePasquale	0
2	John	Lewis	1
3	Jason	Smithson	2

Location

locationID	city	state
0	Portsmouth	RI
1	Blacksburg	VA
2	Maple Glen	PA
3	San Jose	CA

Thus, by using the `locationID` value in Peter's record in the Person table, and finding that value in the Location table, we can determine that Peter lives in Portsmouth, RI. Matthew Williamson lives there as well. But what do these tables really get us? There are several answers to that question. First of all, we pointed out that both Matt and Peter live in the same town in the same state. If we didn't have a `Location` table in use, the city and state strings would probably be a part of the `Person` table, so we would be using more space in our database to replicate the same values over and over. By using the Location table, we can conserve space by avoiding data replication.

There's another advantage to the use of tables in this manner. We can easily query our database (ask it a question) to determine which people live in Portsmouth, RI. Each record of each person that we have stored in the `Person` table will contain the value 0 in the `locationID` field. There's actually a more complex way of *joining* the tables to determine the answer to our query, but that's really outside the scope of this text.

KEY CONCEPT
A relational database creates relationships between records in different tables by using unique identifiers.

Our query can be quickly executed because we are using the strength of the relational database model. If we were replicating our data ("Portsmouth", "RI") over and over in several records of the Person table, searching for all residents of Portsmouth would be rather inefficient and time-consuming, because we'd be searching each record looking for two string values.

Tables in our database are related by the use of the `locationID`. A `locationID` value helps to relate a specific person record in the `Person` table to a specific record in the Location table. Additionally, as you may have noted, each record in the Location table has its own unique identifier (`locationID`). The use of these identifiers and the way we utilize them between tables enable us to establish the relationships between the records in our tables.

KEY CONCEPT
The JDBC API is used to establish a connection to a database.

For the purposes of this discussion, we will be using the open source database MySQL (see http://www.mysql.com to obtain a copy). There are other databases that we could have chosen, including Oracle, SQLServer, Access, and PostgreSQL. Some of these databases are open source and freely available; others can be purchased for a fee.

Before we can interact with a database in a Java program, we must first establish a connection to it. To do that, we'll use the Java Database Connectivity (JDBC) API. The JDBC API provides us the classes and methods required to manage our data from within the Java program. Fortunately, the JDBC API has been a component of the Java Development Environment (JDK) since JDK 1.1, so we won't have to download any additional software to obtain the API functionality. However, to connect to our database, we will need a database-specific driver, discussed in the next section.

25.2 Establishing a Connection to a Database

In order to establish communications to our database, we'll need a specialized piece of software that communicates our database requests to the database application (known as the server, which generally resides on another machine). This software is known as a *driver*. The response from the database is communicated back to our program via the driver, as well.

Obtaining a Database Driver

There are over two hundred JDBC drivers available for various databases. To find one for your system, check Java's JDBC Data Access API web page at developers.sun.com/product/jdbc/drivers. We'll be using the MySQL connector driver (www.mysql.com/products/connector/) to provide our connection to a database hosted on another computer. We downloaded our connector in jar file form from the MySQL website and placed in it a directed named "connector."

Once installed, we need only refer to its location via the CLASSPATH environment variable (during compilation and execution). Depending on your configuration, you can save these values in a shell configuration or just provide them on the command line. As we demonstrate the coding steps necessary to create and issue queries, and obtain and process responses from the server, we'll also show you how to include the driver into the CLASSPATH.

In Listing 25.1, we have a simple program that includes the use of several JDBC-specific classes. The code demonstrates the loading of our JDBC driver, attempting to establish a connection to our database server, and closing the connection once we have confirmed that the connection was opened.

The program starts by attempting to load our JDBC driver (`com.mysql.jdbc.Driver`) from our downloaded jar file. When our driver is loaded, it will create an instance of itself and register itself with the `DriverManager` (`java.sql.DriverManager`) class.

Next, we attempt to establish a connection to our database through the `DriverManager` class. The `DriverManager` will attempt to select an appropriate driver from among the set of drivers registered with the `DriverManager`. In our case, this will be easy; we're using only one driver for this program, and it is the only possible driver that can be selected. The call to the `getConnection` method of the `DriverManager` accepts a URL that defines our database instance. It comprises several components that we must provide, including the hostname (the name of the machine where our database server is residing), the name of the database, and the username and password that will provide us access to the selected database.

