9 Building database applications with JDBC

Exam objectives covered in this chapter	What you need to know
[9.1] Describe the interfaces that make up the core of the JDBC API (including the `Driver`, `Connection`, `Statement`, and `ResultSet`) and their relationships to provider implementations	Identification of the interfaces that make up the core of the JDBC API How an application creates objects of these interfaces
[9.2] Identify the components required to connect to a database using class `DriverManager` (including the JDBC URL)	The steps and components required to connect to a database using class `DriverManager`, for JDBC API version 3.0 and before and JDBC API version 4.0 and later
[9.3] Submit queries and read results from the database (including creating statements, returning result sets, iterating through the results, and properly closing result sets, statements, and connections)	How to execute SQL statements against a database using the interfaces `Connection`, `Statement`, and `ResultSet` How to combat the `SQLException` thrown by database operations How to use `try-catch-finally` and `try`-with-resources statements to close database resources
[9.4] Use JDBC transactions (including disabling auto-commit mode, committing and rolling back transactions, and setting and rolling back to savepoints)	How to create transactions, `Savepoint`, commit, and roll back changes Effects of auto-commit mode on a transaction
[9.5] Construct and use `RowSet` objects using class `RowSetProvider` and the `RowSetFactory` interface	How to create `RowSet` objects using class `RowSetProvider` and the interface `RowSetFactory` The specialized types of `RowSet` objects

Exam objectives covered in this chapter	What you need to know
[9.6] Create and use `PreparedStatement` and `CallableStatement` objects	How to use `PreparedStatement` and `CallableStatement` statements to execute static and dynamic SQL statements and stored procedures

Given multiple options, choosing the right technology to connect a Java application to a database is challenging.

At present, there are multiple approaches and technologies to connect a Java application with a database. Examples include the EJB Entity Beans, Java Persistence API (JPA), Java Database Objects (JDO), and others. Hibernate™ is a popular JPA implementation. Hibernate is an Object Relational Mapping (ORM) framework for the Java language. Each of these approaches or implementations has its own set of advantages and disadvantages. For example, you can't use JPA entities or EJB Entity Beans with Java's Standard Edition (JSE). Apart from accessibility, other factors like license fees, ease of use, backward compatibility, and availability of expertise in a particular technology might affect your choice of approach or API to use to connect to a database.

The exam has included a simple industry standard to connect your Java application to any tabular data: Java Database Connectivity (JDBC). JDBC has been around since Java version 1.1.

Before I go further, let me clarify that the exam won't query you on your capabilities to connect tables in a database or write complex SQL queries. The exam will focus on testing your capabilities on how to connect a Java application to a database using the JDBC API. It will query you on the prerequisites for establishing a connection, such as the existence of JDBC drivers, the connection URL, submitting queries, reading results, performing transactions, and using multiple flavors of results (`ResultSet` and `RowSet`) and statements (`Statement`, `CallableStatement`, and `PreparedStatement`).

Let's get started with an introduction to the JDBC API so that you understand what it is and how it connects your Java class to a database, before getting our hands dirty with the actual code.

9.1 Introduction

JDBC is a Java API. It's an industry standard for database-independent connections from Java applications to tabular data stored in relational databases, flat files, or spreadsheets. The JDBC API defines multiple interfaces, which must be implemented by the database vendors or a third party, and is supplied as a driver. As shown in figure 9.1, you can think of the bridge between a Java application and a database to be made of the JDBC API and driver implementation classes. The JDBC API provides your classes with the interfaces and classes to connect to a database and process the results. The actual classes that implement the interfaces in the JDBC API are defined by the driver for a database.

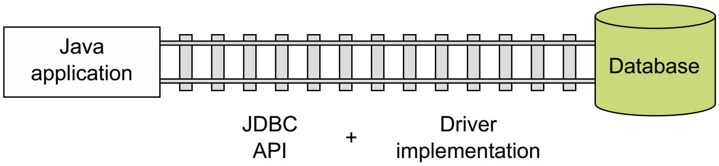


Figure 9.1 JDBC API and driver implementation classes are a bridge between a Java application and a database

Any Java class can use JDBC to connect with a database using a driver. JDBC has been around since 1997, and was added to JSE 1.1 as JDBC 1.0. Since its first version, multiple enhancements have been made to the JDBC API. Java 7 ships with JDBC version 4.1. JDBC classes are defined in Java packages `java.sql` and `javax.sql`.

9.1.1 JDBC API overview

By using the classes and interfaces from the JDBC API, Java classes can connect to a data source, execute SQL (Structured Query Language, the standard language for relational database management systems, RDBMS), and process the results. At the heart of the JDBC API lies the use of SQL to create, modify, and delete database objects like tables, and query and update them, as shown in figure 9.2.

JDBC provides vendor-neutral access to the common database features. So if you don't use proprietary features of a database, you can easily change the database that you connect to. The JDBC API includes an abstraction of a database connection, statements, and result sets.

NOTE Though different in meaning, the terms *database* and *database engine* are often used interchangeably by developers. A database refers to the collection of data and the relationships supporting the data. The database software refers to the software like MySQL to access the data. A database engine is a process instance of the software that accesses the database. A database server is the computer on which the database engine runs.

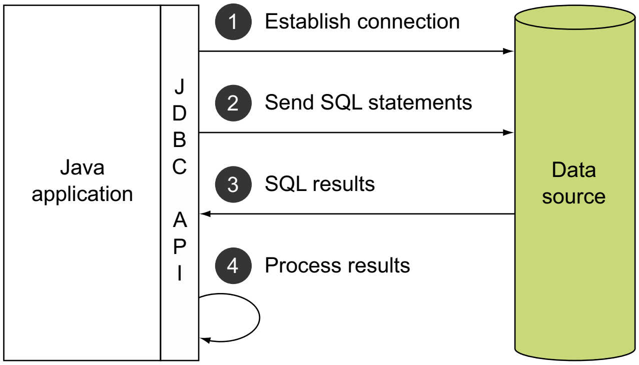


Figure 9.2 JDBC API usage overview

Let's look at the classes and interfaces that make up the core of the JDBC API.

9.1.2 JDBC architecture

With the JDBC API, you have a choice of connecting to a local or remote data store either directly or through an application server. The JDBC API supports the use of a two-tier model or *n*-tier (multiple-tier) model to connect to a database or data source. In a two-tier model, your Java application connects directly with a local or remote data source using a JDBC driver. In an *n*-tier model, a Java or non-Java application sends a command to an application server, which connects to a remote or local database using the JDBC API. Figure 9.3 shows an application connecting to a data source using two-tier and *n*-tier architecture.

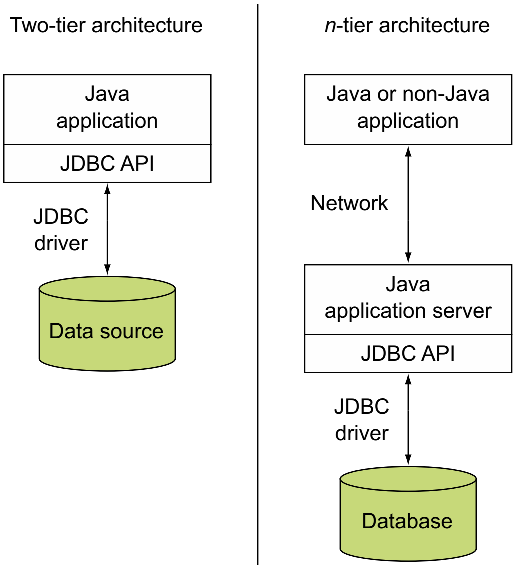


Figure 9.3 JDBC architecture

JDBC drivers provided by the database or a third party work as interpreters to enable the Java applications to "talk" with the databases. These come in multiple flavors, as discussed in the next section.

9.1.3 JDBC drivers

The JDBC API includes two major set of interfaces:

- A JDBC API for application developers
- A lower-level JDBC driver API for writing drivers

JDBC drivers are the implementation of the lower-level JDBC driver API as defined in the JDBC specifications. Depending on whether the drivers are implemented using only Java code, native code, or partial Java, they are categorized as follows:

- Type 4—pure Java JDBC driver
- Type 3—pure Java driver for database middleware
- Type 2—native API, partly Java driver
- Type 1—JDBC-ODBC bridge

Figure 9.4 shows the different types of drivers.

For the exam, you won't be questioned on the types of JDBC drivers, how they are implemented, and other driver-related specific details. The exam covers how you'd make your Java application access a given driver, using the correct connection string to establish a connection with the underlying database. Also, for the exam,

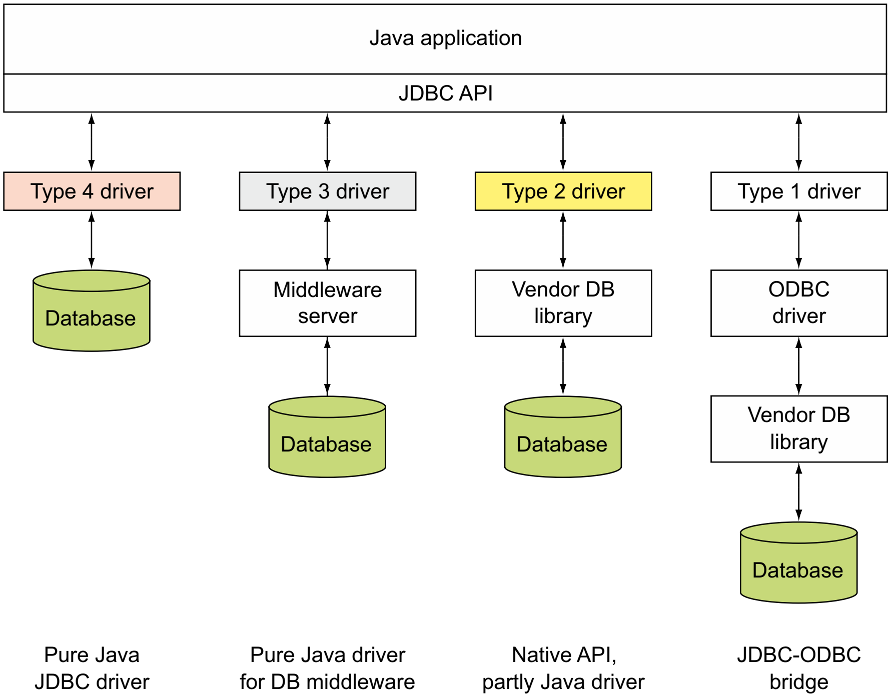


Figure 9.4 Multiple flavors of JDBC drivers

you'll only need to connect with the type 4 JDBC driver—that is, the pure Java JDBC driver.

With an overview of the JDBC API, its architecture, and the various drivers that you can use to connect a Java application to a database, let's look next at the main interfaces that make up the core of the JDBC API.

9.2 Interfaces that make up the JDBC API core

[9.1] Describe the interfaces that make up the core of the JDBC API (including the Driver, Connection, Statement, and ResultSet interfaces and their relationship to provider implementations)

The JDBC API defines multiple interfaces, which are implemented by the database vendor or a third party that your Java application is connecting with. The implementation of these interfaces is bundled in a driver and made available to the Java application. JDBC calls from your Java applications are converted to database calls by a driver. Figure 9.5 shows the main interfaces from packages `java.sql` and `javax.sql` that you'll work with in this chapter (and that are covered on the exam).

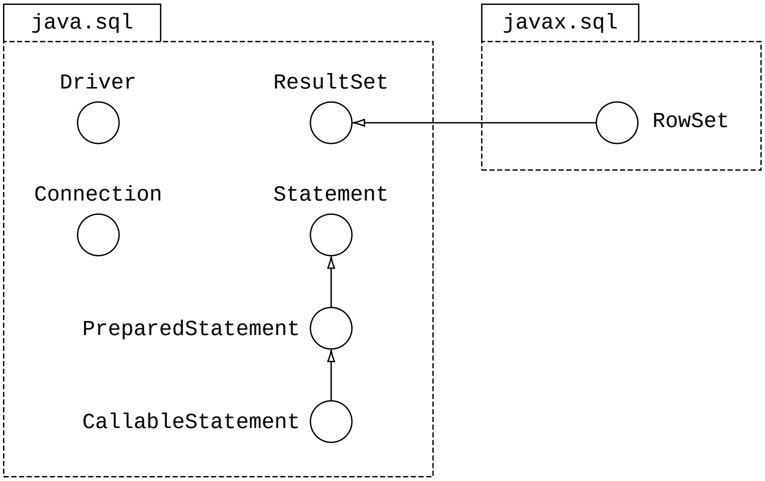


Figure 9.5 Interfaces that make up the core of the JDBC API

Every driver *must* implement a minimum set of interfaces defined in the JDBC API to conform to the JDBC specifications. The interface `java.sql.Driver` is one that must be implemented by all JDBC drivers, as discussed in the next section.

