Database Connectivity 24

Chapter Topics

- Understanding the basics of relational databases: relational tables and SQL
- Understanding the role of the JDBC driver, and establishing a database connection using such a driver
- Understanding the handling of JDBC resources, and closing them in a responsible way
- Writing code to create and execute SQL statements: basic statement, prepared statement, and callable statement in the JDBC API
- Writing code to process query results, including customizing the result set returned by a query
- Discovering database capabilities and reading result set metadata
- Writing code to implement transaction control

Java SE 17 Developer Exam Objectives	
[11.1] Implement localization using locales, resource bundles, parse and format messages, dates, times, and numbers including currency and percentage values	§24.1, p. 1512, *to* §24.8, p. 1545.

Java SE 11 Developer Exam Objectives	
[11.1] Connect to and perform database SQL operations, process query results using JDBC API	§24.1, p. 1512, *to* §24.8, p. 1545.

Database is a general term that describes organized data storage, controlled by a *Database Management System* (DBMS). There are many different such systems from different providers and vendors, both commercial and open source.

The responsibilities of a DBMS typically include tasks such as secure and control data access, keep data safe and consistent, and query the data. Databases can be classified by the way in which they store information, such as hierarchical, network, object, and relational, each with their own advantages and disadvantages. Many modern DBMS present a mix of different database features and capabilities.

This chapter discusses how to develop Java applications that use a relational database, which is one of the most commonly used database models. The Java Database Connectivity (JDBC) specification allows development of Java applications that can leverage a relational database independent of a specific DBMS.

The coverage of JDBC in this chapter is in no way exhaustive. The topics covered provide the basic essentials to write Java applications that use the JDBC API to interact with a relational database in an independent way. Prior knowledge of databases is advantageous, but not essential, as the basics are covered in this chapter. However, to be proficient, the large body of literature and software in this area should be consulted.

24.1 Introduction to Relational Databases

Relational databases are based on the *relational model* that provides a standardized way to represent and query data. The data is stored in *tables* and queried using a *query language*.

Relational Tables

Relational databases store information in *tables*. Each table is designed to represent a business entity that applications need to store information about. Each table is composed of *columns* (a.k.a. *attributes*). Each column represents specific information that describes a specific characteristic of a business entity that the table represents.

Each column has a *name* and describes information of a specific *type*, such as fixed-or variable-length character data (type `CHAR` or `VARCHAR`), numeric values (type `NUMBER` or `INTEGER`), date and time values (type `DATE` or `TIMESTAMP`), and many other types. Different database providers may support different data types.

Each table may be defined with a number of *constraints* that enforce business rules associated with this entity. For example, constraints may be used to enforce uniqueness of column values (`PRIMARY KEY`), or that these values must always be present (`NOT NULL`), or that they are restricted to a specific list or range of values. Different database providers may support different constraint types.

Actual business data is stored in tables as *rows* (a.k.a. *tuples*). Each row is composed of values that follow the exact structure of columns defined by a given table.

As an example, a music catalogue application needs to store information about music compositions in a database. Let us assume that each composition has the following attributes:

- An International Standard Recording Code (*ISRC*), which uniquely identifies a composition
- A title
- A duration

Thus to store this information in a relational database would require a `compositions` table with `isrc`, `title`, and `duration` columns. Assume that all of these are mandatory pieces of information for every composition object, and that `isrc` and `title` are both of type `VARCHAR`, and `duration` is of type `INT`. The definition of the `compositions` table is shown in **Table 24.1**.

Table 24.1 *Definition of the* `compositions` *Table*

Column name	Column type	Column constraints
`isrc`	`VARCHAR(12)`	`PRIMARY KEY`
`title`	`VARCHAR(40)`	`NOT NULL`
`duration`	`INT`	`NOT NULL`

Note that column types often specify specific length or precision characteristics— for example, `VARCHAR(12)` in **Table 24.1**. Also, a `PRIMARY KEY` constraint is usually implicitly mandatory `NOT NULL` (does not allow `null` values). In **Table 24.1**, the `isrc` column name uniquely identifies a composition, hence its constraint is `PRIMARY KEY` and implicitly `NOT NULL`. The other two column names `title` and `duration` are mandatory for each composition, hence their constraint is also `NOT NULL`.

Table 24.2 shows example data (four rows) stored in the `compositions` table, which satisfies the *table definition* given in **Table 24.1**. Each row in **Table 24.2** represents a unique music composition.

A table definition can be loosely compared to a class that describes a number of attributes, and a row can be compared to an object (instance) of that specific class. A spreadsheet is another example of organizing data in rows and columns.

Table 24.2 *Data in the* `compositions` *Table*

`isrc`	`title`	`duration`
`ushm91736697`	`Vacation`	`231`
`ushm91736698`	`Rage`	`308`
`ushm91736699`	`Why Don't`	`178`
`ushm91736700`	`Something Happened`	`147`

Basic SQL Statements

Structured Query Language (SQL) is used to perform relational database operations. Despite the existence of an ANSI (American National Standards Institute) SQL standard, different database providers may provide SQL implementations that are not exactly the same and may contain nuanced differences and proprietary additions.

SQL statements can be logically grouped into a number of languages, in which they are identified by SQL keywords:

- *Data Manipulation Language* (*DML*) comprises the `SELECT`, `INSERT`, `UPDATE`, and `DELETE` statements, used to query, create, modify, and remove table rows. Sometimes the `SELECT` statement is described to be in its own *Data Query Language* (*DQL*) group.
- *Data Definition Language* (*DDL*) comprises the `CREATE`, `ALTER`, and `DROP` statements, whose purpose is to create, modify, and remove tables. We will refer to the `CREATE`, `ALTER`, and `DROP` statements as *DDL operations*.
- *Transaction Control Language* (*TCL*) comprises the `COMMIT` and `ROLLBACK` statements, whose purpose is to save or undo pending changes, respectively.

Often the `INSERT`, `SELECT`, `UPDATE`, and `DELETE` statements (in that order) are referred to as *CRUD operations*, denoting the *Create*, *Read*/*Retrieve*, *Update*, and *Delete* operations implied by the acronym, respectively. Note the name mismatch of the CRUD operations and the SQL statements is implied. In particular, the set of CRUD operations does *not* include the `CREATE` statement in SQL that creates relational tables.

The rest of this subsection presents basic syntax patterns and examples of creating relational tables and the CRUD operations in SQL. Note that the semicolon (`;`) is a *statement terminator* in SQL.

The `CREATE` *Statement*

The `CREATE` statement can be used to create a new table in the database:

Click here to view code image

```
CREATE TABLE table_name (column_definitions);
```

where *table_name* is the name of the table to be created, and *column_definitions* is a comma-delimited list that describes each column in this table:

Click here to view code image

```
column_name column_type constraints, ...
```

The following statement creates an empty `compositions` table:

Click here to view code image

```
CREATE TABLE compositions (isrc VARCHAR(12) PRIMARY KEY,
                           title VARCHAR(40) NOT NULL, duration INT NOT NULL);
```

The `CREATE` statement above will create an empty relational table whose columns will correspond to the table shown in Table 24.2.

The `INSERT` *Statement*

The `INSERT` statement can be used to insert a new row into a table:

Click here to view code image

```
INSERT INTO table_name VALUES (actual_values);
```

where *actual_values* in the `VALUES` clause is a comma-delimited list that provides values for the columns of the table specified by *table_name*.

The following statement inserts a new row into the `compositions` table:

Click here to view code image

```
INSERT INTO compositions VALUES ('ushm91736697', 'Vacation', 231);
```

The number of rows in the `compositions` table increases by 1. By default, the values specified are interpreted in the same order as the columns are defined in the table. Note that string literals in SQL are enclosed in single quotes (`'`).

The `SELECT` *Statement*

The `SELECT` statement can be used to query or read rows from the database:

Click here to view code image

```
SELECT column_list FROM table_name WHERE row_filter;
```

The *column_list* is a comma-delimited list of column names, whose values are selected from the table specified with *table_name* in the `FROM` clause. The `'*'` symbol can be used as *column_list* to specify that all columns should be included in the returned result. A list of conditions is specified by *row_filter* in the `WHERE` clause to determine which specific rows should be selected. Such conditions may be constructed using a variety of operators to compare column values, such as `=` (*equals*), `>` (*greater than*), `<` (*less than*), `LIKE` (which can use `"%"` wildcards), and many others. Conditions in *row_filter* can be combined using `AND` and `OR` operators.

Click here to view code image

```
column_name = some_value AND column_name LIKE some_value
```

The following statement selects values of `isrc` and `title` for all rows in the `compositions` table whose `duration` is greater than `200` and whose `title` starts with `'V'`:

Click here to view code image

```
SELECT isrc, title FROM compositions WHERE duration > 200 AND title LIKE 'V%';
```

Applied to Table 24.2, the above query will return the values of `isrc` and `title` in the first row, which is the only row that meets the *row_filter* criteria—that is, `(ushm91736697, Vacation)`.

The `WHERE` clause is optional. In that case, the values of *column_list* in the entire table are returned.

```
SELECT * FROM compositions;
```

The above `SELECT` statement will return all the rows in the specified table.

The `UPDATE` *Statement*

The `UPDATE` statement can be used to update specific rows in a table. In other words, this statement modifies zero or more rows in the table:

Click here to view code image

```
UPDATE table_name SET column_name_value_pairs WHERE row_filter;
```

where *column_name_value_pairs* in the `SET` clause is a comma-delimited list of pairs of column names and values:

Click here to view code image

```
column_name = column_value, ...
```

and *row_filter* in the `WHERE` clause specifies a list of conditions to indicate which specific rows should be updated.

All rows that meet the *row_filter* criteria in *table_name* will have their columns specified in the *column_name_value_pairs* set to the corresponding values in *column_name_value_pairs*.

The following statement updates a `duration` value for an existing row in the `compositions` table:

Click here to view code image

```
UPDATE compositions SET duration = 240 WHERE isrc LIKE '%91736697'
                                             OR duration > 231;
```

The above `UPDATE` statement will update the first row `(isrc '91736697')` in Table 24.2 by setting the `duration` to `240`.

The `WHERE` clause is optional. When omitted, all rows will have their columns specified in the `column_name_value_pairs` set to the corresponding values in `column_name_value_pairs`.

Click here to view code image

```
UPDATE compositions SET duration = 250;
```

The above `UPDATE` statement will update the `duration` column for all rows to `250` in **Table 24.2**.

The `DELETE` *Statement*

The `DELETE` statement can be used to delete specific rows in a table that meet the criteria in the `WHERE` clause. This statement deletes zero or more rows from the specified table:

Click here to view code image

```
DELETE FROM table_name WHERE row_filter;
```

where *row_filter* in the `WHERE` clause is the criteria for which rows should be deleted from *table_name* in the `FROM` clause.

