Chapter 24
Accessing Databases with JDBC

Java How to Program, 10/e

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.

24.1 Introduction

- A database is an organized collection of data.
- A database management system (DBMS) provides mechanisms for storing, organizing, retrieving and modifying data for many users.
- SQL is the international standard language used with relational databases to perform queries and to manipulate data.
- Popular relational database management systems (RDBMSs)
  - Microsoft SQL Server®
  - Oracle®
  - Sybase®
  - IBM DB2®
  - Informix®
  - PostgreSQL
  - MySQL™

24.1 Introduction (cont.)

- Java programs interact with databases 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 24.1

The JDBC API is portable—the same code can manipulate databases in various RDBMSs.

24.1 Introduction (cont.)

Java Persistence API (JPA)

- In online Chapter 29, we introduce Java Persistence API (JPA).
- In that chapter, you'll learn how to autogenerate Java classes that represent the tables in a database and the relationships between them—known as object-relational mapping—then use objects of those classes to interact with a database.
- As you'll see, storing data in and retrieving data from a database will be handled for you—the techniques you learn in this chapter will typically be hidden from you by JPA.

24.2 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.
- Tables are composed of rows, each describing a single entity—in Fig. 24.1, an employee.
- Rows are composed of columns in which values are stored.
- Primary key—a column (or group of columns) with a value that is unique for each row.

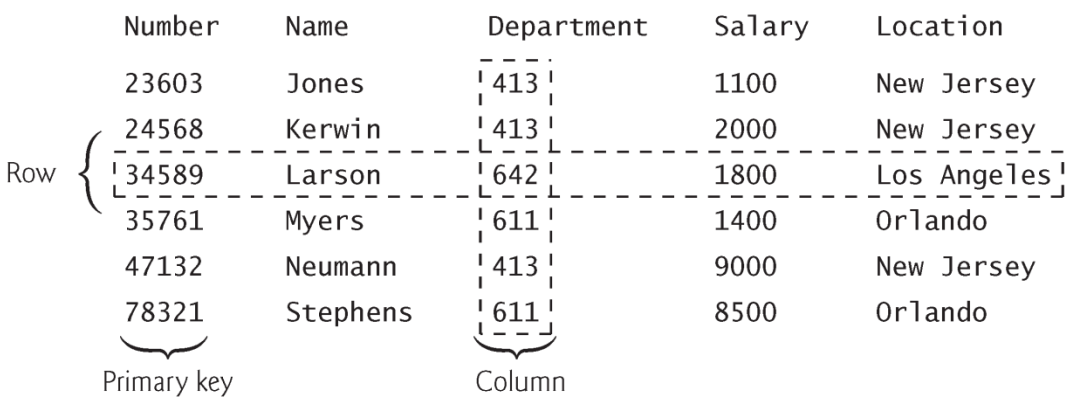


Fig. 24.1 | `Employee` table sample data.

Department	Location
413	New Jersey
611	Orlando
642	Los Angeles

Fig. 24.2 | Distinct `Department` and `Location` data from the `Employees` table.

24.3 A books Database

- We introduce relational databases in the context of this chapter's `books` database, which you'll use in several examples.
- The database consists of three tables: `Authors`, `AuthorISBN` and `Titles`.
- `AuthorISBN` table consists of two columns that maintain ISBNs for each book and their corresponding authors' ID numbers.
- The `AuthorID` column is a foreign key—a column in this table that matches the primary-key column in another table.
- Every foreign-key value must appear as another table's primary-key value so the DBMS can ensure that the foreign key value is valid—this is known as the Rule of Referential Integrity.
- There is a one-to-many relationship between a primary key and a corresponding foreign key.

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. Autoincremented columns are so-called identity columns. The SQL script we provide for this database uses the SQL `IDENTITY` keyword to mark the `AuthorID` column as an identity column. For more information on using the `IDENTITY` keyword and creating databases, see the Java DB Developer's Guide at `http://docs.oracle.com/javadb/10.10.1.1/devguide/derbydev.pdf`.
`FirstName`	Author's first name (a string).
`LastName`	Author's last name (a string).

Fig. 24.3 | `Authors` table from the `books` database.

AuthorID	FirstName	LastName
1	Paul	Deitel
2	Harvey	Deitel
3	Abbey	Deitel
4	Dan	Quirk
5	Michael	Morgano

Fig. 24.4 | Sample data from the `Authors` table.

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. 24.5 | `Titles` table from the `books` database.

ISBN	Title	EditionNumber	Copyright
0132151006	Internet & World Wide Web How to Program	5	2012
0133807800	Java How to Program	10	2015
0132575655	Java How to Program, Late Objects Version	10	2015
013299044X	C How to Program	7	2013
0132990601	Simply Visual Basic 2010	4	2013
0133406954	Visual Basic 2012 How to Program	6	2014
0133379337	Visual C# 2012 How to Program	5	2014
0136151574	Visual C++ 2008 How to Program	2	2008
0133378713	C++ How to Program	9	2014
0133570924	Android How to Program	2	2015
0133570924	Android for Programmers: An App-Driven Approach, Volume 1	2	2014

Fig. 24.6 | Sample data from the `Titles` table of the **books** database (Part 1 of 2.).

ISBN	Title	EditionNumber	Copyright
0132121360	Android for Programmers: An App-Driven Approach	1	2012

Fig. 24.6 | Sample data from the `Titles` table of the **books** database (Part 2 of 2.).

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. 24.7 | `AuthorISBN` table from the `books` database.

AuthorID	ISBN	AuthorID	ISBN
1	0132151006	2	0133379337
2	0132151006	1	0136151574
3	0132151006	2	0136151574
1	0133807800	4	0136151574
2	0133807800	1	0133378713
1	0132575655	2	0133378713
2	0132575655	1	0133764036
1	013299044X	2	0133764036
2	013299044X	3	0133764036
1	0132990601	1	0133570924
2	0132990601	2	0133570924
3	0132990601	3	0133570924

Fig. 24.8 | Sample data from the `AuthorISBN` table of `books`. (Part 1 of 2.)

AuthorID	ISBN	AuthorID	ISBN
1	0133406954	1	0132121360
2	0133406954	2	0132121360
3	0133406954	3	0132121360
1	0133379337	5	0132121360

Fig. 24.8 | Sample data from the `AuthorISBN` table of `books`. (Part 2 of 2.)

24.3 Relational Database Overview: The books Database (cont.)

- Entity-relationship (ER) diagram for the books database.
  - Shows the *database tables* and the *relationships* among them.
  - 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.

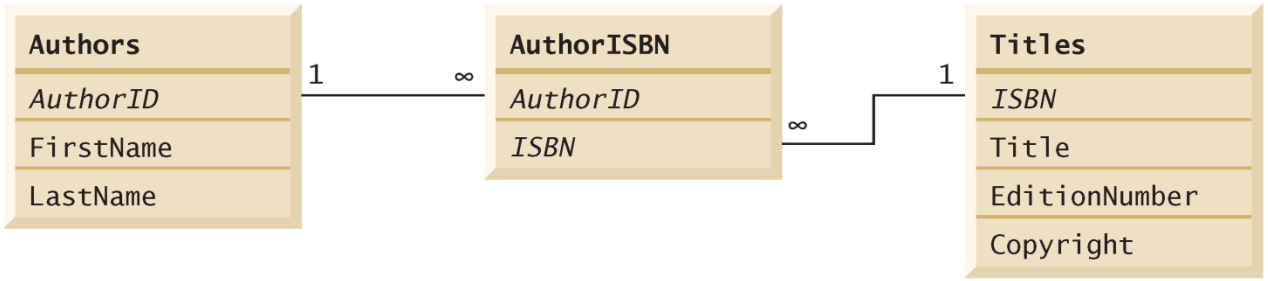


Fig. 24.9 | Table relationships in the **books** database.

24.4 SQL

- The next several subsections discuss the SQL queries and statements using the SQL keywords.
- Other SQL keywords are beyond this text's scope.

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. 24.10 | SQL query keywords.