9.2.1 Interface java.sql.Driver

Every driver class must implement the interface `java.sql.Driver`. A Java system allows for the existence and registration of multiple drivers. When a database vendor or third party develops a JDBC driver, it must implement this interface. When a class implementing this interface is loaded into memory, it must create an instance of itself and register itself with class `DriverManager`. It does this by defining a static initializer block, which instantiates it and registers itself with class `DriverManager`. When a Java application requests to establish a connection with a database, class `DriverManager` searches for an appropriate driver from its list of registered drivers to connect with the target database.

You can manually load and register a driver by calling method `forName()` from class `java.lang.Class`. For example, let's assume that the name of the class that implements the interface `java.sql.Driver` in MySQL's JDBC driver is `com.mysql.jdbc.Driver`. To load and register this driver, you should execute the following:

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

Manual loading of the drivers was required until JDBC version 3.0. Starting with JDBC 4.0 and later, Java applications can automatically load and register the drivers using the Service Provider Mechanism (SPM), introduced with Java 6 and JDBC 4.0. For SPM, the driver needs to include the configuration file META-INF/services/

java.sql.Driver in the .jar file. This file contains the name of the JDBC driver's implementation of `java.sql.Driver`. An example for MySQL's JDBC driver is `com.mysql.jdbc.Driver`. When `DriverManager` requests a database connection, it loads driver classes.

9.2.2 Interface java.sql.Connection

The interface `java.sql.Connection` represents a connection session with the specified database. It's used to create the SQL statements `Statement`, `PreparedStatement`, and `CallableStatement`, which can be executed against database objects. A `Connection` object can be used to initiate transactions by creating savepoints, and committing or rolling back all changes to specified savepoints.

The `Connection` object can also be used to get a database's metadata—that is, information regarding the data stored in a database. It makes accessible the SQL grammar supported by the database, its stored procedures, and other database-related information.

9.2.3 Interface java.sql.Statement

The interface `java.sql.Statement` is used to create and execute static SQL statements and retrieve their results. These SQL statements can be used to create, modify, and delete database objects like tables, or insert, retrieve, modify, and delete table data. The results from the database are returned as `ResultSet` objects or `int` values indicating the number of affected rows.

The interface `Statement` is extended by interfaces `java.sql.PreparedStatement` and `java.sql.CallableStatement`. Objects of both of these subinterfaces represent precompiled SQL statements, which execute faster than their uncompiled counterparts. They can also be used to execute SQL statements with placeholders, using ?. `CallableStatements` are used to execute database-stored procedures.

9.2.4 Interface java.sql.ResultSet

The interface `java.sql.ResultSet` is retrieved as a result of executing a SQL `SELECT` statement against a database. It represents a table of data. The `ResultSet` object can be read-only, scrollable, or updatable. By default, a `ResultSet` object is only read-only and can be traversed in only one (forward) direction. You can create a scrollable `ResultSet` (that can be traversed forward and backward) and/or an updatable `ResultSet` by passing the relevant parameters during the creation of a `Statement` object.

Let's now dive deep into the first step of connecting to a database using class `java.sql.DriverManager`.

9.3 Connecting to a database

[9.2] Identify the components required to connect to a database using class DriverManager (including the JDBC URL)

The first step to make a Java class communicate with a database is to establish a connection between them. Class `java.sql.DriverManager` talks with the JDBC driver to return a `Connection` object, which can be used by your Java classes for all further communication with the database.

EXAM TIP For the exam, it's important to note the difference between a JDBC driver (lowercase d) and a `Driver` class (uppercase D). A JDBC driver is a set of classes provided by the database vendor, or a third party, usually in a .jar or .zip file, to support the JDBC API. A `Driver` class is an implementation of the interface `java.sql.Driver` in a JDBC driver. For example, for MySQL, its platform-independent JDBC driver can be downloaded as mysql-connector-java-5.1.27.zip. The name of the class that implements `java.sql.Driver` in MySQL Connector/J (JDBC driver) is `com.mysql.jdbc.Driver`.

To connect with a database, you need class `DriverManager` only once. When a JDBC client needs to connect with a database, it connects with class `DriverManager`, which finds an appropriate driver, establishes a connection with the database, and returns the `Connection` object to the JDBC client. For further JDBC calls, the JDBC doesn't need class `DriverManager` again. It uses the `Connection` object and other implementations of the JDBC interfaces provided by the JDBC driver to send the SQL queries and get the results. These steps are shown in figure 9.6.

In the next section, let's see how JDBC drivers are loaded in memory and registered with the `DriverManager`.

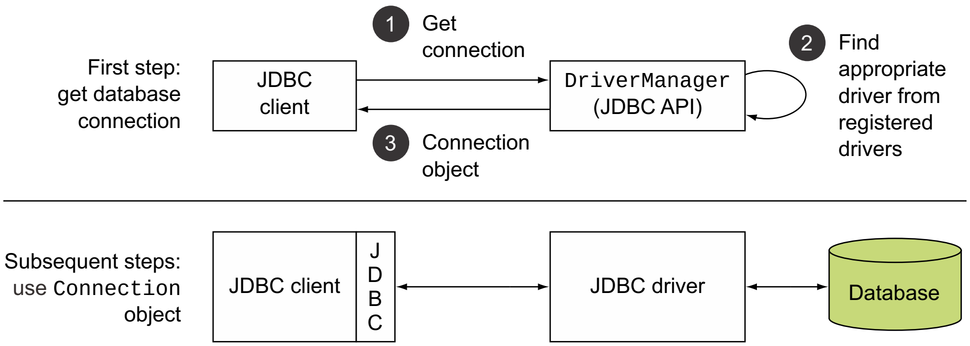

First step: get database connection
JDBC client
1 Get connection
DriverManager (JDBC API)
2 Find appropriate driver from registered drivers
3 Connection object
Subsequent steps: use Connection object
JDBC client
JDBC
JDBC driver
Database

Figure 9.6 To communicate with a data source, a JDBC client needs the `DriverManager` only once.

9.3.1 Loading JDBC drivers

For the exam, it's important to understand how JDBC drivers are loaded and registered. There are two approaches to load JDBC drivers:

- Manual (JDBC API version 3.0 and before)
- Automatic (JDBC API version 4.0 and later)

With JDBC 3.0 and its earlier versions, you need to call `Class.forName()`, passing it the name of the class that implements the interface `java.sql.Driver`. An example of loading a JDBC driver for a MySQL database is as follows:

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

EXAM TIP `Class.forName()` will throw a `ClassNotFoundException` (a checked exception) if the JVM is unable to locate the specified class.

The preceding code will load class `com.mysql.jdbc.Driver` in memory, executing its static initializer block(s). According to the JDBC specification, a driver must register itself with the `DriverManager`. For example, here's the static initializer block defined in MySQL's class `com.mysql.jdbc.Driver` that initializes and registers itself with the `DriverManager`:

```
static {
    try {
        java.sql.DriverManager.registerDriver(new Driver());
    } catch (SQLException E) {
        throw new RuntimeException("Can't register driver!");
    }
}
```

JDBC 4.0 and its later versions support automatic loading and registration of all JDBC drivers accessible via an application's class path. You no longer need to explicitly load the driver in memory using `Class.forName()`. The SPM automates the driver-loading mechanism. Using SPM, every JDBC 4.0 driver implementation must include the configuration file with the name java.sql.Driver within the `META-INF/services` folder in their `.jar` file. The file `java.sql.Driver` contains the full name of the class that the vendor used to implement the interface `jdbc.sql.Driver`. For example, `com.mysql.jdbc.Driver` is the name for the MySQL driver. When `DriverManager` requests a database connection, it loads these driver classes.

The exam might specify the JDBC API version that's being used for an application and then query you on whether you'd need to load the driver explicitly. Figure 9.7 shows the difference in steps in establishing a connection with a database using the JDBC API for versions 3.0 and before and versions 4.0 and later.

JDBC 3.0 and before	Step 1 (Load and register driver)	`Class.forName("----");`
	Step 2 (Establish DB connection)	`DriverManager.getConnection("----");`
JDBC 4.0 and later	Step 1 (Load and register driver and establish DB connection)	`DriverManager.getConnection("----");`

Figure 9.7 Difference in steps to establish a connection with a database using JDBC API version 3.0 (and before) and JDBC API version 4.0 and later

9.3.2 Use DriverManager to connect to a database

Class `DriverManager` manages all the instances of JDBC driver implementations registered with a JVM. When loaded into memory, its static initializer attempts to load all JDBC drivers that are referred to in the `jdbc.drivers` system property. `DriverManager` is a starting point to obtain database connections for Java SE.

When you invoke method `getConnection()`, the `DriverManager` finds the appropriate drivers from its set of registered drivers, establishes a connection with a database, and returns a `Connection` object. Here are the overloaded `getConnection()` methods:

```
public static Connection getConnection
            (String url) throws SQLException
public static Connection getConnection
            (String url, Properties info) throws SQLException
public static Connection getConnection
            (String url, String user, String pwd) throws SQLException
```

The JDBC URL determines the appropriate driver for a given URL string. For example

```
jdbc:subprotocol://<host>:<port>/<database_name>
```

Following is an example of a URL string for connecting with a MySQL database on a local host:

```
DriverManager.getConnection
    ("jdbc:mysql://localhost/feedback?"
              + "user=sqluser&password=sqluserpw");
```

The default port for MySQL is 3306. Usually, if the default port is being used by the database server, the `:<port>` value of the JDBC URL can be omitted. Here are a few additional examples of JDBC connection strings for MySQL:

```
jdbc:mysql://data.ejavaguru.com:3305/examDB
jdbc:mysql://localhost:3305/mysql?connectTimeout=0
jdbc:mysql://127.0.0.1:3306/examDB
```

Additional parameter connectTimeout

In a connection string, apart from specifying the type of driver, host, port, and database to connect to, you can also specify additional parameters like the default `connectTimeout`, `autoConnect`, and others. But you don't need to worry about these additional parameters for the exam. Following is an example of using the JDBC API to connect a Java class with a MySQL database:

```
import java.sql.*;
class CreateConnection {
    public static void main(String[] args) {
        Connection con = null;
        try {
            String url = "jdbc:mysql://localhost/BookLibrary";
            String username = "test";
            String password = "test";

            con = DriverManager.getConnection(url, username, password);
        }
        catch (SQLException e) {
            System.out.println(e);
        }
        System.out.println(con);
    }
}
```

NOTE To keep it simple, the code in this section doesn't include closing of the `Connection` object. I'll cover that in the next section.

Alternatively, you can also connect with a database by including the username and password as part of the JDBC connection string (changes in bold):

```
import java.sql.*;
class CreateConnection {
    public static void main(String[] args) {
        Connection con = null;
        try {
            con = DriverManager.getConnection
                        ("jdbc:mysql://localhost/BookLibrary?"
                        + "user=test&password=test");
        }
        catch (SQLException e) {
            System.out.println(e);
        }
        System.out.println(con);
    }
}
```

It's common to use `Properties` to specify the login credentials for the JDBC URL:

```
import java.sql.*;
class CreateConnection {
    public static void main(String[] args) {
        Connection con = null;
        try {
            java.util.Properties prop = new java.util.Properties();
            prop.put("user", "test");
            prop.put("password", "test");
            con = DriverManager.getConnection
                        ("jdbc:mysql://localhost/BookLibrary", prop);
        }
        catch (SQLException e) {
            System.out.println(e);
        }
        System.out.println(con);
    }
}
```

EXAM TIP Make note of the property names for specifying the username and password: the keys are "user" and "password". An attempt to use any other key to specify the username and password to connect to a database will throw a SQLException.

Connecting to a data source

There are two ways to connect to a database: by using class java.sql.DriverManager or the interface javax.sql.DataSource. Class DriverManager was included in the JDBC API since its beginning and the interface DataSource was added later in JDBC version 2.0. Class DriverManager is the preferred class to establish database connections with Java SE applications because DataSource works with the Java Naming and Directory Interface (JNDI). JNDI is usually supported by Java applications with a container that supports JNDI like Java Enterprise Edition Server. For the exam, you must only know how to establish a connection using a DriverManager. The use of DataSource isn't on the exam.

9.3.3 Exceptions thrown by database connections

Let's see what happens if your application can't load a JDBC driver. For example, I didn't add the JDBC driver to the application's class-path environment variable on purpose. Here's the exception message that I got when I tried to run this code:

```
java.sql.SQLException: No suitable driver found for jdbc:mysql://localhost/
     BookLibrary?user=test&password=test
```

If the class is unable to connect to a database due to invalid login credentials, you'll get a SQLException:

```
java.sql.SQLException: Access denied for user 'test'@'test' (using password:
     YES)
```

The individual SQL exception message will differ across various databases and their versions. After you connect to a database successfully, the next step is to execute SQL statements that can create database objects, and query, update, and delete them.

