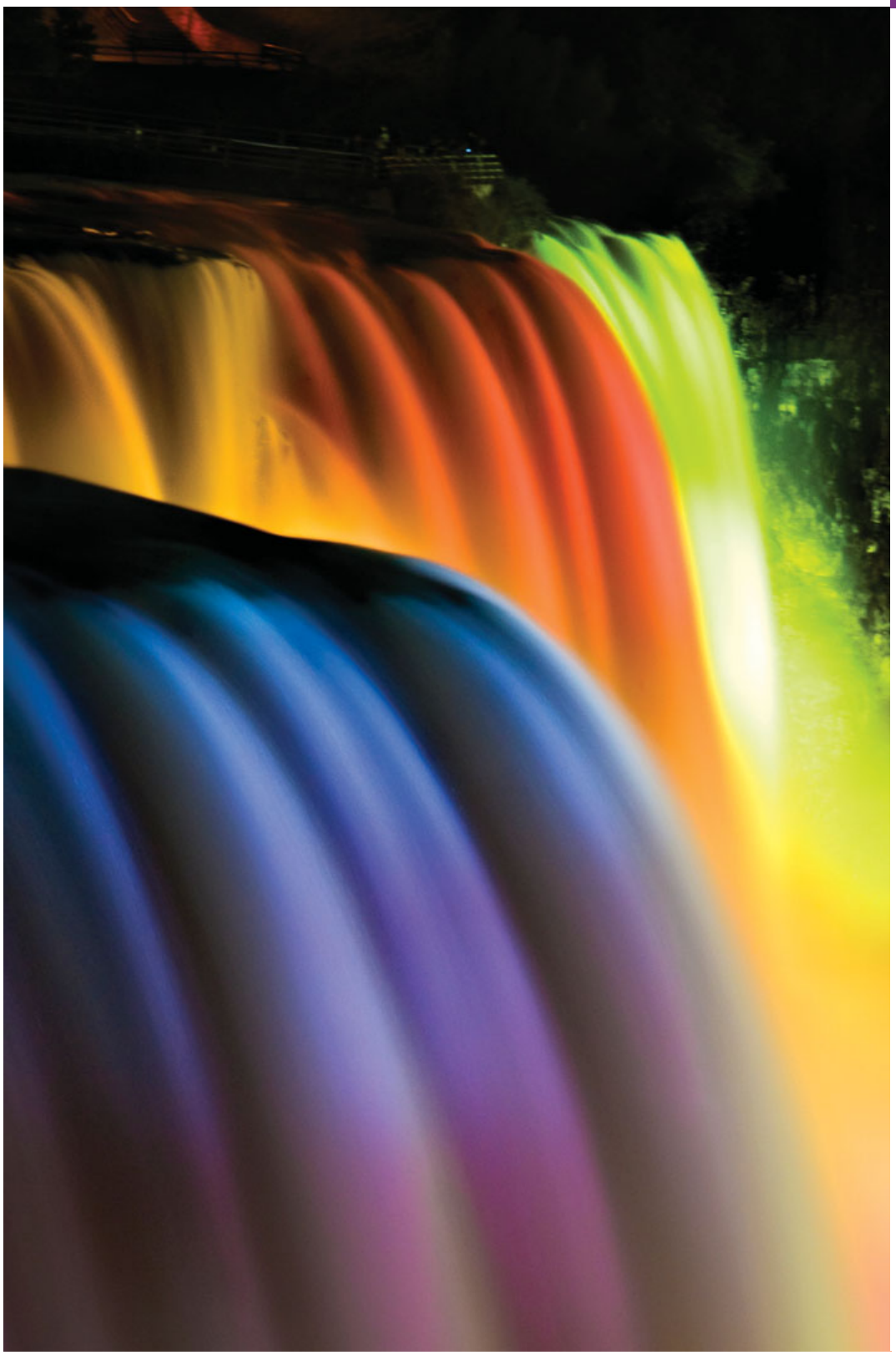
28

Accessing Databases with JDBC



It is a capital mistake to theorize before one has data.
—Arthur Conan Doyle

Now go, write it before them in a table, and note it in a book, that it may be for the time to come for ever and ever.
—The Holy Bible, Isaiah 30:8

Get your facts first, and then you can distort them as much as you please.
—Mark Twain

I like two kinds of men: domestic and foreign.
—Mae West

Objectives

In this chapter you'll learn:

- Relational database concepts.
- To use Structured Query Language (SQL) to retrieve data from and manipulate data in a database.
- To use the JDBC™ API to access databases.
- To use the `RowSet` interface from package `javax.sql` to manipulate databases.
- To use JDBC 4's automatic JDBC driver discovery.
- To create precompiled SQL statements with parameters via `PreparedStatement`s.
- How transaction processing makes database applications more robust.

Outline

28.1 Introduction[1]

A **database** is an organized collection of data. There are many different strategies for organizing data to facilitate easy access and manipulation. A **database management system** (**DBMS**) provides mechanisms for storing, organizing, retrieving and modifying data for many users. Database management systems allow for the access and storage of data without concern for the internal representation of data.

Today's most popular database systems are *relational databases* (Section 28.2). A language called **SQL**—pronounced "sequel," or as its individual letters—is the international standard language used almost universally with relational databases to perform **queries** (i.e., to request information that satisfies given criteria) and to manipulate data. [*Note:* As you learn about SQL, you'll see some authors writing "a SQL statement" (which assumes the pronunciation "sequel") and others writing "an SQL statement" (which assumes that the individual letters are pronounced). In this book we pronounce SQL as "sequel."]

Some popular **relational database management systems** (**RDBMSs**) are Microsoft SQL Server, Oracle, Sybase, IBM DB2, Informix, PostgreSQL and MySQL. The JDK now comes with a pure-Java RDBMS called Java DB—Oracles's version of Apache Derby. In this chapter, we present examples using MySQL and Java DB.

Java programs communicate with databases and manipulate their data using the **Java Database Connectivity (JDBC™) API**. A **JDBC driver** enables Java applications to connect to a database in a particular DBMS and allows you to manipulate that database using the JDBC API.

Software Engineering Observation 28.1

Using the JDBC API enables developers to change the underlying DBMS (for example, from Java DB to MySQL) without modifying the Java code that accesses the database.

1. Before using this chapter, please review the Before You Begin section of the book.

Most popular database management systems now provide JDBC drivers. There are also many third-party JDBC drivers available. In this chapter, we introduce JDBC and use it to manipulate MySQL and Java DB databases. The techniques demonstrated here can also be used to manipulate other databases that have JDBC drivers. Check your DBMS's documentation to determine whether your DBMS comes with a JDBC driver. If not, third-party vendors provide JDBC drivers for many DBMSs.

Software Engineering Observation 28.2

Most major database vendors provide their own JDBC database drivers, and many third-party vendors provide JDBC drivers as well.

For more information on JDBC, visit

```
www.oracle.com/technetwork/java/javase/tech/index-jsp-136101.html
```

which contains JDBC information including the JDBC specification, FAQs, a learning resource center and software downloads.

28.2 Relational Databases

A **relational database** is a logical representation of data that allows the data to be accessed without consideration of its physical structure. A relational database stores data in **tables**. Figure 28.1 illustrates a sample table that might be used in a personnel system. The table name is `Employee`, and its primary purpose is to store the attributes of employees. Tables are composed of **rows**, and rows are composed of **columns** in which values are stored. This table consists of six rows. The `Number` column of each row is the table's **primary key**—a column (or group of columns) with a *unique* value that cannot be duplicated in other rows. This guarantees that each row can be identified by its primary key. Good examples of primary-key columns are a social security number, an employee ID number and a part number in an inventory system, as values in each of these columns are guaranteed to be unique. The rows in Fig. 28.1 are displayed in order by primary key. In this case, the rows are listed in increasing order, but we could also use decreasing order.

Rows in tables are not guaranteed to be stored in any particular order. As we'll demonstrate in an upcoming example, programs can specify ordering criteria when requesting data from a database.

Number	Name	Department	Salary	Location
23603	Jones	413	1100	New Jersey
24568	Kerwin	413	2000	New Jersey
34589	Larson	642	1800	Los Angeles
35761	Myers	611	1400	Orlando
47132	Neumann	413	9000	New Jersey
78321	Stephens	611	8500	Orlando

Row

Primary key

Column

Fig. 28.1 | `Employee` table sample data.

Each column represents a different data attribute. Rows are normally unique (by primary key) within a table, but particular column values may be duplicated between rows. For example, three different rows in the `Employee` table's `Department` column contain number 413.

Different users of a database are often interested in different data and different relationships among the data. Most users require only subsets of the rows and columns. Queries specify which subsets of the data to select from a table. You use SQL to define queries. For example, you might select data from the `Employee` table to create a result that shows where each department is located, presenting the data sorted in increasing order by department number. This result is shown in Fig. 28.2. SQL is discussed in Section 28.4.

Department	Location
413	New Jersey
611	Orlando
642	Los Angeles

Fig. 28.2 | Result of selecting distinct `Department` and `Location` data from table `Employee`.

28.3 Relational Database Overview: The books Database

We now overview relational databases in the context of a sample `books` database we created for this chapter. Before we discuss SQL, we discuss the *tables* of the `books` database. We use this database to introduce various database concepts, including how to use SQL to obtain information from the database and to manipulate the data. We provide a script to create the database. You can find the script in the examples directory for this chapter. Section 28.7 explains how to use this script. The database consists of three tables: `Authors`, `AuthorISBN` and `Titles`.

`Authors` Table

The `Authors` table (described in Fig. 28.3) consists of three columns that maintain each author's unique ID number, first name and last name. Figure 28.4 contains sample data from the `Authors` table of the `books` database.

Column	Description
`AuthorID`	Author's ID number in the database. In the `books` database, this integer column is defined as **autoincremented**—for each row inserted in this table, the `AuthorID` value is increased by 1 automatically to ensure that each row has a unique `AuthorID`. This column represents the table's primary key.
`FirstName`	Author's first name (a string).
`LastName`	Author's last name (a string).

Fig. 28.3 | `Authors` table from the `books` database.

AuthorID	FirstName	LastName
1	Paul	Deitel
2	Harvey	Deitel
3	Abbey	Deitel
4	Michael	Morgano
5	Eric	Kern

Fig. 28.4 | Sample data from the `Authors` table.

`AuthorISBN` Table

The `AuthorISBN` table (described in Fig. 28.5) consists of two columns that maintain each ISBN and the corresponding author's ID number. This table associates authors with their books. Both columns are foreign keys that represent the relationship between the tables `Authors` and `Titles`—one row in table `Authors` may be associated with many rows in table `Titles`, and vice versa. The combined columns of the `AuthorISBN` table represent the table's *primary key*—thus, each row in this table must be a *unique* combination of an `AuthorID` and an `ISBN`. Figure 28.6 contains sample data from the `AuthorISBN` table of the `books` database. [*Note:* To save space, we have split the contents of this table into two columns, each containing the `AuthorID` and `ISBN` columns.] The `AuthorID` column is a **foreign key**—a column in this table that matches the primary-key column in another table (i.e., `AuthorID` in the `Authors` table). Foreign keys are specified when creating a table. The foreign key helps maintain the **Rule of Referential Integrity**—every foreign-key value must appear as another table's primary-key value. This enables the DBMS to determine whether the `AuthorID` value for a particular book is *valid*. Foreign keys also allow related data in multiple tables to be selected from those tables for analytic purposes—this is known as **joining** the data.

Column	Description
`AuthorID`	The author's ID number, a foreign key to the `Authors` table.
`ISBN`	The ISBN for a book, a foreign key to the `Titles` table.

Fig. 28.5 | `AuthorISBN` table from the `books` database.

AuthorID	ISBN	AuthorID	ISBN
1	0132152134	2	0132575663
2	0132152134	1	0132662361
1	0132151421	2	0132662361
2	0132151421	1	0132404168
1	0132575663	2	0132404168

Fig. 28.6 | Sample data from the `AuthorISBN` table of `books`. (Part 1 of 2.)

AuthorID	ISBN	AuthorID	ISBN
1	013705842X	1	0132121360
2	013705842X	2	0132121360
3	013705842X	3	0132121360
4	013705842X	4	0132121360
5	013705842X		

Fig. 28.6 | Sample data from the `AuthorISBN` table of `books`. (Part 2 of 2.)

`Titles` Table

The `Titles` table described in Fig. 28.7 consists of four columns that stand for the ISBN, the title, the edition number and the copyright year. The table is in Fig. 28.8.

Column	Description
`ISBN`	ISBN of the book (a string). The table's primary key. ISBN is an abbreviation for "International Standard Book Number"—a numbering scheme that publishers use to give every book a unique identification number.
`Title`	Title of the book (a string).
`EditionNumber`	Edition number of the book (an integer).
`Copyright`	Copyright year of the book (a string).

Fig. 28.7 | `Titles` table from the `books` database.

ISBN	Title	EditionNumber	Copyright
0132152134	Visual Basic 2010 How to Program	5	2011
0132151421	Visual C# 2010 How to Program	4	2011
0132575663	Java How to Program	9	2012
0132662361	C++ How to Program	8	2012
0132404168	C How to Program	6	2010
013705842X	iPhone for Programmers: An App-Driven Approach	1	2010
0132121360	Android for Programmers: An App-Driven Approach	1	2012

Fig. 28.8 | Sample data from the `Titles` table of the `books` database .

Entity-Relationship (ER) Diagram

There's a one-to-many relationship between a primary key and a corresponding foreign key (e.g., one author can write many books). A foreign key can appear many times in its own table, but only once (as the primary key) in another table. Figure 28.9 is an **entity-**

relationship (ER) diagram for the `books` database. This diagram shows the *database tables* and the *relationships* among them. The first compartment in each box contains the table's name and the remaining compartments contain the table's columns. The names in italic are primary keys. *A table's primary key uniquely identifies each row in the table.* Every row must have a primary-key value, and that value must be unique in the table. This is known as the **Rule of Entity Integrity**. Again, for the `AuthorISBN` table, the primary key is the combination of both columns.

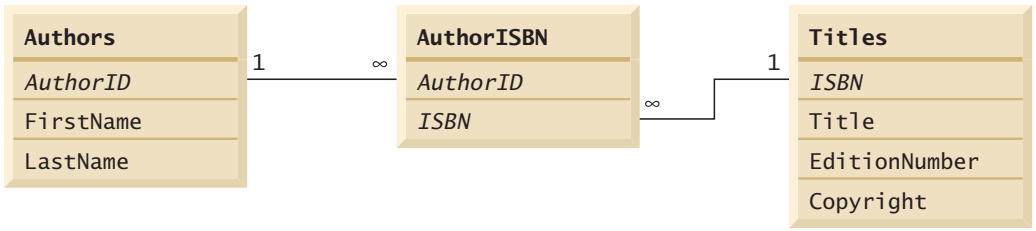


Fig. 28.9 | Table relationships in the `books` database.

Common Programming Error 28.1

Not providing a value for every column in a primary key breaks the Rule of Entity Integrity and causes the DBMS to report an error.

Common Programming Error 28.2

Providing the same primary-key value in multiple rows causes the DBMS to report an error.

The lines connecting the tables (Fig. 28.9) represent the relationships between the tables. Consider the line between the `AuthorISBN` and `Authors` tables. On the `Authors` end of the line is a `1`, and on the `AuthorISBN` end is an infinity symbol (∞), indicating a **one-to-many relationship** in which every author in the `Authors` table can have an arbitrary number of books in the `AuthorISBN` table. The relationship line links the `AuthorID` column in `Authors` (i.e., its primary key) to the `AuthorID` column in `AuthorISBN` (i.e., its foreign key). The `AuthorID` column in the `AuthorISBN` table is a foreign key.