24.4.1 Basic SELECT Query

- A SQL query "selects" rows and columns from one or more tables in a database.
- 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.
- To retrieve all the data in the `Authors` table, use
  - `SELECT * FROM Authors`
- To retrieve only specific columns, replace the asterisk (*) with a comma-separated list of the column names, e.g.,
  - `SELECT AuthorID, LastName FROM Authors`

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

Fig. 24.11 | Sample `AuthorID` and `LastName` data from the `Authors` table.

Software Engineering Observation 24.2

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 24.1

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.

24.4.2 WHERE Clause

- In most cases, only rows that satisfy selection criteria 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*
- To select the `Title`, `EditionNumber` and `Copyright` columns from table `Titles` for which the `Copyright` date is greater than `2013`, use the query

```
SELECT Title, EditionNumber, Copyright
   FROM Titles
   WHERE Copyright > '2013'
```

- Strings in SQL are delimited by single (') rather than double (") quotes.
- The `WHERE` clause criteria can contain the operators <, >, <=, >=, =, <> and `LIKE`.

Title	EditionNumber	Copyright
Java How to Program	10	2015
Java How to Program, Late Objects Version	10	2015
Visual Basic 2012 How to Program	6	2014
Visual C# 2012 How to Program	5	2014
C++ How to Program	9	2014
Android How to Program	2	2015
Android for Programmers: An App-Driven Approach, Volume 1	2	2014

Fig. 24.12 | Sampling of titles with copyrights after 2005 from table `Titles`.

24.4.2 WHERE Clause (cont.)

- Operator LIKE is used for pattern matching with wildcard characters percent (%) and underscore (_).
- 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%'
    ```
    ```

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

Fig. 24.13 | Authors whose last name starts with `D` from the `Authors` table.

Portability Tip 24.1

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 24.2

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

24.4.2 WHERE Clause (cont.)

- An underscore (_) in the `LIKE` pattern string indicates a single wildcard character at that position in the pattern.
- 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%'
    ```
    ```

AuthorID	FirstName	LastName
5	Michael	Morgano

Fig. 24.14 | The only author from the `Authors` table whose last name contains `o` as the second letter.

24.4.3 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
  - ASC specifies ascending order (lowest to highest)
  - DESC specifies descending order (highest to lowest)
  - *column* specifies the column on which the sort is based.

24.4.3 ORDER BY Clause (cont.)

- To obtain the list of authors in ascending order by last name (), use the query
  - ```
    SELECT AuthorID, FirstName, LastName
        FROM Authors
        ORDER BY LastName ASC
    ```
- To obtain the same list of authors in descending order by last name (), use the query
  - ```
    SELECT AuthorID, FirstName, LastName
        FROM Authors
        ORDER BY LastName DESC
    ```
- Multiple columns can be used for sorting with an ORDER BY clause of the form
  - ORDER BY *column1 sortingOrder*, *column2 sortingOrder*, …
  - *sortingOrder* is either ASC or DESC.
- Sort all the rows in ascending order by last name, then by first name.
  - ```
    SELECT AuthorID, FirstName, LastName
        FROM Authors
        ORDER BY LastName, FirstName
    ```
    ```

AuthorID	FirstName	LastName
1	Paul	Deitel
2	Harvey	Deitel
3	Abbey	Deitel
5	Michael	Morgano
4	Dan	Quirk

Fig. 24.15 | Sample data from table `Authors` in ascending order by `LastName`.

AuthorID	FirstName	LastName
4	Dan	Quirk
5	Michael	Morgano
1	Paul	Deitel
2	Harvey	Deitel
3	Abbey	Deitel

Fig. 24.16 | Sample data from table `Authors` in descending order by `LastName`.

AuthorID	FirstName	LastName
3	Abbey	Deitel
2	Harvey	Deitel
1	Paul	Deitel
5	Michael	Morgano
4	Dan	Quirk

Fig. 24.17 | Sample data from `Authors` in ascending order by `LastName` and `FirstName`.

24.4.3 ORDER BY Clause (cont.)

- 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`.

ISBN	Title	EditionNumber	Copyright
0133764036	Android How to Program	2	2015
013299044X	C How to Program	7	2013
0133378713	C++ How to Program	9	2014
0132151006	Internet & World Wide Web How to Program	5	2012
0133807800	Java How to Program	10	2015
0133406954	Visual Basic 2012 How to Program	6	2014
0133379337	Visual C# 2012 How to Program	5	2014
0136151574	Visual C++ 2008 How to Program	2	2008

Fig. 24.18 | Sampling of books from table `Titles` whose titles end with `How to Program` in ascending order by `Title`.

24.4.4 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.
- Often, it is necessary to merge data from multiple tables into a single result.
  - Referred to as joining the tables
- An INNER JOIN merges rows from two tables by matching values in columns that are common to the tables.
  - ```
    SELECT columnName1, columnName2, ...
    FROM table1
    INNER JOIN table2
        ON table1.columnName = table2.columnName
    ```
- The ON clause specifies the columns from each table that are compared to determine which rows are merged—these fields almost always correspond to the foreign-key fields in the tables being joined.
    ```

24.4.4 Merging Data from Multiple Tables: INNER JOIN (cont.)

- 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 syntax *tableName*.*columnName* in the ON clause, called a qualified name, specifies the columns from each table that should be compared to join the tables.

Common Programming Error 24.2

Failure to qualify names for columns that have the same name in two or more tables is an error. In such cases, the statement must precede those column names with their table names and a dot (e.g., `Authors.AuthorID`*).*

FirstName	LastName	ISBN	FirstName	LastName	ISBN
Abbey	Deitel	0132121360	Harvey	Deitel	0133764036
Abbey	Deitel	0133570924	Harvey	Deitel	0133378713
Abbey	Deitel	0133764036	Harvey	Deitel	0136151574
Abbey	Deitel	0133406954	Harvey	Deitel	0133379337
Abbey	Deitel	0132990601	Harvey	Deitel	0133406954
Abbey	Deitel	0132151006	Harvey	Deitel	0132990601
Harvey	Deitel	0132121360	Harvey	Deitel	013299044X
Harvey	Deitel	0133570924	Harvey	Deitel	0132575655
Harvey	Deitel	0133807800	Paul	Deitel	0133406954
Harvey	Deitel	0132151006	Paul	Deitel	0132990601
Paul	Deitel	0132121360	Paul	Deitel	013299044X
Paul	Deitel	0133570924	Paul	Deitel	0132575655
Paul	Deitel	0133764036	Paul	Deitel	0133807800
Paul	Deitel	0133378713	Paul	Deitel	0132151006
Paul	Deitel	0136151574	Michael	Morgano	0132121360
Paul	Deitel	0133379337	Dan	Quirk	0136151574

Fig. 24.19 | Sampling of authors and ISBNs for the books they have written in ascending order by `LastName` and `FirstName`.

24.4.5 INSERT Statement

- The INSERT statement inserts a row into a table.
  - `INSERT INTO` *tableName* `(`*columnName1*`,` *columnName2*`, ...,` *columnNameN*`)`
    `VALUES (`*value1*`,` *value2*`, ...,` *valueN*`)`

  where *tableName* is the table in which to insert the row.
  - *tableName* is followed by a comma-separated list of column names in parentheses
  - 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.

24.4.5 INSERT Statement (cont.)

- The `INSERT` statement
  - `INSERT INTO Authors (FirstName, LastName)`
    `VALUES ('Sue', 'Red')`
- indicates that values are provided for the `FirstName` and `LastName` columns. The corresponding values are `'Sue'` and `'Red'`.
- We do not specify an `AuthorID` in this example because `AuthorID` is an autoincremented column in the `Authors` table.
  - Not every database management system supports autoincremented columns.

Common Programming Error 24.3

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.

Common Programming Error 24.4

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

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

Fig. 24.20 | Sample data from table `Authors` after an `INSERT` operation.