9.4 CRUD (create, retrieve, update, and delete) operations

[9.3] Submit queries and read results from the database (including creating statements, returning result sets, iterating through the results, and properly closing result sets, statements, and connections)

Since its release as JDBC API version 1.0 with JDK 1.1 in 1997, to JDBC 4.1 with JDK 7, one of the goals of the JDBC API has been to maintain its focus on SQL. The focus of this API has always been to access relational data from the Java classes. This is the core on which this JDBC API is built. For the purpose of the exam, you should be able to identify the SQL statements related to

- How to create a table, including reading a definition of a given table
- How to insert rows in a table
- How to retrieve rows in a table, specifying the columns to be selected and search criteria
- How to update rows in a table
- How to delete rows in a table

Figure 9.8 shows an outline of basic steps for executing a SQL statement against a database. A `ResultSet` is returned for SQL `SELECT` statements. SQL `INSERT`, `UPDATE`, `DELETE`, and other statements will return `int` values specifying successful execution or the number of rows affected.

Let's get started with the execution of a SQL statement to create a table.

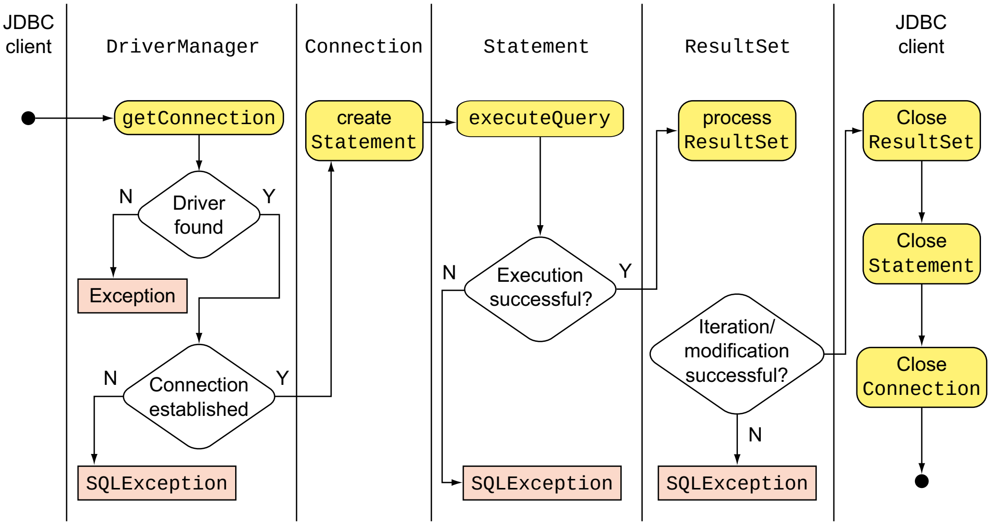


Figure 9.8 Basic steps for executing a SQL SELECT statement against a database

9.4.1 Read table definition and create table

To create a database table, you need a Connection object; use the Connection object to create a Statement object and call method executeUpdate() on the Statement object, passing it the SQL statement. For example

```
import java.sql.*;
class CreateTable {
    public static void main(String[] args) {
        Connection con = null;
        Statement statement = null;
        try {
            String url = "jdbc:mysql://localhost/BookLibrary";
            String username = "test";
            String password = "test";

            con = DriverManager.getConnection(url, username, password);

            statement = con.createStatement();

            int result = statement.executeUpdate("CREATE TABLE book " +
                                   " (id INT PRIMARY KEY, " +
                                   " title VARCHAR(1000), " +
                                   " author CHAR(255), " +
                                   " publication_year INT, " +
                                   " unit_price REAL)");
            System.out.println(result);
        }
        catch (SQLException e) {
            System.out.println(e);
        }
        finally {
            try {
                if (statement != null) statement.close();
                if (con != null) con.close();
            }
            catch (SQLException e) {}
        }
        System.out.println(con);
    }
}
```

Establish a connection with database

Create Statement

Call executeUpdate on Statement to create new table

createConnection, createStatement, and executeUpdate can throw SQLException

Connection and Statement objects must be closed.

NOTE In the book, you'll see the use of uppercase for keywords in all SQL statements. Though the case of SQL is ignored by the underlying database, the use of uppercase for keywords is recommended and practiced for better readability.

Note that you need to call method executeUpdate() on Statement and not executeQuery() to execute a Data Definition Language (DDL) request. What happens if you try to execute the preceding SQL statement using method executeQuery()?

```
statement.executeQuery("CREATE TABLE book " +
                       " (id INT PRIMARY KEY, " +
                       " title VARCHAR(1000), " +
                       " author CHAR(255), " +
                       " publication_year INT, " +
                       " unit_price REAL)");
```

The use of `executeQuery()` will throw a `SQLException` at runtime (the exact exception message might vary across systems):

```
java.sql.SQLException: Can not issue data manipulation statements with
     executeQuery().
```

EXAM TIP Method `executeUpdate()` is used to execute SQL queries to insert new rows in a table, and update and delete existing rows. It's also used to execute DDL queries, such as the creation, modification, and deletion of database objects like tables. If you use method `executeQuery()` for any of these operations, you'll get a `SQLException` at runtime.

The SQL statements that you include in your code are defined as string values. This essentially means that even if any of the SQL statements are invalid, the code compiles successfully. The Java application passes the SQL statements to the database, which determines whether the SQL is formed correctly or not. When a database receives an incorrect SQL statement, it notifies the Java application with an appropriate exception and message at runtime.

9.4.2 Mapping SQL data types to Java data types

For the exam, you might be given a table definition and shown related code to insert data into the table, update its values, or delete the values. To do so, you must know how to read a table definition and how SQL types map to Java types. For example, note the following definition of table `Book`:

```
Table Book
id, INTEGER: PK,
title, VARCHAR(100)
author, CHAR(255),
publication_year, INTEGER
unit_price, REAL
```

What data type can you use to insert values into column `title`, which has a SQL type of `VARCHAR`? You can use table 9.1 to answer this question, which shows the different types of data that can be used in a table and their equivalent Java data types.

EXAM TIP The term `PK` in table 9.1 refers to the table's *primary key*. A primary key is a column, which doesn't accept duplicate values. It's used to uniquely identify a row in a table. In the preceding example, each book can be assigned and identified using a unique `id` (identification) number. On the exam, you might see this term in a table definition, but you won't be asked questions about it.

Table 9.1 Data types in a database and their equivalent Java data types

SQL data type	Java data type	Description
CHAR	java.lang.String	Character data of fixed length
VARCHAR	java.lang.String	Character data of variable length
VARCHAR2	java.lang.String	Character data of variable length
INT	int	Integer values
INTEGER	int	Integer values
INTEGER(4)	int	Integer values
REAL/DOUBLE	double	Decimal numbers
REAL(4,2)/DOUBLE(4,2)	double	Decimal numbers
DATE	java.sql.Date	Date

You don't need to memorize these SQL data types and their corresponding Java data types. But remember the simple data conversion rules. For example, if you try to insert a `Date` object into a column with the SQL data type `INT`, you might get a `SQLException`.

In the next section, let's work with an example to read a table definition and insert data into it.

9.4.3 Insert rows in a table

On the exam, you might be given the definition of a table and questioned on the code to insert rows in it. Here's a sample of the definition of table `Book` from the preceding section (a table definition in SQL may refer to a "table" as an "item"):

```
Item Book
id, INTEGER: PK,
title, VARCHAR(100)
author, CHAR(255),
publication_year, INTEGER
unit_price, REAL
```

It's usual for the exam to include the description of the book as `Item Book`. Let's work with the code to insert rows in this table using the `try`-with-resources construct that auto-closes resources `Statement` and `Connection`:

```
class InsertIntoTable {
    public static void main(String[] args) {
        try (Connection con = getConnection();
             Statement statement = con.createStatement()){
```

Step 1: Establish a connection with database

Step 2: Create Statement object

Step 3: Call executeUpdate on statement to insert new row in table; returns number of rows affected

```
      int ret = statement.executeUpdate
        ("INSERT INTO book VALUES (" +
         "1, 'Expert In Java', 'Mantheakis', 2009, 59.9)");

      System.out.println(ret);
    }
    catch (SQLException e) {
      System.out.println(e);
    }
  }
  static Connection getConnection() throws SQLException{
    String url = "jdbc:mysql://localhost/BookLibrary";
    java.util.Properties prop = new java.util.Properties();
    prop.put("user", "test");
    prop.put("password", "test");
    return DriverManager.getConnection(url, prop);
  }
}
```

Prints "1"

EXAM TIP Because the interfaces `Connection`, `Statement`, and `ResultSet` extend the interface `AutoCloseable`, you can create their instances in a `try`-with-resources statement.

In the preceding code, the SQL statement used to insert a row in table `Book` didn't specify the column names. You might also see SQL statements on the exam that specify the name of the columns:

Column names included in SQL query

```
int ret = statement.executeUpdate
  ("INSERT INTO book (id, title, author, publication_year, unit_price)"+
  " VALUES (1, 'Expert In Java', 'Mantheakis', 2009, 59.9)");
```

EXAM TIP You'll not be tested on how to write SQL statements on this exam. But you should know how to read basic SQL statements to insert, retrieve, update, and delete table data. Including and excluding column names while inserting new rows in a table are two ways to insert data in a table.

It's easy to get confused with the method name used to insert data into a table. The preceding example used method `executeUpdate()` (and not `execute()`). Method `executeUpdate()` is also used to execute SQL `UPDATE` and `DELETE` statements. Here are the overloaded versions of this method, as mentioned in the Java API documentation:

Executes given SQL statement, which may be an INSERT, UPDATE, or DELETE statement or SQL statement that returns nothing, such as a SQL DDL statement

Executes given SQL statement and signals driver with given flag whether auto-generated keys produced by Statement object should be made available for retrieval

```
int executeUpdate(String sql)
int executeUpdate(String sql, int autoGeneratedKeys)
int executeUpdate(String sql, int[] columnIndexes)
int executeUpdate(String sql, String[] columnNames)
```

Executes given SQL statement and signals driver that auto-generated keys indicated in given array should be made available for retrieval

9.4.4 Update data in a table

To update the data in a table, use method `executeUpdate()` from the interface `Statement`. You can use a single SQL query to update single or multiple rows. To update data, the database must find and mark the rows that need to be updated, specified using the `WHERE` clause of a SQL statement. Let's update the `unit_price` of book *Expert in Java* to `99.9`:

```
class UpdateTable {
    public static void main(String[] args) {
        try (Connection con = getConnection();
             Statement statement = con.createStatement()){
            int ret = statement.executeUpdate
                                ("UPDATE book " +
                                 "SET unit_price = 99.9 " +
                                 "WHERE title = 'Expert In Java'");

            System.out.println(ret);
        }
        catch (SQLException e) {
            System.out.println(e);
        }
    }
    static Connection getConnection() throws SQLException{
        String url = "jdbc:mysql://localhost/BookLibrary";
        String username = "test";
        String password = "test";
        return DriverManager.getConnection(url, username, password);
    }
}
```

Step 1: Establish a connection with database

Step 2: Create Statement object

Step 3: Call executeUpdate() on statement to update an existing row for title "Expert In Java"

Prints "1"

The following SQL query would update all the existing rows in a table:

```
statement.executeUpdate
         ("UPDATE book " +
          "SET unit_price = 99.9");
```

If no WHERE condition, UPDATE SQL statement will update all rows

You can also update multiple columns and define multiple conditions using `AND`, `OR`, and comparison operators to find and update matching rows. The following SQL query would update columns `unit_price` and `author` for all books of author Mantheakis and having a unit price greater than 39.9:

```
int ret = statement.executeUpdate
                    ("UPDATE book " +
                     "SET unit_price = 99.9, " +
                     "author = 'Harry Mantheakis' " +
                     "WHERE author = 'Mantheakis' " +
                     "AND unit_price > 39.9");
```

EXAM TIP For all SQL operations on a database, the preferred programming approach is to close the `Connection` and `Statement` objects. You must either close them explicitly by calling `close()` on them or use them with a `try`-with-resources statement, which auto-closes them. You're very likely to see a question that asks you about closing these resources and in their correct order—first `Statement` and then `Connection` object.