The following statement deletes a row from the `compositions` table:

Click here to view code image

```
DELETE FROM compositions WHERE isrc = 'ushm91736697';
```

The row with `isrc` having the value `'ushm91736697'` in the `compositions` table will be deleted from **Table 24.2**.

The `WHERE` clause is optional. Care must be taken, as omitting it will delete all rows from the table.

```
DELETE FROM compositions;
```

There are many other statements available in SQL described by the ANSI SQL standard and additional statements which may be implemented by database providers.

This book does not have a goal of covering SQL in detail, but provides a brief introduction to ensure a minimum level of understanding of how relational databases operate and thereby an understanding of how Java programs interact with relational databases.

Java certification focuses on Java, not on SQL. However, it is essential to learn database concepts and understand SQL if one wants to work with relational databases.

24.2 Introduction to JDBC

The *Java Database Connectivity* (JDBC) protocol provides database interaction capabilities for Java programs. The JDBC specification is not database provider specific, allowing Java programs to interact with any database.

The *JDBC API* is defined in the `java.sql` package. It comprises a number of interfaces that Java programs use to represent database connections, SQL statements, query results, and much more.

The JDBC API is implemented by a number of database-specific *JDBC drivers*. The job of a JDBC driver is to provide database-specific, native protocol implementation of the JDBC API. For example, Oracle, MySQL, DerbyDB, and others provide JDBC drivers that all implement the same JDBC API, but in a different database-specific manner.

A Java application can dynamically load the JDBC driver as required and connect to different databases, provided that the JDBC driver is present in the class path or module path of the application.

Figure 24.1 illustrates the layered approach to database connectivity. Java developers write code that utilizes JDBC interfaces defined by the `java.sql` package. These interfaces are implemented by different JDBC drivers. Java developers should only write code that utilizes JDBC API interfaces and not use JDBC drivers directly, so that software portability is not compromised. This allows Java applications to maintain database-provider neutrality, and potentially switch JDBC drivers and database providers without modifying application code. This approach is designed to decouple the application code from specific database providers, potentially allowing the application to switch between different databases and even use different databases at the same time.

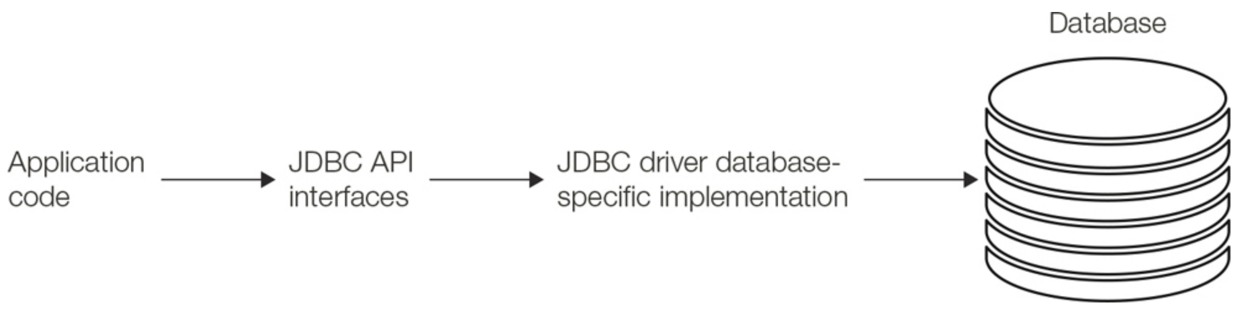

Figure 24.1 *Layered Database Connectivity*

In order to connect to a database, the Java application has to perform the following tasks:

1. Ensure that the relevant JDBC driver is available in the class path or module path of the application.
2. Load the JDBC driver to memory.

3. Establish the database connection.

Once a database connection is established, a typical interaction scenario with the database proceeds as follows:

1. *Create* SQL statements.
2. *Execute* SQL statements.
3. *Process* query results.
4. *Close* the JDBC resources.

Closing JDBC Resources

It is important to ensure that all JDBC resources are properly closed once they are no longer needed.

All JDBC API methods can throw a `java.sql.SQLException`. A `SQLException` is a checked exception, which in addition to the usual Java error message, also wraps up database error information such as the SQL state and an error code.

The skeletal code below will always catch the `SQLException` and handle it:

Click here to view code image

```
try {
/* execute JDBC operations */
} catch (SQLException e) {
  String state = e.getSQLState();
  int code = e.getErrorCode();
}
```

The logic of the JDBC application can be generalized with the following skeletal code:

Click here to view code image

```
try {
  /* establish database connection     */
  /* create and execute SQL statements */
  /* process results                   */
} catch (SQLException e) {
  /* handle any errors */
} finally {
  /* close result sets */
  /* close statements  */
  /* close connection  */
}
```

It is important to remember that the closing order of these objects is significant: First close any result set objects, then close statements, and then close connections. No Java exceptions are actually produced if you try to close in a different order. However, this would prevent resources from being promptly released and thus can result in memory leaks.

JDBC interfaces that represent connection, statement, and result set objects all implement the `AutoCloseable` interface. Therefore, they can be used in the `try`-with-resources construct, in which case they are automatically closed in the implicit `finally` block inserted by the compiler, as shown below.

Click here to view code image

```
try (/* establish database connection     */
     /* create and execute SQL statements */)
{
  /* process results */
} catch (SQLException e) {
  /* handle any exceptions */
} /* implicit finally block closes resources*/
```

24.3 Establishing a Database Connection

Interaction between the application and the database is in the context of a *connection*. Database connections are represented by the `java.sql.Connection` interface. JDBC drivers provide database-specific implementations of this interface.

The class `java.sql.DriverManager` provides the overloaded factory method `getConnection()` that initializes and returns a database `Connection` object.

Click here to view code image

```
static Connection getConnection(String jdbcURL)
static Connection getConnection(String jdbcURL, String username,
                                String password)
static Connection getConnection(String jdbcURL, Properties info)
```

All versions of the `getConnection()` method require a JDBC URL and all throw a `SQLException`. Additional information may be required to establish a database connection, such as username and password, or other means of authentication such as digital signatures that can be set using the `Properties` object (§18.2, p. 1100). Note also that the `Connection` object returned is `AutoCloseable`, and therefore it is best handled in a `try`-with-resources construct to ensure that the connection is closed when done.

JDBC URL

The general syntax of the *JDBC URL* is as follows:

Click here to view code image

```
protocol:provider:driver_type:database_specific_connection_details
```

The *protocol* is always specified as `"jdbc"`. Other details may vary depending on the database provider. For example, to connect to the Apache Derby database you need to specify the *provider* as `"derby"`, followed by the description of where this database is located, which would include *host*, *port*, and *database name* information.

Click here to view code image

```
jdbc:derby:localhost:1521:musicDB
```

The example below shows a more sophisticated JDBC driver that may provide different connectivity mechanisms, each identified by an appropriate *subprotocol*. In this case, the JDBC URL example shows how one can specify an Oracle database connection using the most frequently used *thin* protocol implementation that provides enhanced security features.

Click here to view code image

```
jdbc:oracle:thin:@localhost:1521:musicDB
```

Not all JDBC drivers provide alternative connectivity methods. In the Oracle JDBC driver case, alternative connectivity mechanisms, such as *thin*, *OCI*, or *kprb*, can be used. This could be the case with many other providers of JDBC drivers. In the case of a Derby database, it can be accessed as a separate process or be embedded inside a Java program, which would make the JDBC URL look different. It is best to refer to JDBC driver-specific documentation to identify available choices and the rationale behind selecting one or the other connectivity mechanism.

Getting a Database Connection

To create a connection to a Derby database, the following code can be emulated:

Click here to view code image

```
String jdbcURL  = "jdbc:derby:localhost:1521:musicDB";
String username = "joe";
String password = "welcome1";
try (Connection connection = DriverManager.getConnection(jdbcURL,
                                                          username, password)) {
  /* use the connection. */
} catch (SQLException e) {
  e.printStackTrace();
}
```

A portion of the URL that indicates a driver instructs the `DriverManager` to load the appropriate driver implementation to memory. This of course would fail if this JDBC driver is not found in the class path or module path.

Connecting to the `musicDB` *Database*

We will use the `musicDB` database which consists of the `compositions` table shown in Table 24.2. Instructions for downloading the files for the `musicDB` database and the Derby JAR files can be found on this book's website. The structure of the working directory is assumed to be the following:

Click here to view code image

```
Working directory
 ?
 ??? db-derby-10.15.2.0-lib/  <== Directory with the Apache Derby distribution
 ?   ?
 ?   ??? lib/                 <== Directory with the Derby JAR files
 ?
 ??? musicDB/                 <== Directory with the  musicDB  database files
 ?
 ??? dbDemo/                  <== Package with examples used in this chapter
 ?   ?
 ?   ??? JDBCConnection.java
 ?   ?
... ...
```

Example 24.1 shows how to obtain a connection to the `musicDB` database. Note that in the JDBC URL, the relative path of the `musicDB` directory in the working directory is `musicDB`. The class path must also be set to access the Derby JAR files. In the code below, the implicit `finally` clause of the `try`-with-resources statement guarantees to close the connection when done.

Example 24.1 *Connecting to the* `musicDB` *Database*

Click here to view code image

```
package dbDemo;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class JDBCConnection {
  public static void main(String[] args) {
    final String jdbcURL = "jdbc:derby:musicDB";
    try (Connection connection = DriverManager.getConnection(jdbcURL)) {
      /* use the connection. */
      System.out.println(connection);
    } catch (SQLException e) {
      e.printStackTrace();
    }
  }
}
```

Probable output from the program:

Click here to view code image

```
org.apache.derby.impl.jdbc.EmbedConnection@1201454821 (XID = 1301),
(SESSIONID = 1), (DATABASE = musicDB), (DRDAID = null)
```

Legacy JDBC Driver Management

Prior to JDBC version 4.0, drivers had to be loaded into memory explicitly, before using them to establish database connections. There are several ways in which this could be achieved:

- Instantiate and register the database-specific JDBC driver class using the `java.sql.DriverManager.registerDriver()` method:
  Click here to view code image

  ```
  DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());
  ```

- Load a database-specific JDBC driver class using the j `ava.lang.Class.forName()` method:
  Click here to view code image

```
try {
  Class.forName("oracle.jdbc.driver.OracleDriver");
} catch (ClassNotfoundException c) { }
```

- Load a database-specific JDBC driver class using the command-line `-D` option:
  Click here to view code image

  ```
  java -Djdbc.drivers=oracle.jdbc.driver.OracleDriver ...
  ```

From JDBC 4.0 onward, JDBC drivers are automatically loaded, and no explicit driver loading or registration is required. However, old-style code would still function for backward compatibility reasons.