24.4.6 UPDATE Statement

- An UPDATE statement modifies data in a table.
  - UPDATE *tableName*
    SET *columnName1* = *value1*, *columnName2* = *value2*, …, *columnNameN* = *valueN*
    WHERE criteria
- where *tableName* is the table to update.
  - *tableName* is followed by keyword SET and a comma-separated list of *columnName* = *value* pairs.
  - Optional WHERE clause provides criteria that determine which rows to update.
- The UPDATE statement-
  -
```
UPDATE Authors
    SET LastName = 'Black'
    WHERE LastName = 'Red' AND FirstName = 'Sue'
```
- indicates that `LastName` will be assigned the value `Jones` for the row where `LastName` is `Red` and `FirstName` is `Sue`.

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

Fig. 24.21 | Sample data from table `Authors` after an `UPDATE` operation.

24.4.7 DELETE Statement

- A SQL `DELETE` statement removes rows from a table.
  - `DELETE FROM` *tableName* `WHERE` *criteria*
- where *tableName* is the table from which to delete.
  - 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 Jones in the `Authors` table.

AuthorID	FirstName	LastName
1	Paul	Deitel
2	Harvey	Deitel
3	Abbey	Deitel
4	Dan	Quirk
5	Michael	Morgano

Fig. 24.22 | Sample data from table `Authors` after a `DELETE` operation.

24.5 Setting up a Java DB Database

- This chapter's examples use Oracle's pure Java database Java DB, which is installed with Oracle's JDK on Windows, Mac OS X and Linux.
- For this chapter, you'll be using the embedded version of Java DB.
  - The database you manipulate in each example must be located in that example's folder.
  - This chapter's examples are located in two subfolders of the `ch24` examples folder—`books_examples` and `addressbook_example`.

24.5 Setting up a Java DB Database (Cont.)

JDK Installation Folders

- The Java DB software is located in the `db` subdirectory of your JDK's installation directory. The directories listed below are for Oracle's JDK 7 update 51:
  - 32-bit JDK on Windows:
    - `C:\Program Files (x86)\Java\jdk1.7.0_51`
  - 64-bit JDK on Windows:
    - `C:\Program Files\Java\jdk1.7.0_51`
  - Mac OS X:
    - `/Library/Java/JavaVirtualMachines/jdk1.7.0_51.jdk/Contents/Home`
  - Ubuntu Linux:
    - `/usr/lib/jvm/java-7-oracle`

24.5 Setting up a Java DB Database (Cont.)

- For Linux, the install location depends on the installer you use and possibly the version of Linux that you use. We used Ubuntu Linux for testing purposes.
- Depending on your platform, the JDK installation folder's name might differ if you're using a different update of JDK 7 or using JDK 8.
- Java DB comes with several files that enable you to configure and run it.
- Before executing these files from a command window, you must set the environment variable `JAVA_HOME` to refer to the JDK's exact installation directory listed above (or the location where you installed the JDK if it differs from those listed above).
- See the Before You Begin section of this book for information on setting environment variables.

24.5.1 Creating the Chapter's Databases on Windows

- After setting the `JAVA_HOME` environment variable, perform the following steps:
  - Run Notepad as an administrator. To do this on Windows 7, select **Start > All Programs > Accessories**, right click Notepad and select **Run as administrator**. On Windows 8, search for Notepad, right click it in the search results and select Advanced in the app bar, then select **Run as administrator**.
  - From Notepad, open the batch file `setEmbeddedCP.bat` that is located in the JDK installation folder's `db\bin` folder.
  - Locate the line
    - `@rem set DERBY_INSTALL=`
  - and change it to
    - `@set DERBY_INSTALL=%JAVA_HOME%\db`
  - Save your changes and close this file.

24.5.1 Creating the Chapter's Databases on Windows (Cont.)

- Open a Command Prompt window and change directories to the JDK installation folder's `db\bin` folder. Then, type `setEmbeddedCP.bat` and press *Enter* to set the environment variables required by Java DB.
- Use the `cd` command to change to this chapter's `books_examples` directory. This directory contains a SQL script `books.sql` that builds the books database.
- Execute the following command (with the quotation marks):
  - `"%JAVA_HOME%\db\bin\ij"`
- to start the command-line tool for interacting with Java DB. The double quotes are necessary because the path that the environment variable `%JAVA_HOME%` represents contains a space. This will display the `ij>` prompt.

24.5.1 Creating the Chapter's Databases on Windows (Cont.)

- At the `ij>` prompt type
  - `connect 'jdbc:derby:books;create=true;user=deitel; password=deitel';`
- and press *Enter* to create the `books` database in the current directory and to create the user `deitel` with the password `deitel` for accessing the database.
- 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 'books.sql';`
- Once you create the database, you can execute the SQL statements presented in Section 24.4 to confirm their execution.
- Each command you enter at the `ij>` prompt must be terminated with a semicolon (`;`).

24.5.1 Creating the Chapter's Databases on Windows (Cont.)

- To terminate the Java DB command-line tool, type
  - `exit;`
- Change directories to the `addressbook_example` subfolder of the `ch24` examples folder, which contains the SQL script `addressbook.sql` that builds the `addressbook` database.
- Repeat Steps 6–9. In each step, replace `books` with `addressbook`.

24.5.2 Creating the Chapter's Databases on Mac OS X

- After setting the `JAVA_HOME` environment variable, perform the following steps:
  - Open a Terminal, then type:
    - `DERBY_HOME=/Library/Java/JavaVirtualMachines/jdk1.7.0_51.jdk/Contents/Home/db`
  - and press *Enter*. Then type
    - `export DERBY_HOME`
  - and press *Enter*. This specifies where Java DB is located on your Mac.

24.5.2 Creating the Chapter's Databases on Mac OS X (Cont.)

- In the Terminal window, change directories to the JDK installation folder's `db/bin` folder. Then, type `./setEmbeddedCP` and press *Enter* to set the environment variables required by Java DB.
- In the Terminal window, use the `cd` command to change to the `books_examples` directory. This directory contains a SQL script `books.sql` that builds the books database.
- Execute the following command (with the quotation marks):
  - `$JAVA_HOME/db/bin/ij`
- to start the command-line tool for interacting with Java DB. This will display the `ij>` prompt.

24.5.2 Creating the Chapter's Databases on Mac OS X (Cont.)

- Perform Steps 7–9 of Section 24.5.1 to create the `books` database.
- Use the `cd` command to change to the `addressbook_example` directory. This directory contains a SQL script `addressbook.sql` that builds the `addressbook` database.
- Perform Steps 7–9 of Section 24.5.1 to create the `addressbook` database. In each step, replace books with `addressbook`.

24.5.3 Creating the Chapter's Databases on Linux

- After setting the `JAVA_HOME` environment variable, perform the following steps:
  - Open a shell window.
  - Perform the steps in Section 24.5.2, but in Step 1, set `DERBY_HOME` to
    - `DERBY_HOME=YourLinuxJDKInstallationFolder/db`
  - On our Ubuntu Linux system, this was:
    - `DERBY_HOME=/usr/lib/jvm/java-7-oracle/db`

24.6 Manipulating Databases with JDBC

- In this section, we present two examples.
- The first introduces how to connect to a database and query the database.
- The second demonstrates how to display the result of the query in a `JTable`.