Common Programming Error 28.3

Providing a foreign-key value that does not appear as a primary-key value in another table breaks the Rule of Referential Integrity and causes the DBMS to report an error.

The line between `Titles` and `AuthorISBN` illustrates another *one-to-many relationship*; a title can be written by any number of authors. In fact, the sole purpose of the `AuthorISBN` table is to provide a *many-to-many relationship* between `Authors` and `Titles`—an author can write many books and a book can have many authors.

28.4 SQL

We now overview SQL in the context of our `books` database. You'll be able to use the SQL discussed here in the examples later in the chapter and in examples in Chapters 30–31.

The next several subsections discuss the SQL keywords listed in Fig. 28.10 in the context of SQL queries and statements. Other SQL keywords are beyond this text's scope. To

learn other keywords, refer to the SQL reference guide supplied by the vendor of the RDBMS you're using.

SQL keyword	Description
`SELECT`	Retrieves data from one or more tables.
`FROM`	Tables involved in the query. Required in every `SELECT`.
`WHERE`	Criteria for selection that determine the rows to be retrieved, deleted or updated. Optional in a SQL query or a SQL statement.
`GROUP BY`	Criteria for grouping rows. Optional in a `SELECT` query.
`ORDER BY`	Criteria for ordering rows. Optional in a `SELECT` query.
`INNER JOIN`	Merge rows from multiple tables.
`INSERT`	Insert rows into a specified table.
`UPDATE`	Update rows in a specified table.
`DELETE`	Delete rows from a specified table.

Fig. 28.10 | SQL query keywords.

28.4.1 Basic SELECT Query

Let us consider several SQL queries that extract information from database `books`. A SQL query "selects" rows and columns from one or more tables in a database. Such selections are performed by queries with the **`SELECT`** keyword. The basic form of a `SELECT` query is

```
SELECT * FROM tableName
```

in which the **asterisk** (`*`) *wildcard character* indicates that all columns from the *tableName* table should be retrieved. For example, to retrieve all the data in the `Authors` table, use

```
SELECT * FROM Authors
```

Most programs do not require all the data in a table. To retrieve only specific columns, replace the `*` with a comma-separated list of column names. For example, to retrieve only the columns `AuthorID` and `LastName` for all rows in the `Authors` table, use the query

```
SELECT AuthorID, LastName FROM Authors
```

This query returns the data listed in Fig. 28.11.

AuthorID	LastName
1	Deitel
2	Deitel
3	Deitel
4	Morgano
5	Kern

Fig. 28.11 | Sample `AuthorID` and `LastName` data from the `Authors` table.

Software Engineering Observation 28.3

In general, you process results by knowing in advance the order of the columns in the result—for example, selecting `AuthorID` *and* `LastName` *from table* `Authors` *ensures that the columns will appear in the result with* `AuthorID` *as the first column and* `LastName` *as the second column. Programs typically process result columns by specifying the column number in the result (starting from number 1 for the first column). Selecting columns by name avoids returning unneeded columns and protects against changes in the actual order of the columns in the table(s) by returning the columns in the exact order specified.*

Common Programming Error 28.4

If you assume that the columns are always returned in the same order from a query that uses the asterisk (*), the program may process the results incorrectly. If the column order in the table(s) changes or if additional columns are added at a later time, the order of the columns in the result will change accordingly.

28.4.2 WHERE Clause

In most cases, it's necessary to locate rows in a database that satisfy certain **selection criteria**. Only rows that satisfy the selection criteria (formally called **predicates**) are selected. SQL uses the optional **`WHERE` clause** in a query to specify the selection criteria for the query. The basic form of a query with selection criteria is

```
SELECT columnName1, columnName2, ... FROM tableName WHERE criteria
```

For example, to select the `Title`, `EditionNumber` and `Copyright` columns from table `Titles` for which the `Copyright` date is greater than `2010`, use the query

```
SELECT Title, EditionNumber, Copyright
   FROM Titles
   WHERE Copyright > '2010'
```

Strings in SQL are delimited by single (') rather than double (") quotes.Figure 28.12 shows the result of the preceding query.

Title	EditionNumber	Copyright
Visual Basic 2010 How to Program	5	2011
Visual C# 2010 How to Program	4	2011
Java How to Program	9	2012
C++ How to Program	8	2012
Android for Programmers: An App-Driven Approach	1	2012

Fig. 28.12 | Sampling of titles with copyrights after 2005 from table `Titles`.

Pattern Matching: Zero or More Characters

The `WHERE` clause criteria can contain the operators <, >, <=, >=, =, <> and `LIKE`. Operator **`LIKE`** is used for **pattern matching** with wildcard characters **percent** (**%**) and **underscore** (_). Pattern matching allows SQL to search for strings that match a given pattern.

A pattern that contains a percent character (%) searches for strings that have zero or more characters at the percent character's position in the pattern. For example, the next query locates the rows of all the authors whose last name starts with the letter D:

```
SELECT AuthorID, FirstName, LastName
   FROM Authors
   WHERE LastName LIKE 'D%'
```

This query selects the two rows shown in Fig. 28.13—three of the five authors have a last name starting with the letter D (followed by zero or more characters). The % symbol in the WHERE clause's LIKE pattern indicates that any number of characters can appear after the letter D in the LastName. The pattern string is surrounded by single-quote characters.

AuthorID	FirstName	LastName
1	Paul	Deitel
2	Harvey	Deitel
3	Abbey	Deitel

Fig. 28.13 | Authors whose last name starts with D from the Authors table.

Portability Tip 28.1

See the documentation for your database system to determine whether SQL is case sensitive on your system and to determine the syntax for SQL keywords.

Portability Tip 28.2

Read your database system's documentation carefully to determine whether it supports the LIKE operator as discussed here.

Pattern Matching: Any Character

An underscore (_) in the pattern string indicates a single wildcard character at that position in the pattern. For example, the following query locates the rows of all the authors whose last names start with any character (specified by _), followed by the letter o, followed by any number of additional characters (specified by %):

```
SELECT AuthorID, FirstName, LastName
   FROM Authors
   WHERE LastName LIKE '_o%'
```

The preceding query produces the row shown in Fig. 28.14, because only one author in our database has a last name that contains the letter o as its second letter.

AuthorID	FirstName	LastName
4	Michael	Morgano

Fig. 28.14 | The only author from the Authors table whose last name contains o as the second letter.

28.4.3 ORDER BY Clause

The rows in the result of a query can be sorted into ascending or descending order by using the optional **ORDER BY clause**. The basic form of a query with an `ORDER BY` clause is

```
SELECT columnName1, columnName2, ... FROM tableName ORDER BY column ASC
SELECT columnName1, columnName2, ... FROM tableName ORDER BY column DESC
```

where `ASC` specifies ascending order (lowest to highest), `DESC` specifies descending order (highest to lowest) and *column* specifies the column on which the sort is based. For example, to obtain the list of authors in ascending order by last name (Fig. 28.15), use the query

```
SELECT AuthorID, FirstName, LastName
   FROM Authors
   ORDER BY LastName ASC
```

AuthorID	FirstName	LastName
1	Paul	Deitel
2	Harvey	Deitel
3	Abbey	Deitel
5	Eric	Kern
4	Michael	Morgano

Fig. 28.15 | Sample data from table `Authors` in ascending order by `LastName`.

Sorting in Descending Order

The default sorting order is ascending, so `ASC` is optional. To obtain the same list of authors in descending order by last name (Fig. 28.16), use the query

```
SELECT AuthorID, FirstName, LastName
   FROM Authors
   ORDER BY LastName DESC
```

AuthorID	FirstName	LastName
4	Michael	Morgano
5	Eric	Kern
1	Paul	Deitel
2	Harvey	Deitel
3	Abbey	Deitel

Fig. 28.16 | Sample data from table `Authors` in descending order by `LastName`.

Sorting By Multiple Columns

Multiple columns can be used for sorting with an `ORDER BY` clause of the form

```
ORDER BY column1 sortingOrder, column2 sortingOrder, ...
```

where *sortingOrder* is either `ASC` or `DESC`. The *sortingOrder* does not have to be identical for each column. The query

```
SELECT AuthorID, FirstName, LastName
   FROM Authors
   ORDER BY LastName, FirstName
```

sorts all the rows in ascending order by last name, then by first name. If any rows have the same last-name value, they're returned sorted by first name (Fig. 28.17).

AuthorID	FirstName	LastName
3	Abbey	Deitel
2	Harvey	Deitel
1	Paul	Deitel
5	Eric	Kern
4	Michael	Morgano

Fig. 28.17 | Sample data from `Authors` in ascending order by `LastName` and `FirstName`.

Combining the `WHERE` and `ORDER BY` Clauses

The `WHERE` and `ORDER BY` clauses can be combined in one query, as in

```
SELECT ISBN, Title, EditionNumber, Copyright
   FROM Titles
   WHERE Title LIKE '%How to Program'
   ORDER BY Title ASC
```

which returns the `ISBN`, `Title`, `EditionNumber` and `Copyright` of each book in the `Titles` table that has a `Title` ending with `"How to Program"` and sorts them in ascending order by `Title`. The query results are shown in Fig. 28.18.

ISBN	Title	Edition-Number	Copy-right
0132404168	C How to Program	6	2010
0132662361	C++ How to Program	8	2012
0132575663	Java How to Program	9	2012
0132152134	Visual Basic 2005 How to Program	5	2011
0132151421	Visual C# 2005 How to Program	4	2011

Fig. 28.18 | Sampling of books from table `Titles` whose titles end with `How to Program` in ascending order by `Title`.

28.4.4 Merging Data from Multiple Tables: INNER JOIN

Database designers often split related data into separate tables to ensure that a database does not store data redundantly. For example, in the `books` database, we use an `AuthorISBN` table to store the relationship data between authors and their corresponding titles. If we did

not separate this information into individual tables, we'd need to include author information with each entry in the `Titles` table. This would result in the database's storing *duplicate* author information for authors who wrote multiple books. Often, it's necessary to merge data from multiple tables into a single result. Referred to as joining the tables, this is specified by an **`INNER JOIN`** operator, which merges rows from two tables by matching values in columns that are common to the tables. The basic form of an `INNER JOIN` is:

```
SELECT columnName1, columnName2, ...
FROM table1
INNER JOIN table2
   ON table1.columnName = table2.columnName
```

The **`ON` clause** of the `INNER JOIN` specifies the columns from each table that are compared to determine which rows are merged. For example, the following query produces a list of authors accompanied by the ISBNs for books written by each author:

```
SELECT FirstName, LastName, ISBN
FROM Authors
INNER JOIN AuthorISBN
   ON Authors.AuthorID = AuthorISBN.AuthorID
ORDER BY LastName, FirstName
```

The query merges the `FirstName` and `LastName` columns from table `Authors` with the `ISBN` column from table `AuthorISBN`, sorting the result in ascending order by `LastName` and `FirstName`. Note the use of the syntax *tableName*`.`*columnName* in the `ON` clause. This syntax, called a **qualified name**, specifies the columns from each table that should be compared to join the tables. The "*tableName*`.`" syntax is required if the columns have the same name in both tables. The same syntax can be used in any SQL statement to distinguish columns in different tables that have the same name. In some systems, table names qualified with the database name can be used to perform cross-database queries. As always, the query can contain an `ORDER BY` clause. Figure 28.19 shows the results of the preceding query, ordered by `LastName` and `FirstName`. [*Note:* To save space, we split the result of the query into two columns, each containing the `FirstName`, `LastName` and `ISBN` columns.]

FirstName	LastName	ISBN	FirstName	LastName	ISBN
Abbey	Deitel	013705842X	Paul	Deitel	0132151421
Abbey	Deitel	0132121360	Paul	Deitel	0132575663
Harvey	Deitel	0132152134	Paul	Deitel	0132662361
Harvey	Deitel	0132151421	Paul	Deitel	0132404168
Harvey	Deitel	0132575663	Paul	Deitel	013705842X
Harvey	Deitel	0132662361	Paul	Deitel	0132121360
Harvey	Deitel	0132404168	Eric	Kern	013705842X
Harvey	Deitel	013705842X	Michael	Morgano	013705842X
Harvey	Deitel	0132121360	Michael	Morgano	0132121360
Paul	Deitel	0132152134			

Fig. 28.19 | Sampling of authors and ISBNs for the books they have written in ascending order by `LastName` and `FirstName`.

Software Engineering Observation 28.4

If a SQL statement includes columns with the same name from multiple tables, the statement must precede those column names with their table names and a dot (e.g., `Authors.AuthorID`).

Common Programming Error 28.5

Failure to qualify names for columns that have the same name in two or more tables is an error.

28.4.5 INSERT Statement

The **`INSERT`** statement inserts a row into a table. The basic form of this statement is

```
INSERT INTO tableName ( columnName1, columnName2, ..., columnNameN )
   VALUES ( value1, value2, ..., valueN )
```

where *tableName* is the table in which to insert the row. The *tableName* is followed by a comma-separated list of column names in parentheses (this list is not required if the `INSERT` operation specifies a value for every column of the table in the correct order). The list of column names is followed by the SQL keyword **`VALUES`** and a comma-separated list of values in parentheses. The values specified here must match the columns specified after the table name in both order and type (e.g., if *columnName1* is supposed to be the `FirstName` column, then *value1* should be a string in single quotes representing the first name). Always explicitly list the columns when inserting rows. If the table's column order changes or a new column is added, using only `VALUES` may cause an error. The `INSERT` statement

```
INSERT INTO Authors ( FirstName, LastName )
   VALUES ( 'Sue', 'Red' )
```

inserts a row into the `Authors` table. The statement indicates that values are provided for the `FirstName` and `LastName` columns. The corresponding values are `'Sue'` and `'Smith'`. We do not specify an `AuthorID` in this example because `AuthorID` is an autoincremented column in the `Authors` table. For every row added to this table, the DBMS assigns a unique `AuthorID` value that is the next value in the autoincremented sequence (i.e., 1, 2, 3 and so on). In this case, Sue Red would be assigned `AuthorID` number 6. Figure 28.20 shows the `Authors` table after the `INSERT` operation. [*Note:* Not every database management system supports autoincremented columns. Check the documentation for your DBMS for alternatives to autoincremented columns.]

AuthorID	FirstName	LastName
1	Paul	Deitel
2	Harvey	Deitel
3	Abbey	Deitel
4	Michael	Morgano
5	Eric	Kern
6	Sue	Red

Fig. 28.20 | Sample data from table `Authors` after an `INSERT` operation.

Common Programming Error 28.6

It's normally an error to specify a value for an autoincrement column.

Common Programming Error 28.7

SQL delimits strings with single quotes ('). *A string containing a single quote (e.g., O'Malley) must have two single quotes in the position where the single quote appears (e.g., 'O''Malley').* *The first acts as an escape character for the second.* *Not escaping single-quote characters in a string that's part of a SQL statement is a SQL syntax error.*

28.4.6 UPDATE Statement

An **UPDATE** statement modifies data in a table. Its basic form is

```
UPDATE tableName
   SET columnName1 = value1, columnName2 = value2, ..., columnNameN = valueN
   WHERE criteria
```

where *tableName* is the table to update. The *tableName* is followed by keyword **SET** and a comma-separated list of column name/value pairs in the format *columnName* = *value*. The optional `WHERE` clause provides criteria that determine which rows to update. Though not required, the `WHERE` clause is typically used, unless a change is to be made to every row. The `UPDATE` statement

```
UPDATE Authors
   SET LastName = 'Black'
   WHERE LastName = 'Red' AND FirstName = 'Sue'
```

updates a row in the `Authors` table. The statement indicates that `LastName` will be assigned the value `Black` for the row in which `LastName` is equal to `Red` and `FirstName` is equal to `Sue`. [*Note:* If there are multiple rows with the first name "Sue" and the last name "Red," this statement will modify all such rows to have the last name "Black."] If we know the `AuthorID` in advance of the `UPDATE` operation (possibly because we searched for it previously), the `WHERE` clause can be simplified as follows:

```
WHERE AuthorID = 6
```

Figure 28.21 shows the `Authors` table after the `UPDATE` operation has taken place.