LISTING 25.1

```
import java.sql.*;

/**
 * Demonstrates the establishment of a JDBC connector.
 *
 * @author Java Foundations
 * @version 4.0
 */
public class DatabaseConnector
{
   /**
   * Establishes the connection to the database and prints an
   * appropriate confirmation message.
   */
    public static void main (String args[])
    {
       try
       {
          Connection conn = null;

          //  Loads the class object for the mysql driver into the DriverManager.

          Class.forName("com.mysql.jdbc.Driver");

          //  Attempt to establish a connection to the specified database via the
          //  DriverManager

          conn = DriverManager.getConnection("jdbc:mysql://comtor.org/"+
             "javafoundations?user=jf2e&password=hirsch");

          if (conn != null)
          {
             System.out.println("We have connected to our database!");
             conn.close();
          }

       }  catch (SQLException ex) {
          System.out.println("SQLException: " + ex.getMessage());
          ex.printStackTrace();
    }     catch (Exception ex) {
```

LISTING 25.1 *continued*

```
            System.out.println("Exception: " + ex.getMessage());
            ex.printStackTrace();
        }
    }
}
```

If all goes well, and there are no problems (such as the database server not running, or a communications issue between our machine where our Java program is executing and the host on which the database server is executing), we will be returned a `Connection` (`java.sql.Connection`) object. This object represents a single connection to our database and will be the conduit for our queries to, and responses from, the database. Finally, our program checks to determine whether we did in fact receive a non-null `Connection` object, and if so, it prints a success message to the user and then closes the connection to the database.

In order to execute a program called `Example1`, the CLASSPATH will need to reflect the location of the driver. It is not necessary for it to do so when we compile the `Example1` program, because we need only the `Driver` class from the JDBC jar file at run-time when the program attempts to load and register the `Driver` object. Working on the UNIX command line, after successful compilation, we can execute the program with the following command:

```
$ java -cp .:../connector/mysql-connector-java-5.1.7-bin.jar
Example1
```

Under Windows, the command will be

```
java -cp .;..\connector\mysql-connector-java-5.1.7-bin.jar
Example1
```

The resulting output is

```
We have connected to our database!
```

This assumes that the download location of our JDBC jar file is the adjacent connector directory from the current directory. We could have placed our jar file in the same directory where the source code is located, but we prefer not to mix our source code and jar files, so they are placed in the connector directory.

We'll do more with our database in the next section, including creating and altering tables in our database.

25.3 Creating and Altering Database Tables

In the previous section, we showed how to establish a connection to our database, check that the connection was successful, and disconnect. Let's expand on those abilities and create a new database table.

Create Table

The start of the SQL statement to create a database table is `CREATE TABLE <tablename>`. We will also need to specify the name of the table and any fields that will be present at creation. For example, let's create a new table of students. For now, this table will contain an ID value (known as a key) and the students' first and last names. To do so, our creation command will be the following string:

KEY CONCEPT
The `CREATE TABLE` SQL statement is used to create new database tables.

```
CREATE TABLE Student (student_ID INT UNSIGNED NOT NULL
AUTO_INCREMENT, PRIMARY KEY (student_ID), firstName
varchar(255), lastName varchar(255))
```

Each field of the table is specified in the comma-separated list in the parentheses that follow the table name (`Student`). Our table will initially include the following fields:

- `student_ID`—an unsigned integer value, which cannot be null and will automatically increment each time we add a new student to the table.
- `firstName`—a variable-length character string up to 255 characters in length.
- `lastName`—a variable-length character string up to 255 characters in length.

You may be asking yourself about the `PRIMARY KEY` field that we skipped mentioning. It isn't really a field but a setting on the table itself that specifies which field or fields make the record unique among the other records. Because each student will have a unique identifier, the `student_ID` field will be sufficient to be our table's key. Again, this is a database topic that you really need to study in the database course, so we will leave it at that for now.

To create the table in our existing code from Listing 1, we'll add the following lines immediately after the successful connection message.

```
Statement stmt = conn.createStatement();
boolean result = stmt.execute("CREATE TABLE Student " +
  " (student_ID INT UNSIGNED NOT NULL AUTO_INCREMENT, " +
  " PRIMARY KEY (student_ID), firstName varchar(255), " +
  " lastName varchar(255))");
System.out.print("\tTable creation result: " + result + "\t");
System.out.println("(false is the expected result)");
```

The `Statement` class (`java.sql.Statement`) is an interface class, which we will use to prepare and execute our SQL statement(s). We can ask our `Connection` object to create our `Statement` object for us. Once we have our `Statement` object, we can call its `execute` method and pass it the SQL query string (our `CREATE TABLE` string) for execution by the database. The `execute` method returns a `TRUE` value if there is a `ResultSet` object (we'll talk about this shortly) returned after the call, or returns a false otherwise.