9.4.5 Delete data in a table

Deletion of data works in a manner that's similar to updating data in a database, as discussed in the preceding section. Use method `executeUpdate()` to execute a SQL query to delete rows from a table. The following code will delete all rows from table `Book`, if their year of publication is before 1900:

```
class DeleteTableData {
    public static void main(String[] args) {
        try (Connection con = getConnection();
             Statement statement = con.createStatement()){
            int ret = statement.executeUpdate
                            ("DELETE FROM book " +
                             "WHERE publication_year < 1900");

            System.out.println("ret="+ret);
        }
        catch (SQLException e) {
            System.out.println(e);
        }
    }
    static Connection getConnection() throws SQLException{
        String url = "jdbc:mysql://localhost/BookLibrary";
        String username = "test";
        String password = "test";
        return DriverManager.getConnection(url, username, password);
    }
}
```

Step 1: Establish a connection with database

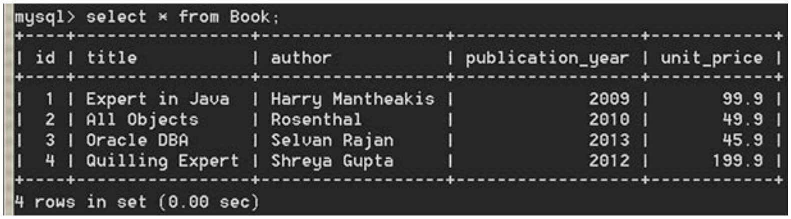
Step 2: Create Statement object

Step 3: Call executeUpdate on Statement to delete all rows with publication_year < 1900.

Prints number of rows deleted

9.4.6 Querying database

Imagine you need to check your library for the existence of all books with a unit price greater than 47.7. Figure 9.9 shows all existing rows in table `Book`.



```
mysql> select * from Book;
+----+----------------+----------------+------------------+------------+
| id | title          | author         | publication_year | unit_price |
+----+----------------+----------------+------------------+------------+
|  1 | Expert in Java | Harry Mantheakis |             2009 |       99.9 |
|  2 | All Objects    | Rosenthal      |             2010 |       49.9 |
|  3 | Oracle DBA     | Selvan Rajan   |             2013 |       45.9 |
|  4 | Quilling Expert | Shreya Gupta  |             2012 |      199.9 |
+----+----------------+----------------+------------------+------------+
4 rows in set (0.00 sec)
```

Figure 9.9 MySQL client showing results of selecting all rows from table `Book`

To select the required information, you can issue an appropriate SQL SELECT query and the database would return to you a set of information as table rows as a ResultSet object. For example

```
class QueryTableData {
    public static void main(String[] args) {
        try (Connection con = getConnection();
             Statement statement = con.createStatement()){
     ResultSet rs = statement.executeQuery
                                    ("SELECT * FROM book " +
                                     "WHERE unit_price > 47.7");){

            while (rs.next()) {
                System.out.print(rs.getInt("id") + "-");
                System.out.print(rs.getString("title") + "-");
                System.out.print(rs.getString("author") + "-");
                System.out.print(rs.getInt("publication_year") + "-");
                System.out.println(rs.getDouble("unit_price"));
            }
        }
        catch (SQLException e) {
            System.out.println(e);
        }
    }
    static Connection getConnection() throws SQLException{
        String url = "jdbc:mysql://localhost/BookLibrary";
        String username = "test";
        String password = "test";
        return DriverManager.getConnection(url, username, password);
    }
}
```

Call executeQuery() to execute SQL SELECT statement

next() moves current cursor in ResultSet object to next available row

EXAM TIP Method executeQuery() is used for SQL SELECT statements.

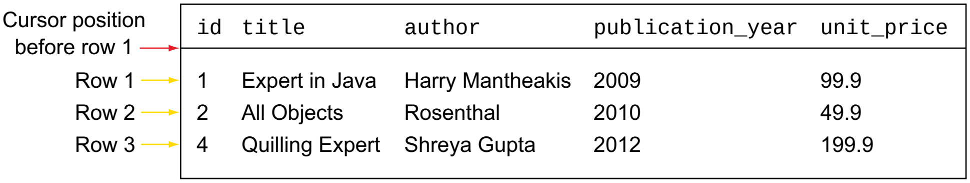
As you can see in figure 9.9, table Book has three matching rows for a unit_price greater than 47.7. The output of the preceding code is as follows:

```
1-Expert in Java-Harry Mantheakis-2009-99.9
2-All Objects-Rosenthal-2010-49.9
4-Quilling Expert-Shreya Gupta-2012-199.9
```

Let's take a closer look at the structure of the ResultSet object, returned by method executeQuery(), as shown in figure 9.10.

At the beginning, the cursor position is *before* the first row. Method next() returns a boolean value indicating whether more rows are available in the ResultSet *and* moves the cursor position to the next row, if it's available. The code iterates through all the rows of the ResultSet object and prints the column values by calling ResultSet's methods getString(), getInt(), and getDouble().

For the exam, you must understand multiple concepts when you access a ResultSet using a SQL SELECT query. The preceding example used * to specify that the resulting set must include all the columns for the selected row. You can restrict the



id	title	author	publication_year	unit_price
1	Expert in Java	Harry Mantheakis	2009	99.9
2	All Objects	Rosenthal	2010	49.9
4	Quilling Expert	Shreya Gupta	2012	199.9

Figure 9.10 When you issue a SQL SELECT statement, `executeQuery()` returns a `ResultSet`. A `ResultSet` is a link to a database cursor with selected results—an object that maintains a cursor (or a pointer) to the current row. At the beginning, the cursor is placed before the beginning of the first row.

column data that's returned to you by specifying the column names. The following query will select three columns—`id`, `title`, and `publication_year`—from table `Book` where the value for the author is `'Harry Mantheakis'`:

```
ResultSet rs = statement.executeQuery
                       ("SELECT id, title, publication_year FROM book " +
                        "WHERE author='Harry Mantheakis'");
```

Take note of method `getString()` and its method arguments (column names) used to retrieve the value for a column in a row. The overloaded method `getString()` in the interface `ResultSet` accepts either the column position or column label and returns the table value as a string. It takes the following forms:

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

EXAM TIP Although everything in Java is 0-based, column indexes in a `ResultSet` are 1-based.

The interface `ResultSet` also defines overloaded methods to retrieve the values from the `ResultSet` as specific data types, accepting column positions and column labels. Table 9.2 lists these methods.

Table 9.2 Overloaded methods in interface `ResultSet` to retrieve individual data value as a particular data type

Method return type	Method name
`int`	`getInt(String columnLabel)`
`int`	`getInt(int columnIndex)`
`long`	`getLong(String columnLabel)`
`long`	`getLong(int columnIndex)`
`float`	`getFloat(String columnLabel)`
`float`	`getFloat(int columnIndex)`

Table 9.2 Overloaded methods in interface `ResultSet` to retrieve individual data value as a particular data type *(continued)*

Method return type	Method name
`double`	`getDouble(String columnLabel)`
`double`	`getDouble(int columnIndex)`
`java.lang.String`	`getString(String columnLabel)`
`java.lang.String`	`getString(int columnIndex)`
`java.sql.Date`	`getDate(String columnLabel)`
`java.sql.Date`	`getDate(int columnIndex)`

What happens if the target table doesn't find any matching rows in the table for your `SELECT` query? In this case, the `ResultSet` object isn't set to `null`. It's initialized, but it doesn't contain any rows. For example

```
class QueryTableDataNoData {
    public static void main(String[] args) {
        try (Connection con = getConnection();
             Statement statement = con.createStatement();
             ResultSet rs = statement.executeQuery
                                 ("SELECT id, title, author FROM book " +
                                  "WHERE unit_price=155.99");){

            System.out.println(rs);

            ResultSetMetaData rsmd = rs.getMetaData();

            System.out.println(rsmd.getColumnLabel(1));
            System.out.println(rsmd.getColumnLabel(2));
            System.out.println(rsmd.getColumnLabel(3));
        }
        catch (SQLException e) {
            System.out.println(e);
        }
    }
    static Connection getConnection() throws SQLException{
        //code to establish and return connection
    }
}
```

No matching rows for unit_price 155.99.

Won't print null.

Get MetaData for this result set.

Retrieve and print column names for positions 1, 2, and 3.

EXAM TIP If your SQL statement doesn't return any rows, the `ResultSet` object doesn't point to a `null` value. In this case, you'll get a `ResultSet` object that won't include any rows.

Even though the SQL statement in the preceding code didn't return any rows, the resultant `ResultSet` doesn't refer to a `null` value. This is verified by calling methods on the `ResultSet` object `rs`. The preceding example retrieves the metadata (data about data) from the `ResultSet` by calling method `getMetaData()`.

In this section, you worked with modification of data in a database as independent operations. However, in the real world, most often, a simple function like updating the last name of an employee might include execution of multiple database statements. In the next section, we'll cover how database transactions logically group multiple database operations, which can be committed or rolled back.

9.5 JDBC transactions

[9.4] Use JDBC transactions (including disabling auto-commit mode, committing and rolling back transactions, and setting and rolling back to savepoints)

Imagine that a programmer, Selvan, needs to transfer $55 from his bank account to Paul's account. This simple transfer would require multiple steps like a withdrawal of $55 from Selvan's account, depositing the same amount to Paul's bank account, and modification of the total available balances of both these bank accounts.

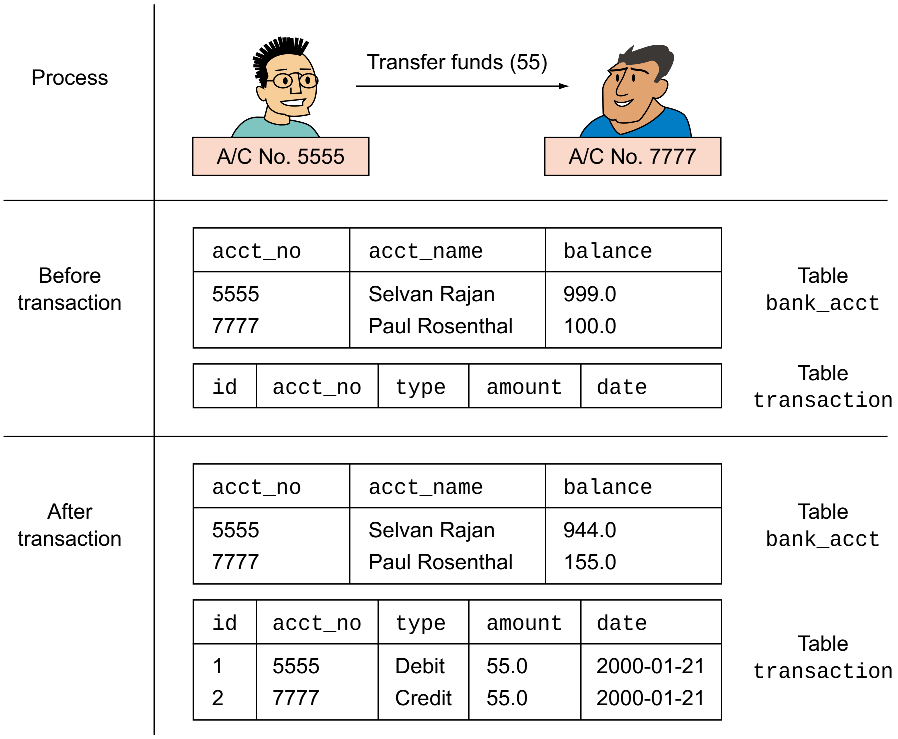
A typical fund transfer like this might involve multiple changes to the relevant database records. For such cases, you wouldn't like partial completion of steps. For example, what happens if Selvan's bank debits the $55 amount from his bank account but doesn't credit it to Paul's account? Similarly, if the bank credits the amount to Paul's account without debiting it from Selvan's account, its own data would be invalid or inconsistent. For this process to be marked *complete* and *valid*, all individual database changes should complete successfully. If either of them fails, none should be reflected in the database.

By default, the JDBC API initiates all database changes as soon as you execute SQL queries. To ensure that all or none of a set of database modifications happen, you can define them as a single JDBC transaction. The multiple database changes in a JDBC transaction execute as a single unit so that all or none of them execute. Actually, all the changes within a transaction are executed in the database, but they're persisted when they're committed. This is an important conceptual difference.

In the next section, you'll see the typical database changes that might be required to complete a simple funds transfer.

9.5.1 A transaction example

When you establish a connection with a database, by default it's in the *auto-commit* mode—that is, all changes performed by SQL statements are immediately committed to the database. Let's modify this behavior and execute a set of SQL statements to transfer funds ($55) from a bank account (ID: 5555) to another bank account (ID: 7777). Let's assume that the database defines tables `bank_acct` and `transaction` to store the details of each bank account and their related transactions, as shown in figure 9.11. To complete the process of transferring funds, the class must modify the account balances in table `bank_acct` and insert rows in table `transaction`.



Before transaction — Table bank_acct

acct_no	acct_name	balance
5555	Selvan Rajan	999.0
7777	Paul Rosenthal	100.0

Table transaction

id	acct_no	type	amount	date

After transaction — Table bank_acct

acct_no	acct_name	balance
5555	Selvan Rajan	944.0
7777	Paul Rosenthal	155.0

Table transaction

id	acct_no	type	amount	date
1	5555	Debit	55.0	2000-01-21
2	7777	Credit	55.0	2000-01-21

Figure 9.11 A simple process of a funds transfer can include multiple data modifications in the database.