AuthorID	FirstName	LastName
1	Paul	Deitel
2	Harvey	Deitel
3	Abbey	Deitel
4	Michael	Morgano
5	Eric	Kern
6	Sue	Black

Fig. 28.21 | Sample data from table `Authors` after an `UPDATE` operation.

28.4.7 DELETE Statement

A SQL **DELETE** statement removes rows from a table. Its basic form is

```
DELETE FROM tableName WHERE criteria
```

where *tableName* is the table from which to delete. The optional `WHERE` clause specifies the criteria used to determine which rows to delete. If this clause is omitted, all the table's rows are deleted. The `DELETE` statement

```
DELETE FROM Authors
   WHERE LastName = 'Black' AND FirstName = 'Sue'
```

deletes the row for Sue Black in the `Authors` table. If we know the `AuthorID` in advance of the `DELETE` operation, the `WHERE` clause can be simplified as follows:

```
WHERE AuthorID = 5
```

Figure 28.22 shows the `Authors` table after the `DELETE` operation has taken place.

AuthorID	FirstName	LastName
1	Paul	Deitel
2	Harvey	Deitel
3	Abbey	Deitel
4	Michael	Morgano
5	Eric	Kern

Fig. 28.22 | Sample data from table `Authors` after a `DELETE` operation.

28.5 Instructions for Installing MySQL and MySQL Connector/J

MySQL Community Edition is an open-source database management system that executes on many platforms, including Windows, Linux, and Mac OS X. Complete information about MySQL is available from `www.mysql.com`. The examples in Sections 28.8–28.9 manipulate MySQL databases using MySQL 5.5.8—the latest release at the time of this writing.

Installing MySQL

To install MySQL Community Edition on Windows, Linux or Mac OS X, see the installation overview for your platform at:

- Windows: `dev.mysql.com/doc/refman/5.5/en/windows-installation.html`
- Linux: `dev.mysql.com/doc/refman/5.5/en/linux-installation-rpm.html`
- Mac OS X: `dev.mysql.com/doc/refman/5.5/en/macosx-installation.html`

Carefully follow the instructions for downloading and installing the software on your platform. The downloads are available from:

```
dev.mysql.com/downloads/mysql/
```

For the following steps, we assume that you're installing MySQL on Windows. When you execute the installer, the **MySQL Server 5.5 Setup Wizard** window will appear. Perform the following steps:

1. Click the **Next** button.
2. Read the license agreement, then check the **I accept the terms in the License Agreement** checkbox and click the **Next** button. [*Note:* If you do not accept the license terms, you will not be able to install MySQL.]
3. Click the **Typical** button in the **Choose Setup Type** screen then click **Install**.
4. When the installation completes, click **Next >** twice.
5. In the **Completed the MySQL Server 5.5 Setup Wizard** screen, ensure that the **Launch the MySQL Instance Configuration Wizard** checkbox is checked, then click **Finish** to begin configuring the server.

The **MySQL Instance Configuration Wizard** window appears. To configure the server:

1. Click **Next >**, then select **Standard Configuration** and click **Next >** again.
2. You have the option of installing MySQL as a Windows service, which enables the MySQL server to begin executing automatically each time your system starts. For our examples, this is unnecessary, so you can uncheck **Install as a Windows Service** if you wish. Check **Include Bin Directory in Windows PATH**. This will enable you to use the MySQL commands in the Windows Command Prompt. Click **Next >**, then click **Execute** to perform the server configuration.
3. Click **Finish** to close the wizard.

You've now completed the MySQL installation.

Installing MySQL Connector/J

To use MySQL with JDBC, you also need to install **MySQL Connector/J** (the J stands for Java)—a JDBC driver that allows programs to use JDBC to interact with MySQL. MySQL Connector/J can be downloaded from

```
dev.mysql.com/downloads/connector/j/
```

The documentation for Connector/J is located at

```
dev.mysql.com/doc/refman/5.5/en/connector-j.html
```

At the time of this writing, the current generally available release of MySQL Connector/J is 5.1.14. To install MySQL Connector/J, carefully follow the installation instructions at:

```
dev.mysql.com/doc/refman/5.5/en/connector-j-installing.html
```

We *do not* recommend modifying your system's `CLASSPATH` environment variable, which is discussed in the installation instructions. Instead, we'll show you how use MySQL Connector/J by specifying it as a command-line option when you execute your applications.

28.6 Instructions for Setting Up a MySQL User Account

For the MySQL examples to execute correctly, you need to set up a user account that allows users to create, delete and modify a database. After MySQL is installed, follow the

steps below to set up a user account (these steps assume MySQL is installed in its default installation directory):

1. Open a Command Prompt and start the database server by executing the command `mysqld.exe`. This command has no output—it simply starts the MySQL server. Do not close this window—doing so terminates the server.

1. Next, you'll start the MySQL monitor so you can set up a user account, open another Command Prompt and execute the command

```
mysql -h localhost -u root
```

   The `-h` option indicates the host (i.e., computer) on which the MySQL server is running—in this case your local computer (`localhost`). The `-u` option indicates the user account that will be used to log in to the server—`root` is the default user account that is created during installation to allow you to configure the server. Once you've logged in, you'll see a `mysql>` prompt at which you can type commands to interact with the MySQL server.

1. At the `mysql>` prompt, type

```
USE mysql;
```

   and press *Enter* to select the built-in database named `mysql`, which stores server information, such as user accounts and their privileges for interacting with the server. Each command must end with a semicolon. To confirm the command, MySQL issues the message "`Database changed`."

1. Next, you'll add the `deitel` user account to the `mysql` built-in database. The `mysql` database contains a table called `user` with columns that represent the user's name, password and various privileges. To create the `deitel` user account with the password `deitel`, execute the following commands from the `mysql>` prompt:

```
create user 'deitel'@'localhost' identified by 'deitel';
grant select, insert, update, delete, create, drop, references,
   execute on *.* to 'deitel'@'localhost';
```

   This creates the `deitel` user with the privileges needed to create the databases used in this chapter and manipulate them.

1. Type the command

```
exit;
```

   to terminate the MySQL monitor.

28.7 Creating Database books in MySQL

For each MySQL database we discuss, we provide a SQL script in a `.sql` file that sets up the database and its tables. You can execute these scripts in the MySQL monitor. In this chapter's examples directory, you'll find the script `books.sql` to create the `books` database. For the following steps, we assume that the MySQL server (`mysqld.exe`) is still running. To execute the `books.sql` script:

1. Open a Command Prompt and use the `cd` command to change directories to the location that contains the `books.sql` script.

2. Start the MySQL monitor by typing

```
mysql -h localhost -u deitel -p
```

The `-p` option prompts you for the password for the `deitel` user account. When prompted, enter the password `deitel`.

3. Execute the script by typing

```
source books.sql;
```

This creates a new directory named `books` in the server's `data` directory—located by default on Windows at `C:\ProgramData\MySQL\MySQL Server 5.5\data`. This new directory contains the `books` database.

4. Type the command

```
exit;
```

to terminate the MySQL monitor. You're now ready to proceed to the first JDBC example.

28.8 Manipulating Databases with JDBC

This section presents two examples. The first introduces how to connect to a database and query it. The second demonstrates how to display the result of the query in a `JTable`.

28.8.1 Connecting to and Querying a Database

The example of Fig. 28.23 performs a simple query on the `books` database that retrieves the entire `Authors` table and displays the data. The program illustrates connecting to the database, querying the database and processing the result. The discussion that follows presents the key JDBC aspects of the program. [*Note:* Sections 28.5–28.7 demonstrate how to start the MySQL server, configure a user account and create the `books` database. These steps *must* be performed before executing the program of Fig. 28.23.]

```
1  // Fig. 28.23: DisplayAuthors.java
2  // Displaying the contents of the Authors table.
3  import java.sql.Connection;
4  import java.sql.Statement;
5  import java.sql.DriverManager;
6  import java.sql.ResultSet;
7  import java.sql.ResultSetMetaData;
8  import java.sql.SQLException;
9
10 public class DisplayAuthors
11 {
12    // database URL
13    static final String DATABASE_URL = "jdbc:mysql://localhost/books";
14
15    // launch the application
16    public static void main( String args[] )
17    {
```

Fig. 28.23 | Displaying the contents of the `Authors` table. (Part 1 of 3.)

```
18          Connection connection = null; // manages connection
19          Statement statement = null; // query statement
20          ResultSet resultSet = null; // manages results
21
22          // connect to database books and query database
23          try
24          {
25             // establish connection to database
26             connection = DriverManager.getConnection(
27                DATABASE_URL, "deitel", "deitel" );
28
29             // create Statement for querying database
30             statement = connection.createStatement();
31
32             // query database
33             resultSet = statement.executeQuery(
34                "SELECT AuthorID, FirstName, LastName FROM Authors" );
35
36             // process query results
37             ResultSetMetaData metaData = resultSet.getMetaData();
38             int numberOfColumns = metaData.getColumnCount();
39             System.out.println( "Authors Table of Books Database:\n" );
40
41             for ( int i = 1; i <= numberOfColumns; i++ )
42                System.out.printf( "%-8s\t", metaData.getColumnName( i ) );
43             System.out.println();
44
45             while ( resultSet.next() )
46             {
47                for ( int i = 1; i <= numberOfColumns; i++ )
48                   System.out.printf( "%-8s\t", resultSet.getObject( i ) );
49                System.out.println();
50             } // end while
51          }  // end try
52          catch ( SQLException sqlException )
53          {
54             sqlException.printStackTrace();
55          } // end catch
56          finally // ensure resultSet, statement and connection are closed
57          {
58             try
59             {
60                resultSet.close();
61                statement.close();
62                connection.close();
63             } // end try
64             catch ( Exception exception )
65             {
66                exception.printStackTrace();
67             } // end catch
68          } // end finally
69       } // end main
70    } // end class DisplayAuthors
```

Fig. 28.23 | Displaying the contents of the `Authors` table. (Part 2 of 3.)

```
Authors Table of Books Database:

AuthorID        FirstName       LastName
1               Harvey          Deitel
2               Paul            Deitel
3               Andrew          Goldberg
4               David           Choffnes
```

Fig. 28.23 | Displaying the contents of the `Authors` table. (Part 3 of 3.)

Lines 3–8 import the JDBC interfaces and classes from package `java.sql` used in this program. Line 13 declares a string constant for the database URL. This identifies the name of the database to connect to, as well as information about the protocol used by the JDBC driver (discussed shortly). Method `main` (lines 16–69) connects to the `books` database, queries the database, displays the result of the query and closes the database connection.

In past versions of Java, programs were required to load an appropriate database driver before connecting to a database. JDBC 4.0 and higher support **automatic driver discovery**—you're no longer required to load the database driver in advance. To ensure that the program can locate the database driver class, you must include the class's location in the program's classpath when you execute the program. For MySQL, you include the file `mysql-connector-java-5.1.14-bin.jar` (in the `C:\mysql-connector-java-5.1.14` directory) in your program's classpath, as in:

```
java -classpath .;c:\mysql-connector-java-5.1.14\mysql-connector-
java-5.1.14-bin.jar DisplayAuthors
```

If the period (`.`) at the beginning of the classpath information is missing, the JVM will not look for classes in the current directory and thus will not find the `DisplayAuthors` class file. You may also copy the `mysql-connector-java-5.1.14-bin.jar` file to your JDK's `\jre\lib\ext` folder. After doing so, you can run the application simply using the command

```
java DisplayAuthors
```

Connecting to the Database

Lines 26–27 of Fig. 28.23 create a **`Connection`** object (package `java.sql`) referenced by `connection`. An object that implements interface `Connection` manages the connection between the Java program and the database. `Connection` objects enable programs to create SQL statements that manipulate databases. The program initializes `connection` with the result of a call to `static` method **`getConnection`** of class **`DriverManager`** (package `java.sql`), which attempts to connect to the database specified by its URL. Method `getConnection` takes three arguments—a `String` that specifies the database URL, a `String` that specifies the username and a `String` that specifies the password. The username and password are set in Section 28.6. If you used a different username and password, you need to replace the username (second argument) and password (third argument) passed to method `getConnection` in line 27. The URL locates the database (possibly on a network or in the local file system of the computer). The URL `jdbc:mysql://localhost/books` specifies the protocol for communication (`jdbc`), the **subprotocol** for communication (`mysql`) and the location of the database (`//localhost/books`, where `localhost` is the host running the MySQL server and `books` is the database name). The subprotocol `mysql`

indicates that the program uses a MySQL-specific subprotocol to connect to the MySQL database. If the `DriverManager` cannot connect to the database, method `getConnection` throws a **`SQLException`** (package `java.sql`). Figure 28.24 lists the JDBC driver names and database URL formats of several popular RDBMSs.

RDBMS	Database URL format
MySQL	`jdbc:mysql://`*hostname*`:`*portNumber*`/`*databaseName*
ORACLE	`jdbc:oracle:thin:@`*hostname*`:`*portNumber*`:`*databaseName*
DB2	`jdbc:db2:`*hostname*`:`*portNumber*`/`*databaseName*
PostgreSQL	`jdbc:postgresql://`*hostname*`:`*portNumber*`/`*databaseName*
Java DB/Apache Derby	`jdbc:derby:`*dataBaseName* (embedded) `jdbc:derby://`*hostname*`:`*portNumber*`/`*databaseName* (network)
Microsoft SQL Server	`jdbc:sqlserver://`*hostname*`:`*portNumber*`;databaseName=`*dataBaseName*
Sybase	`jdbc:sybase:Tds:`*hostname*`:`*portNumber*`/`*databaseName*

Fig. 28.24 | Popular JDBC database URL formats.

Software Engineering Observation 28.5

Most database management systems require the user to log in before accessing the database contents. `DriverManager` method `getConnection` is overloaded with versions that enable the program to supply the user name and password to gain access.

Creating a `Statement` for Executing Queries

Line 30 invokes `Connection` method **`createStatement`** to obtain an object that implements interface `Statement` (package `java.sql`). The program uses the **`Statement`** object to submit SQL statements to the database.

Executing a Query

Lines 33–34 use the `Statement` object's **`executeQuery`** method to submit a query that selects all the author information from table `Authors`. This method returns an object that implements interface **`ResultSet`** and contains the query results. The `ResultSet` methods enable the program to manipulate the query result.

Processing a Query's `ResultSet`

Lines 37–50 process the `ResultSet`. Line 37 obtains the metadata for the `ResultSet` as a **`ResultSetMetaData`** (package `java.sql`) object. The **metadata** describes the `ResultSet`'s contents. Programs can use metadata programmatically to obtain information about the `ResultSet`'s column names and types. Line 38 uses `ResultSetMetaData` method **`getColumnCount`** to retrieve the number of columns in the `ResultSet`. Lines 41–42 display the column names.

Software Engineering Observation 28.6

Metadata enables programs to process `ResultSet` contents dynamically when detailed information about the `ResultSet` is not known in advance.