When we execute our code, we anticipate that we will be returned a `FALSE` value and the table will be created. In fact, that is what happens (see the output below). However, note that if an attempt to execute the same program is again made, the program will throw an exception stating that the table already exists. That's fine for now. We'll discuss removal of tables later in this chapter.

```
We have connected to our database!
  Table creation result: false (false is the expected result)
```

Alter Table

So far, so good! Our new database has a new table called `Student`. Now let's add a few fields in our new table. Often the structure of a table will be established when it is created and will not be changed subsequently, but from time to time the need to alter an existing table arises. We could drop (delete) the existing table and create a new one from scratch. But if we do, we'll lose the data currently stored in the table. It's usually preferable to add new fields or remove existing fields (and their data) from an existing table.

KEY CONCEPT

The `ALTER TABLE` SQL statement can be used to modify an existing database table.

Let's add `age` and `gpa` fields for our students. For the `age` field, we can use the smallest unsigned integer field possible (conserving the most amount of space). This is an unsigned `tinyint` for MySQL (be sure to check the datatype listing for your database); it will range from 0 to 255, sufficient for an `age` field. We'll also need an additional column for our `gpa` field. Here we will use an unsigned `float` that uses three digits total, and two digits for the fractional part.

In order to modify our table and add the missing fields, we need to create a new `Statement` object and use the `ALTER TABLE <tablename> ADD COLUMN` SQL statement. Following the `ADD COLUMN` portion of the string, we specify our new fields in a parenthetical list, separated by commas. Once the string is constructed, we call the execute method of the `Statement` object to execute the query. Again here, we expect a `FALSE` value to be returned, indicating that no result set is returned to us.


```
Statement stmt2 = conn.createStatement();
result = stmt2.execute("ALTER TABLE Student ADD COLUMN " +
  " (age tinyint UNSIGNED, gpa FLOAT (3,2) unsigned)");

System.out.print("\tTable modification result: " + result + "\t");
System.out.println("(false is the expected result)");
```

Our output is not surprising:

```
We have connected to our database!
  Table creation result: false (false is the expected result)
  Table modification result: false (false is the expected result)
```

Drop Column

Of course, we may also wish to alter a table by dropping a column, rather than adding one or more. To do so, we can use the `ALTER TABLE` Sql statement and follow the table name with the `DROP COLUMN` command. `DROP COLUMN` is followed by one or more column names, separated by commas. For example, if we wanted to drop the `firstName` column of our `Student` table, we could use the following statements:

```
Statement stmt3 = conn.createStatement();
result = stmt3.execute("ALTER TABLE Student DROP COLUMN first-
Name");

System.out.print("\tTable modification result: " + result + "\t");
System.out.println("(false is the expected result)");
```

Again, our output is fairly straightforward:

```
We have connected to our database!
  Table creation result: false (false is the expected result)
  Table modification result: false (false is the expected result)
  Table modification result: false (false is the expected result)
```

But how do we know this is really modifying the table? What we really want to be able to do is ask the database to tell us what the structure of our table is at any point in time. We'll discuss that in the next section.

25.4 Querying the Database

At this point, we have a single table in our database with no data. One of the activities we would like to do at this point is query the database regarding the structure of our table. To do this, we will build another `Statement` and send it to

the database server for processing. However, the difference here, compared to our earlier examples, is that we expect to be returned a `ResultSet` object—an object that manages a set of records that contains our result.

How to obtain and use a `ResultSet` is an important piece of the knowledge we will need to master in order to obtain information from our database. A `ResultSet` functions in much the same way a `Scanner` object does; it provides a method of accessing and traversing through a set of data (in this case, data obtained from our database). A default `ResultSet` object permits moving through the data from the first object to the last object in the set, and it cannot be updated. Other variations of the `ResultSet` object can permit bidirectional movement and the ability to be updated.

Show Columns

The simplest example we can provide of using a `ResultSet` is querying the database about the structure of our table. In the following sections of this chapter, we will use the `ResultSet` in more complex ways. In the code below, we form our query and submit it to the server. Apart from the syntax of the `SHOW COLUMNS <tablename>` statement, the remainder of the code is very straightforward and similar to our previous examples. Note that we are expecting a `ResultSet` object to be returned from our execution of the query.