Here's the complete code:

```
class TransactionTranferFunds {
    public static void main(String[] args) {
        Connection con = null;
        Statement statement = null;
        try {
            con = getConnection();
            con.setAutoCommit(false);
            statement = con.createStatement();

            int result = statement.executeUpdate
                        ("INSERT INTO transaction VALUES " +
                         " (1, '5555', 'db', 55.0, '2000-01-21')");

            result = statement.executeUpdate
                         ("INSERT INTO transaction VALUES " +
                          " (2, '7777', 'cr', 55.0, '2000-01-21')");

            result = statement.executeUpdate
                         ("UPDATE bank_account " +
                          "SET balance = 944.0 " +
                          "WHERE acct_no='5555'");
```

1 **Start transaction by calling setAutoCommit(false) on Connection object**

2 **Insert row 1 in table transaction.**

3 **Insert row 2 in table transaction.**

4 **Update balance for acct_no 5555 in table bank_account.**

```
            result = statement.executeUpdate
                          ("UPDATE bank_account " +
                           "SET balance = 155.0 " +
                           "WHERE acct_no='7777'");

            con.commit();
        }
        catch (SQLException e) {
            System.out.println(e);
            try {
                con.rollback();
            }
            catch(SQLException ex) {
                System.out.println(ex);
            }
        }
    }
    static Connection getConnection() throws SQLException{
        // code to create and return Connection object
    }
}
```

❺ **Update balance for acct_no 7777 in table bank_account.**

❻ **Commit all changes to database**

❼ **If any exception is thrown, call rollback() on Connection object**

To start a transaction that includes multiple SQL statements, the code calls `setAutoCommit(false)` on the `Connection` object ❶. The code at ❷ and ❸ inserts rows in table `transaction`. The code at ❹ and ❺ modifies the balance of account numbers 5555 and 7777 in table `bank_account`. At the end of the execution of all the SQL statements, the code at ❻ calls `commit()` on the `Connection` object. If any `SQLException` is thrown during the execution of any of the SQL statements, the exception handler calls `rollback()` on the `Connection` object ❼ so that a partial set of statements isn't persisted in the database.

As mentioned earlier, all statements are sent to the database and all requested changes are performed by the database system. Methods `commit()` and `rollback()` only decide whether these changes are persisted (that is, confirmed) in the database or not.

EXAM TIP Method `executeUpdate()` returns a count of the rows that are or would be affected in the database for row insertions, modifications, and deletion. The value is returned even if the statement isn't committed. This method returns `0` for SQL DDL statements, which create database objects and modify their structure or delete them.

Rolling back or losing all the transactions might be undesirable in most situations. In the next section, you'll see how you define *savepoints* in your transactions, so that if any exceptions occur during the course of the transaction, you might consider rolling back to particular savepoints.

9.5.2 Create savepoints and roll back partial transactions

By using a savepoint, you can exercise finer control over the work done by a set of SQL statements in a transaction. Within a transaction, you can create and set a savepoint, and later roll back the work done because you set the savepoint. You can create multiple savepoints in a transaction and release them or roll them back.

The following example defines multiple SQL statements to credit the salary of three account holders with IDs 5555, 7777, and 9999. For each account holder, a row is added to table `transaction` and then its balance is updated in table `bank_account`. After executing the SQL statements for these account holders, the bank realizes that it used an incorrect transaction amount. Notice how the following application uses savepoints in a transaction to roll back the transaction:

```
class TransactionSavepoint {
    public static void main(String[] args) {
        Connection con = null;
        Statement statement = null;
        try {
            con = getConnection();
            con.setAutoCommit(false);
            statement = con.createStatement();

            int result = statement.executeUpdate
                  ("INSERT INTO transaction values " +
                   " (101, '5555', 'db', 2099.0, '2020-10-01')");
            result = statement.executeUpdate
                         ("UPDATE bank_account " +
                          "SET balance = balance + 2099.0 " +
                          "WHERE acct_no='5555'");

            Savepoint sp5555 = con.setSavepoint();

            result = statement.executeUpdate
                   ("INSERT INTO transaction values " +
                    " (102, '7777', 'db', 12099.0, '2020-10-01')");
            result = statement.executeUpdate
                         ("UPDATE bank_account " +
                          "SET balance = balance + 12099.0 " +
                          "WHERE acct_no='7777'");

            Savepoint sp7777 = con.setSavepoint("CrSalaryFor7777");

            result = statement.executeUpdate
                   ("INSERT INTO transaction values " +
                    " (103, '9999', 'db', 5099.0, '2020-10-01')");
            result = statement.executeUpdate
                         ("UPDATE bank_account " +
                          "SET balance = balance + 5099.0 " +
                          "WHERE acct_no='9999'");
            Savepoint savepoint = con.setSavepoint("CrSalaryFor9999");

            con.rollback(sp7777);
            con.commit();
        }
        catch (SQLException e) {
            System.out.println(e);
            try {
                con.rollback();
            }
```

❶ **To start a transaction, set Connection's auto-commit mode to false.**

❷ **SQL statements to credit salary for acct_no 5555**

❸ **Set unnamed Savepoint sp5555.**

❹ **SQL statements to credit salary for acct_no 7777**

❺ **Set named Savepoint sp7777 to CrSalaryFor7777.**

❻ **SQL statements to credit salary for acct_no 9999**

❼ **Set named Savepoint sp9999 to CrSalary-For9999.**

❽ **Roll back all transactions to point sp7777.**

❾ **Commit the rest of the transactions.**

❿ **If SQLException, roll back all database changes.**

```
            catch(SQLException ex) {
                System.out.println(ex);
            }
        }
    }
    static Connection getConnection() throws SQLException{
        // code to create and return Connection object
    }
}
```

In the preceding code, ❶ sets the auto-commit mode of a `Connection` object to `false`. This is important. The default auto-commit mode is `true`, which if not changed will make the `commit` and `rollback` methods throw a `SQLException`. The code at ❷ executes a SQL statement to credit the salary of the account holder with ID 5555 by inserting a row in table `transaction` and updating the balance in table `bank_account`. The code at ❸ sets the first savepoint. The code at ❹ defines a SQL statement to credit the salary for the account holder with ID 7777, and the code at ❺ creates a named savepoint, `CrSalaryFor7777`. The code at ❻ defines a SQL statement to credit the salary for the account holder with ID 9999, and the code at ❼ creates another named savepoint, `CrSalaryFor9999`. The code at ❽ rolls back the transaction, discarding all executed statements, going backward, to the point where savepoint `CrSalaryFor7777` is created. The code at ❾ commits all the database changes that were executed before the creation of savepoint `CrSalaryFor7777`. In case the code encounters a `SQLException`, the code at ❿ rolls ba.ck the complete transaction.

9.5.3 Commit modes and JDBC transactions

A JDBC connection can have two automatic `commit` (auto-commit) modes:

- `true` (default)
- `false`

To work with a transaction, you must set the auto-commit mode to `false` or methods `rollback()` and `commit()` will throw a `SQLException`.

If the commit mode of a connection is set to `false` and you try to set it to `true` during the course of a transaction, the code will throw a `SQLException` when you call any transaction-related method like `commit()` or `rollback()`.

9.6 RowSet objects

[9.5] Construct and use RowSet objects using the RowSetProvider class and the RowSetFactory interface

Imagine that you need to show a graph in your application depicting the performance of your organization. The data for the graph needs to be retrieved from a database. This task doesn't seem to be difficult. You can connect to the database using the JDBC

API, send a SQL query to the database, get a ResultSet, extract the values from it, and display the graph accordingly. Now imagine that you need to update the graph whenever any change is made to the underlying data in the database, using any application. A RowSet can help here.

You can configure a RowSet object by setting its properties, connecting to a JDBC data source, executing a SQL statement, and getting the results.

You can register listeners with a RowSet object so that when an event occurs on a RowSet object (like any modification to its value), the registered listeners can be notified.

RowSet objects can be *connected* or *disconnected*. A connected RowSet object, like JdbcRowSet, maintains a connection with its data source throughout its life. On the other hand, a disconnected RowSet object, like CachedRowSet, establishes a connection with the data source, gets the values, and then disconnects itself. It can still update its values and later update them in the data source by reconnecting to it. Figure 9.12 shows the relationship among the interface ResultSet and the interface RowSet and its subinterfaces.

In the next section, let's work with the classes and interfaces that can be used to create RowSet objects.

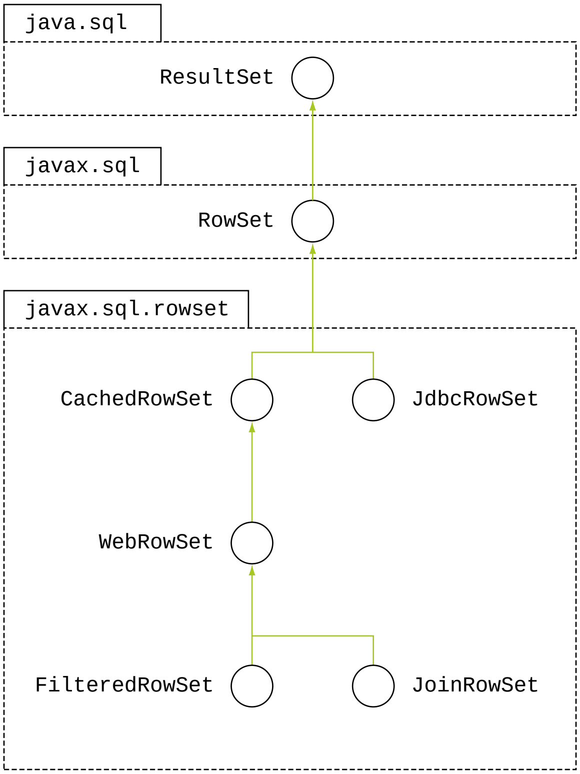


Figure 9.12 Relationship among interfaces ResultSet and RowSet and the subinterfaces of Rowset

9.6.1 *Interface RowSetFactory*

The interface `javax.sql.rowset.RowSetFactory` defines the implementation of a factory that can be used to obtain different types of `RowSet` implementations. Table 9.3 lists its methods.

Table 9.3 Methods of interface `RowSetFactory`

Method	Description
`CachedRowSet createCachedRowSet()`	Creates a new instance of a `CachedRowSet`
`FilteredRowSet createFilteredRowSet()`	Creates a new instance of a `FilteredRowSet`
`JdbcRowSet createJdbcRowSet()`	Creates a new instance of a `JdbcRowSet`
`JoinRowSet createJoinRowSet()`	Creates a new instance of a `JoinRowSet`
`WebRowSet createWebRowSet()`	Creates a new instance of a `WebRowSet`

To get access to an object of `RowSetFactory`, you can use class `RowSetProvider`, discussed in the next section.

9.6.2 *Class RowSetProvider*

Class `javax.sql.rowset.RowSetProvider` defines factory methods to get a `RowSetFactory` implementation. The `RowSetFactory` can then be used to create objects of different types of `RowSet` implementations. Here's how you can create an instance of this default implementation and a `JdbcRowSet` using it:

```
RowSetFactory rowsetFactory = RowSetProvider.newFactory();
JdbcRowSet crs = rowsetFactory.createJdbcRowSet();
```

You can also specify a custom factory implementation by specifying its name. For example

```
RowSetFactory rowsetFactory = RowSetProvider.newFactory(
                     "com.ejava.sql.rowset.CustomRowSetFactory", null);
```

This interface defines the following methods to create objects of `RowSetFactory` implementations:

Creates new instance of default RowSetFactory implementation

```
static RowSetFactory newFactory()
static RowSetFactory newFactory(String factoryClassName, ClassLoader cl)
```

Creates new instance of RowSetFactory from specified factory class name

9.6.3 *An example of working with JdbcRowSet*

Let's create and use a `JdbcRowSet` by using `RowSetFactory` and `RowSetProvider`. For this example, we'll use the default factory implementation provided by Oracle. The custom implementation is out of the scope of this exam, so it isn't discussed here. A `JdbcRowSet` object accomplishes the following:

- It establishes a connection with a database (so you don't need a separate `Connection` object).
- It creates a `PreparedStatement` object for the query to execute (so you don't need a separate `Statement` object).
- It executes the SQL statement to create a `ResultSet` object.

EXAM TIP If the `execute` method isn't successful, you can only call `execute` and `close` methods on it. The rest of the methods will throw an exception.