Lines 45–50 display the data in each `ResultSet` row. First, the program positions the `ResultSet` cursor (which points to the row being processed) to the first row in the `ResultSet` with method **`next`** (line 45). Method `next` returns `boolean` value `true` if it's able to position to the next row; otherwise, the method returns `false`.

Common Programming Error 28.8

Initially, a `ResultSet` cursor is positioned before the first row. A `SQLException` occurs if you attempt to access a `ResultSet`'s contents before positioning the `ResultSet` cursor to the first row with method `next`.

If there are rows in the `ResultSet`, lines 47–48 extract and display the contents of each column in the current row. When a `ResultSet` is processed, each column can be extracted as a specific Java type. In fact, `ResultSetMetaData` method **`getColumnType`** returns a constant integer from class **`Types`** (package `java.sql`) indicating the type of a specified column. Programs can use these values in a `switch` statement to invoke `ResultSet` methods that return the column values as appropriate Java types. If the type of a column is `Types.INTEGER`, `ResultSet` method **`getInt`** returns the column value as an `int`. `ResultSet` *get* methods typically receive as an argument either a column number (as an `int`) or a column name (as a `String`) indicating which column's value to obtain. Visit

```
java.sun.com/javase/6/docs/technotes/guides/jdbc/getstart/
   GettingStartedTOC.fm.html
```

for detailed mappings of SQL data types to Java types and to determine the appropriate `ResultSet` method to call for each SQL data type.

Performance Tip 28.1

If a query specifies the exact columns to select from the database, the `ResultSet` contains the columns in the specified order. In this case, using the column number to obtain the column's value is more efficient than using the column name. The column number provides direct access to the specified column. Using the column name requires a search of the column names to locate the appropriate column.

Error-Prevention Tip 28.1

Using column names to obtain values from a `ResultSet` produces code that is less error prone than obtaining values by column number—you don't need to remember the column order. Also, if the column order changes, your code does not have to change.

For simplicity, this example treats each value as an `Object`. We retrieve each column value with `ResultSet` method **`getObject`** (line 48) then print the `Object`'s `String` representation. Unlike array indices, `ResultSet` *column numbers start at 1*. The `finally` block (lines 56–68) closes the `ResultSet`, the `Statement` and the database `Connection`. [*Note:* Lines 60–62 will throw `NullPointerExceptions` if the `ResultSet`, `Statement` or `Connection` objects were not created properly. For code used in industry, you should check the variables that refer to these objects to see if they're `null` before you call `close`.]

Common Programming Error 28.9

Specifying column 0 when obtaining values from a `ResultSet` causes a `SQLException`.

Common Programming Error 28.10

A `SQLException` occurs if you attempt to manipulate a `ResultSet` after closing the `Statement` that created it. The `ResultSet` is discarded when the `Statement` is closed.

Software Engineering Observation 28.7

Each `Statement` object can open only one `ResultSet` object at a time. When a `Statement` returns a new `ResultSet`, the `Statement` closes the prior `ResultSet`. To use multiple `ResultSet`s in parallel, separate `Statement` objects must return the `ResultSet`s.

Java SE 7: Automatically Closing `Connection`s, `Statement`s and `ResultSet`s

As of Java SE 7, the interfaces `Connection`, `Statement` and `ResultSet` each extend the `AutoCloseable` interface, so you can use objects that implement these interfaces with the new `try`-with-resources statement, which was introduced in Section 11.13. In the folder for the example of Fig. 28.23, the subfolder `JavaSE7Version` contains a version of the example that uses the `try`-with-resources statement to allocate the `Connection`, `Statement` and `ResultSet` objects. These objects are automatically closed at the end of the `try` block or if an exception occurs while executing the code in the `try` block.

28.8.2 Querying the books Database

The next example (Fig. 28.25 and Fig. 28.28) allows the user to enter any query into the program. The example displays the result of a query in a **`JTable`**, using a **`TableModel`** object to provide the `ResultSet` data to the `JTable`. A `JTable` is a swing GUI component that can be bound to a database to display the results of a query. Class `ResultSetTableModel` (Fig. 28.25) performs the connection to the database via a `TableModel` and maintains the `ResultSet`. Class `DisplayQueryResults` (Fig. 28.28) creates the GUI and specifies an instance of class `ResultSetTableModel` to provide data for the `JTable`.

`ResultSetTableModel` Class

Class `ResultSetTableModel` (Fig. 28.25) extends class **`AbstractTableModel`** (package `javax.swing.table`), which implements interface `TableModel`. `ResultSetTableModel` overrides `TableModel` methods **`getColumnClass`**, **`getColumnCount`**, **`getColumnName`**, **`getRowCount`** and **`getValueAt`**. The default implementations of `TableModel` methods `isCellEditable` and `setValueAt` (provided by `AbstractTableModel`) are not overridden, because this example does not support editing the `JTable` cells. The default implementations of `TableModel` methods **`addTableModelListener`** and **`removeTableModelListener`** (provided by `AbstractTableModel`) are not overridden, because the implementations of these methods in `AbstractTableModel` properly add and remove event listeners.

```
1  // Fig. 28.25: ResultSetTableModel.java
2  // A TableModel that supplies ResultSet data to a JTable.
3  import java.sql.Connection;
4  import java.sql.Statement;
5  import java.sql.DriverManager;
6  import java.sql.ResultSet;
7  import java.sql.ResultSetMetaData;
```

Fig. 28.25 | A `TableModel` that supplies `ResultSet` data to a `JTable`. (Part 1 of 5.)

```
 8  import java.sql.SQLException;
 9  import javax.swing.table.AbstractTableModel;
10
11  // ResultSet rows and columns are counted from 1 and JTable
12  // rows and columns are counted from 0. When processing
13  // ResultSet rows or columns for use in a JTable, it is
14  // necessary to add 1 to the row or column number to manipulate
15  // the appropriate ResultSet column (i.e., JTable column 0 is
16  // ResultSet column 1 and JTable row 0 is ResultSet row 1).
17  public class ResultSetTableModel extends AbstractTableModel
18  {
19     private Connection connection;
20     private Statement statement;
21     private ResultSet resultSet;
22     private ResultSetMetaData metaData;
23     private int numberOfRows;
24
25     // keep track of database connection status
26     private boolean connectedToDatabase = false;
27
28     // constructor initializes resultSet and obtains its meta data object;
29     // determines number of rows
30     public ResultSetTableModel( String url, String username,
31        String password, String query ) throws SQLException
32     {
33        // connect to database
34        connection = DriverManager.getConnection( url, username, password );
35
36        // create Statement to query database
37        statement = connection.createStatement(
38           ResultSet.TYPE_SCROLL_INSENSITIVE,
39           ResultSet.CONCUR_READ_ONLY );
40
41        // update database connection status
42        connectedToDatabase = true;
43
44        // set query and execute it
45        setQuery( query );
46     } // end constructor ResultSetTableModel
47
48     // get class that represents column type
49     public Class getColumnClass( int column ) throws IllegalStateException
50     {
51        // ensure database connection is available
52        if ( !connectedToDatabase )
53           throw new IllegalStateException( "Not Connected to Database" );
54
55        // determine Java class of column
56        try
57        {
58           String className = metaData.getColumnClassName( column + 1 );
59
```

Fig. 28.25 | A `TableModel` that supplies `ResultSet` data to a `JTable`. (Part 2 of 5.)

```
60            // return Class object that represents className
61            return Class.forName( className );
62         } // end try
63         catch ( Exception exception )
64         {
65            exception.printStackTrace();
66         } // end catch
67
68         return Object.class; // if problems occur above, assume type Object
69      } // end method getColumnClass
70
71      // get number of columns in ResultSet
72      public int getColumnCount() throws IllegalStateException
73      {
74         // ensure database connection is available
75         if ( !connectedToDatabase )
76            throw new IllegalStateException( "Not Connected to Database" );
77
78         // determine number of columns
79         try
80         {
81            return metaData.getColumnCount();
82         } // end try
83         catch ( SQLException sqlException )
84         {
85            sqlException.printStackTrace();
86         } // end catch
87
88         return 0; // if problems occur above, return 0 for number of columns
89      } // end method getColumnCount
90
91      // get name of a particular column in ResultSet
92      public String getColumnName( int column ) throws IllegalStateException
93      {
94         // ensure database connection is available
95         if ( !connectedToDatabase )
96            throw new IllegalStateException( "Not Connected to Database" );
97
98         // determine column name
99         try
100        {
101           return metaData.getColumnName( column + 1 );
102        } // end try
103        catch ( SQLException sqlException )
104        {
105           sqlException.printStackTrace();
106        } // end catch
107
108        return ""; // if problems, return empty string for column name
109     } // end method getColumnName
110
```

Fig. 28.25 | A `TableModel` that supplies `ResultSet` data to a `JTable`. (Part 3 of 5.)

```
111    // return number of rows in ResultSet
112    public int getRowCount() throws IllegalStateException
113    {
114       // ensure database connection is available
115       if ( !connectedToDatabase )
116          throw new IllegalStateException( "Not Connected to Database" );
117
118       return numberOfRows;
119    } // end method getRowCount
120
121    // obtain value in particular row and column
122    public Object getValueAt( int row, int column )
123       throws IllegalStateException
124    {
125       // ensure database connection is available
126       if ( !connectedToDatabase )
127          throw new IllegalStateException( "Not Connected to Database" );
128
129       // obtain a value at specified ResultSet row and column
130       try
131       {
132          resultSet.absolute( row + 1 );
133          return resultSet.getObject( column + 1 );
134       } // end try
135       catch ( SQLException sqlException )
136       {
137          sqlException.printStackTrace();
138       } // end catch
139
140       return ""; // if problems, return empty string object
141    } // end method getValueAt
142
143    // set new database query string
144    public void setQuery( String query )
145       throws SQLException, IllegalStateException
146    {
147       // ensure database connection is available
148       if ( !connectedToDatabase )
149          throw new IllegalStateException( "Not Connected to Database" );
150
151       // specify query and execute it
152       resultSet = statement.executeQuery( query );
153
154       // obtain meta data for ResultSet
155       metaData = resultSet.getMetaData();
156
157       // determine number of rows in ResultSet
158       resultSet.last(); // move to last row
159       numberOfRows = resultSet.getRow(); // get row number
160
161       // notify JTable that model has changed
162       fireTableStructureChanged();
163    } // end method setQuery
```

Fig. 28.25 | A `TableModel` that supplies `ResultSet` data to a `JTable`. (Part 4 of 5.)

```
164
165    // close Statement and Connection
166    public void disconnectFromDatabase()
167    {
168       if ( connectedToDatabase )
169       {
170          // close Statement and Connection
171          try
172          {
173             resultSet.close();
174             statement.close();
175             connection.close();
176          } // end try
177          catch ( SQLException sqlException )
178          {
179             sqlException.printStackTrace();
180          } // end catch
181          finally  // update database connection status
182          {
183             connectedToDatabase = false;
184          } // end finally
185       } // end if
186    } // end method disconnectFromDatabase
187 } // end class ResultSetTableModel
```

Fig. 28.25 | A `TableModel` that supplies `ResultSet` data to a `JTable`. (Part 5 of 5.)

`ResultSetTableModel` Constructor

The `ResultSetTableModel` constructor (lines 30–46) accepts four `String` arguments—the URL of the database, the username, the password and the default query to perform. The constructor throws any exceptions that occur in its body back to the application that created the `ResultSetTableModel` object, so that the application can determine how to handle the exception (e.g., report an error and terminate the application). Line 34 establishes a connection to the database. Lines 37–39 invoke `Connection` method `createStatement` to create a `Statement` object. This example uses a version of method `createStatement` that takes two arguments—the result set type and the result set concurrency. The **result set type** (Fig. 28.26) specifies whether the `ResultSet`'s cursor is able to scroll in both directions or forward only and whether the `ResultSet` is sensitive to changes made to the underlying data.

`ResultSet` constant	Description
`TYPE_FORWARD_ONLY`	Specifies that a `ResultSet`'s cursor can move only in the forward direction (i.e., from the first to the last row in the `ResultSet`).
`TYPE_SCROLL_INSENSITIVE`	Specifies that a `ResultSet`'s cursor can scroll in either direction and that the changes made to the underlying data during `ResultSet` processing are not reflected in the `ResultSet` unless the program queries the database again.

Fig. 28.26 | `ResultSet` constants for specifying `ResultSet` type. (Part 1 of 2.)

ResultSet constant	Description
TYPE_SCROLL_SENSITIVE	Specifies that a ResultSet's cursor can scroll in either direction and that the changes made to the underlying data during ResultSet processing are reflected immediately in the ResultSet.

Fig. 28.26 | ResultSet constants for specifying ResultSet type. (Part 2 of 2.)

Portability Tip 28.3

Some JDBC drivers do not support scrollable ResultSets. In such cases, the driver typically returns a ResultSet in which the cursor can move only forward. For more information, see your database driver documentation.

Common Programming Error 28.11

Attempting to move the cursor backward through a ResultSet when the database driver does not support backward scrolling causes a SQLFeatureNotSupportedException.

ResultSets that are sensitive to changes reflect those changes immediately after they're made with methods of interface ResultSet. If a ResultSet is insensitive to changes, the query that produced the ResultSet must be executed again to reflect any changes made. The **result set concurrency** (Fig. 28.27) specifies whether the ResultSet can be updated with ResultSet's update methods.

ResultSet static concurrency constant	Description
CONCUR_READ_ONLY	Specifies that a ResultSet cannot be updated (i.e., changes to the ResultSet contents cannot be reflected in the database with ResultSet's update methods).
CONCUR_UPDATABLE	Specifies that a ResultSet can be updated (i.e., changes to its contents can be reflected in the database with ResultSet's update methods).

Fig. 28.27 | ResultSet constants for specifying result properties.

Portability Tip 28.4

Some JDBC drivers do not support updatable ResultSets. In such cases, the driver typically returns a read-only ResultSet. For more information, see your database driver documentation.

Common Programming Error 28.12

Attempting to update a ResultSet when the database driver does not support updatable ResultSets causes SQLFeatureNotSupportedExceptions.

This example uses a ResultSet that is scrollable, insensitive to changes and read only. Line 45 invokes our method setQuery (lines 144–163) to perform the default query.

`ResultSetTableModel` *Method* `getColumnClass`

Method `getColumnClass` (lines 49–69) returns a `Class` object that represents the superclass of all objects in a particular column. The `JTable` uses this information to configure the default cell renderer and cell editor for that column in the `JTable`. Line 58 uses `ResultSetMetaData` method **`getColumnClassName`** to obtain the fully qualified class name for the specified column. Line 61 loads the class and returns the corresponding `Class` object. If an exception occurs, the `catch` in lines 63–66 prints a stack trace and line 68 returns `Object.class`—the `Class` instance that represents class `Object`—as the default type. [*Note:* Line 58 uses the argument `column + 1`. Like arrays, `JTable` row and column numbers are counted from 0. However, `ResultSet` row and column numbers are counted from 1. Thus, when processing `ResultSet` rows or columns for use in a `JTable`, it's necessary to add 1 to the row or column number to manipulate the appropriate `ResultSet` row or column.]

`ResultSetTableModel` *Method* `getColumnCount`

Method `getColumnCount` (lines 72–89) returns the number of columns in the model's underlying `ResultSet`. Line 81 uses `ResultSetMetaData` method **`getColumnCount`** to obtain the number of columns in the `ResultSet`. If an exception occurs, the `catch` in lines 83–86 prints a stack trace and line 88 returns 0 as the default number of columns.

`ResultSetTableModel` *Method* `getColumnName`

Method `getColumnName` (lines 92–109) returns the name of the column in the model's underlying `ResultSet`. Line 101 uses `ResultSetMetaData` method **`getColumnName`** to obtain the column name from the `ResultSet`. If an exception occurs, the `catch` in lines 103–106 prints a stack trace and line 108 returns the empty string as the default column name.