24.6.1 Connecting to and Querying a Database

- The example of illustrates connecting to the database, querying the database and processing the result.

```
1  // Fig. 24.23: 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    public static void main(String args[])
13    {
14       final String DATABASE_URL = "jdbc:derby:books";
15       final String SELECT_QUERY =
16          "SELECT authorID, firstName, lastName FROM authors";
17
18       // use try-with-resources to connect to and query the database
19       try (
20          Connection connection = DriverManager.getConnection(
21             DATABASE_URL, "deitel", "deitel");
22          Statement statement = connection.createStatement();
23          ResultSet resultSet = statement.executeQuery(SELECT_QUERY))
```

Fig. 24.23 | Displaying the contents of the `Authors` table. (Part 1 of 3.)

```
24          {
25             // get ResultSet's meta data
26             ResultSetMetaData metaData = resultSet.getMetaData();
27             int numberOfColumns = metaData.getColumnCount();
28
29             System.out.printf("Authors Table of Books Database:%n%n");
30
31             // display the names of the columns in the ResultSet
32             for (int i = 1; i <= numberOfColumns; i++)
33                System.out.printf("%-8s\t", metaData.getColumnName(i));
34             System.out.println();
35
36             // display query results
37             while (resultSet.next())
38             {
39                for (int i = 1; i <= numberOfColumns; i++)
40                   System.out.printf("%-8s\t", resultSet.getObject(i));
41                System.out.println();
42             }
43          } // AutoCloseable objects' close methods are called now
```

Fig. 24.23 | Displaying the contents of the `Authors` table. (Part 2 of 3.)

```
44         catch (SQLException sqlException)
45         {
46            sqlException.printStackTrace();
47         }
48      }
49   } // end class DisplayAuthors
```

```
Authors Table of Books Database:

AUTHORID        FIRSTNAME       LASTNAME
1               Paul            Deitel
2               Harvey          Deitel
3               Abbey           Deitel
4               Dan             Quirk
5               Michael         Morgano
```

Fig. 24.23 | Displaying the contents of the `Authors` table. (Part 3 of 3.)

24.6.1 Connecting to and Querying a Database (cont.)

- The database URL identifies the name of the database to connect to, as well as information about the protocol used by the JDBC driver.
- JDBC supports automatic driver discovery
  - It loads the database driver into memory for you.
  - 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.

24.6.1 Connecting to and Querying a Database (cont.)

Connecting to the Database

- The JDBC interfaces we use in this example each extend the `AutoCloseable` interface, so you can use objects that implement these interfaces with the `try-with-resources` statement (introduced in Section 11.12).
- Each object created in the parentheses following keyword `try` must be separated from the next by a semicolon (`;`).

24.6.1 Connecting to and Querying a Database (cont.)

- `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.

24.6.1 Connecting to and Querying a Database (cont.)

- The URL `jdbc:derby:books` specifies
  - the protocol for communication (`jdbc`)
  - the subprotocol for communication (`derby`)
  - the location of the database (`books`).
- The subprotocol `derby` indicates that the program uses a Java DB/Apache Derby-specific subprotocol to connect to the database.

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. 24.24 | Popular JDBC database URL formats.

Software Engineering Observation 24.3

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 username and password to gain access.*

24.6.1 Connecting to and Querying a Database (cont.)

- `Connection` method createStatement obtains an object that implements interface `Statement` (package `java.sql`).
  - Used to submit SQL statements to the database.
- The `Statement` object's executeQuery method submits a query to the database.
  - Returns an object that implements interface ResultSet and contains the query results.
  - The `ResultSet` methods enable the program to manipulate the query result.
- A `ResultSet`'s ResultSetMetaData describes the `ResultSet`'s contents.
  - Can be used programatically to obtain information about the `ResultSet`'s column names and types.
- `ResultSetMetaData` method getColumnCount retrieves the number of columns in the `ResultSet`.

Software Engineering Observation 24.4

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

24.6.1 Connecting to and Querying a Database (cont.)

- The first call to `ResultSet` method next positions the `ResultSet` cursor to the first row
  - Returns `boolean` value `true` if it is able to position to the next row; otherwise, the method returns `false`.
- `ResultSetMetaData` method getColumnType returns a constant integer from class Types (package `java.sql`) indicating the type of a specified column.
- `ResultSet` method getInt can be used to get 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.
- `ResultSet` method getObject prints the `Object`'s `String` representation.

Common Programming Error 24.5

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`*.*

Performance Tip 24.1

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 24.1

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.*

Common Programming Error 24.6

Specifying column index 0 when obtaining values from a `ResultSet` *causes a* `SQLException`*—the first column index in a* `ResultSet` *is always 1.*

Common Programming Error 24.7

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 24.5

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.*

24.6.2 Querying the books Database

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

24.8.2 Querying the books Database (cont.)

- `ResultSetTableModel` overrides `TableModel` methods getColumnClass, getColumnCount, getColumnName, getRowCount and getValueAt (inherited from AbstractTableModel),

```
1  // Fig. 24.25: 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;
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 final Connection connection;
20    private final Statement statement;
21    private ResultSet resultSet;
22    private ResultSetMetaData metaData;
23    private int numberOfRows;
```

Fig. 24.25 | A `TableModel` that supplies `ResultSet` data to a `JTable`. (Part 1 of 9.)

```
24
25      // keep track of database connection status
26      private boolean connectedToDatabase = false;
27
28      // constructor initializes resultSet and obtains its metadata 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      }
```

Fig. 24.25 | A `TableModel` that supplies `ResultSet` data to a `JTable`. (Part 2 of 9.)

```
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
60            // return Class object that represents className
61            return Class.forName(className);
62         }
63         catch (Exception exception)
64         {
65            exception.printStackTrace();
66         }
67
68         return Object.class; // if problems occur above, assume type Object
69      }
```

Fig. 24.25 | A `TableModel` that supplies `ResultSet` data to a `JTable`. (Part 3 of 9.)

```
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         }
83         catch (SQLException sqlException)
84         {
85            sqlException.printStackTrace();
86         }
87
88         return 0; // if problems occur above, return 0 for number of columns
89      }
90
```

Fig. 24.25 | A `TableModel` that supplies `ResultSet` data to a `JTable`. (Part 4 of 9.)

```
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        }
103        catch (SQLException sqlException)
104        {
105           sqlException.printStackTrace();
106        }
107
108        return ""; // if problems, return empty string for column name
109     }
110
```

Fig. 24.25 | A `TableModel` that supplies `ResultSet` data to a `JTable`. (Part 5 of 9.)

```
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     }
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
```

Fig. 24.25 | A `TableModel` that supplies `ResultSet` data to a `JTable`. (Part 6 of 9.)

```
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         }
135         catch (SQLException sqlException)
136         {
137            sqlException.printStackTrace();
138         }
139
140         return ""; // if problems, return empty string object
141      }
```

Fig. 24.25 | A `TableModel` that supplies `ResultSet` data to a `JTable`. (Part 7 of 9.)

```
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 metadata 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     }
```

Fig. 24.25 | A `TableModel` that supplies `ResultSet` data to a `JTable`. (Part 8 of 9.)

```
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           }
177           catch (SQLException sqlException)
178           {
179              sqlException.printStackTrace();
180           }
181           finally  // update database connection status
182           {
183              connectedToDatabase = false;
184           }
185        }
186     }
187  } // end class ResultSetTableModel
```

Fig. 24.25 | A `TableModel` that supplies `ResultSet` data to a `JTable`. (Part 9 of 9.)

24.6.2 Querying the books Database (cont.)

- `Connection` method `createStatement` with two arguments receives the result set type and the result set concurrency.
- The result set type () 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`s that are sensitive to changes reflect those changes immediately after they are 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 () specifies whether the `ResultSet` can be updated with `ResultSet`'s update methods.

`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.
`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. 24.26 | `ResultSet` constants for specifying `ResultSet` type.

Portability Tip 24.3

Some JDBC drivers do not support scrollable `ResultSet`*s. 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 24.8

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

`ResultSet` static concurrency constant	Description
`CONCUR_READ_ONLY`	Specifies that a `ResultSet` can't be updated—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. 24.27 | `ResultSet` constants for specifying result properties.