Here's an example:

```
class RowSetExample {
    public static void main(String[] args) {
        JdbcRowSet jdbcRS = null;
        try {

            RowSetFactory rowSetFactory = RowSetProvider.newFactory();
            jdbcRS = rowSetFactory.createJdbcRowSet();

            jdbcRS.setUrl("jdbc:mysql://localhost/BookLibrary");
            jdbcRS.setUsername("test");
            jdbcRS.setPassword("test");
            jdbcRS.setCommand("SELECT * FROM book");
            jdbcRS.execute();

            while (jdbcRS.next()) {

                System.out.print(jdbcRS.getString("id") + "-");
                System.out.print(jdbcRS.getString("title") +"-");
                System.out.print(jdbcRS.getString("unit_price"));
                System.out.println();
            }
        }
        catch (SQLException e) {
            System.out.println(e);
        }
        finally {
            try {
                jdbcRS.close();
            }
            catch (SQLException e) {}
        }
    }
}
```

1 Use RowSetProvider to get default implementation of RowSetFactory

2 Use RowSetFactory to create object of JdbcRowSet

3 Call setURL to define connection URL

4 Call setUsername and setPassword to set username and password

5 Call setCommand to define SQL statement

6 Call execute to execute SQL statement

7 Call next to determine whether more rows are available and to move cursor to next row

8 Call getString to extract column value in a row

9 Call close() on JdbcRowSet object

In the preceding code, note how the `JdbcRowSet` object manages all the database operations without any separate `Connection`, `Statement`, or `ResultSet` objects. The code at ❶ and ❷ uses the RowSetProvider and RowSetFactory to get an object of JdbcRowSet. The code at ❸ defines the database connection string using method setURL(), and at ❹ it defines the database username and password values. The code at ❺ defines the SQL statement by using RowSet's method setCommand. The code at ❻ executes the SQL statement using method execute(). It receives the result of the SQL statement and then can check for the existence of more rows by calling the next method ❼. The JdbcRowSet object accesses the columns by calling method getString() and passing the relevant column names ❽. With database objects, it's important to call close on the JdbcRowSet object ❾.

9.7 Precompiled statements

[9.6] Create and use PreparedStatement and CallableStatement objects

Unlike the objects of the interface Statement, the objects of interfaces PreparedStatement and CallableStatement represent precompiled SQL statements. Precompiled SQL statements are compiled in the database system. The precompiled statements execute faster than their noncompiled counterparts. Another major advantage offered by PreparedStatement and CallableStatement is their ability to include placeholders in SQL statements using a ?. You can assign values to these placeholders by calling one of the appropriate setDataType(parameterIndex, value) on these objects. And the most critical advantage of using parameters with PreparedStatement and CallableStatement is that they prevent SQL injection attacks. Let's get started with using the PreparedStatement.

9.7.1 Prepared statements

The interface java.sql.PreparedStatement extends the interface java.sql.Statement. Its objects represent precompiled SQL statements. The first difference that you'd notice when you compare PreparedStatement with Statement is in its creation. Unlike Statement, you must specify the relevant SQL statement when you create an object of PreparedStatement. The following code replaces Statement with a PreparedStatement object; the rest of the code remains the same:

```
class QueryPrepStatement {
    public static void main(String[] args) {
        try (Connection con = getConnection();
             PreparedStatement stmt = con.prepareStatement
                                            ("SELECT * FROM book " +
                                             "WHERE unit_price > 47.5");
             ResultSet rs = stmt.executeQuery();){
            while (rs.next()) {
                System.out.print(rs.getString("id") + "-");
                System.out.print(rs.getString("title") +"-");
                System.out.print(rs.getString("author") + "-");
                System.out.print(rs.getString("publication_year") + "-");
```

SQL query is specified when you create an object of PreparedStatement

```
                System.out.print(rs.getString("unit_price"));
                System.out.println();
            }
        }
        catch (SQLException e) {
            System.out.println(e);
        }
    }
    static Connection getConnection() throws SQLException{
        // code to get a valid connection
    }
}
```

EXAM TIP Unlike the interface Statement, where you specify the SQL query with the issue of method executeQuery() or executeUpdate(), you must specify the SQL query when you create objects of PreparedStatement.

As shown in the preceding example, you can use the PreparedStatement objects for SQL statements that don't include placeholders. The fact that you can include parameters makes PreparedStatement objects even more useful. Imagine that you need to update the unit price of all books in table book by reading the new prices (column unit_price) from table new_book_price. Here's an example:

```
class UpdateBookPricePrepStatement {
    public static void main(String[] args) throws Exception {
        try {
            Connection con = getConnection();
            PreparedStatement bookUpdStmt = con.prepareStatement
                                ("UPDATE book SET " +
                                 "unit_price = ? WHERE id = ?");
            PreparedStatement bookNewPrStmt = con.prepareStatement
                  ("SELECT id, unit_price FROM new_book_price");

            ResultSet bookNewPrRs = bookNewPrStmt.executeQuery();

            while (bookNewPrRs.next()) {
                bookUpdStmt.setDouble
                            (1, bookNewPrRs.getDouble("unit_price"));
                bookUpdStmt.setString(2, bookNewPrRs.getString("id"));
                bookUpdStmt.executeUpdate();
            }
        }
        catch (SQLException e) {
            System.out.println(e);
        }
    }
    static Connection getConnection() throws SQLException{
        // code to get a valid connection
    }
}
```

PreparedStatement for input parameters for price and id

Prepared-Statement without any parameters

Call executeQuery to execute SQL SELECT statement

For all data retrieved from table new_book_price

Read id and unit_price from new_book_price and set their values as parameter values for bookUpdStmt.

Call executeUpdate to execute SQL UPDATE statement

In the preceding code, note how values are assigned to the parameters in a PreparedStatement. The interface PreparedStatement defines methods to assign values to its parameters by using its setDataType (parameterIndex, value) methods. Because column unit_price is SQL type REAL, you use the setDouble(int parameterIndex, double value) method to assign the unit price. Because column id is SQL type CHAR, you use the setString(int parameterIndex, String value) method to assign the ID. Table 9.4 lists important methods of the interface PreparedStatement.

Table 9.4 Important methods of interface PreparedStatement

Method name	Method description
void clearParameters()	Clears the current parameter values immediately.
boolean execute()	Executes the SQL statement in this PreparedStatement object, which may be any kind of SQL statement.
ResultSet executeQuery()	Executes the SQL query in this PreparedStatement object and returns the ResultSet object generated by the query.
int executeUpdate()	Executes the SQL statement in this PreparedStatement object, which must be a SQL DML statement, such as INSERT, UPDATE, or DELETE, or a SQL statement that returns nothing, such as a DDL statement.
void setByte(int parameterIndex, byte x)	Sets the designated parameter to the given Java byte value.
void setDate(int parameterIndex, Date x)	Sets the designated parameter to the given java.sql.Date value using the default time zone of the virtual machine that's running the application.
void setDouble(int parameterIndex, double x)	Sets the designated parameter to the given Java double value.
void setFloat(int parameterIndex, float x)	Sets the designated parameter to the given Java float value.
void setInt(int parameterIndex, int x)	Sets the designated parameter to the given Java int value.
void setLong(int parameterIndex, long x)	Sets the designated parameter to the given Java long value.

EXAM TIP PreparedStatement defines three methods to execute its SQL statement: execute(), executeQuery(), and executeUpdate(). Method execute() can execute any type of SQL statement and returns a boolean value—true when a query executes successfully, false when an error occurs. Method executeQuery() executes a SQL SELECT statement and returns a ResultSet. Method executeUpdate() executes a DDL query, like CREATE TABLE, and INSERT, UPDATE, and DELETE SQL statements. It returns `0` for DDL statements and the number of rows affected for SQL `INSERT`, `UPDATE`, and `DELETE` statements.

In the next section, let's see how you can use existing database-stored procedures using `java.sql.CallableStatement`.

9.7.2 Interface CallableStatement

Compare a stored procedure with a method in Java. The same way you can define multiple statements in a method to form a logical unit, a stored procedure defines multiple SQL statements to form a logical unit. For example, a database procedure could update the price of all the books in a table by reading the new values from another table. For database operations, using stored procedures would improve your Java application's performance. On creation, the stored procedures are compiled and stored in the database. It's much more efficient to execute multiple, precompiled SQL statements together in a procedure, rather than sending individual, noncompiled SQL statements from your application.

You can execute the database-stored procedures from your Java applications by using `CallableStatement` in the JDBC API. Let's begin with creating a simple database-stored procedure using a JDBC call. The creation process is simple. You can use objects of `Statement` or `PreparedStatement` to send this logical group of SQL statements to the database. Here's an example using MySQL of the creation of a database procedure that joins tables `book` and `new_book_price` and lists `id` and `author` columns from table `book` and the `unit_price` column from table `new_book_price`. Don't worry about the fine SQL details to join the tables. What's important is to note that this procedure doesn't accept any parameters and can be created using a JDBC call:

```
class CreateDatabaseProcedure{
    public static void main(String[] args) {
        try {
            Connection con = getConnection();
            PreparedStatement statement = con.prepareStatement
                    ("CREATE PROCEDURE book_details_new_prices() "+
                     "BEGIN "+
                        "SELECT A.id, A.author, B.unit_price " +
                        "FROM book A, new_book_price B " +
                        "WHERE A.id = B.id; "+
                     "END;");
            int result = statement.executeUpdate();
            System.out.println(result);
        }
        catch (SQLException e) {
            System.out.println(e);
        }
    }
    static Connection getConnection() throws SQLException{
        // code to get a valid Connection object
    }
}
```

Create a stored procedure

> **NOTE** Most of the popular databases support stored procedures but they might have variations with the creation and use of stored procedures. Don't worry; the exam won't query you with these fine details. The exam expects you to know how to create objects of the interface `CallableStatement` and use it to call stored procedures from your Java applications.

Execution of the preceding code will create a procedure called `book_details_new_prices` in the database. It's important to note database objects are rarely created via the JDBC API, but this is shown in the preceding example code for demonstration purposes.

Let's see how you can call this procedure from your Java application:

Call prepareCall to create object of CallableStatement, passing it name of stored procedure

Because stored procedure book_details_new_prices executes SQL SELECT statement, it will return ResultSet

```
class CallProcedure{
    public static void main(String[] args) {
        try {
            Connection con = getConnection();
            CallableStatement cs = con.prepareCall
                                   ("{call book_details_new_prices()}");

            ResultSet rs = cs.executeQuery();

            while(rs.next()) {
                System.out.println(rs.getString("id") + "--" +
                                   rs.getString("author") + "--" +
                                   rs.getDouble("unit_price"));
            }
        }
        catch (SQLException e) {
            System.out.println(e);
        }
    }
    static Connection getConnection() throws SQLException{
        // code to get a valid connection
    }
}
```

Note the similarity in creating objects of `PreparedStatement` and `CallableStatement`. Unlike the `Statement` object, where you specify the SQL statement while calling methods `executeQuery()` or `executeUpdate()`, you must specify the SQL statement when you create objects of `PreparedStatement` and `CallableStatement`. Also note the SQL statement used to call a procedure:

```
call book_details_new_prices()
```

Because the database-stored procedure `book_details_new_prices` doesn't accept any parameters, it's acceptable to drop the `()` following the procedure name:

```
CallableStatement cs = con.prepareCall
          ("{call book_details_new_prices}");
```

9.7.3 Database-stored procedures with parameters

The way it's quite usual for you to define method parameters for your Java methods, it's also usual to define parameters for a database-stored procedure. Let's work with an example of a stored procedure that accepts a parameter. The following code creates the relevant database procedure:

```
class DatabaseCreateProcedure{
    public static void main(String[] args) {
        try {
            Connection con = getConnection();
            PreparedStatement statement = con.prepareStatement
                ("CREATE PROCEDURE proc_book_count (OUT count INT) "+
                 "BEGIN "+
                      "SELECT COUNT(*) INTO count FROM book; "+
                 "END;");
            int result = statement.executeUpdate();
            System.out.println(result);
        }
        catch (SQLException e) {
            System.out.println(e);
        }
    }
    static Connection getConnection() throws SQLException{
        // code to get a valid Connection
    }
}
```

Code to create a stored procedure with a parameter

Unlike the stored procedure shown in the preceding example, the procedure in the following example accepts multiple parameters: an IN (input) parameter to send values to the procedure, and an OUT (output) parameter to return values from the procedure. The database procedure proc_author_row_count accepts the author name as a CHAR value, and it selects the total book count from table book for author names that match its input parameter. It stores the book count in its OUT parameter:

```
class CreateProcedureRowCount{
    public static void main(String[] args) {
        try {
            Connection con = getConnection();
            PreparedStatement statement = con.prepareStatement
                  ("CREATE PROCEDURE proc_author_row_count " +
                       "(IN author_name CHAR(50), OUT count INT) "+
                   "BEGIN "+
                      "SELECT COUNT(*) INTO count FROM book " +
                      "WHERE author = author_name; " +
                   "END;");
            int result = statement.executeUpdate();
            System.out.println(result);
        }
        catch (SQLException e) {
            System.out.println(e);
        }
    }
```

Database-stored procedure with IN and OUT parameters

```
    static Connection getConnection() throws SQLException{
        // get a valid Connection object
    }
}
```

You can define parameters of type `IN`, `OUT`, and `INOUT` with a database-stored procedure. The parameters of type `IN` can be used to pass values to a procedure, the parameters of type `OUT` can be used to return values from a procedure, and the parameters of type `INOUT` can be used to do both—pass values to a procedure and return values from it.