24.4 Creating and Executing SQL Statements

The JDBC API defines three interfaces that represent SQL *statements*:

- `java.sql.Statement` represents a basic statement (p. 1523). This type of statement can be used to execute any SQL operation, such as CRUD or DDL operations. However, it is associated with performance and security issues. Thus it is not generally recommended, except for very simple operations that do not require dynamic parameterization and that are not reused within a program.
- `java.sql.PreparedStatement` is a subinterface of the `Statement` interface, that represents a statement capable of *substitutional parameterization* (p. 1526). This type of statement has the same capabilities as a basic statement, but is considered to be better from both performance and security perspectives, and thus is a recommended choice for representing SQL operations in general.
- `java.sql.CallableStatement` is a subinterface of the `PreparedStatement` interface, that represents a statement that invokes a *stored procedure* or *function* (p. 1530) and is also capable of *substitutional parameterization*.

The result of executing a `SELECT` statement is a table of rows represented by the following interface:

- `java.sql.ResultSet` that represents a table whose contents can be read and processed

All statements are created using a previously initialized JDBC connection object. A statement is created using one of the methods of the `Connection` interface shown below. Depending on the method, a specific statement is returned. Note that the statement object returned is `AutoCloseable`, and should be used in a `try`-with-resources statement to ensure proper closing of recourses such objects represent.

The following methods are defined in the `Connection` interface:

Click here to view code image

```
Statement         createStatement()
PreparedStatement prepareStatement(String sql)
CallableStatement prepareCall(String sql)
```

For overloaded versions of these methods in the `Connection` interface that can be used to customize the result set, see p. 1541.

Basic Statement

We first look at how to create and execute a basic statement. A `Statement` object should first be obtained from the `Connection` object. Note the declarations in the header of the `try`-with-resources statement that obtain a `Connection` object and create a `Statement` object, respectively. The `try`-with-resources statement will ensure that these resources are closed in the right order—that is, the reverse of their declarations.

Click here to view code image

```
try (Connection connection = DriverManager.getConnection(jdbcURL);
     Statement statement
         = connection.createStatement()) { // Obtain a Statement object.
  /* execute a query */
} catch (SQLException e) {
  e.printStackTrace();
}
```

Once a statement has been obtained, the following methods of the `Statement` interface can be used to execute and process a SQL query.

Click here to view code image

```
boolean execute(String sql)
```

The `execute()` method can be called to execute any operation, returning a `boolean` value which is `true` for the `SELECT` statement, `false` otherwise.

Click here to view code image

```
ResultSet executeQuery(String sql)
```

The `executeQuery()` method can only be called to execute `SELECT` statements, returning a `ResultSet` that is the result of executing the query. A `ResultSet` is also `AutoCloseable` and must be closed when done. Passing a non-`SELECT` operation will throw a `SQLException`.

Click here to view code image

```
int executeUpdate(String sql)
```

The `executeUpdate()` method can be called to execute any operations, except `SELECT` operations, returning an `int` value indicating the number of rows affected by the non-`SELECT` operation. Passing a `SELECT` operation will throw a `SQLException`.

```
ResultSet getResultSet()
```

Returns the `ResultSet` that was the result of executing a `SELECT` query using the current statement.

```
int getUpdateCount()
```

Returns the number of rows that were affected by the execution of a non-`SELECT` query using the current statement.

An appropriate execute method can be called depending on the type of SQL statement and on passing the SQL operation as a `String` parameter. The code below executes a `SELECT` query by calling the `execute()` method of the `Statement` interface, as shown at (1). This method returns a `boolean` value. The output from the code at (2) indicates that a `SELECT` query was processed.

Click here to view code image

```
final String jdbcURL = "jdbc:derby:musicDB";
try (Connection connection = DriverManager.getConnection(jdbcURL);
     Statement statement
         = connection.createStatement()) {      // Obtain a Statement object.
  String sql = "select * from compositions";    // SELECT query: select all rows.
  boolean isSelectStmt = statement.execute(sql);          // (1) Execute the query
  System.out.println("SELECT statement? " + isSelectStmt);// (2) SELECT statement?
```

```
                                                                      //     true
  } catch (SQLException e) {
    e.printStackTrace();
  }
```

When calling the `execute()` method with a SQL statement as a `String` parameter, it might not be possible to know a priori what type of SQL statement will be executed. However, this method would return `true` if the statement happens to be a `SELECT` query; otherwise, it returns `false`. As shown in the code below, which analyzes the resulting value in an `if` statement, a `ResultSet` object can be retrieved or the number of rows affected by the statement, depending on the result.

Click here to view code image

```
boolean isSelectStmt = statement.execute(sql);
if (isSelectStmt) {
  try (ResultSet resultSet
           = statement.getResultSet()) { // SELECT statement: retrieve ResultSet.
    System.out.println("SELECT statement: processing ResultSet");
  }
} else {                                                // Update statement:
  int rowCount = statement.getUpdateCount();            // Retrieve the number of
                                                        // rows affected.
  System.out.println("Update statement: Rows affected " + rowCount);
}
```

The method `executeQuery()` is used exclusively to execute a `SELECT` query. This method returns a `ResultSet` object that represents the rows retrieved as a result of executing the query. As a `ResultSet` is `AutoCloseable`, the method is called in a nested `try`-with-resources statement at (1) to ensure closure of the `ResultSet`.

Click here to view code image

```
try (Connection connection = DriverManager.getConnection(jdbcURL);
     Statement statement
         = connection.createStatement()) {         // Obtain a Statement object.
  String sql = "select * from compositions";       // SQL query: select all rows.

  try (ResultSet resultSet
           = statement.executeQuery(sql)) {    // (1) Nested try-with-resources
    System.out.println("Processing ResultSet");
  }
} catch (SQLException e) {
  e.printStackTrace();
}
```

The method `executeUpdate()` can be used to execute any SQL statement except a `SELECT` statement. An `int` value indicating the number of rows affected by the statement is returned as a result. The code below doubles the duration of each composition in the `compositions` table.

Click here to view code image

```
try (Connection connection = DriverManager.getConnection(jdbcURL);      // (1)
     Statement statement = connection.createStatement()) {              // (2)
  String sql = "update compositions set duration = duration * 2";       // (3)
  int count = statement.executeUpdate(sql);                             // (4)
  System.out.println("Rows modified: " + count);
} catch (SQLException e) {
  e.printStackTrace();
}
```

The code above illustrates the procedure to use a basic statement. The numbered comments below correspond to the numbered lines in the code:

1. Create a `Connection` with the `DriverManager.getConnection()` method.
2. Obtain a `Statement` from the connection by calling its `createStatement()` method.
3. Formulate a SQL operation.
4. Call the appropriate execute method of the statement, passing the SQL operation as the `String` parameter.

SQL Injection

Passing the SQL operation as a string in the execute method presents the risk of a *SQL injection*, as malicious SQL code can be injected into the query string. Assume that there is a basic statement that queries a table, where the query is dependent on a value:

Click here to view code image

```
ResultSet resultSet = statement.executeQuery(
    "select * from compositions where title like '" + value + "'");
```

A user may try to exploit this code by submitting a malicious value, such as the following:

Click here to view code image

```
String value = "SomeValue'; drop table compositions;";
```

The above query would be equivalent to the following statement, where the `SELECT` operation would be executed, followed by dropping the `compositions` table:

Click here to view code image

```
ResultSet resultSet2 = statement.executeQuery(
       "select * from compositions where title like '" +
       "SomeValue'; drop table compositions;");
```

To address this issue, one can use either a `PreparedStatement` object or the `Statement.enquoteLiteral()` method to sanitize the parameter value.

Prepared Statement

In contrast to the `Statement` object, which defines the SQL operation as a `String` parameter to the execute method, a `PreparedStatement` defines a SQL operation immediately at the point when the prepared statement is created with the `prepareStatement()` method. In other words, the SQL operation string is passed as an argument in the `prepareStatement()` method call, as shown at (1). The prepared statement executes the precompiled SQL operation at (2) by calling the `execute()` method.

Click here to view code image

```
String sql = "select * from compositions where duration > 200"; // SQL operation
try (Connection connection
         = DriverManager.getConnection(jdbcURL);                // Get connection
     PreparedStatement pStatement
         = connection.prepareStatement(sql)) {           // (1) Prepare statement
  boolean result = pStatement.execute();                 // (2) Execute
  System.out.println(result);
} catch (SQLException e) {
  e.printStackTrace();
}
```

Substitutional Parameterization

Consider the following code:

Click here to view code image

```
String sql = "select * from compositions where duration > ?"; // (1) SQL operation
                                                        // with 1 marker parameter.
```

```
PreparedStatement pStatement
    = connection.prepareStatement(sql);            // (2) A prepared statement
```

The SQL operation at (1) now contains a *marker parameter* (also known as *bind parameter* or *bind variable*). Prepared statements use *positional notation* where each question mark (`?`) indicates the position of a marker parameter starting with 1 from left to right. This could be any SQL operation, such as an `INSERT`, `UPDATE`, or `DELETE` statement with appropriate marker parameters. The SQL operation with marker parameters is passed as an argument to the `prepareStatement()` method to precom-pile this SQL operation into a prepared statement, as shown at (2).

However, before a prepared statement can be executed, all its parameters must be set —that is, all marker parameters must be substituted with a value. One can use the set methods of the `PreparedStatement` interface that are provided for all standard JDBC types to set the value of a marker parameter (see below).

Click here to view code image

```
pStatement.setInt(1,200);         // Sets the value of the marker parameter
                                  // at position 1 to 200.
```

Alternatively, the method `setObject()` can be used to supply a value and explicitly indicate the desired SQL type:

Click here to view code image

```
pStatement.setObject(1, 200, Types.INTEGER);
```

There is no requirement that parameters should be set in any specific order, as long as all parameters are set before executing the statement. Note that the position counter in JDBC is not 0-based as in the rest of the Java language, but starts at 1, which is typical in the database world. The driver converts the value of the Java data type in the method call to the corresponding value of the SQL data type. All parameter-setting methods throw a `SQLException` if the parameter index does not correspond to a marker parameter, or if it is called on a prepared statement that is already closed.