Portability Tip 24.4

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

Common Programming Error 24.9

Attempting to update a `ResultSet` *when the database driver does not support updatable* `ResultSet`*s causes* `SQLFeatureNotSupportedException`*s.*

24.6.2 Querying the books Database (cont.)

- `ResultSetMetaData` method getColumnClassName obtains the fully qualified class name for the specified column.
- `ResultSetMetaData` method getColumnCount obtains the number of columns in the `ResultSet`.
- `ResultSetMetaData` method getColumnName obtains the column name from the `ResultSet`.
- `ResultSet` method absolute positions the `ResultSet` cursor at a specific row.
- `ResultSet` method last positions the `ResultSet` cursor at the last row in the `ResultSet`.
- `ResultSet` method getRow obtains the row number for the current row in the `ResultSet`.
- Method fireTableStructureChanged (inherited from class `AbstractTableModel`) notifies any `JTable` using this `ResultSetTableModel` object as its model that the structure of the model has changed.
  - Causes the `JTable` to repopulate its rows and columns with the new `ResultSet` data.

```
1  // Fig. 24.28: 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;
```

Fig. 24.28 | Display the contents of the `Authors` table in the `books` database. (Part 1 of 12.)

```
23
24  public class DisplayQueryResults extends JFrame
25  {
26     // database URL, username and password
27     private static final String DATABASE_URL = "jdbc:derby:books";
28     private static final String USERNAME = "deitel";
29     private static final String PASSWORD = "deitel";
30
31     // default query retrieves all data from Authors table
32     private static final String DEFAULT_QUERY = "SELECT * FROM Authors";
33
34     private static ResultSetTableModel tableModel;
35
36     public static void main(String args[])
37     {
38        // create ResultSetTableModel and display database table
39        try
40        {
41           // create TableModel for results of query SELECT * FROM Authors
42           tableModel = new ResultSetTableModel(DATABASE_URL,
43              USERNAME, PASSWORD, DEFAULT_QUERY);
44
```

Fig. 24.28 | Display the contents of the `Authors` table in the `books` database. (Part 2 of 12.)

```
45          // set up JTextArea in which user types queries
46          final JTextArea queryArea = new JTextArea(DEFAULT_QUERY, 3, 100);
47          queryArea.setWrapStyleWord(true);
48          queryArea.setLineWrap(true);
49
50          JScrollPane scrollPane = new JScrollPane(queryArea,
51             ScrollPaneConstants.VERTICAL_SCROLLBAR_AS_NEEDED,
52             ScrollPaneConstants.HORIZONTAL_SCROLLBAR_NEVER);
53
54          // set up JButton for submitting queries
55          JButton submitButton = new JButton("Submit Query");
56
57          // create Box to manage placement of queryArea and
58          // submitButton in GUI
59          Box boxNorth = Box.createHorizontalBox();
60          boxNorth.add(scrollPane);
61          boxNorth.add(submitButton);
62
63          // create JTable based on the tableModel
64          JTable resultTable = new JTable(tableModel);
65
```

Fig. 24.28 | Display the contents of the `Authors` table in the `books` database. (Part 3 of 12.)

```
66          JLabel filterLabel = new JLabel("Filter:");
67          final JTextField filterText = new JTextField();
68          JButton filterButton = new JButton("Apply Filter");
69          Box boxSouth = Box.createHorizontalBox();
70
71          boxSouth.add(filterLabel);
72          boxSouth.add(filterText);
73          boxSouth.add(filterButton);
74
75          // place GUI components on JFrame's content pane
76          JFrame window = new JFrame("Displaying Query Results");
77          add(boxNorth, BorderLayout.NORTH);
78          add(new JScrollPane(resultTable), BorderLayout.CENTER);
79          add(boxSouth, BorderLayout.SOUTH);
80
```

Fig. 24.28 | Display the contents of the `Authors` table in the `books` database. (Part 4 of 12.)

```
81          // create event listener for submitButton
82          submitButton.addActionListener(
83             new ActionListener()
84             {
85                public void actionPerformed(ActionEvent event)
86                {
87                   // perform a new query
88                   try
89                   {
90                      tableModel.setQuery(queryArea.getText());
91                   }
92                   catch (SQLException sqlException)
93                   {
94                      JOptionPane.showMessageDialog(null,
95                         sqlException.getMessage(), "Database error",
96                         JOptionPane.ERROR_MESSAGE);
97
```

Fig. 24.28 | Display the contents of the **Authors** table in the **books** database. (Part 5 of 12.)

```
98                      // try to recover from invalid user query
99                      // by executing default query
100                     try
101                     {
102                        tableModel.setQuery(DEFAULT_QUERY);
103                        queryArea.setText(DEFAULT_QUERY);
104                     }
105                     catch (SQLException sqlException2)
106                     {
107                        JOptionPane.showMessageDialog(null,
108                           sqlException2.getMessage(), "Database error",
109                           JOptionPane.ERROR_MESSAGE);
110
111                        // ensure database connection is closed
112                        tableModel.disconnectFromDatabase();
113
114                        System.exit(1); // terminate application
115                     }
116                  }
117               }
118            }
119         );
```

Fig. 24.28 | Display the contents of the `Authors` table in the `books` database. (Part 6 of 12.)

```
120
121          final TableRowSorter<TableModel> sorter =
122             new TableRowSorter<TableModel>(tableModel);
123          resultTable.setRowSorter(sorter);
124
125          // create listener for filterButton
126          filterButton.addActionListener(
127             new ActionListener()
128             {
129                // pass filter text to listener
130                public void actionPerformed(ActionEvent e)
131                {
132                   String text = filterText.getText();
133
134                   if (text.length() == 0)
135                      sorter.setRowFilter(null);
136                   else
137                   {
138                      try
139                      {
140                         sorter.setRowFilter(
141                            RowFilter.regexFilter(text));
142                      }
```

Fig. 24.28 | Display the contents of the `Authors` table in the `books` database. (Part 7 of 12.)

```
143                         catch (PatternSyntaxException pse)
144                         {
145                            JOptionPane.showMessageDialog(null,
146                               "Bad regex pattern", "Bad regex pattern",
147                               JOptionPane.ERROR_MESSAGE);
148                         }
149                      }
150                   }
151                }
152             );
153             // dispose of window when user quits application (this overrides
154             // the default of HIDE_ON_CLOSE)
155             window.setDefaultCloseOperation(DISPOSE_ON_CLOSE);
156             window.setSize(500, 250);
157             window.setVisible(true);
158
```

Fig. 24.28 | Display the contents of the `Authors` table in the `books` database. (Part 8 of 12.)

```
159            // ensure database is closed when user quits application
160            addWindowListener(
161               new WindowAdapter()
162               {
163                  // disconnect from database and exit when window has closed
164                  public void windowClosed(WindowEvent event)
165                  {
166                     tableModel.disconnectFromDatabase();
167                     System.exit(0);
168                  }
169               }
170            );
171         }
172         catch (SQLException sqlException)
173         {
174            JOptionPane.showMessageDialog(null, sqlException.getMessage(),
175               "Database error", JOptionPane.ERROR_MESSAGE);
176            tableModel.disconnectFromDatabase();
177            System.exit(1); // terminate application
178         }
179      }
180   } // end class DisplayQueryResults
```

Fig. 24.28 | Display the contents of the `Authors` table in the `books` database. (Part 9 of 12.)

a) Displaying all authors from the `Authors` table

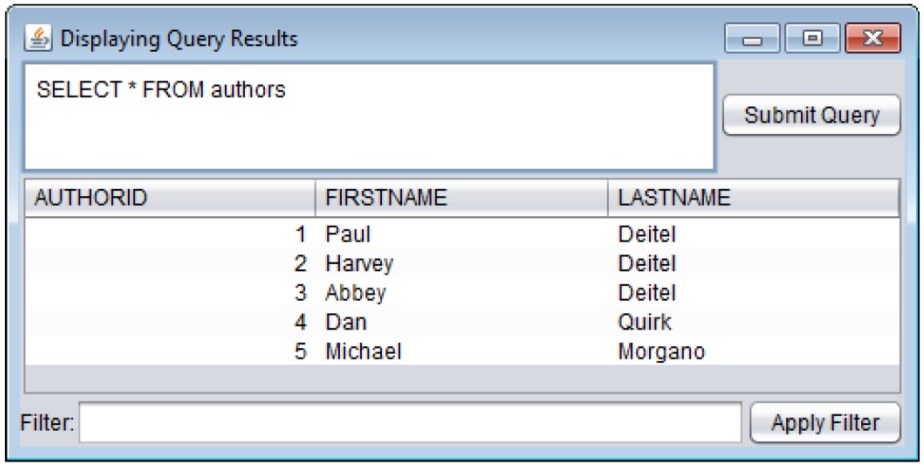


Fig. 24.28 | Display the contents of the `Authors` table in the `books` database. (Part 10 of 12.)