KEY CONCEPT

The `SHOW COLUMNS` SQL statement can be used to obtain a list of a table's columns and configuration settings.

```
Statement stmt5 = conn.createStatement();
ResultSet rSet = stmt5.executeQuery("SHOW COLUMNS FROM Student");
```

Once we have our results returned from the query, we can obtain some information from the `ResultSet`'s `ResultSetMetaData` object (which contains "meta" information about the results returned). For instance, in the next two lines of code, we use information from the `ResultSetMetaData` object to produce output for the user (including the table name and the number of columns in the result).

```
ResultSetMetaData rsmd = rSet.getMetaData();
int numColumns = rsmd.getColumnCount();
```

Our `ResultSet` is essentially a two-dimensional table that contains columns (our fields) and rows of data (records in the result). By utilizing the metadata regarding the number of columns in the `ResultSet`, we can print out some basic information about the structure of our `Student` table.

```
String resultString = null;
if (numColumns > 0)
{
  resultString = "Table: Student\n" +
    "=======================================================" +
    "=======================================================\n\t";
  for (int colNum = 1; colNum <= numColumns; colNum++)
    resultString += rsmd.getColumnLabel(colNum) + '\t';
}
System.out.println(resultString);
System.out.println(
  "=======================================================" +
  "=======================================================\n\t");
```


The program will now print out a table of our column headers in the `ResultSet`, which is really a list of property names for each field in our table.

```
We have connected to our database!
  Table creation result: false (false is the expected result)
  Table modification result: false (false is the expected result)
  Table modification result: false (false is the expected result)
Table: Student
=======================================================
  Field Type Null Key Default Extra
=======================================================
```

Let's get the rows of data as well from the `ResultSet` so that we can see the full structure of our table. We need to add additional statements to our program that will iterate through the `ResultSet` rows and obtain the value of each column as a string.

```
We have connected to our database!
  Table creation result: false (false is the expected result)
  Table modification result: false (false is the expected result)
  Table modification result: false (false is the expected result)
Table: Student
=======================================================
Field   Type   Null   Key   Default    Extra
=======================================================
student_ID   int(10) unsigned   NO   PRI   auto_increment
-------------------------------------------------------
lastName   varchar(255)   YES
-------------------------------------------------------
age   tinyint(3) unsigned   YES
-------------------------------------------------------
gpa   float(3,2) unsigned   YES
-------------------------------------------------------
```

It's not very pretty, but we can now see the basic configuration information for each field in our table. Each record from the `ResultSet` contains the field name, the field (data) type, whether the field can hold a null value, whether the

field is part of a key, the default value for the field (if not provided), and any extra information.

Let's turn this printed table into something easier to read. With a little more work, we could have attempted to output the table structure in a format like the one shown in the following table. This is left up to you as a string formatting exercise.

Field	Type	Null	Key	Default	Extra
student_ID	int(10) unsigned	NO	PRI	auto_increment	
lastName	varchar(255)	YES			
age	tinyint(3) unsigned	YES			
gpa	float(3, 2) unsigned	YES			

We used our `Statement` object as the vehicle to obtain information from our database. In this case, the information we sought was the structure of the `Student` table. However, we can change our `Statement` string and query the database quite easily.

In terms of outputting the results, we'll do the same iteration through the `ResultSet`. However, there's one problem. Our `Student` table does not yet have any data in it. We'll address that problem in the next section and then show how to query the table.

25.5 Inserting, Viewing, and Updating Data

A table without data is like a day without sunshine. It's time to put some real data into our `Student` table. Since the `studentID` field is an `auto_increment` field (it will automatically increment by 1 each time data are placed into the table), we really only needed to insert data for student `lastnames`, `age`, and `gpa`. Let's add the following values into our database.

lastName	age	gpa
Campbell	19	3.79
Garcia	28	2.37
Fuller	19	3.18
Cooper	26	2.13
Walker	27	2.14
Griego	31	2.10

Insert

To insert the data, we need to create a new `Statement` and use the `INSERT <tablename>` SQL statement. The `INSERT` statement takes the form

```
INSERT <tablename> (column name, ...) VALUES (expression, ...)
```

Following the `tablename`, one or more columns are specified by name. Then the values to be placed in the specified fields, respectively, are listed in the same order in which the columns are specified. For example, to insert the Campbell row into the database, we will create a SQL statement that looks like this:

KEY CONCEPT
The `INSERT SQL` statement is used to add new data to a database table.

```
INSERT Student (lastName, age, gpa) VALUES ("Campbell", 19, 3.79)
```

We can use the following two lines of source code to perform the insertion on the table.

```
Statement stmt2 = conn.createStatement(ResultSet.TYPE_FORWARD_ONLY,
  ResultSet.CONCUR_UPDATABLE);
int rowCount = stmt2.executeUpdate("INSERT Student " +
  "(lastName, age, gpa) VALUES (\"Campbell\", 19, 3.79)");
```

In order for the update of the table's data values to take place, we need to make two method calls different from those we have done previously. First, when we construct the `Statement` object (via the `createStatement` method), we specify that the resulting `ResultSet`'s pointer may move only forward and that changes to the `ResultSet` are passed on to the database.