The `PreparedStatement` interface is a subinterface of the `Statement` interface. Once a prepared statement has been obtained with its precompiled SQL operation and values for its marker parameters have been supplied, the execute methods of the `PreparedStatement` interface can be used to execute the SQL operation. These execute methods are analogous to the corresponding methods in the `Statement` interface. Note that since the SQL operation to execute is already precompiled with the prepared statement, the execute methods of the `PreparedStatement` interface do not take any parameter. Specifying an argument will throw a `SQLException`, as will any marker parameter that is not substituted with a value.

```
boolean execute()
ResultSet executeQuery()
int executeUpdate()
```

The `PreparedStatement` interface provides the following set methods for setting the values of marker parameters:

Click here to view code image

```
void setString(int index, String value)
```

Sets a `String` value for the parameter designated by the index.

Click here to view code image

```
void setBoolean(int index, boolean value)
```

Sets a `boolean` value for the parameter designated by the index.

Click here to view code image

```
void setInt(int index, int value)
```

Sets an `int` value for the parameter designated by the index.

Click here to view code image

```
void setLong(int index, long value)
```

Sets a `long` value for the parameter designated by the index.

Click here to view code image

```
void setDouble(int index, double value)
```

Sets a `double` value for the parameter designated by the index.

Click here to view code image

```
void setBigDecimal(int index, BigDecimal value)
```

Sets a `BigDecimal` value for the parameter designated by the index.

Click here to view code image

```
void setDate(int index, Date value)
```

Sets a `Date` value for the parameter designated by the index.

Click here to view code image

```
void setObject(int index, Object value, int sqlType)
```

Sets any `Object` value for the parameter designated by the index, and the required SQL type which is a named constant in the `java.sql.Types` class—for example, `Types.INTEGER`, `Types.DATE`, or `Types.VARCHAR`.

Example 24.2 *Executing a Prepared Statement*

Click here to view code image

```
package dbDemo;
import java.sql.*;

public class XQTPreparedStatement {
  public static void main(String[] args) {
    final String jdbcURL = "jdbc:derby:musicDB";
    String sql = "select * from compositions where duration > ?";         // (1)
    try (Connection connection = DriverManager.getConnection(jdbcURL);     // (2)
        PreparedStatement pStatement = connection.prepareStatement(sql);) { // (3)
      pStatement.setInt(1, 200);                                            // (4)
      boolean result = pStatement.execute();                                // (5)
      System.out.println(result);
    } catch (SQLException e) {
      e.printStackTrace();
    }
    }
  }
```

Probable output:

```
true
```

The procedure to use a prepared statement is illustrated by **Example 24.2**. The numbered comments below correspond to the numbered lines of code in the example.

1. Formulate a SQL operation with marker parameters.
2. Create a `Connection` with the `DriverManager.getConnection()` method.
3. Create a `PreparedStatement` from the connection by calling its `prepareStatement()` method and passing the SQL operation as a parameter.
4. Set the values of all marker parameters.
5. Call the appropriate execute method of the prepared statement.

Note that a prepared statement can be reused by substituting different values for its marker parameters. In the `try`-with-resources statement in **Example 24.2**, we can include the following code to change the condition in the `WHERE` clause at (1) by substituting another value for the marker parameter before executing the prepared statement:

Click here to view code image

```
pStatement.setInt(1, 100);
result = pStatement.execute();
```

Substitutional parameterization allows SQL operations to be precompiled and reused, leading to more efficient execution. Supplying values using marker parameters is more flexible, faster, and safer than using a basic statement because the prepared statement is precompiled and marker parameters are not prone to SQL injection.

Example 24.3 illustrates prepared statements to execute `INSERT`, `UPDATE`, and `DELETE` statements with marker parameters that are defined first. A database connection and three prepared statements are created in the header of the `try`-with-resources statement, respectively. The marker parameters are set in each prepared statement, before the statement is executed using the `executeUpdate()` method of the `PreparedStatement` interface. The program prints three integers that indicate the number of rows affected by each SQL operation, which of course is dependent on the data in the `compositions` table.

Example 24.3 *Prepared Statement to Execute* `INSERT`, `UPDATE`, *and* `DELETE`

Click here to view code image

```
package dbDemo;
import java.sql.DriverManager;
import java.sql.SQLException;

public class PreparedStatementExecuteUpdate {
  public static void main(String[] args) {

    final String insSql = "insert into compositions VALUES(?, ?, ?)";
    final String updSql = "update compositions set title = ? where title = ?";
    final String delSql = "delete from compositions where duration = ?";

    final String jdbcURL = "jdbc:derby:musicDB";
    try (var connection = DriverManager.getConnection(jdbcURL);
        var pStatement1 = connection.prepareStatement(insSql);
        var pStatement2 = connection.prepareStatement(updSql);
        var pStatement3 = connection.prepareStatement(delSql)) {

        pStatement1.setInt(3, 150);
        pStatement1.setString(2, "Java Jazz");
        pStatement1.setString(1, "ushm91736991");
        int result1 = pStatement1.executeUpdate();
        System.out.println(result1);

        pStatement2.setString(1, "Java Jive");
        pStatement2.setString(2, "Java Jazz");
        int result2 = pStatement2.executeUpdate();
        System.out.println(result2);

        pStatement3.setInt(1, 200);
        int result3 = pStatement3.executeUpdate();
        System.out.println(result3);
    } catch (SQLException e) {
      e.printStackTrace();
    }
  }
}
```

Probable output:

```
1
1
0
```

Callable Statement

The `Callable` interface is a subinterface of the `PreparedStatement` interface. Callable statements are very similar to prepared statements because both use marker parameters. The main difference is that callable statements are used to invoke *named stored procedures and stored functions that reside on the database side*. The difference between a function and a procedure is that a function is intended to return a value. Both are precompiled code on the database side and can be invoked via a callable statement on the application side.

Syntax of Calling Stored Procedures and Functions

The syntax for calling stored procedures and functions is shown below at (1) and (2), respectively. The parameter list of a stored procedure or function can have any number of marker parameters, but the value returned by a function call is assigned to the first marker on the left-hand side of the `=` sign.

Click here to view code image

```
String procedureCall
    = "{ call some_procedure(?,?) }";          // (1) Call a stored procedure.
String functionCall = "? = { call some_function(?) }"; // (2) Call a function.
CallableStatement cStatement1 = connection.prepareCall(procedureCall); // (3)
CallableStatement cStatement2 = connection.prepareCall(functionCall);  // (4)
```

Callable statements are prepared analogous to prepared statements by calling the `prepareCall()` factory method of the `Connection` interface, as shown at (3) and (4). The marker parameters in the call must be set up in accordance with the parameters of the stored procedure or function before executing the callable statement.

Three kinds of marker parameters can be specified in a call to a stored procedure or function on the database side: IN, OUT, and INOUT parameters.

IN Parameters

These parameters pass values to the stored procedure or function. These are marker parameters whose values are initialized with the inherited `set`*`XXX`*`()` methods from the superinterface `PreparedStatement`. The IN parameters for callable statements are handled exactly as those for prepared statements—a prepared statement has only IN parameters. The parameter index starts at 1. It goes without saying that IN parameters must be initialized before the callable statement is executed.

Click here to view code image

```
cStatement1.setString(1, "Hi");              // Set value of IN parameter
                                             // at marker 1
cStatement2.setObject(2, 42, Types.INTEGER); // Set value of IN parameter
                                             // at marker 2
```

OUT Parameters

These parameters are marker parameters that hold values returned by a stored procedure or a function. Before executing callable statements with OUT parameters, the `registerOutputParameter()` method of the callable statement should be used to specify the expected types of the values returned in the OUT parameters.

Click here to view code image

```
cStatement1.registerOutParameter(2, Types.VARCHAR); // Register SQL VARCHAR as
                                                    // type for the
                                                    // 2nd marker parameter.
cStatement2.registerOutParameter(1, Types.INTEGER); // Register SQL INTEGER as
                                                    // type for the

                                                    // 1st marker parameter.
```

Click here to view code image

```
void registerOutParameter(int parameterIndex, int sqlType)
```

Registers the type of the OUT parameter at the marker parameter index to be of the specified JDBC type. The JDBC type is a constant specified by `java.sql.Types`—for example, `Types.INTEGER` and `Types.VARCHAR`.

Note there can be several OUT parameters, and the *first* marker parameter of a stored function call must also register the expected type of its return value.

The `Callable` interface inherits the execute methods of the `PreparedStatement` interface. The code below executes callable statements.

Click here to view code image

```
boolean result1 = cStatement1.execute();
boolean result2 = cStatement2.execute();
```

After execution, the `getXXX()` methods of the `CallableStatement` interface can be used to retrieve OUT parameter values of relevant types, based on the parameter index.

Click here to view code image

```
String result1 = cStatement1.getString(2);     // Retrieve value of OUT parameter
                                               // at marker 2.
```

The generic method `getObject()` can also be used to retrieve OUT parameter values, specifying both an index and the expected Java type of the value (see below).

Click here to view code image

```
String result2 = cStatement2.getObject(1, Integer.class);// Retrieve value of OUT
                                                         // parameter at marker 1.
```

Following are selected get methods in the `CallableStatement` interface to retrieve values of OUT parameters:

Click here to view code image

```
String getString(int paramIndex)
```

Retrieves the value of the designated marker parameter as a `String`.

Click here to view code image

```
int getBoolean(int paramIndex)
```

Retrieves the value of the designated marker parameter as a `boolean`.

Click here to view code image

```
int getInt(int paramIndex)
```

Retrieves the value of the designated marker parameter as an `int`.

Click here to view code image

```
double getLong(int paramIndex)
```

Retrieves the value of the designated marker parameter as a `long`.

Click here to view code image

```
double getDouble(int paramIndex)
```

Retrieves the value of the designated marker parameter as a `double`.

Click here to view code image

```
BigDecimal getBigDecimal(int paramIndex)
```

Retrieves the value of the designated marker parameter as a `BigDecimal`.

```
Date getDate(int paramIndex)
```

Retrieves the value of the designated marker parameter as a `Date`.

Click here to view code image

```
<T> T getObject(int paramIndex, Class<T> type)
```

Retrieves the value of the designated marker parameter as an object of the indicated Java type.

INOUT Parameters

An INOUT parameter acts as both an IN parameter to pass a value to the stored procedure or function and an OUT parameter to return a value from the stored procedure or function. Before the execution of the callable statement, its value can be initialized with a `set`*`XXX`*`()` method as for an IN parameter and can register the JDBC type of the return value by calling the `registerOutputParameter()` method as for an OUT parameter. The returned values can be retrieved using the appropriate `get`*`XXX`*`()` method as we have seen for OUT parameters.

Example 24.4 illustrates the basics of calling stored procedures and functions. The nitty-gritty of creating and deploying stored methods and procedures is database specific, and beyond the scope of this book.