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

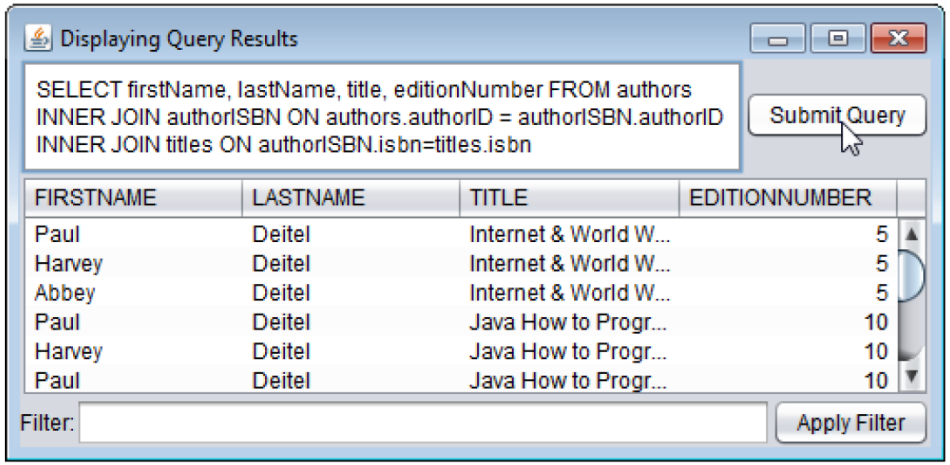


Fig. 24.28 | Display the contents of the `Authors` table in the `books` database. (Part 11 of 12.)

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

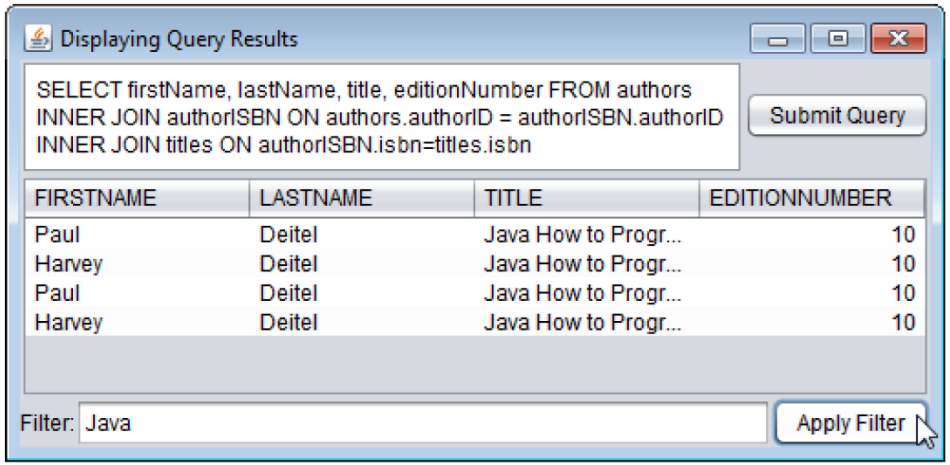


Fig. 24.28 | Display the contents of the `Authors` table in the `books` database. (Part 12 of 12.)

24.6.2 Querying the books Database (cont.)

- Any local variable that will be used in an anonymous inner class *must* be declared final; otherwise, a compilation error occurs. (In Java SE 8, this program would compile without declaring these variables final because these variables would be effectively `final`, as discussed in Chapter 17.)
- Class TableRowSorter (from package javax.swing.table) can be used to sort rows in a `JTable`.
  - 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.
- `JTable` method setRowSorter specifies the `TableRowSorter` for the `JTable`.

24.6.2 Querying the books Database (cont.)

- `JTable`s can now show subsets of the data from the underlying `TableModel`.
  - This is known as filtering the data.
- `JTable` method setRowFilter specifies a RowFilter- (from package `javax.swing`) for a `JTable`.
- `RowFilter` `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.

24.7 RowSet Interface

- The interface RowSet provides several *set* methods that allow you to specify the properties needed to establish a connectionand create a `Statement`.
  - `RowSet` also provides several *get* methods that return these properties.
- 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 is disconnected.
  - Modified data can be updated in the database after a disconnected `RowSet` reestablishes the connection with the database.

24.7 RowSet Interface (cont.)

- 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`.
  - 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.
  - A `CachedRowSet` object is scrollable and updatable by default.
  - Also serializable, so it can be passed between Java applications through a network, such as the Internet.

Portability Tip 24.5

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

```
1  // Fig. 24.29: JdbcRowSetTest.java
2  // Displaying the contents of the Authors table using JdbcRowSet.
3  import java.sql.ResultSetMetaData;
4  import java.sql.SQLException;
5  import javax.sql.rowset.JdbcRowSet;
6  import javax.sql.rowset.RowSetProvider;
7
8  public class JdbcRowSetTest
9  {
10    // JDBC driver name and database URL
11    private static final String DATABASE_URL = "jdbc:derby:books";
12    private static final String USERNAME = "deitel";
13    private static final String PASSWORD = "deitel";
14
```

Fig. 24.29 | Displaying the contents of the `Authors` table using `JdbcRowSet`. (Part 1 of 4.)

```
15      public static void main(String args[])
16      {
17         // connect to database books and query database
18         try (JdbcRowSet rowSet =
19            RowSetProvider.newFactory().createJdbcRowSet())
20         {
21            // specify JdbcRowSet properties
22            rowSet.setUrl(DATABASE_URL);
23            rowSet.setUsername(USERNAME);
24            rowSet.setPassword(PASSWORD);
25            rowSet.setCommand("SELECT * FROM Authors"); // set query
26            rowSet.execute(); // execute query
27
28            // process query results
29            ResultSetMetaData metaData = rowSet.getMetaData();
30            int numberOfColumns = metaData.getColumnCount();
31            System.out.printf("Authors Table of Books Database:%n%n");
32
33            // display rowset header
34            for (int i = 1; i <= numberOfColumns; i++)
35               System.out.printf("%-8s\t", metaData.getColumnName(i));
36            System.out.println();
37
```

Fig. 24.29 | Displaying the contents of the `Authors` table using `JdbcRowSet`. (Part 2 of 4.)

```
38             // display each row
39             while (rowSet.next())
40             {
41                for (int i = 1; i <= numberOfColumns; i++)
42                   System.out.printf("%-8s\t", rowSet.getObject(i));
43                System.out.println();
44             }
45          }
46          catch (SQLException sqlException)
47          {
48             sqlException.printStackTrace();
49             System.exit(1);
50          }
51       }
52    } // end class JdbcRowSetTest
```

Fig. 24.29 | Displaying the contents of the `Authors` table using `JdbcRowSet`. (Part 3 of 4.)

```
Authors Table of Books Database:

AUTHORID        FIRSTNAME       LASTNAME
1               Paul            Deitel
2               Harvey          Deitel
3               Abbey           Deitel
4               Dan             Quirk
5               Michael         Morgano
```

Fig. 24.29 | Displaying the contents of the `Authors` table using `JdbcRowSet`. (Part 4 of 4.)

24.7 RowSet Interface (cont.)

- Class `RowSetProvider` (package `javax.sql.rowset`) provides static method `newFactory` which returns a an object that implements interface `RowSetFactory` (package `javax.sql.rowset`) that can be used to create various types of `RowSet`s.
- The `try`-with-resources statement use `RowSetFactory` method `createJdbcRowSet` to obtain a `JdbcRowSet` object.
- `JdbcRowSet` method `setUrl` specifies the database URL.
- `JdbcRowSet` method `setUsername` specifies the username.
- `JdbcRowSet` method `setPassword` specifies the password.
- `Jdbc-RowSet` method `setCommand` specifies the SQL query that will be used to populate the `RowSet`.