`ResultSetTableModel` *Method* `getRowCount`

Method `getRowCount` (lines 112–119) returns the number of rows in the model's underlying `ResultSet`. When method `setQuery` (lines 144–163) performs a query, it stores the number of rows in variable `numberOfRows`.

`ResultSetTableModel` *Method* `getValueAt`

Method `getValueAt` (lines 122–141) returns the `Object` in a particular row and column of the model's underlying `ResultSet`. Line 132 uses `ResultSet` method **`absolute`** to position the `ResultSet` cursor at a specific row. Line 133 uses `ResultSet` method `getObject` to obtain the `Object` in a specific column of the current row. If an exception occurs, the `catch` in lines 135–138 prints a stack trace and line 140 returns an empty string as the default value.

`ResultSetTableModel` *Method* `setQuery`

Method `setQuery` (lines 144–163) executes the query it receives as an argument to obtain a new `ResultSet` (line 152). Line 155 gets the `ResultSetMetaData` for the new `ResultSet`. Line 158 uses `ResultSet` method **`last`** to position the `ResultSet` cursor at the last row in the `ResultSet`. [*Note:* This can be slow if the table contains many rows.] Line 159 uses `ResultSet` method **`getRow`** to obtain the row number for the current row in the `ResultSet`. Line 162 invokes method **`fireTableStructureChanged`** (inherited from class `AbstractTableModel`) to notify any `JTable` using this `ResultSetTableModel` object as its model that the structure of the model has changed. This causes the `JTable` to repopulate its rows and columns with the new `ResultSet` data. Method `setQuery` throws any exceptions that occur in its body back to the application that invoked `setQuery`.

`ResultSetTableModel` *Method* `disconnectFromDatabase`

Method `disconnectFromDatabase` (lines 166–186) implements an appropriate termination method for class `ResultSetTableModel`. A class designer should provide a `public` method that clients of the class must invoke explicitly to free resources that an object has used. In this case, method `disconnectFromDatabase` closes the `ResultSet`, `Statement` and `Connection` (lines 173–175), which are considered limited resources. Clients of the `ResultSetTableModel` class should always invoke this method when the instance of this class is no longer needed. Before releasing resources, line 168 verifies whether the connection is already terminated. If not, the method proceeds. The other methods in class `ResultSetTableModel` each throw an `IllegalStateException` if `connectedToDatabase` is `false`. Method `disconnectFromDatabase` sets `connectedToDatabase` to `false` (line 183) to ensure that clients do not use an instance of `ResultSetTableModel` after that instance has already been terminated. `IllegalStateException` is an exception from the Java libraries that is appropriate for indicating this error condition.

`DisplayQueryResults` *Class*

Class `DisplayQueryResults` (Fig. 28.28) implements the application's GUI and interacts with the `ResultSetTableModel` via a `JTable` object. This application also demonstrates the `JTable` sorting and filtering capabilities.

```
1  // Fig. 28.28: DisplayQueryResults.java
2  // Display the contents of the Authors table in the books database.
3  import java.awt.BorderLayout;
4  import java.awt.event.ActionListener;
5  import java.awt.event.ActionEvent;
6  import java.awt.event.WindowAdapter;
7  import java.awt.event.WindowEvent;
8  import java.sql.SQLException;
9  import java.util.regex.PatternSyntaxException;
10 import javax.swing.JFrame;
11 import javax.swing.JTextArea;
12 import javax.swing.JScrollPane;
13 import javax.swing.ScrollPaneConstants;
14 import javax.swing.JTable;
15 import javax.swing.JOptionPane;
16 import javax.swing.JButton;
17 import javax.swing.Box;
18 import javax.swing.JLabel;
19 import javax.swing.JTextField;
20 import javax.swing.RowFilter;
21 import javax.swing.table.TableRowSorter;
22 import javax.swing.table.TableModel;
23
24 public class DisplayQueryResults extends JFrame
25 {
26    // database URL, username and password
27    static final String DATABASE_URL = "jdbc:mysql://localhost/books";
28    static final String USERNAME = "deitel";
29    static final String PASSWORD = "deitel";
```

Fig. 28.28 | Displays contents of the database `books`. (Part 1 of 5.)

```
30
31      // default query retrieves all data from Authors table
32      static final String DEFAULT_QUERY = "SELECT * FROM Authors";
33
34      private ResultSetTableModel tableModel;
35      private JTextArea queryArea;
36
37      // create ResultSetTableModel and GUI
38      public DisplayQueryResults()
39      {
40         super( "Displaying Query Results" );
41
42         // create ResultSetTableModel and display database table
43         try
44         {
45            // create TableModel for results of query SELECT * FROM Authors
46            tableModel = new ResultSetTableModel( DATABASE_URL,
47               USERNAME, PASSWORD, DEFAULT_QUERY );
48
49            // set up JTextArea in which user types queries
50            queryArea = new JTextArea( DEFAULT_QUERY, 3, 100 );
51            queryArea.setWrapStyleWord( true );
52            queryArea.setLineWrap( true );
53
54            JScrollPane scrollPane = new JScrollPane( queryArea,
55               ScrollPaneConstants.VERTICAL_SCROLLBAR_AS_NEEDED,
56               ScrollPaneConstants.HORIZONTAL_SCROLLBAR_NEVER );
57
58            // set up JButton for submitting queries
59            JButton submitButton = new JButton( "Submit Query" );
60
61            // create Box to manage placement of queryArea and
62            // submitButton in GUI
63            Box boxNorth = Box.createHorizontalBox();
64            boxNorth.add( scrollPane );
65            boxNorth.add( submitButton );
66
67            // create JTable based on the tableModel
68            JTable resultTable = new JTable( tableModel );
69
70            JLabel filterLabel = new JLabel( "Filter:" );
71            final JTextField filterText = new JTextField();
72            JButton filterButton = new JButton( "Apply Filter" );
73            Box boxSouth = Box.createHorizontalBox();
74
75            boxSouth.add( filterLabel );
76            boxSouth.add( filterText );
77            boxSouth.add( filterButton );
78
79            // place GUI components on content pane
80            add( boxNorth, BorderLayout.NORTH );
81            add( new JScrollPane( resultTable ), BorderLayout.CENTER );
82            add( boxSouth, BorderLayout.SOUTH );
```

Fig. 28.28 | Displays contents of the database books. (Part 2 of 5.)

```
83
84          // create event listener for submitButton
85          submitButton.addActionListener(
86
87             new ActionListener()
88             {
89                // pass query to table model
90                public void actionPerformed( ActionEvent event )
91                {
92                   // perform a new query
93                   try
94                   {
95                      tableModel.setQuery( queryArea.getText() );
96                   } // end try
97                   catch ( SQLException sqlException )
98                   {
99                      JOptionPane.showMessageDialog( null,
100                        sqlException.getMessage(), "Database error",
101                        JOptionPane.ERROR_MESSAGE );
102
103                     // try to recover from invalid user query
104                     // by executing default query
105                     try
106                     {
107                        tableModel.setQuery( DEFAULT_QUERY );
108                        queryArea.setText( DEFAULT_QUERY );
109                     } // end try
110                     catch ( SQLException sqlException2 )
111                     {
112                        JOptionPane.showMessageDialog( null,
113                           sqlException2.getMessage(), "Database error",
114                           JOptionPane.ERROR_MESSAGE );
115
116                        // ensure database connection is closed
117                        tableModel.disconnectFromDatabase();
118
119                        System.exit( 1 ); // terminate application
120                     } // end inner catch
121                  } // end outer catch
122               } // end actionPerformed
123            }  // end ActionListener inner class
124         ); // end call to addActionListener
125
126         final TableRowSorter< TableModel > sorter =
127            new TableRowSorter< TableModel >( tableModel );
128         resultTable.setRowSorter( sorter );
129         setSize( 500, 250 ); // set window size
130         setVisible( true ); // display window
131
132         // create listener for filterButton
133         filterButton.addActionListener(
134            new ActionListener()
135            {
```

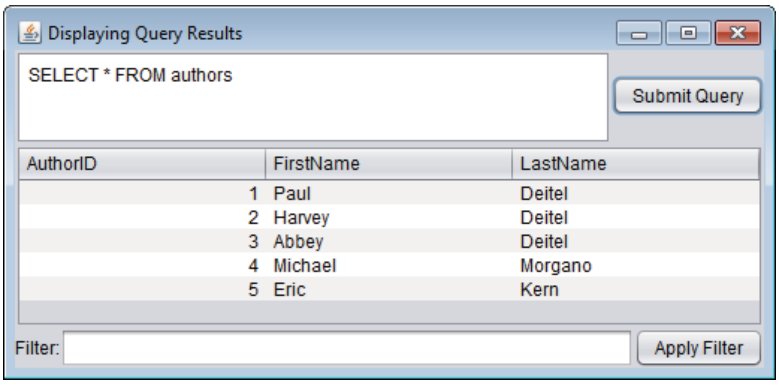
Fig. 28.28 | Displays contents of the database books. (Part 3 of 5.)

```
136                // pass filter text to listener
137                public void actionPerformed( ActionEvent e )
138                {
139                   String text = filterText.getText();
140
141                   if ( text.length() == 0 )
142                      sorter.setRowFilter( null );
143                   else
144                   {
145                      try
146                      {
147                         sorter.setRowFilter(
148                            RowFilter.regexFilter( text ) );
149                      } // end try
150                      catch ( PatternSyntaxException pse )
151                      {
152                         JOptionPane.showMessageDialog( null,
153                            "Bad regex pattern", "Bad regex pattern",
154                            JOptionPane.ERROR_MESSAGE );
155                      } // end catch
156                   } // end else
157                } // end method actionPerfomed
158             } // end annonymous inner class
159          ); // end call to addActionLister
160       } // end try
161       catch ( SQLException sqlException )
162       {
163          JOptionPane.showMessageDialog( null, sqlException.getMessage(),
164             "Database error", JOptionPane.ERROR_MESSAGE );
165
166          // ensure database connection is closed
167          tableModel.disconnectFromDatabase();
168
169          System.exit( 1 ); // terminate application
170       } // end catch
171
172       // dispose of window when user quits application (this overrides
173       // the default of HIDE_ON_CLOSE)
174       setDefaultCloseOperation( DISPOSE_ON_CLOSE );
175
176       // ensure database connection is closed when user quits application
177       addWindowListener(
178
179          new WindowAdapter()
180          {
181             // disconnect from database and exit when window has closed
182             public void windowClosed( WindowEvent event )
183             {
184                tableModel.disconnectFromDatabase();
185                System.exit( 0 );
186             } // end method windowClosed
187          } // end WindowAdapter inner class
```

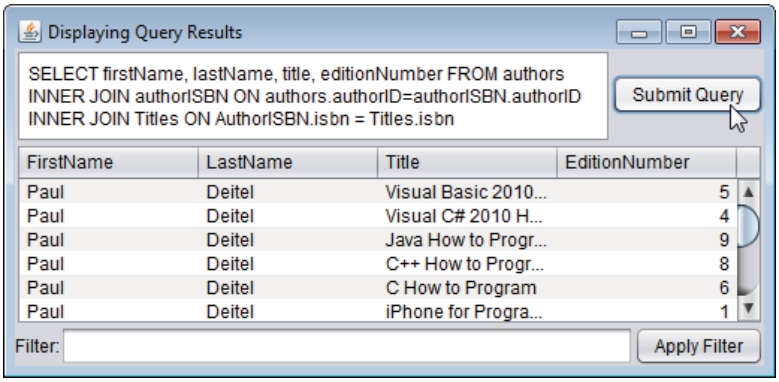
Fig. 28.28 | Displays contents of the database books. (Part 4 of 5.)

```
188         ); // end call to addWindowListener
189      } // end DisplayQueryResults constructor
190
191      // execute application
192      public static void main( String args[] )
193      {
194         new DisplayQueryResults();
195      } // end main
196   } // end class DisplayQueryResults
```

a) Displaying all authors from the `Authors` table



b) Displaying the the authors' first and last names joined with the titles and edition numbers of the books they've authored



c) Filtering the results of the previous query to show only the books with Java in the title

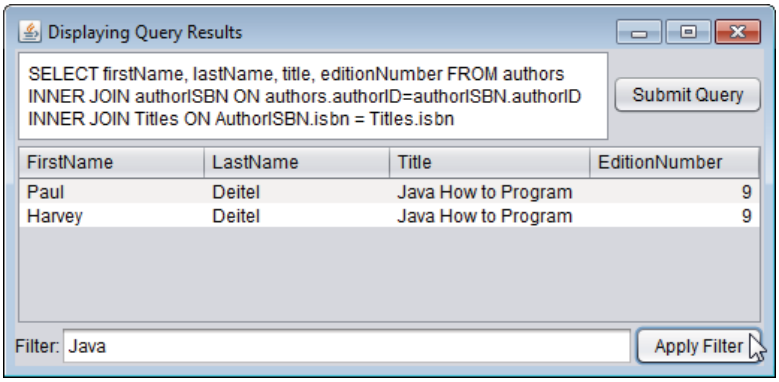


Fig. 28.28 | Displays contents of the database `books`. (Part 5 of 5.)

Lines 27–29 and 32 declare the URL, username, password and default query that are passed to the `ResultSetTableModel` constructor to make the initial connection to the

database and perform the default query. The `DisplayQueryResults` constructor (lines 38–189) creates a `ResultSetTableModel` object and the GUI for the application. Line 68 creates the `JTable` object and passes a `ResultSetTableModel` object to the `JTable` constructor, which then registers the `JTable` as a listener for `TableModelEvents` generated by the `ResultSetTableModel`.

The local variables `filterText` (line 71) and `sorter` (lines 126–127) are declared `final`. These are both used from an event handler that is implemented as an anonymous inner class (lines 134–158). Any local variable that will be used in an anonymous inner class *must* be declared final; otherwise, a compilation error occurs.

Lines 85–124 register an event handler for the `submitButton` that the user clicks to submit a query to the database. When the user clicks the button, method `actionPerformed` (lines 90–122) invokes method `setQuery` from the class `ResultSetTableModel` to execute the new query (line 95). If the user's query fails (e.g., because of a syntax error in the user's input), lines 107–108 execute the default query. If the default query also fails, there could be a more serious error, so line 117 ensures that the database connection is closed and line 119 exits the program. The screen captures in Fig. 28.28 show the results of two queries. The first screen capture shows the default query that retrieves all the data from table `Authors` of database `books`. The second screen capture shows a query that selects each author's first name and last name from the `Authors` table and combines that information with the title and edition number from the `Titles` table. Try entering your own queries in the text area and clicking the **Submit Query** button to execute the query.

Lines 177–188 register a **`WindowListener`** for the **`windowClosed`** event, which occurs when the user closes the window. Since `WindowListeners` can handle several window events, we extend class **`WindowAdapter`** and override only the `windowClosed` event handler.

Sorting Rows in a `JTable`

`JTables` allow users to sort rows by the data in a specific column. Lines 126–127 use the **`TableRowSorter`** class (from package **`javax.swing.table`**) to create an object that uses our `ResultSetTableModel` to sort rows in the `JTable` that displays query results. When the user clicks the title of a particular `JTable` column, the `TableRowSorter` interacts with the underlying `TableModel` to reorder the rows based on the data in that column. Line 128 uses `JTable` method **`setRowSorter`** to specify the `TableRowSorter` for `resultTable`.