Example 24.4 *Stored Procedures and Functions*

Click here to view code image

```
public void storedProcedureCall(Connection connection) {
  final String callProc
      = "{call longCompositionsProc(?, ?)}";            // (1) 1: IN 2:OUT
  try (CallableStatement cStmt = connection.prepareCall(callProc)) { // (2)
    int duration = 100;                                             // (3)
    cStmt.setInt(1, duration);                                      // (4)
    cStmt.registerOutParameter(2, Types.INTEGER);                   // (5)
    cStmt.execute();                                                // (6)
    int returnedValue = cStmt.getInt(2);                            // (7)
    System.out.println("Compositions with duration greater than "
        + duration + ": " + returnedValue);
  } catch (SQLException e) {
    e.printStackTrace();
  }
}

public void storedFunctionCall(Connection connection) {
  final String callFunc
      = "? = {call longCompositionsFunc(?)}";             // (8) 1:OUT 2:IN
  try (CallableStatement cStmt = connection.prepareCall(callFunc)) {
    cStmt.registerOutParameter(1, Types.INTEGER);                    // (9)
    int duration = 100;
    cStmt.setInt(2, duration);                                       // (10)
    cStmt.execute();
    int returnedValue = cStmt.getInt(1);                             // (11)
    System.out.println("Compositions with duration greater than "
        + duration + ": " + returnedValue);
  } catch (SQLException e) {

    e.printStackTrace();
  }
}
```

The two methods `storedProcedureCall()` and `storedFunctionCall()`, in **Example 24.4**, call a stored procedure and a stored function, respectively. Both the stored procedure named `longCompositionsProc` and the stored function named `longCompositionsFunc` compute the number of rows in the `compositions` table whose duration is greater than the duration specified in the call. We assume that the stored procedure and function are implemented and deployed on the database side. The result returned will depend on the state of the `compositions` table. The numbered comments below correspond to the numbered lines in **Example 24.4**.

1. The stored procedure `longCompositionsProc` has one IN and one OUT parameter designated by the marker positions 1 and 2, respectively.

2. The `prepareCall()` method prepares a callable statement for the stored procedure call.
3. The variable `duration` is declared and initialized with the value `100`.
4. The IN parameter at marker 1 is initialized with the value of the variable `duration` (`100`). We are interested in finding the number of rows with duration greater than 100.
5. The OUT parameter at marker 2 is registered to return a value of JDBC type `INTEGER` which will be retrieved as a value of Java type `int`.
6. The `execute()` method executes the stored procedure call in the callable statement.
7. After execution, the `int` value returned in the OUT parameter at marker 2 can be retrieved. Of course, this value depends on the state of the database.
8. The stored function `longCompositionsCall` has one IN and one OUT parameter designated by the marker positions 2 and 1, respectively.
9. Note that the OUT parameter at marker 1 is reserved for the return value from the stored function call, but its JDBC type must be registered, in this case JDBC type `INTEGER`.
10. The IN parameter at marker 2 is initialized with the value `100` as the lower limit for duration.
11. After execution, the returned value at marker 1 can be retrieved.

24.5 Processing Query Results

A *result set* represents a table of rows that is the result of executing a database query. By default, it is not updatable—that is, it cannot be modified. `SELECT` queries can be executed by the different types of statements that we have encountered so far to create result sets (p. 1522). It is quite obvious that basic and prepared statements can produce such results, but it is also possible to get a result set as an output value from a stored procedure or function.

The first thing to note is that a `ResultSet` object is `AutoCloseable`. Of course, it will be automatically closed and its resources released if handled in a `try`-with-resources statement. However, it can be prudent to call its `close()` method to release its resources immediately after use rather than wait for it to happen automatically. This might improve overall performance and facilitate scalability of the application.

Traversing the Result Set

Associated with a result set is a *cursor* that indicates the current row in the table of rows associated with the result set. This cursor should not be confused with a *database cursor* which represents a `ResultSet` object itself in database terminology. Initially the cursor is positioned just before the first row. A result set can be traversed by moving the cursor forward by one row by calling the `next()` method of the `ResultSet` interface.

- If the `next()` method returns `true`, the cursor has been moved forward and the row it points to is now the current row whose column values can be read from the result set using the get methods of the `ResultSet` interface (p. 1537).
- If the `next()` method returns `false`, the cursor now points past the last row and there are no more rows to traverse in the result set.

A result set can only be traversed once from the first row to the last row. The `next()` method can be used as the condition of a `while` loop to process all rows in the result set.

```
while (resultSet.next()) {
  /* Process current row. */
}
```

An `if` statement can be used to determine whether the result set is empty or not. For example, a `SELECT` statement parameterized with a primary key value would expect to return a single row. However, it is also possible that no rows are returned if the marker parameter specifies a nonexistent key value. In any case, the `next()` method should be invoked to check if there are any rows in the result set.

```
if (resultSet.next()) {
  /* Result set not empty. */
}
```

Result Set Navigation Methods

The following methods of the `ResultSet` interface can be used to navigate a result set. By default, the cursor can only be moved in the forward direction by calling the `next()` method. The other navigation methods (`previous()`, `next()`, `first()`, `last()`, `absolute()`, and `relative(int row)`) can only be used if scrolling is first enabled by setting the appropriate `ResultSet type` (Table 24.3, p. 1539).

```
boolean next()
```

Returns `true` if it is possible to move the cursor forward by one row from its current position; otherwise, it returns `false`. The return value `false` implies that either the cursor is after the last row or the result set is empty. Calling the `next()` method after it returns `false` will throw a `SQLException`.

```
boolean previous()
```

Returns `true` if it is possible to move the cursor to the previous row from its current position; otherwise, it returns `false`. The return value `false` implies that either the cursor is before the first row or the result set is empty. In both case, the cursor does not point to a valid row—in which case, calling a method on the `ResultSet` that requires a valid row will throw a `SQLException`.

```
boolean first()
```

Returns `true` if it is possible to move the cursor to the first row; otherwise, it returns `false`. The return value `false` implies that the result set is empty.

```
boolean last()
```

Returns `true` if it is possible to move the cursor to the last row; otherwise, it returns `false`. The return value `false` implies that the result set is empty.

Click here to view code image

```
boolean absolute(int rowNumber)
```

If the specified row number is positive, the cursor is moved to the absolute row number with respect to the beginning of the result set. Calling `absolute(1)` is the same as calling `first()`.

If the given row number is negative, the cursor is moved to the absolute row number with respect to the end of the result set. Calling `absolute(-1)` is the same as calling `last()`.

If the row number is zero, the cursor is moved to before the first row.

An attempt to position the cursor beyond the first or the last row in the result set leaves the cursor before the first row or after the last row, respectively.

```
boolean relative(int rows)
```

A positive/negative value in rows moves the cursor that many rows forward/ backward from the current position, respectively. Attempting to move beyond the first or the last row in the result set will set the cursor before the first row or after the last row, respectively.

Calling the method `relative(1)` is the same as calling the method `next()`.

Calling the method `relative(-1)` is the same as calling the method `previous()`.

Calling `relative(0)` is valid, but does not change the cursor position.

Optimizing the Fetch Size

To optimize the processing of rows in the result set that resulted from executing a query, only a certain number of these rows are fetched at a time from the database when they are needed by the result set. The default number to fetch, called the default *fetch size*, is set by the `Statement` object that created the result set.

The fetch size achieves a balance between the time it takes to fetch a row (depending on the row size), the total number of rows, and the number of round trips necessary between the database and the application to fetch all the rows.

However, we can improve the performance of processing the result set by providing a hint about the fetch size in the `setFetchSize()` method. Setting the fetch size only provides the JDBC driver with a hint as to the number of rows that should be fetched at a time from the database when more rows are needed by the result set. Setting the fetch size value to `0` is ignored by the JDBC driver, and it will then decide what fetch size to use. The fetch size can be set at the level of the `Statement` or the `ResultSet`.

Click here to view code image

```
PreparedStatement statement
    = connection.prepareStatement("select * from compositions");
statement.setFetchSize(20);             // Rows would be downloaded 20 at a time.
ResultSet resultSet = statement.executeQuery();
while (resultSet.next()){
 /* Rows are processed one at a time regardless of the fetch size */
  if (/* some condition */) {
    resultSet.setFetchSize(10);        // Rows would be downloaded 10 at a time.
  }
}
```

Calling the `next()` method will still only move the cursor to the next row, if there are still rows in the result set to traverse, regardless of the actual fetch size that is set. In other words, the program logic does not change in any way regardless of the fetch size optimization.

Extracting Column Values from the Current Row

The column values of the current row indicated by the cursor can be extracted using the get methods provided by the result set (see below).

Click here to view code image

```
String strValue1 = resultSet.getString(1);              // Using column index.
String strValue2 = resultSet.getString("column_name");    // Using column name.
```

We can also use the `getObject()` generic method, specifying either the column index or the column name together with the corresponding SQL type.

Click here to view code image

```
String strValue3 = resultSet.getObject(1, String.class);
String strValue4 = resultSet.getObject("column_name", String.class);
```

If the result set is not processed in a `try`-with-resources statement, consider closing the result set explicitly as soon as possible, after it has been processed.

```
resultSet.close();
```

Result Set Methods to Extract Column Values of the Current Row

The following get methods of the `ResultSet` interface can be used to extract the column values of the current row indicated by the cursor in the result set. The column index or the column label can be passed as a parameter to designate the appropriate column in the table. The `CallableStatement` interface provides analogous `get`*`XXX`*`()` methods to retrieve values of OUT parameters (p. 1530).

Selected get methods from the `ResultSet` interface to extract column values of the current row are shown below:

Click here to view code image

```
String getString(int columnIndex)
String getString(String columnLabel)
```

Retrieve the value of the column as a `String`.

Click here to view code image

```
int getBoolean(int columnIndex)
int getBoolean(String columnLabel)
```

Retrieve the value of the column as a `boolean`.

Click here to view code image

```
int getInt(int columnIndex)
int getInt(String columnLabel)
```

Retrieve the value of the column as an `int`.

Click here to view code image

```
double getLong(int columnIndex)
double getLong(String columnLabel)
```

Retrieve the value of the column as a `long`.

Click here to view code image

```
double getDouble(int columnIndex)
double getDouble(String columnLabel)
```

Retrieve the value of the column as a `double`.

Click here to view code image

```
BigDecimal getBigDecimal(int columnIndex)
BigDecimal getBigDecimal(String columnLabel)
```

Retrieve the value of the column as a `BigDecimal`.

Click here to view code image

```
Date getDate(int columnIndex)
Date getDate(int columnLabel)
```

Retrieve the value of the column as a `Date`.