24.7 RowSet Interface (cont.)

- Jdbc-RowSet method execute executes the SQL query.
- Method execute performs four actions
  - Establishes a Connection to the database
  - Prepares the query Statement
  - Executes the query
  - Stores the ResultSet returned by query.
- The Connection, Statement and ResultSet are encapsulated in the JdbcRowSet object.
- Jdbc-RowSet method close closes the RowSet's encapsulated ResultSet, Statement and Connection.

24.8 PreparedStatements

- Interface PreparedStatement enables you to create compiled SQL statements that execute more efficiently than `Statement` objects.
- Can also specify parameters, making them more flexible than `Statement`s.
  - Programs can execute the same query repeatedly with different parameter values.

24.8 PreparedStatements (cont.)

- To locate all book titles for an author with a specific last name and first name:

```
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 (?) are placeholders for values that will be passed as part of the query to the database.

24.8 PreparedStatements (cont.)

- 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" );
    ```
- Parameter numbers are counted from 1, starting with the first question mark (?).
- Interface `PreparedStatement` provides *set* methods for each supported SQL type.
    ```

Performance Tip 24.2

`PreparedStatement`s are more efficient than `Statement`s when executing SQL statements multiple times and with different parameter values.

Error-Prevention Tip 24.2

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

Error-Prevention Tip 24.3

`PreparedStatement`s help prevent SQL injection attacks, which typically occur in SQL statements that include user input improperly. To avoid this security issue, use `PreparedStatement`s in which user input can be supplied only via parameters—indicated with ? when creating a `PreparedStatement`. Once you've created such a `PreparedStatement`, you can use its `set` *methods to specify the user input as arguments for those parameters.*

24.8 PreparedStatements (cont.)

- Our `AddressBook` Java DB database contains an `Addresses` table with the columns `addressID`, `FirstName`, `LastName`, `Email` and `PhoneNumber`.
- The column `addressID` is an identity column in the `Addresses` table.

24.8 PreparedStatements (cont.)

- Invoke `Connection` method prepareStatement to create a `PreparedStatement`.
- Calling `PreparedStatement` method executeQuery returns a `ResultSet` containing the rows that match the query.
- `PreparedStatement` method executeUpdate executes a SQL statement that modifies the database.

```
1  // Fig. 24.30: 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    // constructor
12    public Person()
13    {
14    }
15
```

Fig. 24.30 | `Person` class that represents an entry in an `AddressBook`. (Part 1 of 5.)

```
16      // constructor
17      public Person(int addressID, String firstName, String lastName,
18         String email, String phoneNumber)
19      {
20         setAddressID(addressID);
21         setFirstName(firstName);
22         setLastName(lastName);
23         setEmail(email);
24         setPhoneNumber(phoneNumber);
25      }
26
27      // sets the addressID
28      public void setAddressID(int addressID)
29      {
30         this.addressID = addressID;
31      }
32
33      // returns the addressID
34      public int getAddressID()
35      {
36         return addressID;
37      }
38
```

Fig. 24.30 | `Person` class that represents an entry in an `AddressBook`. (Part 2 of 5.)

```
39      // sets the firstName
40      public void setFirstName(String firstName)
41      {
42         this.firstName = firstName;
43      }
44
45      // returns the first name
46      public String getFirstName()
47      {
48         return firstName;
49      }
50
51      // sets the lastName
52      public void setLastName(String lastName)
53      {
54         this.lastName = lastName;
55      }
56
57      // returns the last name
58      public String getLastName()
59      {
60         return lastName;
61      }
```

Fig. 24.30 | `Person` class that represents an entry in an `AddressBook`. (Part 3 of 5.)

```
62
63      // sets the email address
64      public void setEmail(String email)
65      {
66         this.email = email;
67      }
68
69      // returns the email address
70      public String getEmail()
71      {
72         return email;
73      }
74
75      // sets the phone number
76      public void setPhoneNumber(String phone)
77      {
78         this.phoneNumber = phone;
79      }
80
```

Fig. 24.30 | `Person` class that represents an entry in an `AddressBook`. (Part 4 of 5.)

```
81      // returns the phone number
82      public String getPhoneNumber()
83      {
84         return phoneNumber;
85      }
86   } // end class Person
```

Fig. 24.30 | `Person` class that represents an entry in an `AddressBook`. (Part 5 of 5.)

```
1  // Fig. 24.31: 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; // manages connection
18    private PreparedStatement selectAllPeople;
19    private PreparedStatement selectPeopleByLastName;
20    private PreparedStatement insertNewPerson;
21
```

Fig. 24.31 | `PreparedStatements` used by the Address Book application. (Part 1 of 9.)

```
22      // constructor
23      public PersonQueries()
24      {
25         try
26         {
27            connection =
28               DriverManager.getConnection(URL, USERNAME, PASSWORD);
29
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         }
```

Fig. 24.31 | `PreparedStatements` used by the Address Book application. (Part 2 of 9.)

```
44          catch (SQLException sqlException)
45          {
46             sqlException.printStackTrace();
47             System.exit(1);
48          }
49       }
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
```

Fig. 24.31 | `PreparedStatements` used by the Address Book application. (Part 3 of 9.)

```
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            }
72         }
73         catch (SQLException sqlException)
74         {
75            sqlException.printStackTrace();
76         }
```

Fig. 24.31 | `PreparedStatements` used by the Address Book application. (Part 4 of 9.)

```
77          finally
78          {
79             try
80             {
81                resultSet.close();
82             }
83             catch (SQLException sqlException)
84             {
85                sqlException.printStackTrace();
86                close();
87             }
88          }
89
90          return results;
91       }
92
```

Fig. 24.31 | `PreparedStatements` used by the Address Book application. (Part 5 of 9.)

```
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
```

Fig. 24.31 | `PreparedStatements` used by the Address Book application. (Part 6 of 9.)

```
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          }
116       }
117       catch (SQLException sqlException)
118       {
119          sqlException.printStackTrace();
120       }
121       finally
122       {
123          try
124          {
125             resultSet.close();
126          }
127          catch (SQLException sqlException)
128          {
129             sqlException.printStackTrace();
130             close();
```

Fig. 24.31 | `PreparedStatements` used by the Address Book application. (Part 7 of 9.)

```
131             }
132          }
133
134          return results;
135       }
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          }
```

Fig. 24.31 | `PreparedStatements` used by the Address Book application. (Part 8 of 9.)

```
154         catch (SQLException sqlException)
155         {
156            sqlException.printStackTrace();
157            close();
158         }
159
160         return result;
161      }
162
163      // close the database connection
164      public void close()
165      {
166         try
167         {
168            connection.close();
169         }
170         catch (SQLException sqlException)
171         {
172            sqlException.printStackTrace();
173         }
174      }
175   } // end class PersonQueries
```

Fig. 24.31 | `PreparedStatements` used by the Address Book application. (Part 9 of 9.)

```
1  // Fig. 24.32: 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;
20
```

Fig. 24.32 | A simple address book. (Part 1 of 19.)

```
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;
```

Fig. 24.32 | A simple address book. (Part 2 of 19.)

```
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      // 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();
```

Fig. 24.32 | A simple address book. (Part 3 of 19.)

```
68          displayPanel = new JPanel();
69          idLabel = new JLabel();
70          idTextField = new JTextField(10);
71          firstNameLabel = new JLabel();
72          firstNameTextField = new JTextField(10);
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
```

Fig. 24.32 | A simple address book. (Part 4 of 19.)

```
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               }
102            }
103         );
104
105         navigatePanel.add(previousButton);
106         navigatePanel.add(Box.createHorizontalStrut(10));
107
```

Fig. 24.32 | A simple address book. (Part 5 of 19.)

```
108         indexTextField.setHorizontalAlignment(
109            JTextField.CENTER);
110         indexTextField.addActionListener(
111            new ActionListener()
112            {
113               public void actionPerformed(ActionEvent evt)
114               {
115                  indexTextFieldActionPerformed(evt);
116               }
117            }
118         );
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));
126
127         maxTextField.setHorizontalAlignment(
128            JTextField.CENTER);
129         maxTextField.setEditable(false);
130         navigatePanel.add(maxTextField);
131         navigatePanel.add(Box.createHorizontalStrut(10));
```

Fig. 24.32 | A simple address book. (Part 6 of 19.)