Filtering Rows in a `JTable`

`JTables` can now show subsets of the data from the underlying `TableModel`. This is known as filtering the data. Lines 133–159 register an event handler for the `filterButton` that the user clicks to filter the data. In method `actionPerformed` (lines 137–157), line 139 obtains the filter text. If the user did not specify filter text, line 142 uses `JTable` method **`setRowFilter`** to remove any prior filter by setting the filter to `null`. Otherwise, lines 147–148 use `setRowFilter` to specify a **`RowFilter`** (from package `javax.swing`) based on the user's input. Class `RowFilter` provides several methods for creating filters. The `static` method **`regexFilter`** receives a `String` containing a regular expression pattern as its argument and an optional set of indices that specify which columns to filter. If no indices are specified, then all the columns are searched. In this example, the regular expression pattern is the text the user typed. Once the filter is set, the data displayed in the `JTable` is updated based on the filtered `TableModel`.

28.9 RowSet Interface

In the preceding examples, you learned how to query a database by explicitly establishing a `Connection` to the database, preparing a `Statement` for querying the database and executing the query. In this section, we demonstrate the **`RowSet` interface**, which configures the database connection and prepares query statements automatically. The interface `RowSet` provides several *set* methods that allow you to specify the properties needed to establish a connection (such as the database URL, user name and password of the database) and create a `Statement` (such as a query). `RowSet` also provides several *get* methods that return these properties.

Connected and Disconnected `RowSet`*s*

There are two types of `RowSet` objects—connected and disconnected. A **connected `RowSet`** object connects to the database once and remains connected while the object is in use. A **disconnected `RowSet`** object connects to the database, executes a query to retrieve the data from the database and then closes the connection. A program may change the data in a disconnected `RowSet` while it's disconnected. Modified data can be updated in the database after a disconnected `RowSet` reestablishes the connection with the database.

Package **`javax.sql.rowset`** contains two subinterfaces of `RowSet`—`JdbcRowSet` and `CachedRowSet`. **`JdbcRowSet`**, a connected `RowSet`, acts as a wrapper around a `ResultSet` object and allows you to scroll through and update the rows in the `ResultSet`. Recall that by default, a `ResultSet` object is nonscrollable and read only—you must explicitly set the result set type constant to `TYPE_SCROLL_INSENSITIVE` and set the result set concurrency constant to `CONCUR_UPDATABLE` to make a `ResultSet` object scrollable and updatable. A `JdbcRowSet` object is scrollable and updatable by default. **`CachedRowSet`**, a disconnected `RowSet`, caches the data of a `ResultSet` in memory and disconnects from the database. Like `JdbcRowSet`, a `CachedRowSet` object is scrollable and updatable by default. A `CachedRowSet` object is also *serializable*, so it can be passed between Java applications through a network, such as the Internet. However, `CachedRowSet` has a limitation—the amount of data that can be stored in memory is limited. Package `javax.sql.rowset` contains three other subinterfaces of `RowSet`.

Portability Tip 28.5

A `RowSet` can provide scrolling capability for drivers that do not support scrollable `ResultSet`s.

Using a `RowSet`

Figure 28.29 reimplements the example of Fig. 28.23 using a `RowSet`. Rather than establish the connection and create a `Statement` explicitly, Fig. 28.29 uses a `JdbcRowSet` object to create a `Connection` and a `Statement` automatically.

```
1  // Fig. 28.29: JdbcRowSetTest.java
2  // Displaying the contents of the Authors table using JdbcRowSet.
3  import java.sql.ResultSetMetaData;
4  import java.sql.SQLException;
```

Fig. 28.29 | Displaying the `Authors` table using `JdbcRowSet`. (Part 1 of 3.)

```
5  import javax.sql.rowset.JdbcRowSet;
6  import com.sun.rowset.JdbcRowSetImpl; // Sun's JdbcRowSet implementation
7
8  public class JdbcRowSetTest
9  {
10    // JDBC driver name and database URL
11    static final String DATABASE_URL = "jdbc:mysql://localhost/books";
12    static final String USERNAME = "deitel";
13    static final String PASSWORD = "deitel";
14
15    // constructor connects to database, queries database, processes
16    // results and displays results in window
17    public JdbcRowSetTest()
18    {
19       // connect to database books and query database
20       try
21       {
22          // specify properties of JdbcRowSet
23          JdbcRowSet rowSet = new JdbcRowSetImpl();
24          rowSet.setUrl( DATABASE_URL ); // set database URL
25          rowSet.setUsername( USERNAME ); // set username
26          rowSet.setPassword( PASSWORD ); // set password
27          rowSet.setCommand( "SELECT * FROM Authors" ); // set query
28          rowSet.execute(); // execute query
29
30          // process query results
31          ResultSetMetaData metaData = rowSet.getMetaData();
32          int numberOfColumns = metaData.getColumnCount();
33          System.out.println( "Authors Table of Books Database:\n" );
34
35          // display rowset header
36          for ( int i = 1; i <= numberOfColumns; i++ )
37             System.out.printf( "%-8s\t", metaData.getColumnName( i ) );
38          System.out.println();
39
40          // display each row
41          while ( rowSet.next() )
42          {
43             for ( int i = 1; i <= numberOfColumns; i++ )
44                System.out.printf( "%-8s\t", rowSet.getObject( i ) );
45             System.out.println();
46          } // end while
47
48          // close the underlying ResultSet, Statement and Connection
49          rowSet.close();
50       } // end try
51       catch ( SQLException sqlException )
52       {
53          sqlException.printStackTrace();
54          System.exit( 1 );
55       } // end catch
56    } // end DisplayAuthors constructor
57
```

Fig. 28.29 | Displaying the `Authors` table using `JdbcRowSet`. (Part 2 of 3.)

```
58      // launch the application
59      public static void main( String args[] )
60      {
61         JdbcRowSetTest application = new JdbcRowSetTest();
62      } // end main
63   } // end class JdbcRowSetTest
```

```
Authors Table of Books Database:

AuthorID        FirstName       LastName
1               Paul            Deitel
2               Harvey          Deitel
3               Abbey           Deitel
4               Michael         Morgano
5               Eric            Kern
```

Fig. 28.29 | Displaying the `Authors` table using `JdbcRowSet`. (Part 3 of 3.)

The package **`com.sun.rowset`** provides Oracle's reference implementations of the interfaces in package `javax.sql.rowset`. Line 23 uses Sun's reference implementation of the `JdbcRowSet` interface—**`JdbcRowSetImpl`**—to create a `JdbcRowSet` object. We used class `JdbcRowSetImpl` here to demonstrate the capability of the `JdbcRowSet` interface. Other databases may provide their own `RowSet` implementations.

Lines 24–26 set the `RowSet` properties that the `DriverManager` uses to establish a database connection. Line 24 invokes `JdbcRowSet` method **`setUrl`** to specify the database URL. Line 25 invokes `JdbcRowSet` method **`setUsername`** to specify the username. Line 26 invokes `JdbcRowSet` method **`setPassword`** to specify the password. Line 27 invokes `JdbcRowSet` method **`setCommand`** to specify the SQL query that will be used to populate the `RowSet`. Line 28 invokes `JdbcRowSet` method **`execute`** to execute the SQL query. Method `execute` performs four actions—it establishes a `Connection` to the database, prepares the query `Statement`, executes the query and stores the `ResultSet` returned by query. The `Connection`, `Statement` and `ResultSet` are encapsulated in the `JdbcRowSet` object.

The remaining code is almost identical to Fig. 28.23, except that line 31 obtains a `ResultSetMetaData` object from the `JdbcRowSet`, line 41 uses the `JdbcRowSet`'s `next` method to get the next row of the result and line 44 uses the `JdbcRowSet`'s `getObject` method to obtain a column's value. Line 49 invokes `JdbcRowSet` method **`close`**, which closes the `RowSet`'s encapsulated `ResultSet`, `Statement` and `Connection`. In a `CachedRowSet`, invoking `close` also releases the resources held by that `RowSet`. The output of this application is the same as that of Fig. 28.23.

28.10 Java DB/Apache Derby

In this section and Section 28.11, we use Oracle's pure Java database **Java DB**. Please refer to the Before You Begin section after the Preface for information on installing Java DB. Section 28.11 uses the embedded version of Java DB. There's also a network version that executes similarly to the MySQL DBMS introduced earlier in the chapter.

Before you can execute the application in Section 28.11, you must set up the `AddressBook` database in Java DB. For the purpose of the following steps, we assume

you're running Microsoft Windows with Java installed in its default location. Mac OS X and Linux will need to perform similar steps.

1. Java DB comes with several batch files to configure and run it. Before executing these batch files from a command prompt, you must set the environment variable `JAVA_HOME` to refer to the JDK's installation directory—for example, `C:\Program Files\Java\jdk1.6.0_23`. Be sure to use the exact installation directory of the JDK on your computer.
2. Open the batch file `setEmbeddedCP.bat` (typically located in `C:\Program Files\Sun\JavaDB\bin`) in a text editor such as Notepad. Locate the line

```
@rem set DERBY_INSTALL=
```

   and change it to

```
@set DERBY_INSTALL=C:\Program Files\Sun\JavaDB
```

   Save your changes and close this file. [*Note:* You might need to run Notepad as an Administrator to edit this file. To do so, open the Start menu and type `Notepad` in the **Search programs and files** field. Then, right click **Notepad** at the top of the menu and select **Run as administrator**.]
3. Open a Command Prompt as an administrator (as you did for Notepad in the previous step) and change directories to

```
C:\Program Files\Sun\JavaDB\bin
```

   Then, type `setEmbeddedCP.bat` and press *Enter* to set the environment variables required by Java DB.
4. An embedded Java DB database must reside in the same location as the application that manipulates the database. For this reason, change to the directory that contains the code for Figs. 28.30–28.32. This directory contains a SQL script `address.sql` that builds the `AddressBook` database.
5. Execute the command

```
"C:\Program Files\Sun\JavaDB\bin\ij"
```

   to start the command-line tool for interacting with Java DB. The double quotes are necessary because the path contains a space. This will display the `ij>` prompt.
6. At the `ij>` prompt type

```
connect 'jdbc:derby:AddressBook;create=true;user=deitel;
   password=deitel';
```

   and press *Enter* to create the `AddressBook` database in the current directory and to create the user `deitel` with the password `deitel` for accessing the database.
7. To create the database table and insert sample data in it, we've provided the file `address.sql` in this example's directory. To execute this SQL script, type

```
run 'address.sql';
```

8. To terminate the Java DB command-line tool, type

```
exit;
```

You're now ready to execute the `AddressBook` application in Section 28.11. MySQL or any other database that supports JDBC `PreparedStatements` could also be used.

28.11 PreparedStatements

A **`PreparedStatement`** enables you to create compiled SQL statements that execute more efficiently than `Statements`. `PreparedStatements` can also specify parameters, making them more flexible than `Statements`—you can execute the same query repeatedly with different parameter values. For example, in the `books` database, you might want to locate all book titles for an author with a specific last and first name, and you might want to execute that query for several authors. With a `PreparedStatement`, that query is defined as follows:

```
PreparedStatement authorBooks = connection.prepareStatement(
   "SELECT LastName, FirstName, Title " +
   "FROM Authors INNER JOIN AuthorISBN " +
      "ON Authors.AuthorID=AuthorISBN.AuthorID " +
   "INNER JOIN Titles " +
      "ON AuthorISBN.ISBN=Titles.ISBN " +
   "WHERE LastName = ? AND FirstName = ?" );
```

The two question marks (?) in the the preceding SQL statement's last line are placeholders for values that will be passed as part of the query to the database. Before executing a `PreparedStatement`, the program must specify the parameter values by using the `PreparedStatement` interface's *set* methods.

For the preceding query, both parameters are strings that can be set with `PreparedStatement` method **`setString`** as follows:

```
authorBooks.setString( 1, "Deitel" );
authorBooks.setString( 2, "Paul" );
```

Method `setString`'s first argument represents the parameter number being set, and the second argument is that parameter's value. Parameter numbers are *counted from 1*, starting with the first question mark (?). When the program executes the preceding `PreparedStatement` with the parameter values set above, the SQL passed to the database is

```
SELECT LastName, FirstName, Title
FROM Authors INNER JOIN AuthorISBN
   ON Authors.AuthorID=AuthorISBN.AuthorID
INNER JOIN Titles
   ON AuthorISBN.ISBN=Titles.ISBN
WHERE LastName = 'Deitel' AND FirstName = 'Paul'
```

Method `setString` automatically escapes `String` parameter values as necessary. For example, if the last name is O'Brien, the statement

```
authorBooks.setString( 1, "O'Brien" );
```

escapes the `'` character in O'Brien by replacing it with two single-quote characters, so that the `'` appears correctly in the database.

Performance Tip 28.2

`PreparedStatements` are more efficient than `Statements` when executing SQL statements multiple times and with different parameter values.

Error-Prevention Tip 28.2

Use `PreparedStatements` with parameters for queries that receive `String` values as arguments to ensure that the `String`s are quoted properly in the SQL statement.

Interface `PreparedStatement` provides *set* methods for each supported SQL type. It's important to use the *set* method that is appropriate for the parameter's SQL type in the database—`SQLException`s occur when a program attempts to convert a parameter value to an incorrect type.

Address Book Application that Uses `PreparedStatements`

We now present an address book application that enables you to browse existing entries, add new entries and search for entries with a specific last name. Our `AddressBook` Java DB database contains an `Addresses` table with the columns `addressID`, `FirstName`, `LastName`, `Email` and `PhoneNumber`. The column `addressID` is a so-called *identity column*. This is the SQL standard way to represent an *autoincremented column*. The SQL script we provide for this database uses the SQL `IDENTITY` keyword to mark the `addressID` column as an identity column. For more information on using the `IDENTITY` keyword and creating databases, see the Java DB Developer's Guide at `download.oracle.com/javadb/10.6.1.0/devguide/devguide-single.html`.

Class `Person`

Our address book application consists of three classes—`Person` (Fig. 28.30), `PersonQueries` (Fig. 28.31) and `AddressBookDisplay` (Fig. 28.32). Class `Person` is a simple class that represents one person in the address book. The class contains fields for the address ID, first name, last name, email address and phone number, as well as *set* and *get* methods for manipulating these fields.

```
1  // Fig. 28.30: Person.java
2  // Person class that represents an entry in an address book.
3  public class Person
4  {
5     private int addressID;
6     private String firstName;
7     private String lastName;
8     private String email;
9     private String phoneNumber;
10
11    // no-argument constructor
12    public Person()
13    {
14    } // end no-argument Person constructor
15
16    // constructor
17    public Person( int id, String first, String last,
18       String emailAddress, String phone )
19    {
20       setAddressID( id );
21       setFirstName( first );
```

Fig. 28.30 | `Person` class that represents an entry in an `AddressBook`. (Part 1 of 3.)

```
22          setLastName( last );
23          setEmail( emailAddress );
24          setPhoneNumber( phone );
25       } // end five-argument Person constructor
26
27       // sets the addressID
28       public void setAddressID( int id )
29       {
30          addressID = id;
31       } // end method setAddressID
32
33       // returns the addressID
34       public int getAddressID()
35       {
36          return addressID;
37       } // end method getAddressID
38
39       // sets the firstName
40       public void setFirstName( String first )
41       {
42          firstName = first;
43       } // end method setFirstName
44
45       // returns the first name
46       public String getFirstName()
47       {
48          return firstName;
49       } // end method getFirstName
50
51       // sets the lastName
52       public void setLastName( String last )
53       {
54          lastName = last;
55       } // end method setLastName
56
57       // returns the last name
58       public String getLastName()
59       {
60          return lastName;
61       } // end method getLastName
62
63       // sets the email address
64       public void setEmail( String emailAddress )
65       {
66          email = emailAddress;
67       } // end method setEmail
68
69       // returns the email address
70       public String getEmail()
71       {
72          return email;
73       } // end method getEmail
74
```

Fig. 28.30 | `Person` class that represents an entry in an `AddressBook`. (Part 2 of 3.)

```
75      // sets the phone number
76      public void setPhoneNumber( String phone )
77      {
78         phoneNumber = phone;
79      } // end method setPhoneNumber
80
81      // returns the phone number
82      public String getPhoneNumber()
83      {
84         return phoneNumber;
85      } // end method getPhoneNumber
86   } // end class Person
```

Fig. 28.30 | `Person` class that represents an entry in an `AddressBook`. (Part 3 of 3.)