Let's see how you can set the variables for method parameters while calling the database-stored procedure:

```
class CallStoredProcedureWithParameters{
    public static void main(String[] args) {
        try {
            Connection con = getConnection();
            CallableStatement cs = con.prepareCall
                            ("{call proc_author_row_count(?, ?)}");

            int rowCount = 10;
            String authorName = "Shreya";

            cs.setString(1, authorName);
            cs.registerOutParameter(2, Types.NUMERIC);
            cs.setInt(2, rowCount);

            cs.execute();

            System.out.println("rowCount = " + rowCount);
        }
        catch (SQLException e) {
            System.out.println(e);
        }
    }
    static Connection getConnection() throws SQLException{
        // code to get a valid Connection object
    }
}
```

Use ? for procedure parameters

Set first procedure parameter using parameter index position and value

Register OUT parameter

Assign second parameter

Execute CallableStatement

Print value returned in OUT paramater

9.8 Summary

This chapter introduces the JDBC API, and its overview, architecture, and JDBC drivers. The JDBC API is an industry standard for connecting Java to tabular data, typically stored in database systems and also in flat files or Excel spreadsheets. It's a mature and simple technology that has been around since Java Standard Edition version 1.1.

To connect a JDBC client to a database, the JDBC client should have access to a JDBC driver. JDBC drivers implement the interfaces defined in the JDBC API. JDBC drivers are provided by the database vendor or a third party and come in multiple flavors (like type 1 and type 2). The JDBC API is a call-level API. At its core, it works with sending SQL statements to a data source and retrieving the results. It can also modify the return values and update the underlying database.

We worked with the interfaces that make up the core of the JDBC API: `java.sql.Driver`, `java.sql.Connection`, `java.sql.Statement`, `javax.sql.RowSet`, and `java.sql.ResultSet`. Class `DriverManager` manages the registration and loading of JDBC drivers and establishes a connection with a database.

We used `Statement` objects to send SQL statements to the database to create tables, query them, and update and modify their values. `ResultSet` objects store the results returned by the database. The values in a `ResultSet` can be modified and committed to the database. Most of the database operations throw a `SQLException`; therefore almost all the methods of these core interfaces throw a `SQLException`.

We worked with the commit modes of a database connection: auto-commit and manual commit. To start a transaction, you must set the auto-commit mode of the database connection to `false`. You witnessed how to exercise finer control over a subset of the SQL statements in a transaction by creating savepoints and rolling back partial transactions, in the case where an exception is thrown.

The interface `RowSet` extends the interface `ResultSet`. It can be used as a JavaBean component and it defines multiple properties, which can be set to connect to a database, execute SQL statements, retrieve results, and update them back to the database. `RowSet` is extended by multiple interfaces, such as `JdbcRowSet`, `CachedRowSet`, and others, offering specialized functionalities.

At the end of the chapter, we worked with `PreparedStatement` and `CallableStatement`. Objects of these interfaces represent precompiled statements. `PreparedStatement` can be used to execute SQL statements with or without placeholders. `CallableStatement` can be used to execute stored database procedures with method parameters passed as parameters.

REVIEW NOTES

This section lists the main points covered in this chapter.

Introduction

- JDBC is part of the core Java API; you don't need to download it separately to use it in your Java applications.
- By using the standard classes and interfaces from the JDBC API, Java classes can connect to a database, execute SQL, and process the results.
- JDBC is a standard specification—any Java class can use JDBC to connect with a database using a JDBC driver.
- Contrary to the JDBC API, JDBC drivers are external class bundles not included in the JDK.
- JDBC has been around since 1997 and was added to JSE 1.1 as JDBC 1.0. Since its first version, multiple enhancements have been made to JDBC. Java 7 ships with JDBC 4.1.

- JDBC classes are defined in Java packages `java.sql` and `javax.sql`.
- You can access tabular data stored on relational databases, flat files, and Excel spreadsheets using the JDBC API.
- JDBC provides vendor-neutral access to the common database features. So if you don't use proprietary features of a database, you can easily change the database that you connect to. The JDBC API includes an abstraction of a database connection, statements, and result sets.
- With the JDBC API, you have a choice of connecting to a local or remote data store either directly or through an application server.
- JDBC drivers are the implementation of the lower-level JDBC driver API as defined in the JDBC specifications.
- Depending on whether the drivers are implemented using only Java code, native code, or partial Java, they're categorized as type 4, pure Java JDBC driver; type 3, pure Java driver for database middleware; type 2, native API, partial Java driver; and type 1, JDBC–ODBC bridge.

Interfaces that make up the JDBC API core

- The interfaces that make up the core of the JDBC API are `java.sql.Driver`, `java.sql.Connection`, `java.sql.Statememt`, `java.sql.ResultSet`, and `javax.sql.RowSet`.
- Every JDBC driver implementation *must* implement the interface `Driver`.
- Every driver must implement a minimum set of interfaces defined in the JDBC API to conform to the JDBC specifications.
- `Connection` represents a connection session with the specified database. It's used to create SQL statements, execute them against the database, start and commit transactions, and retrieve other details.
- The interface `Statement` is used to create and execute static SQL statements and retrieve their results.
- Interfaces `PreparedStatement` and `CallableStatement` extend the `Statement` interface. They represent precompiled statements.
- `PreparedStatement` can be used to execute static or dynamic SQL statements.
- `CallableStatement` is used to execute stored database procedures.
- A `ResultSet` is retrieved as a result of executing a SQL `SELECT` statement against a database. It represents a table of data.

Connecting to a database

- The first step to make a Java class communicate with a database is to establish a connection between them.
- Class `java.sql.DriverManager` talks with the JDBC driver to return a `Connection` object, which can be used by your Java classes for all further communication with the database.

- For the exam, it's important to note the difference between a JDBC driver (lowercase d) and a `Driver` (uppercase D). A JDBC driver is a set of classes provided by the database vendor or a third party, usually in a .jar or .zip file, to support the JDBC API. A `Driver` class is an implementation of the interface `java.sql.Driver`.
- To connect with a data source, you need class `DriverManager` only once.
- Manual loading of drivers is required for JDBC API version 3.0 and before.
- JDBC drivers should be manually loaded by calling `Class.forName()`, passing it the name of the `Driver` class.
- If `Class.forName()` can't load the JDBC driver, it throws a `ClassNotFoundException` (a checked exception).
- For JDBC API 4.0 and later, JDBC drivers can be automatically loaded and registered by class `DriverManager`.
- When a class is loaded in memory, its static initializer block executes. According to the JDBC specifications, a driver must register itself with the `DriverManager`.
- The JVM loads class `DriverManager` when you call any of its methods.
- Class `DriverManager` manages all the instances of JDBC driver implementations registered with a system.
- When you invoke method `getConnection()`, class `DriverManager` finds the appropriate drivers from its set of registered drivers, establishes a connection with a database, and returns the `Connection` object.
- There are three overloaded versions of method `getConnection()`.
- You can connect to a database by including the username and password as part of the JDBC connect URL string.
- It's common to use `Properties` to specify the login credentials for the JDBC URL in method `getConnection()`.
- The property names for specifying the username and password to establish a connection are `"user"` and `"password"`. An attempt to use any other key to specify and use username and password will throw a `SQLException`.
- If your application can't load a JDBC driver or connect to a database due to invalid login credentials, it will throw an exception.

CRUD (create, retrieve, update, and delete) operations

- To create a `Statement` object, call method `createStatement()` on a `Connection` object.
- To define and execute a static SQL statement for a `Statement` object, call `executeQuery()` or `executeUpdate()` on `Statement`.
- Method `executeQuery()` returns `ResultSet`.
- Method `executeUpdate()` returns an `int` value specifying the number of affected rows.
- Method `executeUpdate()` is used to execute SQL queries to insert new rows in a table, and update and delete existing rows. It's also used to execute DDL queries

like the creation, modification, and deletion of database objects like tables. If you use method executeQuery() for any of these operations, you'll get a SQLException at runtime.

- The SQL statements that you include in your code are defined as string values. This essentially means that if any of the SQL statement is invalid, no compilation errors are thrown.
- If a SQL SELECT returns no rows, ResultSet doesn't refer to a null value. It refers to an initialized ResultSet object with zero rows.
- Connection, Statement, and ResultSet objects should be closed by calling their method close() either implicitly or explicitly.
- If you create Connection, Statement, and ResultSet objects using a try-with-resources statement, it will auto-close them.

JDBC transactions

- A transaction is a logical set of SQL statements. Either all or none of the statements must execute from a transaction.
- To initiate a transaction, set the default database auto-commit mode to false.
- If the auto-commit mode of a connection is set to true, calling any of the transaction methods like commit() or rollback() will throw a SQLException.
- Method executeUpdate() returns a count of the rows that are or would be affected in the database for row insertions, modifications, and deletion. The value is returned even if the statement isn't committed to a database.
- By using a savepoint, you can exercise finer control over the work done by a set of SQL statements in a transaction.

RowSet objects

- You can use RowSet objects as JavaBeans components, which can be created and configured at design time.
- You can configure a RowSet object by setting its properties, connecting to a JDBC data source, executing a SQL statement, and getting the results.
- The interface javax.sql.RowSet extends the interface java.sql.ResultSet.
- You can register listeners with a RowSet object so that when an event occurs on a RowSet object (like any modification to its value), the registered listeners can be notified.
- RowSet objects can be *connected* or *disconnected*. A connected RowSet object, like JdbcRowSet, maintains a connection with its data source throughout its life. On the other hand, a disconnected RowSet object, like CachedRowSet, establishes a connection with the data source, gets the values, and then disconnects itself.
- The interface javax.sql.rowset.RowSetFactory defines the implementation of a factory that can be used to obtain different types of RowSet implementations.

- Class javax.sql.rowset.RowSetProvider defines factory methods to get a RowSetFactory implementation. The RowSetFactory can then be used to create objects of different types of RowSet implementations. The Java API defines a default implementation of RowSetFactory.
- If the execute method isn't successful on a RowSet object, you can only call execute and close methods on it. The rest of the methods will throw an exception.

Precompiled statements

- Interfaces java.sql.PreparedStatement and java.sql.CallableStatement extend the java.sql.Statement interface.
- The objects of interfaces PreparedStatement and CallableStatement represent precompiled SQL statements.
- Precompiled statements execute faster than their noncompiled counterparts.
- Another major advantage offered by PreparedStatement and CallableStatement is their ability to include placeholders in SQL statements using ?. You can assign values to these placeholders by calling one of the appropriate setDataType(parameterIndex, value) on these objects.
- Unlike Statement, you must specify the relevant SQL statement when you create an object of PreparedStatement.
- PreparedStatement defines three methods to execute its SQL statement: execute(), executeQuery(), and executeUpdate(). Method execute() can execute any type of SQL statement and returns a boolean value. Method executeQuery() executes a SQL SELECT statement and returns a ResultSet. Method executeUpdate() executes DDL statements and table INSERT, UPDATE, and DELETE SQL statements. It returns 0 for DDL statements and the number of rows affected for SQL INSERT, UPDATE, and DELETE statements.
- You can execute the database-stored procedures from your Java applications by using CallableStatement in the JDBC API.
- If a database-stored procedure doesn't accept any parameter, it's acceptable to drop the () following the procedure name in a call to execute it using CallableStatement.
- A database stored procedure can accept multiple parameters: an IN (input) parameter to send values to the procedure, and an OUT (output) parameter to return values from the procedure.
- You can define parameters of type IN, OUT, and INOUT with a database-stored procedure using CallableStatement.

SAMPLE EXAM QUESTIONS

Q 9-1. Given that method getConnection() returns a valid Connection object, the query variable defines a valid SQL statement, and class PrepStatement prints 0, select the options for the following code that can be correct individually:

```
class PrepStatement {
    public static void main(String[] args) {
        try {
            String query = ".....";                         //line1
            Connection con = getConnection();
            PreparedStatement statement =
                           con.prepareStatement(query);
            System.out.println(
                statement.executeUpdate());
        }
        catch (SQLException e) {
            System.out.println(e);
        }
    }
}
```

- **a** Line 1 defines a SQL SELECT statement that returned zero rows.
- **b** Line 1 defines a SQL UPDATE statement that affected zero rows.
- **c** Line 1 defines a SQL DELETE statement that affected zero rows.
- **d** Line 1 defines a SQL CREATE TABLE statement, which would always return zero rows.