Second, we called the `executeUpdate` method, rather than the `execute` or `executeQuery` methods used earlier. The `executeUpdate` method returns the number of rows affected by the update query; in this case, only 1 row was modified. Similar `executeQuery` statements can follow, enabling us to insert each row of students shown in Table 1. Alternatively, we could read our data from an input file and use a loop to process the data from the input file into the database.

It's been a while since we have seen the full body of code of our program. Along the way we have made a number of additions and changes. Listing 25.2 on the next page updates us on the source code we're using in discussing our database connections.

SELECT . . . FROM

One of the actions we perform most frequently on a database is issuing queries to view to retrieve the data. The `SELECT . . . FROM` SQL statement permits users to construct a request for data based on a number of criteria. The basic syntax of the `SELECT . . . FROM` statement is as follows:

KEY CONCEPT
The `SELECT` SQL statement is used to retrieve data from a database table.

LISTING 25.2

```
import java.sql.*;

/**
 * Demonstrates interaction between a Java program and a database.
 *
 * @author Java Foundations
 * @version 4.0
 */
public class DatabaseModification
{
   /**
   * Carries out various CRUD operations after establishing the
   * database connection.
   */
   public static void main (String args[])
   {
      Connection conn = null;
      try
      {

         //  Loads the class object for the mysql driver into the DriverManager.
         Class.forName("com.mysql.jdbc.Driver");

         //  Attempt to establish a connection to the specified database via the
         //  DriverManager

         conn = DriverManager.getConnection("jdbc:mysql://comtor.org/" +
           "javafoundations?user=jf2e&password=hirsch");

      // Check the connection

      if (conn != null)
      {
         System.out.println("We have connected to our database!");

          //  Create the table and show the table structure

          Statement stmt = conn.createStatement();
          boolean result = stmt.execute("CREATE TABLE Student " +
            " (student_ID INT UNSIGNED NOT NULL AUTO_INCREMENT, " +
            " PRIMARY KEY (student_ID), lastName varchar(255), " +
            " age tinyint UNSIGNED, gpa FLOAT (3,2) unsigned)");
```


LISTING 25.2 *continued*

```
          System.out.println("\tTable creation result: " + result);
          DatabaseModification.showColumns(conn);

          //  Insert the data into the database and show the values in the table

          Statement stmt2 = conn.createStatement(ResultSet.TYPE_FORWARD_ONLY,
            ResultSet.CONCUR_UPDATABLE);
          int rowCount = stmt2.executeUpdate("INSERT Student " +
            "(lastName, age, gpa) VALUES (\"Campbell\", 19, 3.79)");
          DatabaseModification.showValues(conn);

          //  Close the database

          conn.close();
       }

      }  catch (SQLException ex) {
         System.out.println("SQLException: " + ex.getMessage());
         ex.printStackTrace();

      }  catch (Exception ex) {
         System.out.println("Exception: " + ex.getMessage());
         ex.printStackTrace();
      }
   }

/**
 * Obtains and displays a ResultSet from the Student table.
 */
public static void showValues(Connection conn)
{
   try
   {
      Statement stmt = conn.createStatement();
      ResultSet rset = stmt.executeQuery("SELECT * FROM Student");
      DatabaseModification.showResults("Student", rset);
   }  catch (SQLException ex) {
      System.out.println("SQLException: " + ex.getMessage());
      ex.printStackTrace();
   }
}

  /**
   * Displays the structure of the Student table.
   */
```

LISTING 25.2 *continued*

```
public static void showColumns(Connection conn)
{
   try
   {
      Statement stmt = conn.createStatement();
      ResultSet rset = stmt.executeQuery("SHOW COLUMNS FROM Student");
      DatabaseModification.showResults("Student", rset);
   }  catch (SQLException ex) {
      System.out.println("SQLException: " + ex.getMessage());
      ex.printStackTrace();
   }
}

/**
* Displays the contents of the specified ResultSet.
*/
public static void showResults(String tableName, ResultSet rSet)
{
   try
   {
      ResultSetMetaData rsmd = rSet.getMetaData();
      int numColumns = rsmd.getColumnCount();
      String resultString = null;
      if (numColumns > 0)
      {
         resultString = "\nTable: " + tableName + "\n" +
           "=======================================================\n";
         for (int colNum = 1; colNum <= numColumns; colNum++)
           resultString += rsmd.getColumnLabel(colNum) + " ";
      }
      System.out.println(resultString);
      System.out.println(
        "=======================================================");

      while (rSet.next())
      {
        resultString = "";
        for (int colNum = 1; colNum <= numColumns; colNum++)
        {
           String column = rSet.getString(colNum);
           if (column != null)
             resultString += column + " ";
        }
        System.out.println(resultString + '\n' +
        "-------------------------------------------------------------");
   }
```


LISTING 25.2 *continued*

```
      }  catch (SQLException ex) {
         System.out.println("SQLException: " + ex.getMessage());
         ex.printStackTrace();
      }
   }
}
```

```
SELECT <columns, ...> FROM <tablename> WHERE <condition, ...>
```

The `SELECT` statement has many parts (most of which are beyond the scope of this text). We will talk about just a few to get you started using the statement.