Class `PersonQueries`

Class `PersonQueries` (Fig. 28.31) manages the address book application's database connection and creates the `PreparedStatements` that the application uses to interact with the database. Lines 18–20 declare three `PreparedStatement` variables. The constructor (lines 23–49) connects to the database at lines 27–28.

```
1  // Fig. 28.31: PersonQueries.java
2  // PreparedStatements used by the Address Book application.
3  import java.sql.Connection;
4  import java.sql.DriverManager;
5  import java.sql.PreparedStatement;
6  import java.sql.ResultSet;
7  import java.sql.SQLException;
8  import java.util.List;
9  import java.util.ArrayList;
10
11 public class PersonQueries
12 {
13    private static final String URL = "jdbc:derby:AddressBook";
14    private static final String USERNAME = "deitel";
15    private static final String PASSWORD = "deitel";
16
17    private Connection connection = null; // manages connection
18    private PreparedStatement selectAllPeople = null;
19    private PreparedStatement selectPeopleByLastName = null;
20    private PreparedStatement insertNewPerson = null;
21
22    // constructor
23    public PersonQueries()
24    {
25       try
26       {
27          connection =
28             DriverManager.getConnection( URL, USERNAME, PASSWORD );
29
```

Fig. 28.31 | `PreparedStatements` used by the Address Book application. (Part 1 of 4.)

```
30            // create query that selects all entries in the AddressBook
31            selectAllPeople =
32               connection.prepareStatement( "SELECT * FROM Addresses" );
33
34            // create query that selects entries with a specific last name
35            selectPeopleByLastName = connection.prepareStatement(
36               "SELECT * FROM Addresses WHERE LastName = ?" );
37
38            // create insert that adds a new entry into the database
39            insertNewPerson = connection.prepareStatement(
40               "INSERT INTO Addresses " +
41               "( FirstName, LastName, Email, PhoneNumber ) " +
42               "VALUES ( ?, ?, ?, ? )" );
43         } // end try
44         catch ( SQLException sqlException )
45         {
46            sqlException.printStackTrace();
47            System.exit( 1 );
48         } // end catch
49      } // end PersonQueries constructor
50
51      // select all of the addresses in the database
52      public List< Person > getAllPeople()
53      {
54         List< Person > results = null;
55         ResultSet resultSet = null;
56
57         try
58         {
59            // executeQuery returns ResultSet containing matching entries
60            resultSet = selectAllPeople.executeQuery();
61            results = new ArrayList< Person >();
62
63            while ( resultSet.next() )
64            {
65               results.add( new Person(
66                  resultSet.getInt( "addressID" ),
67                  resultSet.getString( "FirstName" ),
68                  resultSet.getString( "LastName" ),
69                  resultSet.getString( "Email" ),
70                  resultSet.getString( "PhoneNumber" ) ) );
71            } // end while
72         } // end try
73         catch ( SQLException sqlException )
74         {
75            sqlException.printStackTrace();
76         } // end catch
77         finally
78         {
79            try
80            {
81               resultSet.close();
82            } // end try
```

Fig. 28.31 | `PreparedStatements` used by the Address Book application. (Part 2 of 4.)

```
83            catch ( SQLException sqlException )
84            {
85               sqlException.printStackTrace();
86               close();
87            } // end catch
88         } // end finally
89
90         return results;
91      } // end method getAllPeople
92
93      // select person by last name
94      public List< Person > getPeopleByLastName( String name )
95      {
96         List< Person > results = null;
97         ResultSet resultSet = null;
98
99         try
100        {
101           selectPeopleByLastName.setString( 1, name ); // specify last name
102
103           // executeQuery returns ResultSet containing matching entries
104           resultSet = selectPeopleByLastName.executeQuery();
105
106           results = new ArrayList< Person >();
107
108           while ( resultSet.next() )
109           {
110              results.add( new Person( resultSet.getInt( "addressID" ),
111                 resultSet.getString( "FirstName" ),
112                 resultSet.getString( "LastName" ),
113                 resultSet.getString( "Email" ),
114                 resultSet.getString( "PhoneNumber" ) ) );
115           } // end while
116        } // end try
117        catch ( SQLException sqlException )
118        {
119           sqlException.printStackTrace();
120        } // end catch
121        finally
122        {
123           try
124           {
125              resultSet.close();
126           } // end try
127           catch ( SQLException sqlException )
128           {
129              sqlException.printStackTrace();
130              close();
131           } // end catch
132        } // end finally
133
134        return results;
135     } // end method getPeopleByName
```

Fig. 28.31 | `PreparedStatements` used by the Address Book application. (Part 3 of 4.)

```
136
137    // add an entry
138    public int addPerson(
139       String fname, String lname, String email, String num )
140    {
141       int result = 0;
142
143       // set parameters, then execute insertNewPerson
144       try
145       {
146          insertNewPerson.setString( 1, fname );
147          insertNewPerson.setString( 2, lname );
148          insertNewPerson.setString( 3, email );
149          insertNewPerson.setString( 4, num );
150
151          // insert the new entry; returns # of rows updated
152          result = insertNewPerson.executeUpdate();
153       } // end try
154       catch ( SQLException sqlException )
155       {
156          sqlException.printStackTrace();
157          close();
158       } // end catch
159
160       return result;
161    } // end method addPerson
162
163    // close the database connection
164    public void close()
165    {
166       try
167       {
168          connection.close();
169       } // end try
170       catch ( SQLException sqlException )
171       {
172          sqlException.printStackTrace();
173       } // end catch
174    } // end method close
175 } // end class PersonQueries
```

Fig. 28.31 | `PreparedStatements` used by the Address Book application. (Part 4 of 4.)

Creating `PreparedStatements`

Lines 31–32 invoke `Connection` method **`prepareStatement`** to create the `PreparedStatement` named `selectAllPeople` that selects all the rows in the `Addresses` table. Lines 35–36 create the `PreparedStatement` named `selectPeopleByLastName` with a parameter. This statement selects all the rows in the `Addresses` table that match a particular last name. Notice the `?` character that's used to specify the last-name parameter. Lines 39–42 create the `PreparedStatement` named `insertNewPerson` with four parameters that represent the first name, last name, email address and phone number for a new entry. Again, notice the `?` characters used to represent these parameters.

`PersonQueries` *Method* `getAllPeople`

Method `getAllPeople` (lines 52–91) executes `PreparedStatement` `selectAllPeople` (line 60) by calling method **`executeQuery`**, which returns a `ResultSet` containing the rows that match the query (in this case, all the rows in the `Addresses` table). Lines 61–71 place the query results in an `ArrayList` of `Person` objects, which is returned to the caller at line 90. Method `getPeopleByLastName` (lines 94–135) uses `PreparedStatement` method `setString` to set the parameter to `selectPeopleByLastName` (line 101). Then, line 104 executes the query and lines 106–115 place the query results in an `ArrayList` of `Person` objects. Line 134 returns the `ArrayList` to the caller.

`PersonQueries` *Methods* `addPerson` *and* `Close`

Method `addPerson` (lines 138–161) uses `PreparedStatement` method `setString` (lines 146–149) to set the parameters for the `insertNewPerson` `PreparedStatement`. Line 152 uses `PreparedStatement` method **`executeUpdate`** to insert the new record. This method returns an integer indicating the number of rows that were updated (or inserted) in the database. Method `close` (lines 164–174) simply closes the database connection.

Class `AddressBookDisplay`

The `AddressBookDisplay` (Fig. 28.32) application uses a `PersonQueries` object to interact with the database. Line 59 creates the `PersonQueries` object. When the user presses the **Browse All Entries** `JButton`, the `browseButtonActionPerformed` handler (lines 309–335) is called. Line 313 calls the method `getAllPeople` on the `PersonQueries` object to obtain all the entries in the database. The user can then scroll through the entries using the **Previous** and **Next** `JButtons`. When the user presses the **Find** `JButton`, the `queryButtonActionPerformed` handler (lines 265–287) is called. Lines 267–268 call method `getPeopleByLastName` on the `PersonQueries` object to obtain the entries in the database that match the specified last name. If there are several such entries, the user can then scroll through them using the **Previous** and **Next** `JButtons`.

```
1  // Fig. 28.32: AddressBookDisplay.java
2  // A simple address book
3  import java.awt.event.ActionEvent;
4  import java.awt.event.ActionListener;
5  import java.awt.event.WindowAdapter;
6  import java.awt.event.WindowEvent;
7  import java.awt.FlowLayout;
8  import java.awt.GridLayout;
9  import java.util.List;
10 import javax.swing.JButton;
11 import javax.swing.Box;
12 import javax.swing.JFrame;
13 import javax.swing.JLabel;
14 import javax.swing.JPanel;
15 import javax.swing.JTextField;
16 import javax.swing.WindowConstants;
17 import javax.swing.BoxLayout;
18 import javax.swing.BorderFactory;
19 import javax.swing.JOptionPane;
```

Fig. 28.32 | A simple address book. (Part 1 of 9.)

```
20
21  public class AddressBookDisplay extends JFrame
22  {
23     private Person currentEntry;
24     private PersonQueries personQueries;
25     private List< Person > results;
26     private int numberOfEntries = 0;
27     private int currentEntryIndex;
28
29     private JButton browseButton;
30     private JLabel emailLabel;
31     private JTextField emailTextField;
32     private JLabel firstNameLabel;
33     private JTextField firstNameTextField;
34     private JLabel idLabel;
35     private JTextField idTextField;
36     private JTextField indexTextField;
37     private JLabel lastNameLabel;
38     private JTextField lastNameTextField;
39     private JTextField maxTextField;
40     private JButton nextButton;
41     private JLabel ofLabel;
42     private JLabel phoneLabel;
43     private JTextField phoneTextField;
44     private JButton previousButton;
45     private JButton queryButton;
46     private JLabel queryLabel;
47     private JPanel queryPanel;
48     private JPanel navigatePanel;
49     private JPanel displayPanel;
50     private JTextField queryTextField;
51     private JButton insertButton;
52
53     // no-argument constructor
54     public AddressBookDisplay()
55     {
56        super( "Address Book" );
57
58        // establish database connection and set up PreparedStatements
59        personQueries = new PersonQueries();
60
61        // create GUI
62        navigatePanel = new JPanel();
63        previousButton = new JButton();
64        indexTextField = new JTextField( 2 );
65        ofLabel = new JLabel();
66        maxTextField = new JTextField( 2 );
67        nextButton = new JButton();
68        displayPanel = new JPanel();
69        idLabel = new JLabel();
70        idTextField = new JTextField( 10 );
71        firstNameLabel = new JLabel();
72        firstNameTextField = new JTextField( 10 );
```

Fig. 28.32 | A simple address book. (Part 2 of 9.)

```
73          lastNameLabel = new JLabel();
74          lastNameTextField = new JTextField( 10 );
75          emailLabel = new JLabel();
76          emailTextField = new JTextField( 10 );
77          phoneLabel = new JLabel();
78          phoneTextField = new JTextField( 10 );
79          queryPanel = new JPanel();
80          queryLabel = new JLabel();
81          queryTextField = new JTextField( 10 );
82          queryButton = new JButton();
83          browseButton = new JButton();
84          insertButton = new JButton();
85
86          setLayout( new FlowLayout( FlowLayout.CENTER, 10, 10 ) );
87          setSize( 400, 300 );
88          setResizable( false );
89
90          navigatePanel.setLayout(
91             new BoxLayout( navigatePanel, BoxLayout.X_AXIS ) );
92
93          previousButton.setText( "Previous" );
94          previousButton.setEnabled( false );
95          previousButton.addActionListener(
96             new ActionListener()
97             {
98                public void actionPerformed( ActionEvent evt )
99                {
100                  previousButtonActionPerformed( evt );
101               } // end method actionPerformed
102            } // end anonymous inner class
103         ); // end call to addActionListener
104
105         navigatePanel.add( previousButton );
106         navigatePanel.add( Box.createHorizontalStrut( 10 ) );
107
108         indexTextField.setHorizontalAlignment(
109            JTextField.CENTER );
110         indexTextField.addActionListener(
111            new ActionListener()
112            {
113               public void actionPerformed( ActionEvent evt )
114               {
115                  indexTextFieldActionPerformed( evt );
116               } // end method actionPerformed
117            } // end anonymous inner class
118         ); // end call to addActionListener
119
120         navigatePanel.add( indexTextField );
121         navigatePanel.add( Box.createHorizontalStrut( 10 ) );
122
123         ofLabel.setText( "of" );
124         navigatePanel.add( ofLabel );
125         navigatePanel.add( Box.createHorizontalStrut( 10 ) );
```

Fig. 28.32 | A simple address book. (Part 3 of 9.)

```
126
127        maxTextField.setHorizontalAlignment(
128           JTextField.CENTER );
129        maxTextField.setEditable( false );
130        navigatePanel.add( maxTextField );
131        navigatePanel.add( Box.createHorizontalStrut( 10 ) );
132
133        nextButton.setText( "Next" );
134        nextButton.setEnabled( false );
135        nextButton.addActionListener(
136           new ActionListener()
137           {
138              public void actionPerformed( ActionEvent evt )
139              {
140                 nextButtonActionPerformed( evt );
141              } // end method actionPerformed
142           } // end anonymous inner class
143        ); // end call to addActionListener
144
145        navigatePanel.add( nextButton );
146        add( navigatePanel );
147
148        displayPanel.setLayout( new GridLayout( 5, 2, 4, 4 ) );
149
150        idLabel.setText( "Address ID:" );
151        displayPanel.add( idLabel );
152
153        idTextField.setEditable( false );
154        displayPanel.add( idTextField );
155
156        firstNameLabel.setText( "First Name:" );
157        displayPanel.add( firstNameLabel );
158        displayPanel.add( firstNameTextField );
159
160        lastNameLabel.setText( "Last Name:" );
161        displayPanel.add( lastNameLabel );
162        displayPanel.add( lastNameTextField );
163
164        emailLabel.setText( "Email:" );
165        displayPanel.add( emailLabel );
166        displayPanel.add( emailTextField );
167
168        phoneLabel.setText( "Phone Number:" );
169        displayPanel.add( phoneLabel );
170        displayPanel.add( phoneTextField );
171        add( displayPanel );
172
173        queryPanel.setLayout(
174           new BoxLayout( queryPanel, BoxLayout.X_AXIS) );
175
176        queryPanel.setBorder( BorderFactory.createTitledBorder(
177           "Find an entry by last name" ) );
178        queryLabel.setText( "Last Name:" );
```