Q 9-2. Which SQL standard is JDBC 4.1 (Java 7) consistent with?

- **a** SQL99
- **b** SQL:2003
- **c** SQL:2010
- **d** SQL7

Q 9-3. Given the following table

```
Item tree
id INT PRIMARY KEY,
average_age REAL,
name CHAR(100)
```

which statements are true about the following code if con is a valid Connection object?

```
try (Connection con = getConnection();
     Statement statement = con.createStatement();
     ResultSet rs = statement.executeQuery("SELECT * FROM tree");){
     if (rs.next()) rs.getString(2);      //line1
}
```

```
catch (SQLException e) {
    System.out.println(e);
}
```

- a If table tree has one or more rows, the code throws a runtime exception at line 1.
- b The code prints the average_age for all rows in table tree.
- c The code prints the name for all rows in table tree.
- d If there are no rows in table tree, the code won't throw an exception.
- e If table tree has five rows, the code prints the value for the average_age for the first row.
- f If table tree has five rows, the code prints the value for name for the first row.

Q 9-4. A programmer is unable to connect her Java application to a database using JDBC. Assuming that she is using Java 7 on her system and trying to connect with a remote MySQL database, what are the possible solutions that can help her?

- a Add the driver .jar file—that is, mysql-connector-java-5.1.26-bin.jar—to the Java application's class path.
- b Add the driver .jar file—that is, mysql-connector-java-5.1.26-bin.jar—to the system's class path.
- c Add the driver .jar file—that is, mysql-connector-java-5.1.26-bin.jar—to the remote system's class path hosting the MySQL database.
- d Call method Class.forName() passing it the string value mysql-connector-java-5.1.26-bin.jar.
- e Call method DriverManager.loadDriver() to load the MySQL driver in memory.
- f Create a new Driver instance by calling new and pass it to DriverManager's method connect().
- g Call DriverManager's method getConnection(), passing it the database URL, username, and password.
- h Call DriverManager's method connect(), passing it the database URL, username, and password.

Q 9-5. Which of the following code options would create objects of RowSet?

a
```
RowSetFactory aFactory = RowSetProvider.newFactory();
CachedRowSet rowset = aFactory.createCachedRowSet();
```

b
```
RowSetFactory aFactory = RowSetProvider.newFactory();
JDBCRowSet rowset = aFactory.createJDBCRowSet();
```

c
```
RowSetFactory aFactory = RowSetProvider.createFactory();
CachedRowSet rowset = aFactory.cachedRowSet();
```

d
```
RowSetFactory aFactory = RowSetProvider.newFactory();
ComparableRowSet rowset = aFactory.createComparableRowSet();
```

Q 9-6. Which of the following options will populate `ResultRet rs` with all rows from table `Contact`, assuming `getConnection()` returns a valid `Connection` object?

a
```
public static void main(String[] args) throws Exception{
    Connection con = getConnection();
    Statement statement = con.createStatement();
    ResultSet rs = statement.executeQuery("SELECT * FROM contact");
}
```

b
```
public static void main(String[] args) {
    Connection con = getConnection();
    Statement statement = con.createStatement();
    ResultSet rs = statement.executeQuery("SELECT * FROM contact");
}
```

c
```
public static void main(String[] args) throws SQLException{
    Connection con = getConnection();
    Statement statement = con.prepareStatement();
    ResultSet rs = statement.executeQuery("SELECT * FROM contact");
}
```

d
```
public static void main(String[] args) throws Throwable{
    Connection con = getConnection();
    Statement statement = con.callableStatement("SELECT * FROM contact");
    ResultSet rs = statement.executeUpdate();
}
```

Q 9-7. Select the correct option to be inserted at `//INSERT CODE HERE` that calls the database-stored procedure `hello_world`.

```
class Procedure{
    public static void main(String[] args) {
        try {
            Connection con = getConnection();
            //INSERT CODE HERE
            cs.setString(1, "eJavaGuru");
            cs.registerOutParameter(2, Types.NUMERIC);
            cs.setInt(2, 10);
            System.out.println(cs.executeUpdate());
        }
        catch (SQLException e) {}
    }
}
```

a `CallableStatement cs = con.prepareCall("{hello_world(?, ?)}");`

b `CallableStatement cs = con.prepareCall("{call procedure_hello_world (?, ?)}");`

c `CallableStatement cs = con.prepareCall("{call hello_world}");`

d `CallableStatement cs = con.prepareCall("{call hello_world(?, ?)}");`

e `CallableStatement cs = con.prepareCall("hello_world(?, ?)");`

Q 9-8. Given the following table

```
Item Book
id INT PRIMARY KEY,
title CHAR(100),
publisher CHAR(100),
unit_price REAL,
```

what is the result of the following code?

```
class Transact1 {
    public static void main(String[] args) throws SQLException{
        Connection con = null;
        try {
            con = getConnection();     //assume this get a valid connection
            con.setAutoCommit(false);
            Savepoint sv1 = con.setSavepoint();                // line 1
            Statement statement = con.createStatement();
            statement.executeUpdate("UPDATE book " +
                                    "SET unit_price = 10.0");
            Savepoint sv2 = con.setSavepoint("sv2");           // line 2
            statement.executeUpdate("UPDATE book " +
                                    "SET unit_price = 20.0");
            con.rollback();
            con.commit();                                     // line 3
        }
        catch (SQLException e) {
            con.rollback();                                   // line 4
        }
    }
}
```

- **a** The code updates the `unit_price` of all rows in table `book` to `10.0`.
- **b** The code updates the `unit_price` of all rows in table `book` to `20.0`.
- **c** There is no change in the value of `unit_price` in table `book`.
- **d** The code fails to compile on either line 1, line 2, or line 4.
- **e** If the code on line 3 throws a runtime exception other than `SQLException`, the code will update the `unit_price` of all rows in table `book` to `20.0`.

Q 9-9. Which of the following options demonstrates the correct use and closing of database resources?

a

```
public static void main(String[] args) throws Exception {
    Connection con = getConnection();
    Statement statement = con.createStatement();
    ResultSet rs = statement.executeQuery("SELECT name, email FROM
doctor");
}
```

b
```
public static void main(String[] args) throws Exception {
    try (Connection con = getConnection();
    Statement statement = con.createStatement();
    ResultSet rs = statement.executeQuery("SELECT name, email FROM
doctor");) {}
}
```

c
```
public static void main(String[] args) throws Exception {
    try {
        Connection con = getConnection();
        Statement statement = con.createStatement();
        ResultSet rs = statement.executeQuery("SELECT name, email FROM
doctor");
    }
    catch (SQLException e) {
        rs.close();
        statement.close();
        con.close();
    }
}
```

d
```
public static void main(String[] args) throws Exception {
    try {
        Connection con = getConnection();
        Statement statement = con.createStatement();
        ResultSet rs = statement.executeQuery("SELECT name, email FROM
doctor");
    }
    catch (SQLException e) {
        con.close();
        statement.close();
        rs.close();
    }
}
```

e
```
public static void main(String[] args) throws Exception {
    try {
        Connection con = getConnection();
        Statement statement = con.createStatement();
        ResultSet rs = statement.executeQuery("SELECT name, email FROM
doctor");
    }
    catch (SQLException e) {
        try {
            con.close();
            statement.close();
            rs.close();
        }
        catch (Exception e) {}
    }
}
```

ANSWERS TO SAMPLE EXAM QUESTIONS

A 9-1. b, c, d

[9.6] Create and use PreparedStatement and CallableStatement objects

Explanation: You can't use method `executeUpdate()` to execute a SQL `SELECT` query. If you do, you'll get a `SQLException` with a similar message:

```
java.sql.SQLException: Can not issue executeUpdate() for SELECTs
```

Similarly, you can't execute data deletion and modification queries with method `executeQuery()`. If you do so, you'll get a `SQLException`:

```
java.sql.SQLException: Can not issue data manipulation statements with
     executeQuery().
```

A 9-2. b

[9.1] Describe the interfaces that make up the core of the JDBC API (including the Driver, Connection, Statement, and ResultSet interfaces and their relationship to provider implementations)

Explanation: One of the goals of JDBC 4.1, which is shipped with Java 7, is to be consistent with SQL:2003. JDBC 3.0 supported SQL99 features that were widely supported by the industry. JDBC 4.1 is supporting major components of SQL:2003.

A 9-3. d, e

[9.3] Submit queries and read results from the database (including creating statements, returning result sets, iterating through the results, and properly closing result sets, statements, and connections)

Explanation: This question makes you take note of multiple points:

- What happens if the database data type doesn't match with Java's `getXXX()` methods used to retrieve the database column value? You can use method `getString()` to retrieve all types of database values. But if you use other `getXXX()` methods like `getDouble()` to retrieve a database column value, the method will throw a `SQLException` if the column value can't be implicitly converted to Java data type `double`.
- The index of the column values in a `ResultSet` start with position `1` and not `0`. `rs.getString()` will return the value of the second database column—that is, `average_age`.
- Like any `if` statement, if `rs.next()` returns `true`, the code in its block will execute once.

A 9-4. a, b, g

[9.2] Identify the components required to connect to a database using the DriverManager class (including the JDBC URL)

Option (c) is incorrect. To connect a Java application to a database using JDBC, the driver class should be added to the class path of the application or system executing the Java class. Its addition to the class path of the system on which the database is hosted doesn't matter.

Option (d) is incorrect. Starting with Java 6 (JDBC 4.0) you don't need to call `Class.forName()` to explicitly load the JDBC driver. Starting with JDBC 4.0, SPM automates the driver loading and registration process. The drivers must include a configuration file with the name `java.sql.Driver` (containing the name of the actual class file implementing the `java.sql.Driver` interface) within the META-INF folder in their .jar file. `DriverManager` class searches for drivers on the classpath and registers them automatically.

Option (e) is incorrect because method `loadDriver()` doesn't exist.

Option (f) is incorrect because you aren't supposed to create new `Driver` instances yourself.

Option (h) is incorrect because the correct method name is `getConnection()`.

A 9-5. a

[9.5] Construct and use RowSet objects using the RowSetProvider class and the RowSetFactory interface

Option (b) is incorrect because the correct class name is `JdbcRowSet`. Also, the method invoked on `aFactory` is invalid.

Option (c) is incorrect because the correct factory method name is `newFactory()` and not `createFactory()`. Also, the method invoked on `aFactory` is invalid.

Option (d) is incorrect because the Java API doesn't define the interface `ComparableRowSet`.

A 9-6. a

[9.3] Submit queries and read results from the database (including creating statements, returning result sets, iterating through the results, and properly closing result sets, statements, and connections)

Explanation: The code in option (b) won't compile. This code neither handles the checked `SQLException` thrown by the JDBC API statements, nor declares the `main` method to throw it.

Option (c) is incorrect because you must pass the SQL query when you create an object of `PreparedStatement` using `Connection`'s method `prepareStatement()`.

Option (d) is incorrect because class Connection doesn't define method callableStatement(). The correct method to create an object of CallableStatement is prepareCall() (from class Connection).

A 9-7. d

[9.6] Create and use PreparedStatement and CallableStatement objects

Explanation: To call a stored procedure using Java's CallableStatement, you must prefix the procedure name with the word call. Stored procedures that accept method parameters must be followed by parentheses. The question marks used in the code in this question are placeholders to insert variable values.

A 9-8. c

[9.4] Use JDBC transactions (including disabling auto-commit mode, committing and rolling back transactions, and setting and rolling back to savepoints)

Explanation: The code sets multiple savepoints. The first, Savepoint-sv1, isn't tagged with a name. When rollback() is called on a Connection object without the savepoint name, it's rolled back to the unnamed savepoint.

A 9-9. b

[9.3] Submit queries and read results from the database (including creating statements, returning result sets, iterating through the results, and properly closing result sets, statements, and connections)

Option (a) is incorrect because it doesn't care to call close on the database objects con, resultset, and statement.

Option (b) is correct because the try-with-resources statement declares the database resources and closes all of them in reverse order of their creation. The main method declares to throw the exception Exception so the try-with-resources statement doesn't need to be followed by a catch handler to take care of the exceptions thrown during the creation and auto-closing of the resources.

Options (c) and (d) won't compile. The reference variables are created in the try-block, so they are out of scope (and thus unknown) in the catch-block.

Option (e) is incorrect. The code doesn't compile. The reference variables are created in the try-block, so they are out of scope (and thus unknown) in the catch-block. The catch handler around the close() method invocations has exactly the same name (e) as its surrounding catch handler. Also, in the catch handler, the close() methods are invoked in the order Connection, Statement and ResultSet. On the other hand, the automatic resource closing will call close() in the order ResultSet, Statement, and Connection.