```
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               }
142            }
143         );
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
```

Fig. 24.32 | A simple address book. (Part 7 of 19.)

```
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
```

Fig. 24.32 | A simple address book. (Part 8 of 19.)

```
176         queryPanel.setBorder(BorderFactory.createTitledBorder(
177            "Find an entry by last name"));
178         queryLabel.setText("Last Name:");
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               }
193            }
194         );
195
196         queryPanel.add(queryButton);
197         queryPanel.add(Box.createHorizontalStrut(5));
198         add(queryPanel);
199
```

Fig. 24.32 | A simple address book. (Part 9 of 19.)

```
200         browseButton.setText("Browse All Entries");
201         browseButton.addActionListener(
202            new ActionListener()
203            {
204               public void actionPerformed(ActionEvent evt)
205               {
206                  browseButtonActionPerformed(evt);
207               }
208            }
209         );
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               }
221            }
222         );
223
```

Fig. 24.32 | A simple address book. (Part 10 of 19.)

```
224         add(insertButton);
225
226         addWindowListener(
227            new WindowAdapter()
228            {
229               public void windowClosing(WindowEvent evt)
230               {
231                  personQueries.close(); // close database connection
232                  System.exit(0);
233               }
234            }
235         );
236
237         setVisible(true);
238      } // end constructor
239
```

Fig. 24.32 | A simple address book. (Part 11 of 19.)

```
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     }
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     }
263
```

Fig. 24.32 | A simple address book. (Part 12 of 19.)

```
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        }
285        else
286           browseButtonActionPerformed(evt);
287     }
```

Fig. 24.32 | A simple address book. (Part 13 of 19.)

```
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        }
306     }
307
```

Fig. 24.32 | A simple address book. (Part 14 of 19.)

```
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           }
330        }
```

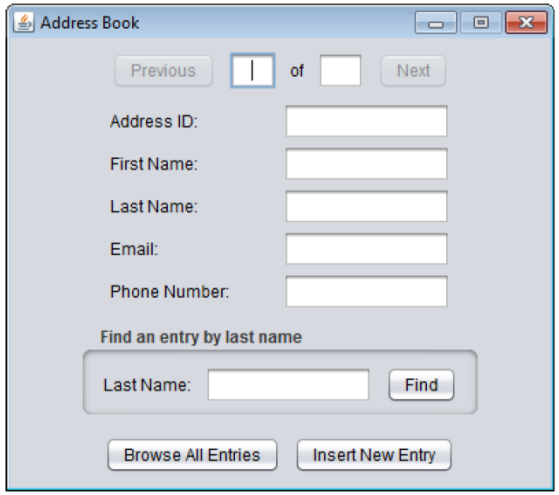
Fig. 24.32 | A simple address book. (Part 15 of 19.)

```
331         catch (Exception e)
332         {
333            e.printStackTrace();
334         }
335      }
336
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      }
353
```

Fig. 24.32 | A simple address book. (Part 16 of 19.)

```
354     // main method
355     public static void main(String args[])
356     {
357        new AddressBookDisplay();
358     }
359  } // end class AddressBookDisplay
```

a) Initial **Address Book** screen.



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

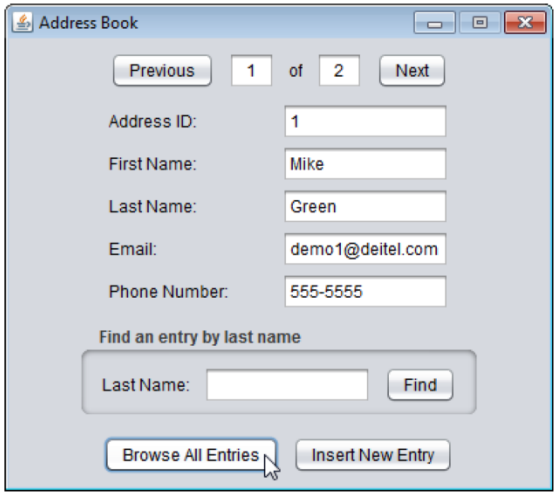
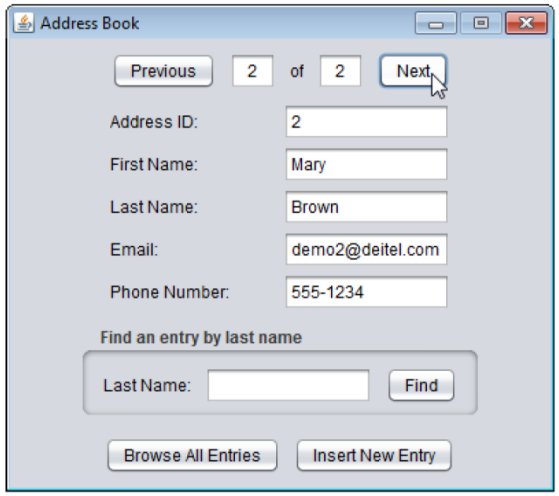


Fig. 24.32 | A simple address book. (Part 17 of 19.)

c) Browsing to the next entry.



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

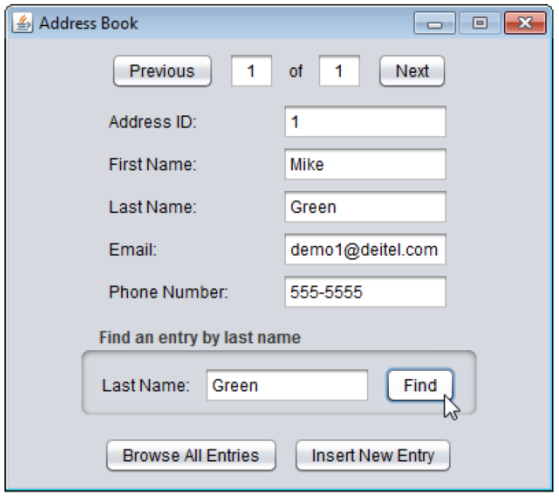


Fig. 24.32 | A simple address book. (Part 18 of 19.)

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

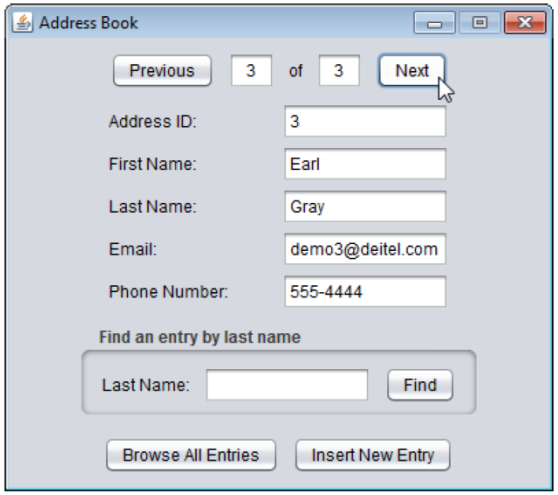


Fig. 24.32 | A simple address book. (Part 19 of 19.)

24.9 Stored Procedures

- Many database management systems can store individual SQL statements 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.
- In addition, `CallableStatement`s can specify output parameters in which a stored procedure can place return values.
- The interface also includes methods to obtain the values of output parameters returned from a stored procedure.
- To learn more about `CallableStatement`s, visit
  - `java.sun.com/javase/6/docs/technotes/guides/jdbc/getstart/callablestatement.html#999652`

Portability Tip 24.6

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 24.7

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* `ResultSet`*s returned from a* `CallableStatement` *before obtaining the values of any output parameters.*

24.10 Transaction Processing

- Many database applications require guarantees that a series of database insertions, updates and deletions executes properly before the applications continue processing the next database operation.
- 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).
- Rolling back the transaction leaves the database in its state prior to the database operation.

24.10 Transaction Processing (cont.)

- `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 or `Connection` method rollback.
- Interface `Connection` also provides method getAutoCommit to determine the autocommit state for the `Connection`.