Click here to view code image

```
<T> T getObject(int columnIndex, Class<T> type)
<T> T getObject(int columnLabel, Class<T> type)
```

Retrieve the value of the column as an object of the indicated Java type.

Example 24.5 *Processing a* `ResultSet`

Click here to view code image

```
package dbDemo;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.time.Duration;

import static java.lang.System.out;

public class ResultSetProcessing {
  public static void main(String[] args) {
    final String jdbcURL = "jdbc:derby:musicDB";
    final String sql = "select * from compositions where duration > ?";    // (1)
    try (var connection = DriverManager.getConnection(jdbcURL);            // (2)
        var pStatement = connection.prepareStatement(sql);) {              // (3)
      pStatement.setInt(1, 0);                                             // (4)
      var resultSet = pStatement.executeQuery();
      try (resultSet) {                                                    // (5)
        while (resultSet.next()) {                                         // (6)
          String isrc = resultSet.getString(1);                            // (7)
          String title = resultSet.getObject(2, String.class);             // (8)
          int duration = resultSet.getInt("duration");                     // (9)
          out.println("[" + isrc + ", " + title + ", " + duration + "]");  // (10)
        }
      } // Closes the result set.
    } catch (SQLException e) {                                             // (11)
      e.printStackTrace();
    } // Closes the prepared statement and the connection.
  }
}
```

Probable output from the program:

Click here to view code image

```
[ushm91736697, Vacation, 231]
[ushm91736698, Rage, 308]
```

```
[ushm91736699, Why Don't, 178]
```

Example 24.5 illustrates the main steps in the interaction between a Java application and a relational database. The example uses the `musicDB` database that has been used so far to illustrate salient features of programming this interaction. The numbered comments below correspond to the numbered code lines in Example 24.5.

1. Define the `SELECT` statement to execute, having one marker parameter for the second operand of the `boolean` expression.
2. Use the `try`-with-resources statement to declare the resources and to close the connection and the prepared statement afterward. Obtain the connection to the database.
3. Create the prepared statement in the header of the `try`-with-resources statement.
4. Substitute the marker parameter with a value in the `SELECT` statement at (1).
5. Use the nested `try`-with-resources statement to close the result set after processing it. The prepared statement is executed in the header of the `try`-with-resources statement, returning a result set. Any exception thrown will be propagated to the `catch` clause at (11) of the outer `try`-with-resources statement.
6. The `while` loop processes the rows in the result set.
7. Extract the column values of the current row denoted by the result set cursor by calling the appropriate get methods of the result set. Extract a string (`isrc`) from column 1 using the `getString()` method.
8. Extract a string (`title`) from column 2 using the `getObject()` method.
9. Extract an `int` value (`duration`) by column name using the `getInt()` method.
10. Print the column values of the current row.
11. Use the `catch` clause of the outer `try`-with-resources statement to catch any `SQLException`.

24.6 Customizing Result Sets

It is possible to customize certain features of the result set. Such features may not necessarily be available across all databases and in some cases may result in performance degradation. In this section we discuss possible customizations and what can be achieved by them.

The following features of the result set can be customized:

- *Result set type*: This feature allows customization of the *navigational direction* of the result set traversal and the *sensitivity* of the result set to reflect changes made in the underlying data while it remains open.
- *Result set concurrency*: This feature allows customization of the *updatability* of the result set—that is, whether the `ResultSet` object can be updated using the

`ResultSet` interface.
- *Result set holdability*: This feature allows customization of whether the result set is retained or closed when the current transaction is committed.

Valid values for these features are defined by static constants in the `ResultSet` interface, shown in Table 24.3.

Table 24.3 *Selected Constants Defined in the* `ResultSet` *Interface*

Result set type constants	**Description**
`TYPE_FORWARD_ONLY`	The type for a `ResultSet` object whose cursor may only move *forward*. This is the default `ResultSet` type`.`
`TYPE_SCROLL_INSENSITIVE`	The type for a `ResultSet` object that is *scrollable* (i.e., the cursor can move forward and backward), but generally *not sensitive* to changes made to the underlying data while the `ResultSet` is open.
`TYPE_SCROLL_SENSITIVE`	The type for a `ResultSet` object that is *scrollable* and generally *sensitive* to changes made to the underlying data while the `ResultSet` is open.
Result set concurrency constants	**Description**
`CONCUR_READ_ONLY`	The concurrency for a `ResultSet` object that may *not* be updated—that is, it is read-only. In other words, *updatability* of the `ResultSet` object is not allowed. This is the default `ResultSet` concurrency.
`CONCUR_UPDATABLE`	The concurrency for a `ResultSet` object that may be updated. That is, *updatability* of the `ResultSet` is allowed.
Result set holdability constants	**Description**
`CLOSE_CURSORS_AT_COMMIT`	Open `ResultSet` objects with this *holdability* will be *closed* when the current transaction is commit-

Result set type constants	Description
	ted. Such a `ResultSet` is said *not to be holdable*.
`HOLD_CURSORS_OVER_COMMIT`	Open `ResultSet` objects with this *holdability* will remain *open* when the current transaction is committed. Such a `ResultSet` said to be *holdable*.

Result Set Type

The *result set type* refers to the direction of navigation in the result set and whether changes in the underlying data get reflected in an open result set.

By default, the result set type is set to `ResultSet.TYPE_FORWARD_ONLY`, which makes the cursor move in the forward direction only, from before the first row and successively to after the last row. This essentially means that the traversal of the rows in the result set is only possible with the `next()` method. This option is considered to be a safe default choice for all result sets because forward-only progression through rows does *not* require the Java application to download the whole *database cursor* (i.e., all rows that comprise the result of the query on the database). It also allows the fetch size to be customized to achieve better performance, especially in a situation where the query might result in a large number of rows.

With the option `ResultSet.TYPE_FORWARD_ONLY`, the result set may reflect changes in the underlying database that occur while the result set is being processed. Not all databases are capable of reflecting changes.

Two other options (`ResultSet.TYPE_SCROLL_INSENSITIVE`, `ResultSet.TYPE_SCROLL_SENSITIVE`) are also available for the result set type, both of which enable scrolling—that is, the arbitrary navigation direction in the result set using the `previous()`, `next()`, `first()`, `last()`, `absolute()`, and `relative()` methods (p. 1534). The cursor can move both forward and backward relative to the current position, or to an absolute position. In order to jump to a particular row in the result set, the entire result set may need to be downloaded first from the database, which can result in performance degradation when dealing with very large result sets returned by the query.

The options `ResultSet.TYPE_SCROLL_INSENSITIVE` and `ResultSet.TYPE_SCROLL_SENSITIVE` make the result set *insensitive* and *sensitive*, respectively, to changes made in the underlying data source while the result set is open.

The `DatabaseMetaData.supportsResultSetType()` method can be called to determine whether a particular result set type is actually supported by the database (p. 1543).

Result Set Concurrency

The *result set concurrency* feature enables or disables whether the result set can be updated or not—that is, it indicates the *concurrency mode* of the result set. By default, the result set concurrency is set to `ResultSet.CONCUR_READ_ONLY`, which means the result set cannot be updated. Alternatively, updatability can be enabled by setting the result set concurrency to be `ResultSet.CONCUR_UPDATABLE`. This allows the result set to be modified.

The `DatabaseMetaData.supportsResultSetConcurrency()` method can be called to identify valid combinations of result set type and result set concurrency supported by the current JDBC driver (p. 1543).

Result Set Holdability

The *result set holdability* refers to whether the result set may remain open or be closed when the current transaction is committed (p. 1545). The default value for the result set holdability is actually database dependent. One option is `ResultSet.HOLD_CURSORS_OVER_COMMIT`, which allows the result sets (cursors) to remain open when the transaction is committed.

Alternatively, holdability can be set to `ResultSet.CLOSE_CURSORS_AT_COMMIT`, which indicates that `ResultSet` objects (cursors) should be closed upon commit. Closing cursors can result in better performance for some applications.

The `DatabaseMetaData.supportsResultSetHoldability()` method can be called to identify the holdability supported by the current JDBC driver (p. 1543).

Please note that not all databases and JDBC drivers support these customizations. Such customizations can be specified when the statement object is created by calling the relevant factory method of the `Connection` interface. Analogous methods with the same result set customizations are also available for a `Statement` and a `Callable`.

The following overloaded method is defined in the `Connection` interface:

Click here to view code image

```
PreparedStatement prepareStatement(String sql,
                          int resultSetType, int resultSetConcurrency,
                          int resultSetHoldability)
```

Example 24.6 illustrates using different `ResultSet` options. Note that in this example, the call to the `updateRow()` method will result in the column value of the underlying row in the database to be updated with the column value of the current row in the result set which was just updated. In other words, the `updateRow()` method commits the changes to the database and thereby the result set is also closed as we have specified `ResultSet.CLOSE_CURSORS_AT_COMMIT`. However, if the automatic commit is disabled, as shown at (1), the changes in the database due to the `updateRow()` method will not be committed and the result set will not be closed until an explicit call to the `commit()` method is executed, as shown at (2) (p. 1545).

Example 24.6 *Using* `ResultSet` *Options*

Click here to view code image

```
package dbDemo;
import java.sql.DriverManager;

import java.sql.ResultSet;
import java.sql.SQLException;
public class ResultSetCustomization {
  public static void main(String[] args) {
    final String jdbcURL = "jdbc:derby:musicDB";
    try (var connection = DriverManager.getConnection(jdbcURL);
        var statement = connection.prepareStatement(
            "select duration from compositions where title = ?",
            ResultSet.TYPE_FORWARD_ONLY,        // Forward direction. May reflect
                                                // database changes.
            ResultSet.CONCUR_UPDATABLE,         // Result set is updatable.
            ResultSet.CLOSE_CURSORS_AT_COMMIT   // Result set is closed on commit.
            )) {
      connection.setAutoCommit(false);          // (1) Disables automatic commit.
      statement.setString(1,"Vacation");
      try (ResultSet resultSet = statement.executeQuery();) {
        if (resultSet.next()) {                 // Moves forward one row.
          resultSet.updateInt("duration", 147); // Updates the current row
                                                // in the result set.
          resultSet.updateRow();                // Updates the underlying
                                                // database.
          System.out.println("Updated");
        }
        connection.commit();                    // (2) Also closes the result set.
      }
    } catch (SQLException e) {
      e.printStackTrace();
    }
    }
  }
```

Possible program output:

```
Updated
```

24.7 Discovering Database and `ResultSet` Metadata

The JDBC API allows Java programs to interact with many different types of databases. Obviously, different database providers have different capabilities and default behaviors, and may or may not support certain SQL features. It is possible to obtain such information using the `java.sql.DatabaseMetaData` object. This object is obtained from the JDBC `Connection` and can be used to investigate various capabilities and many other properties, such as support for type, concurrency, and holdability of a result set.