The `SELECT` statement allows you to ask for only certain columns to be returned following the query. For example, if we wanted only a list of student `lastNames` and `gpas` from our `Student` table, we would construct our query as follows:

```
SELECT lastName, gpa FROM Student
```

If we wanted to make our query even more specific, and get the `lastNames` and `gpas` only of students whose `age` is 21 and over, our statement would look like this:

```
SELECT lastName, gpa FROM Student WHERE age >= 21
```

The `WHERE condition` clause is optional, and if it is not provided, all rows from the specified table will be selected. A condition is an expression that must evaluate to `TRUE` and can contain functions and operators such as `&&` (and), `||` (or), and many others. For example, we can further refine our query to limit results to those students who are 21 years old or older and who have a grade point average of 3.0 or less:

```
SELECT lastName, gpa FROM Student WHERE age >= 21 && gpa <= 3.0
```

The selected items to return (`lastName` and `gpa` in our examples above) are specified by listing them with a comma between them. However, if you wish to have all of the columns from a table returned, you can substitute an asterisk (`*`) for the column listings:

```
SELECT * FROM Student WHERE age >= 21 && gpa <= 3.0
```

As like our INSERT statement, the SELECT statement will be a parameter to an executeQuery method call. The output from the execution of this statement would look something like

```
Table: Student
=======================================================
student_ID   lastName   age   gpa
=======================================================
2   Jones   22   2.40
-------------------------------------------------------
```

There are a number of other capabilities that the SELECT SQL statement provides, including the ability to join two or more tables, to limit our output to a set number of rows, and to group our results by one or more columns. Often, the most useful additional clause is ORDER BY <columnname> <direction>.

The ORDER BY clause, appended to the statement following the where clause, will direct the statement to sort our results in an ascending or descending fashion. To specify ascending, place the string ASC in the direction portion; to specify descending, use DESC. For example, the query

```
SELECT * FROM Student ORDER BY gpa DESC
```

produces the following results:

```
Table: Student
=======================================================
lastName   age   gpa
=======================================================
Hampton   31   3.88
-------------------------------------------------------
Campbell   19   3.79
-------------------------------------------------------
Smith   21   3.69
-------------------------------------------------------
Jones   22   2.40
-------------------------------------------------------
```

Update

Updating data in a database is another frequently used skill. Fortunately, it is a rather simple process. We can break the process down into three steps. First, we obtain a ResultSet and navigate to the row we wish to update. Second, we update the ResultSet's value that we wish to change. Finally, we update the database with the record from the ResultSet.

Let's look at these steps in more detail. We first want to obtain a ResultSet on which we will operate. The type of change you wish to make to the database

(updating one row vs. updating multiple rows) will likely direct the ResultSet you obtain to operate on the data. Generally speaking, it's a good idea to limit your ResultSet to the database rows you wish to modify. Thus, if you are going to change only one row's values, your ResultSet should really comprise only that row.

KEY CONCEPT

Updates to a database can be performed through changes to a ResultSet.

Once we have our ResultSet, we need to navigate the ResultSet cursor (a pointer into the set). You may have noted from Listing 25.2 that we iterated through the ResultSet by repeatedly calling the next method on the ResultSet object. We can actually navigate via a number of methods (first, last, next, previous, and so on). Because the cursor is placed prior to the first row in a ResultSet, we simply used the next method to move forward each time we attempted to read a row of data.

If we are modifying only one row of data and our ResultSet contains only one row, we can just jump to the first row by executing a call to our set's first method. Then, we can modify our data by using the updateXXX methods (updateString, updateFloat, and so on) on our set. We make our changes permanent in the database by calling the updateRow method. Here is an example:

```
ResultSet rst = stmt2.executeQuery("SELECT * FROM Student
WHERE " +
  "lastName=\"Jones\"");
rst.first();
rst.updateFloat("gpa", 3.41f);
rst.updateRow();
```

Obviously, if we are attempting to update multiple rows of records from a ResultSet, the navigation and updating statements will be more involved, possibly utilizing a loop. Additionally, prior to inserting, we can update any number of values in the row or rows in the ResultSet. The updateRow call simply communicates the changes back to the database once the ResultSet contains all of the necessary changes.