Fig. 28.32 | A simple address book. (Part 4 of 9.)

```
179         queryPanel.add( Box.createHorizontalStrut( 5 ) );
180         queryPanel.add( queryLabel );
181         queryPanel.add( Box.createHorizontalStrut( 10 ) );
182         queryPanel.add( queryTextField );
183         queryPanel.add( Box.createHorizontalStrut( 10 ) );
184
185         queryButton.setText( "Find" );
186         queryButton.addActionListener(
187            new ActionListener()
188            {
189               public void actionPerformed( ActionEvent evt )
190               {
191                  queryButtonActionPerformed( evt );
192               } // end method actionPerformed
193            } // end anonymous inner class
194         ); // end call to addActionListener
195
196         queryPanel.add( queryButton );
197         queryPanel.add( Box.createHorizontalStrut( 5 ) );
198         add( queryPanel );
199
200         browseButton.setText( "Browse All Entries" );
201         browseButton.addActionListener(
202            new ActionListener()
203            {
204               public void actionPerformed( ActionEvent evt )
205               {
206                  browseButtonActionPerformed( evt );
207               } // end method actionPerformed
208            } // end anonymous inner class
209         ); // end call to addActionListener
210
211         add( browseButton );
212
213         insertButton.setText( "Insert New Entry" );
214         insertButton.addActionListener(
215            new ActionListener()
216            {
217               public void actionPerformed( ActionEvent evt )
218               {
219                  insertButtonActionPerformed( evt );
220               } // end method actionPerformed
221            } // end anonymous inner class
222         ); // end call to addActionListener
223
224         add( insertButton );
225
226         addWindowListener(
227            new WindowAdapter()
228            {
229               public void windowClosing( WindowEvent evt )
230               {
231                  personQueries.close(); // close database connection
```

Fig. 28.32 | A simple address book. (Part 5 of 9.)

```
232                System.exit( 0 );
233             } // end method windowClosing
234          } // end anonymous inner class
235       ); // end call to addWindowListener
236
237       setVisible( true );
238    } // end no-argument constructor
239
240    // handles call when previousButton is clicked
241    private void previousButtonActionPerformed( ActionEvent evt )
242    {
243       currentEntryIndex--;
244
245       if ( currentEntryIndex < 0 )
246          currentEntryIndex = numberOfEntries - 1;
247
248       indexTextField.setText( "" + ( currentEntryIndex + 1 ) );
249       indexTextFieldActionPerformed( evt );
250    } // end method previousButtonActionPerformed
251
252    // handles call when nextButton is clicked
253    private void nextButtonActionPerformed( ActionEvent evt )
254    {
255       currentEntryIndex++;
256
257       if ( currentEntryIndex >= numberOfEntries )
258          currentEntryIndex = 0;
259
260       indexTextField.setText( "" + ( currentEntryIndex + 1 ) );
261       indexTextFieldActionPerformed( evt );
262    } // end method nextButtonActionPerformed
263
264    // handles call when queryButton is clicked
265    private void queryButtonActionPerformed( ActionEvent evt )
266    {
267       results =
268          personQueries.getPeopleByLastName( queryTextField.getText() );
269       numberOfEntries = results.size();
270
271       if ( numberOfEntries != 0 )
272       {
273          currentEntryIndex = 0;
274          currentEntry = results.get( currentEntryIndex );
275          idTextField.setText( "" + currentEntry.getAddressID() );
276          firstNameTextField.setText( currentEntry.getFirstName() );
277          lastNameTextField.setText( currentEntry.getLastName() );
278          emailTextField.setText( currentEntry.getEmail() );
279          phoneTextField.setText( currentEntry.getPhoneNumber() );
280          maxTextField.setText( "" + numberOfEntries );
281          indexTextField.setText( "" + ( currentEntryIndex + 1 ) );
282          nextButton.setEnabled( true );
283          previousButton.setEnabled( true );
284       } // end if
```

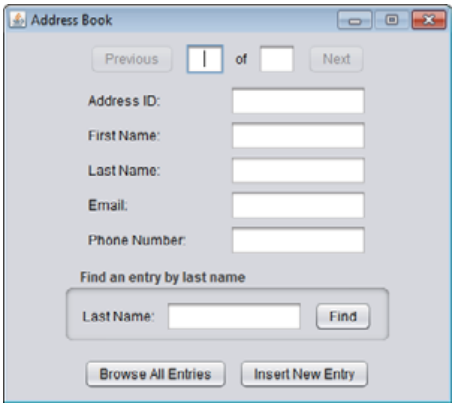
Fig. 28.32 | A simple address book. (Part 6 of 9.)

```
285         else
286            browseButtonActionPerformed( evt );
287      } // end method queryButtonActionPerformed
288
289      // handles call when a new value is entered in indexTextField
290      private void indexTextFieldActionPerformed( ActionEvent evt )
291      {
292         currentEntryIndex =
293            ( Integer.parseInt( indexTextField.getText() ) - 1 );
294
295         if ( numberOfEntries != 0 && currentEntryIndex < numberOfEntries )
296         {
297            currentEntry = results.get( currentEntryIndex );
298            idTextField.setText("" + currentEntry.getAddressID() );
299            firstNameTextField.setText( currentEntry.getFirstName() );
300            lastNameTextField.setText( currentEntry.getLastName() );
301            emailTextField.setText( currentEntry.getEmail() );
302            phoneTextField.setText( currentEntry.getPhoneNumber() );
303            maxTextField.setText( "" + numberOfEntries );
304            indexTextField.setText( "" + ( currentEntryIndex + 1 ) );
305         } // end if
306       } // end method indexTextFieldActionPerformed
307
308      // handles call when browseButton is clicked
309      private void browseButtonActionPerformed( ActionEvent evt )
310      {
311         try
312         {
313            results = personQueries.getAllPeople();
314            numberOfEntries = results.size();
315
316            if ( numberOfEntries != 0 )
317            {
318               currentEntryIndex = 0;
319               currentEntry = results.get( currentEntryIndex );
320               idTextField.setText( "" + currentEntry.getAddressID() );
321               firstNameTextField.setText( currentEntry.getFirstName() );
322               lastNameTextField.setText( currentEntry.getLastName() );
323               emailTextField.setText( currentEntry.getEmail() );
324               phoneTextField.setText( currentEntry.getPhoneNumber() );
325               maxTextField.setText( "" + numberOfEntries );
326               indexTextField.setText( "" + ( currentEntryIndex + 1 ) );
327               nextButton.setEnabled( true );
328               previousButton.setEnabled( true );
329            } // end if
330         } // end try
331         catch ( Exception e )
332         {
333            e.printStackTrace();
334         } // end catch
335      } // end method browseButtonActionPerformed
336
```

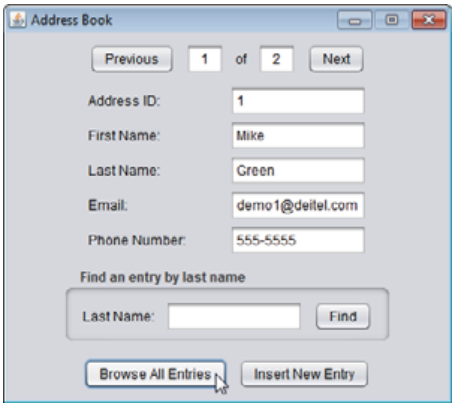
Fig. 28.32 | A simple address book. (Part 7 of 9.)

```
337    // handles call when insertButton is clicked
338    private void insertButtonActionPerformed( ActionEvent evt )
339    {
340       int result = personQueries.addPerson( firstNameTextField.getText(),
341          lastNameTextField.getText(), emailTextField.getText(),
342          phoneTextField.getText() );
343
344       if ( result == 1 )
345          JOptionPane.showMessageDialog( this, "Person added!",
346             "Person added", JOptionPane.PLAIN_MESSAGE );
347       else
348          JOptionPane.showMessageDialog( this, "Person not added!",
349             "Error", JOptionPane.PLAIN_MESSAGE );
350
351       browseButtonActionPerformed( evt );
352    } // end method insertButtonActionPerformed
353
354    // main method
355    public static void main( String args[] )
356    {
357       new AddressBookDisplay();
358    } // end method main
359 } // end class AddressBookDisplay
```

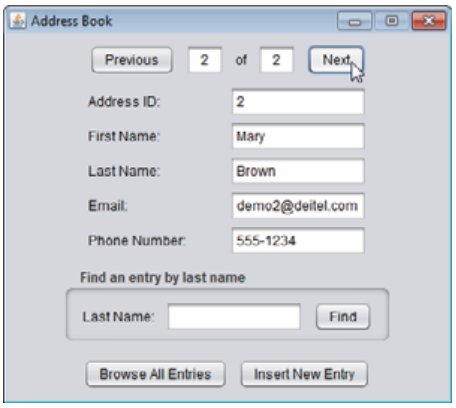
a) Initial **Address Book** screen.



b) Results of clicking **Browse All Entries**.



c) Browsing to the next entry.



d) Finding entries with the last name `Green`.

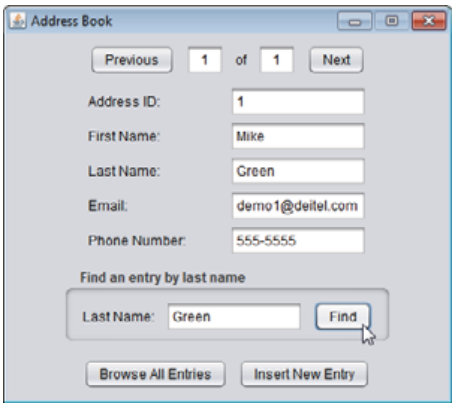


Fig. 28.32 | A simple address book. (Part 8 of 9.)

e) After adding a new entry and browsing to it.

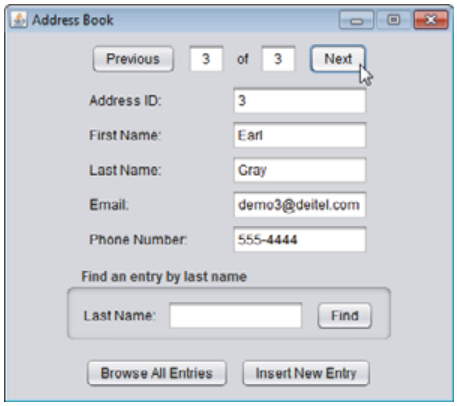


Fig. 28.32 | A simple address book. (Part 9 of 9.)

To add a new entry into the `AddressBook` database, the user can enter the first name, last name, email and phone number (the `AddressID` will *autoincrement*) in the `JTextFields` and press the **Insert New Entry** `JButton`. The `insertButtonActionPerformed` handler (lines 338–352) is called. Lines 340–342 call the method `addPerson` on the `PersonQueries` object to add a new entry to the database. Line 351 calls `browseButtonActionPerformed` to obtain the updated set of people in the address book and update the GUI accordingly.

The user can then view different entries by pressing the **Previous** `JButton` or **Next** `JButton`, which results in calls to methods `previousButtonActionPerformed` (lines 241–250) or `nextButtonActionPerformed` (lines 253–262), respectively. Alternatively, the user can enter a number in the `indexTextField` and press *Enter* to view a particular entry. This results in a call to method `indexTextFieldActionPerformed` (lines 290–306) to display the specified record.

28.12 Stored Procedures

Many database management systems can store individual or sets of SQL statements in a database, so that programs accessing that database can invoke them. Such named collections of SQL statements are called **stored procedures**. JDBC enables programs to invoke stored procedures using objects that implement the interface **`CallableStatement`**. `CallableStatements` can receive arguments specified with the methods inherited from interface `PreparedStatement`. In addition, `CallableStatements` can specify **output parameters** in which a stored procedure can place return values. Interface `CallableStatement` includes methods to specify which parameters in a stored procedure are output parameters. The interface also includes methods to obtain the values of output parameters returned from a stored procedure.

Portability Tip 28.6

Although the syntax for creating stored procedures differs across database management systems, the interface `CallableStatement` provides a uniform interface for specifying input and output parameters for stored procedures and for invoking stored procedures.

Portability Tip 28.7

According to the Java API documentation for interface `CallableStatement`*, for maximum portability between database systems, programs should process the update counts (which indicate how many rows were updated) or* `ResultSets` *returned from a* `CallableStatement` *before obtaining the values of any output parameters.*

28.13 Transaction Processing

Many database applications require guarantees that a series of database insertions, updates and deletions executes properly before the application continues processing the next database operation. For example, when you transfer money electronically between bank accounts, several factors determine if the transaction is successful. You begin by specifying the source account and the amount you wish to transfer from that account to a destination account. Next, you specify the destination account. The bank checks the source account to determine whether its funds are sufficient to complete the transfer. If so, the bank withdraws the specified amount and, if all goes well, deposits it into the destination account to complete the transfer. What happens if the transfer fails after the bank withdraws the money from the source account? In a proper banking system, the bank redeposits the money in the source account. How would you feel if the money was subtracted from your source account and the bank *did not* deposit the money in the destination account?

Transaction processing enables a program that interacts with a database to *treat a database operation (or set of operations) as a single operation*. Such an operation also is known as an **atomic operation** or a **transaction**. At the end of a transaction, a decision can be made either to **commit the transaction** or **roll back the transaction**. Committing the transaction finalizes the database operation(s); all insertions, updates and deletions performed as part of the transaction cannot be reversed without performing a new database operation. Rolling back the transaction leaves the database in its state prior to the database operation. This is useful when a portion of a transaction fails to complete properly. In our bank-account-transfer discussion, the transaction would be rolled back if the deposit could not be made into the destination account.

Java provides transaction processing via methods of interface `Connection`. Method **`setAutoCommit`** specifies whether each SQL statement commits after it completes (a `true` argument) or whether several SQL statements should be grouped as a transaction (a `false` argument). If the argument to `setAutoCommit` is `false`, the program must follow the last SQL statement in the transaction with a call to `Connection` method **`commit`** (to commit the changes to the database) or `Connection` method **`rollback`** (to return the database to its state prior to the transaction). Interface `Connection` also provides method **`getAutoCommit`** to determine the autocommit state for the `Connection`.

28.14 Wrap-Up

In this chapter, you learned basic database concepts, how to query and manipulate data in a database using SQL and how to use JDBC to allow Java applications to interact with MySQL and Java DB databases. You learned about the SQL commands `SELECT`, `INSERT`, `UPDATE` and `DELETE`, as well as clauses such as `WHERE`, `ORDER BY` and `INNER JOIN`. You learned the steps for obtaining a `Connection` to the database, creating a `Statement` to interact with the database's data, executing the statement and processing the results. Then you used a

`RowSet` to simplify the process of connecting to a database and creating statements. You used `PreparedStatements` to create precompiled SQL statements. You also learned how to create and configure databases in both MySQL and Java DB by using predefined SQL scripts. We also provided overviews of `CallableStatements` and transaction processing. In the next chapter, you'll learn about web application development with JavaServer Faces.

28.15 Web Resources

`www.oracle.com/technetwork/java/javadb/overview/index.html`
Oracle Java DB home page.

`db.apache.org/derby/papers/DerbyTut/index.html`
Apache Derby tutorial. Includes Linux installation instructions.

`download.oracle.com/javase/tutorial/jdbc/index.html`
The Java Tutorial's JDBC track.

`www.sql.org`
This SQL portal provides links to many resources, including SQL syntax, tips, tutorials, books, magazines, discussion groups, companies with SQL services, SQL consultants and free software.

`download.oracle.com/javase/6/docs/technotes/guides/jdbc/index.html`
Oracle JDBC API documentation.

`www.mysql.com`
This site is the MySQL database home page. You can download the latest versions of MySQL and MySQL Connector/J and access their online documentation.

`dev.mysql.com/doc/refman/5.5/en/index.html`
MySQL reference manual.

`download.oracle.com/javase/6/docs/technotes/guides/jdbc/getstart/rowsetImpl.html`
Overviews the `RowSet` interface and its subinterfaces. This site also discusses the reference implementations of these interfaces from Sun and their usage.