Following selected methods from the `DatabaseMetaData` API can be used to examine the capabilities of a database:

Click here to view code image

```
String getDatabaseProductName()
```

Retrieves the name of this database product.

Click here to view code image

```
String getDatabaseProductVersion()
```

Retrieves the version number of this database product.

```
String getSQLKeywords()
```

Retrieves a comma-separated list of all SQL keywords supported by this database.

Click here to view code image

```
boolean supportsResultSetType(int resultSettype)
```

Retrieves whether this database supports the given result set type.

Click here to view code image

```
boolean supportsResultSetHoldability(int holdability)
```

Retrieves whether this database supports the given result set holdability.

Click here to view code image

```
boolean supportsResultSetConcurrency(int resultSettype, int concurrency)
```

Retrieves whether this database supports the given combination result set type/concurrency type.

In addition to the database metadata, a `ResultSetMetaData` object can be used to discover information about the structure of a given result set.

Following selected methods from the `ResultSetMetaData` API can be used to examine the structure of the result set:

```
int getColumnCount()
```

Returns the number of columns in this result set.

Click here to view code image

```
String getColumnName(int column)
```

Returns the name of the designated columns in this result set.

```
int getColumnType(int column)
```

Returns the type of the designated columns in this result set. The `int` value returned corresponds to the SQL type designated by constants in `java.sql.Types`. The values `12` and `4` represent the SQL types `VARCHAR` and `INTEGER`, respectively.

Example 24.7 prints various metadata about the database and the result set.

Example 24.7 *Discovering Metadata for the Database and* `ResultSet`

Click here to view code image

```
package dbDemo;
import java.sql.DatabaseMetaData;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.ResultSetMetaData;
import java.sql.SQLException;

public class DBMetadata {
  public static void main(String[] args) {
    final String jdbcURL = "jdbc:derby:musicDB";
    try (var connection = DriverManager.getConnection(jdbcURL)) {

      // Print various information about the database:
      DatabaseMetaData dbMetaData
          = connection.getMetaData();                  // Obtain DatabaseMetaData.
      String dbName                = dbMetaData.getDatabaseProductName();
      String dbVersion             = dbMetaData.getDatabaseProductVersion();
      String sqlKeywords           = dbMetaData.getSQLKeywords();
      boolean forwardOnly = dbMetaData.supportsResultSetType(
          ResultSet.TYPE_FORWARD_ONLY);
      boolean cursorOpen  = dbMetaData.supportsResultSetHoldability(
          ResultSet.HOLD_CURSORS_OVER_COMMIT);
      boolean forwardUpdate = dbMetaData.supportsResultSetConcurrency(
          ResultSet.TYPE_FORWARD_ONLY,
          ResultSet.CONCUR_UPDATABLE);

      System.out.println("Various info about the database:");
      System.out.println("Database name: " + dbName);
      System.out.println("Version: " + dbVersion);
      System.out.println("SQL keywords: " + sqlKeywords);
      System.out.println("TYPE_FORWARD_ONLY: " + forwardOnly);
      System.out.println("HOLD_CURSORS_OVER_COMMIT: " + cursorOpen);
      System.out.println("TYPE_FORWARD_ONLY/CONCUR_UPDATABLE: "
          + forwardUpdate);

      // Create a ResultSet and print its structure:
      String sql = "select * from compositions where duration > ?";
      try (var pStatement = connection.prepareStatement(sql);) {
        pStatement.setInt(1, 100);
        var resultSet = pStatement.executeQuery();
        try (resultSet){
          System.out.println("Structure of ResultSet:");
          ResultSetMetaData rsMetaData
              = resultSet.getMetaData();              // Obtain ResultSetMetadata.
```

```
            int columnCount = rsMetaData.getColumnCount();
            System.out.println("Number of columns:" + columnCount);
            for (int i = 1; i <= columnCount; i++){
              String name = rsMetaData.getColumnName(i);
              int type = rsMetaData.getColumnType(i);   // Value of Types constant.
              System.out.println(name + ": " + type);
            }
          }
        }
      } catch (SQLException e) {
        e.printStackTrace();
      }
    }
  }
```

Probable output from the program:

Click here to view code image

```
Various info about the database:
Database name: Apache Derby
Version: 10.15.2.0 - (1873585)
SQL keywords: ALIAS,BIGINT,BOOLEAN,CALL,CLASS,COPY,DB2J_DEBUG,EXECUTE,… (EDITED)
TYPE_FORWARD_ONLY: true
HOLD_CURSORS_OVER_COMMIT: true
TYPE_FORWARD_ONLY/CONCUR_UPDATABLE: true
Structure of ResultSet:
Number of columns:3
ISRC: 12
TITLE: 12
DURATION: 4
```

24.8 Implementing Transaction Control

The JDBC protocol allows the application to control *database transactions*. A *transaction* is a set of statements that is executed as a logical unit. Either all changes due to the execution of statements in the transaction are committed to the database, or none of the changes made in the transaction are committed. A *commit* results in the changes made in the transaction being made permanent in the database. A commit thereby ends a transaction. A *rollback* results in undoing all changes made in the transaction—that is, restoring the database state to what it was before the transaction commenced. A *save-point* defines a logical rollback point *within* a transaction. A *rollback* can be used to undo all changes made since a savepoint was set up in the transaction—that is, restoring the database state to what it was before the save-point was set up.

By default, JDBC connections are in *auto-commit mode*, which means that commit occurs automatically when the statement processing completes successfully. To change this behavior, the following method call can be used:

Click here to view code image

```
connection.setAutoCommit(false);
```

Once auto-commit is turned off, transactions can be controlled using the `rollback()` and `commit()` methods of the `Connection` interface. Each transaction can include any number of insert, update, and delete actions and must end in either a commit that would make all pending transaction changes permanent, or a rollback if the pending transaction changes should be discarded. Some databases also support save-points, which provides an ability to partially roll back a transaction, discarding only those pending changes that occurred in the transaction after the savepoint was set up. Whether a given database supports savepoints can be determined by calling the `DatabaseMetaData.supportsSavepoints()` method. It is important to remember that normal closing of a connection implicitly causes the transaction to commit. However, the transaction will be rolled back if the program terminates with an uncaught exception.

If an exception is intercepted, the Java runtime considers this exception to be successfully handled and therefore resumes normal execution after the catch block. It is worth reciting the mantra that the JDBC interfaces, including the `Connection` interface, implement `AutoCloseable`, which means that all statements and connections are closed implicitly following the execution of the `try`-with-resources statement. This implies that if the transaction was not explicitly rolled back inside the exception handler, then by default, the transaction will be committed.

Following selected methods of the `java.sql.Connection` interface can be used to control transactions:

Click here to view code image

```
void setAutoCommit(boolean autoCommit)
```

Enables or disables *auto-commit mode* of this connection. The value `true` enables auto-commit mode and `false` disables it.

Click here to view code image

```
Savepoint setSavepoint()
Savepoint setSavepoint(String name)
```

Create an unnamed or a named savepoint in the current transaction and returns the new `Savepoint` object that represents it, respectively.

Click here to view code image

```
void rollback()
void rollback(Savepoint savepoint)
```

Undo all changes made in the current transaction or all changes made since the named savepoint was set up in the current transaction, respectively.

```
void commit()
```

Makes all changes made since the previous commit/rollback permanent.

Example 24.8 *Controlling Transactions*

Click here to view code image

```
package dbDemo;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.Savepoint;

public class Transactions {
  public static void main(String[] args) {
    final String jdbcURL = "jdbc:derby:musicDB";
    try (var connection = DriverManager.getConnection(jdbcURL)) {

      // SQL statements:
      final String insSql = "insert into compositions VALUES(?, ?, ?)";
      final String updSql = "update compositions set title = ? where title = ?";
      final String delSql = "delete from compositions where duration = ?";

      // Create statements:
      try (var insStatement = connection.prepareStatement(insSql);
          var updStatement = connection.prepareStatement(updSql);
          var delStatement = connection.prepareStatement(delSql);) {

        connection.setAutoCommit(false);            // (1) Auto-commit disabled.
```

```
          insStatement.setInt(3, 150);                // (2) Insert a new row.
          insStatement.setString(2, "Java Jazz");
          insStatement.setString(1, "ushm91736991");
          int insResult = insStatement.executeUpdate();
          System.out.println("INSERT: " + insResult);

          updStatement.setString(1, "Java Jive");     // (3) Update an existing row.
          updStatement.setString(2, "Rage");
          int updResult = updStatement.executeUpdate();
          System.out.println("UPDATE: " + updResult);

          Savepoint savePoint = connection.setSavepoint(); // (4) Set a savepoint.
          delStatement.setInt(1, 178);                     // (5) Delete a row.
          int delResult = delStatement.executeUpdate();
          System.out.println("DELETE: " + delResult);

          connection.rollback(savePoint);   // (6) Roll back to safepoint.
          connection.commit();              // (7) Commits only (2) and (3).
        } catch (SQLException e) {
          connection.rollback();            // (8) Roll back any changes.
        }
      } catch (SQLException e) {
        e.printStackTrace();
      }
    }
  }
```

Output from the program:

```
INSERT: 1
UPDATE: 1
DELETE: 1
```

Example 24.8 illustrates how transactions can be controlled with commit and rollback operations. The numbered comments below correspond to the numbered code lines in Example 24.8.

The following rows are in the `compositions` table at the start.

Click here to view code image

```
[ushm91736697, Vacation, 231]
[ushm91736698, Rage, 308]
[ushm91736699, Why Don't, 178]
```

1. Auto-commit is disabled. A new transaction starts.
2. A new row is inserted:
   Click here to view code image

   ```
   [ushm91736991, Java Jazz, 150}
   ```

3. The title `"Rage"` of an existing row:

   ```
   [ushm91736698, Rage, 308]
   ```

   is updated to `"Java Jive"`:
   Click here to view code image

   ```
   [ushm91736698, Java Jive, 308]
   ```

4. A savepoint is set.
5. The row with the duration `178` is deleted:
   Click here to view code image

   ```
   [ushm91736699, Why Don't, 178]
   ```

6. A rollback is performed, rolling back the deletion of the row at (5) that was performed since the last savepoint was set up at (4).
7. A commit is performed, committing the insertion at (2) and the update at (3), leaving the database in the following state and ending the transaction that started at (1):
   Click here to view code image

   ```
   [ushm91736697, Vacation, 231]
   [ushm91736698, Java Jive, 308]
   [ushm91736699, Why Don't, 178]
   [ushm91736991, Java Jazz, 150]
   ```

8. In case a `SQLException` is thrown in the inner `try`-with-resources statement, all changes are rolled back.

Review Questions

24.1 Which of the following statements is true about the JDBC `ResultSet`?

Select the one correct answer.

a. The method `next()` throws an exception when no rows were returned by the query.

b. The method `next()` throws an exception when it is invoked more times than the number of rows in the result set.

c. The method `absolute(1)` navigates to the first row of the result set, if there are rows in the result set.

d. The method `absolute(0)` navigates to the first row of the result set, if there are rows in the result set.