25.6 Deleting Data and Database Tables

The last piece of our JDBC knowledge is the ability to delete data and tables from the database. We'll first look at the deleting of data.

Deleting Data

The SQL statement that deletes data from a table is the DELETE FROM statement. This statement has the syntax

```
DELETE FROM <tablename> WHERE condition
```

KEY CONCEPT
The `DELETE FROM` SQL statement is used to delete data from a database table.

The `WHERE condition` clause is optional, and if it is not provided, all rows from the specified table will be deleted. A condition is an expression that must evaluate to `TRUE` and can contain functions and operators such as `&&` (and), `||` (or), and many others. For example, if we wished to delete all of the students whose `age` is greater than or equal to 30, we would use the following SQL statement.

```
DELETE FROM Student WHERE age >= 30
```

If we wished to delete all students whose `age` is greater than or equal to 30 and whose `gpa` is less than 3.5, we would use

```
DELETE FROM Student WHERE age >=30 && gpa < 3.5
```

As with the `INSERT` statement in the previous section, we will need to use a `Statement` that produces `RecordSet` objects that are forward-scrolling only (`ResultSet.TYPE_FORWARD_ONLY`), and that update the database (`ResultSet.CONCUR_UPDATABLE`).

Deleting Database Tables

Deleting tables from our database is rather simple to accomplish. We use the `executeUpdate` method of a `Statement` object and pass it the `DROP TABLE <tablename>` SQL statement.

KEY CONCEPT
The `DROP TABLE` SQL statement is used to delete data from a database table.

For example, if we wanted to drop the `Student` table, we could do so using the following Java statement in our program.

```
int rowCount = stmt.executeUpdate("DROP TABLE
Student");
```

Keep in mind that when the table is dropped, any data stored within the table are dropped as well.

Summary of Key Concepts

- Databases are software applications used to provide data to other programs.
- A relational database creates relationships between records in different tables by using unique identifiers.
- The JDBC API is used to establish a connection to a database.
- The `CREATE TABLE` SQL statement is used to create new database tables.
- The `ALTER TABLE` SQL statement can be used to modify an existing database table.
- The `SHOW COLUMNS` SQL statement can be used to obtain a list of a table's columns and configuration settings.
- The `INSERT` SQL statement is used to add new data to a database table.
- The `SELECT` SQL statement is used to retrieve data from a database table.
- The `DELETE FROM` SQL statement is used to delete data from a database table.
- The `DROP TABLE` SQL statement is used to delete an entire database table.
- Updates to a database can be performed through changes to a `ResultSet`.

Self-Review Questions

SR 25.1 What are the four primary operations on database data?

SR 25.2 In relational databases, where are the relationships stored, and how are they stored?

SR 25.3 Name two popular database products on the market today.

SR 25.4 Where can one obtain the JDBC?

SR 25.5 What role does a database driver play?

SR 25.6 What does the `java.sql.DriverManager` class assist with?

SR 25.7 Which class do we use to prepare and execute SQL statements?

SR 25.8 Which JDBC class is used to manage a set of records that is usually the result of a database query?

SR 25.9 Which SQL statement is used to add new data to a database table?

SR 25.10 Which SQL statement is used to remove a database table from the database?

Exercises

EX 25.1 Design a table for storing the names and contact information (addresses, phone numbers, email addresses) of your closest friends and family members. What fields would you use?

EX 25.2 Design one or more tables for managing a list of courses run by your university. How many tables do you need? What fields are in each table? Be sure to include such data as the instructor's names, the number of credits awarded for successful completion of the course, the department that offers the course, and the current enrollment in the course.

EX 25.3 Research the SQL statements needed to perform the CRUD operations on an Oracle database, on an Access database, and on a PostgreSQL database. How do they differ from those presented in this chapter?

EX 25.4 Using the MySQL documentation (http://dev.mysql.com/doc), determine how to create a temporary table and what the implications of using a temporary table are.

EX 25.5 How would you go about modifying a table and adding a new column in front of an existing one?

EX 25.6 What is the SQL statement needed to add a new column named `employeeNumber` to a table named `Employees`? Assume that the `employeeNumber` is a 7-digit integer value.

EX 25.7 What is the SQL statement needed to delete a column named `ProductCode` from a table named `Products`?

EX 25.8 Given the `Person` and `Location` tables provided at the beginning of this chapter, indicate what SQL statement is needed to return a query that lists all of the states that any person resides in.

EX 25.9 What SQL statement is needed to insert, in the `Student` table discussed earlier in this chapter, a new field that will store the total number of credits each student has accumulated? What data type should be used, and why?

EX 25.10 What SQL statement is needed to delete the age column from the `Student` table discussed earlier in this chapter?

Programming Projects

Use the MySQL world database (http://dev.mysql.com/doc/world-setup/en/world-setup.html) to populate a database and complete Programming Projects 25.1 through 25.5.

PP 25.1 Write a program to query the world database to obtain a list of all the cities of the world that contain a population that exceeds 5 million residents.

PP 25.2 Write a Java program that queries the world database to determine the total population of all cities in New Jersey.

PP 25.3 Write a program that queries the world database to determine in which country the residents have the greatest life expectancy.

PP 25.4 Research how to accomplish a `JOIN` on two tables (see http://en.wikipedia.org/wiki/Join_(SQL)). Then write a program that queries the world database to list the population of the capitol city for any country in Asia.

PP 25.5 Write a program with a graphical user interface that connects to a database, creates a CDs table (if it does not already exist), and provides the user the ability to manage a music album database. The database should include fields for the album title, artist name, number of tracks, and price. The user should be able to enter new album information, delete existing albums, and obtain a list of albums in the database.

PP 25.6 Using the free tool at http://www.fakenamegenerator.com, create 1,000 fake identifications to a text file. Then write a Java program that will populate a database table with the fake names from the text file. Finally, print a list of individuals in the database whose first (given) name is John and who live in Tennessee.

PP 25.7 Write a program that connects to a database, creates a Movies table (if it does not already exist), and provides the user the ability to manage a movie listing database. The database should include fields for a movie's name, run-time, rating code, date of release, and two primary stars. The user should be able to enter new movie information, delete existing movies, and obtain a list of movies in the database.

PP 25.8 Write a program with a graphical user interface that prompts the user to enter a database hostname, username, password, and database name. Once connected, the program should allow the user to browse the database by selecting a table and then choose to display either the structure of the table (`SHOW COLUMNS`) or the data contained within the selected table.

PP 25.9 Write a program that connects to a database, creates two tables (`Teams` and `Games`, if they don't already exist), and allows the user to manage a series of games between different opponents.

`Games` includes a score value for each team, a unique identifier for each game. `Teams` includes identifiers for each team, as well as their names and overall win/loss records. This problem can be solved with or without the knowledge of table joins in SQL statements. The user should be able enter new game results, add new teams, and navigate through each table or read the data.

PP 25.10 Write program that connects to a database, creates and populates the `Student` table, prints a display of the data contained in the `Student` table, and then deletes the `Student` table along with its data.

Answers to Self-Review Questions

SRA 25.1 The four primary database operations are create, read, update, and delete.

SRA 25.2 Relational databases store their relationships in the database tables themselves in the form of unique identifiers that are used to join the database tables.

SRA 25.3 Some popular database products are MySQL, PostgreSQL, SQLServer, Oracle, and Access.

SRA 25.4 As of JDK 1.1, the JDBC API is part of the JDK.

SRA 25.5 A database driver helps to establish communications from our JDBC statements to the specific database we are attempting to communicate with.

SRA 25.6 The `java.sql.DriverManager` class registers and manages each of the database drivers loaded at run-time.

SRA 25.7 The `java.sql.Statement` is used to prepare and execute SQL statements.

SRA 25.8 The `ResultSet` object is a JDBC class that is used to manage a set of database query result records.

SRA 25.9 The `INSERT` SQL statement is used to add new data to a database table.

SRA 25.10 The `DROP TABLE` SQL statement is used to delete a database table and its data.

Appendix A Glossary

abstract—A Java reserved word that serves as a modifier for classes, interfaces, and methods. An `abstract` class cannot be instantiated and is used to specify bodiless abstract methods that are given definitions by derived classes. Interfaces are inherently abstract.

abstract class—*See* abstract.

abstract data type (ADT)—A collection of data and the operations that are defined on those data. An abstract data type might be implemented in a variety of ways, but the interface operations are consistent.

abstract method—*See* abstract.

Abstract Windowing Toolkit (AWT)—The package in the Java API (`java.awt`) that contains classes related to graphics and GUIs. *See also* Swing.

abstraction—The concept of hiding details. If the right details are hidden at the right times, abstraction can significantly help control complexity and focus attention on appropriate issues.

access—The ability to reference a variable or invoke a method from outside the class in which it is declared. Controlled by the visibility modifier used to declare the variable or method. Also called the level of encapsulation. *See also* visibility modifier.

access modifier—*See* visibility modifier.

actual parameter—The value passed to a method as a parameter. *See also* formal parameter.

adaptor class—*See* listener adaptor class.