24.2 What is the correct order in which the following JDBC resources are closed?

Select the one correct answer.

a. `Connection, Statement, ResultSet`

b. `Statement, ResultSet, Connection`

c. `ResultSet, Statement, Connection`

d. Closure order is irrelevant.

e. Closure order is irrelevant when auto-commit mode is enabled.

24.3 Assume that a database has the following `questions` table with the specified columns and rows of data:

id	**question**	**answer**
`(INTEGER, PRIMARY KEY)`	`(VARCHAR, NOT NULL)`	`(VARCHAR)`
`101`	`What was Deep Thought's answer?`	`42`
`103`	`Where is Wally?`	
`102`	`Is Waldo older than Wally?`	`He is younger.`
`106`	`Which one is Wally?`	`He is the one in a bobble hat.`

id	question	answer
(INTEGER, PRIMARY KEY)	(VARCHAR, NOT NULL)	(VARCHAR)
107	Where am I?	Right here.

What will be the result of executing the following code (assuming that `jdbcUrl`, `username`, and `password` are correctly initialized)?

Click here to view code image

```
String findAnswers = "select * from questions where question like ?";
try (Connection c = DriverManager.getConnection(jdbcUrl, username, password);
     PreparedStatement ps = c.prepareStatement(findAnswers)) {
  ps.setString(1, "Where%");
  try (ResultSet rs = ps.executeQuery()) {
    while (rs.next()) {
      String question = rs.getObject(2, String.class);   // (1)
      String answer = rs.getObject(3, String.class);     // (2)
      answer = (answer == null) ? "No answer." : answer; // (3)
      System.out.println(question + " - " + answer);
    }
  }
} catch (SQLException e) { e.printStackTrace(); }
```

Select the one correct answer.

a. The program will print one line.

b. The program will print two lines.

c. The program will throw an exception at (1).

d. The program will throw an exception at (2).

e. The program will throw an exception at (3).

24.4 Assume that a database has the following `questions` table with the specified columns and rows of data:

id	question	answer
(INTEGER, PRIMARY KEY)	(VARCHAR, NOT NULL)	(VARCHAR)
101	What was Deep Thought's answer?	42
102	Where is Wally?	

What will be the result of executing the following code (assuming that `jdbcUrl`, `user-name`, and `password` are correctly initialized)?

Click here to view code image

```
String findQuestion = "select * from questions where id = ?";
int id = 103;
try (Connection c = DriverManager.getConnection(jdbcUrl, username, password);
     PreparedStatement ps = c.prepareStatement(findQuestion)) {
  ps.setInt(1, id);                                    // (1)
  try (ResultSet rs = ps.executeQuery()) {
    if (rs.next()) {                                   // (2)
      String question = rs.getObject(2, String.class);
      System.out.println(question);
    }
  }
} catch (SQLException e) { e.printStackTrace(); }
```

Select the one correct answer.

a. The program will print one line.

b. The program will print nothing.

c. The program will throw an exception at (1).

d. The program will throw an exception at (2).

24.5 Assume that a database has the following `questions` table with the specified col-umns and rows of data:

Which of the following statements is true about executing the following code (assuming that `jdbcUrl`, `username`, and `password` are correctly initialized)?

`id`	`question`	`answer`

(INTEGER, PRIMARY KEY)	(VARCHAR, NOT NULL)	(VARCHAR)
101	What was Deep Thought’s answer?	42
102	Where is Wally?	

Click here to view code image

```
String findQuestion = "select question from questions where id = ?";
String provideAnswer = "update questions set answer = ? where id = ?";
int id = 102;
try (Connection c = DriverManager.getConnection(jdbcUrl, username, password);
     PreparedStatement ps1 = c.prepareStatement(findQuestion);
     PreparedStatement ps2 = c.prepareStatement(provideAnswer)) {
  c.setAutoCommit(false);
  ps2.setString(1, "Look and you will find!");
  ps2.setInt(2, id);
  ps2.executeUpdate();
  ps1.setInt(2, id);
  try (ResultSet rs = ps1.executeQuery()) {
    while (rs.next()) {
      String question = rs.getString(1);
      String answer = rs.getString(2);
      System.out.println(question + " - " + answer);
    }
  }
  c.commit();
} catch (SQLException e) {
  e.printStackTrace();
}
```

Select the one correct answer.

a. The program updates one row, executes the query, and commits the transaction.

b. The program updates one row, throws a `SQLException`, and commits the transaction.

c. The program updates one row, throws a `SQLException`, and rolls back the transaction.

d. The program throws a `SQLException`, executes the query, and commits the transaction.

e. The program throws a `SQLException`, executes the query, and rolls back the transaction.

24.6 Which of the following statements is true about the following code (assuming that `jdbcUrl`, `username`, `password`, and `sqlQuery` are correctly initialized)?

Click here to view code image

```
try (Connection connection
         = DriverManager.getConnection(jdbcUrl, username, password);
    Statement statement = connection.createStatement();
    ResultSet resultSet = statement.executeQuery(sqlQuery)) {
  while (resultSet.next()) {
    /* Do nothing. */
  }
} catch (SQLException e) {
  e.printStackTrace();
}
```

Select the one correct answer.

a. The program will attempt to close the connection, statement, and result set objects, in that exact order.

b. The program will attempt to close the result set, statement, and connection objects, in that exact order.

c. The program will attempt to close the statement, result set, and connection objects, in that exact order.

d. The program will attempt to close the statement, connection, and result set objects, in that exact order.

e. The program will attempt to close the connection, statement, and result set objects, in an undetermined order.

24.7 Which of the following statements is true?

Select the one correct answer.

a. The method `executeQuery()` of the `PreparedStatement` interface accepts marker parameter values.

b. Each `PreparedStatement` object represents a single SQL statement.

c. Each `PreparedStatement` object represents one or more SQL statements.

d. Setting new values for marker parameters in a `PreparedStatement` object will produce a new SQL statement.

24.8 Given the following code:

Click here to view code image

```
String findQuestion
    = "select question, answer from questions where answer is null";
String jdbcUrl = "jdbc:thin:oracle:@localhost:1521:qa";
String username = "joe";
String password = "welcome1";
try (Connection c = DriverManager.getConnection(jdbcUrl, username, password);
     PreparedStatement ps = c.prepareStatement(findQuestion,
         ResultSet.TYPE_FORWARD_ONLY,
         ResultSet.CONCUR_UPDATABLE,
         ResultSet.CLOSE_CURSORS_AT_COMMIT)) {
  c.setAutoCommit(false);
  try (ResultSet rs = ps.executeQuery()) {
    while (rs.next()) {
      rs.updateString("answer", "no answer");
      rs.updateRow();
    }
  }
  c.commit();
} catch (SQLException e) {
  e.printStackTrace();
}
```

Which of the following statements are true about this code (assuming the database supports all designated `ResultSet` features)?

Select the two correct answers.

a. It selects all rows from the `questions` table that have no value set for the `answer` column.

b. It updates all rows in the `questions` table that have no value set for the `answer` column.

c. It sets the `"no answer"` value for the `answer` column in all rows in the `questions` table.

d. It will roll back changes if an exception is thrown inside the `while` loop.

24.9 Given the following code:

Click here to view code image

```
String jdbcUrl = "jdbc:thin:oracle:@localhost:1521:qa";
String username = "joe";
String password = "welcome1";
try (Connection c = DriverManager.getConnection(jdbcUrl, username, password);
    var ps = c.prepareStatement(
                   "update questions set answer = ? where question = ?")) {
  ps.setString(2, "What?");                    // (1)
  try (ResultSet rs = ps.executeQuery()) {  // (2)
    while (rs.next()) {                        // (3)
      rs.setString(1, "42");                   // (4)
    }
  }
} catch (SQLException e) {
  e.printStackTrace();
}
```

Which of the following statements is true about this program?

Select the one correct answer.

a. It updates rows in the `questions` table, setting the `answer` column value to `"42"`.

b. It throws an exception at (1).

c. It throws an exception at (2).

d. It throws an exception at (3).

e. It throws an exception at (4).

24.10 Given the following code:

Click here to view code image

```
String jdbcUrl = "jdbc:thin:oracle:@localhost:1521:qa";
String username = "joe";
String password = "welcome1";
try (Connection c = DriverManager.getConnection(jdbcUrl, username, password);
    PreparedStatement ps = c.prepareStatement(
        "update questions set answer = ? where question = ?")) {
  ps.setString(2,"What?");            // (1)
  ps.setString(1,"42");               // (2)
```

```
    ps.execute();                      // (3)
  } catch (SQLException e) {
    e.printStackTrace();
  }
```

Which of the following statements is true about this program?

Select the one correct answer.

a. It updates rows in the `questions` table, setting the `answer` column value to `"42"`.

b. It throws an exception at (1).

c. It throws an exception at (2).

d. It throws an exception at (3).

24.11 Which of the following statements are true? Select the three correct answers.

a. A given `PreparedStatement` can only be executed once.

b. A given `PreparedStatement` can be executed multiple times.

c. A given `PreparedStatement` can be executed using both `execute()` and `executeQuery()` methods.

d. A given `PreparedStatement` can be executed using both `execute()` and `executeUpdate()` methods.

e. A given `PreparedStatement` can be executed using both `executeUpdate()` and `executeQuery()` methods.

24.12 Which of the following statements is true?

Select the one correct answer.

a. All databases support forward-only type of result sets.

b. Forward-only result sets automatically reflect database changes in an open result set.

c. By default, result set objects are scroll sensitive.

d. Result sets are always closed at commit.

24.13 Which of the following statements is true about `ResultSet` methods?

Select the one correct answer.

a. Calling the method `relative(1)` is equivalent to calling the method `next()`.

b. Calling the method `relative(1)` is equivalent to calling the method `absolute(0)`.

c. Calling the method `relative(0)` is equivalent to calling the method `previous()`.

d. Calling the method `relative(-1)` is equivalent to calling the method `absolute(-1)`.