Chapter 15

JDBC

OCP EXAM OBJECTIVES COVERED IN THIS CHAPTER:

✓ **Accessing Databases using JDBC**

- Create connections, create and execute basic, prepared and callable statements, process query results and control transactions using JDBC API

JDBC stands for Java Database Connectivity. This chapter introduces you to the basics of accessing databases from Java. We cover the key interfaces for how to connect, perform queries, process the results, and work with transactions.

If you are new to JDBC, note that this chapter covers only the basics of JDBC and working with databases. What we cover is enough for the exam. To be ready to use JDBC on the job, we recommend that you read books on SQL along with Java and databases. For example, you might try *SQL for Dummies,* 9th edition, by Allen G. Taylor (Wiley, 2018) and *Practical Database Programming with Java* by Ying Bai (Wiley-IEEE Press, 2011).

For Experienced Developers

If you are an experienced developer and know JDBC well, you can skip the "Introducing Relational Databases and SQL" section. Read the rest of this chapter carefully, though. We found that the exam covers some topics that developers don't use in practice, in particular, these:

- You probably set up the URL once for a project for a specific database. Often, developers just copy and paste it from somewhere else. For the exam, you have to understand this rather than rely on looking it up.
- You are likely using a DataSource. For the exam, you have to remember or relearn how DriverManager works.

Introducing Relational Databases and SQL

Data is information. A piece of data is one fact, such as your first name. A *database* is an organized collection of data. In the real world, a file cabinet is a type of database. It has file folders, each of which contains pieces of paper. The file folders are organized in some way, often alphabetically. Each piece of paper is like a piece of data. Similarly, the folders on your computer are like a database. The folders provide organization, and each file is a piece of data.

A *relational database* is a database that is organized into *tables*, which consist of rows and columns. You can think of a table as a spreadsheet. There are two main ways to access a relational database from Java:

- *Java Database Connectivity (JDBC)*: Accesses data as rows and columns. JDBC is the API covered in this chapter.
- *Java Persistence API (JPA)*: Accesses data through Java objects using a concept called *object-relational mapping* (ORM). The idea is that you don't have to write as much code, and you get your data in Java objects. JPA is not on the exam, and therefore it is not covered in this chapter.

A relational database is accessed through Structured Query Language (*SQL*). SQL is a programming language used to interact with database records. JDBC works by sending a SQL command to the database and then processing the response.

In addition to relational databases, there is another type of database called a *NoSQL database*. These databases store their data in a format other than tables, such as key/value, document stores, and graph-based databases. NoSQL is out of scope for the exam as well.

In the following sections, we introduce a small relational database that we will be using for the examples in this chapter and present the SQL to access it. We also cover some vocabulary that you need to know.

Running the Examples in the Chapter

In most chapters of this book, you need to write code and try lots of examples. This chapter is different. It's still nice to try the examples, but you can probably get the JDBC questions correct on the exam from just reading this chapter and mastering the review questions.

While the exam is database agnostic, we had to use a database for the examples, and we chose the HyperSQL database. It is a small, in-memory database. In fact, you need only one JAR file to run it. For real projects, we like MySQL and PostgreSQL.

Instructions to download and set up the database for the examples in the chapter are in:

www.selikoff.net/ocp17

For now, you don't need to understand any of the code on the website. It is just to get you set up. In a nutshell, it connects to the database and creates two tables. By the end of this chapter, you should understand how to create a Connection and PreparedStatement in this manner.

There are plenty of tutorials for installing and getting started with any of these. It's beyond the scope of the book and the exam to set up a database,

but feel free to ask questions in the database/JDBC section of CodeRanch. You might even get an answer from the authors.

Identifying the Structure of a Relational Database

Our sample database has two tables. One has a row for each species that is in our zoo. The other has a row for each animal. These two relate to each other because an animal belongs to a species. These relationships are why this type of database is called a relational database. Figure 15.1 shows the structure of our database.

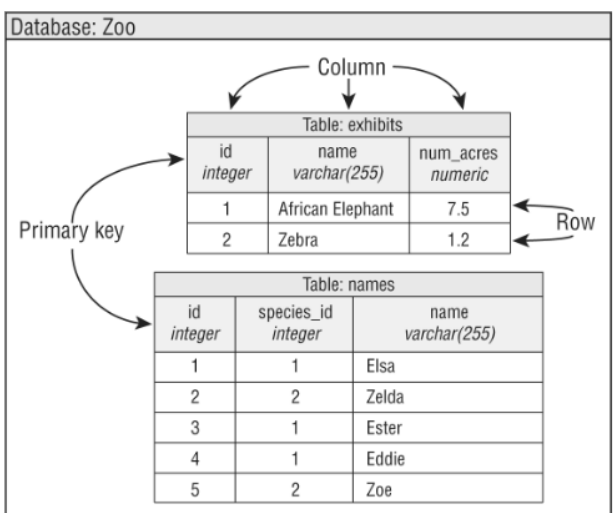


FIGURE 15.1 Tables in our relational database

As you can see in Figure 15.1, we have two tables. One is named exhibits, and the other is named names. Each table has a *primary key,* which gives us a unique way to reference each row. After all, two animals might have the same name, but they can't have the same ID. You don't need to know about keys for the exam. We mention them to give you a bit of context. In our example, it so happens that the primary key is only one column. In some situations, it is a combination of columns called a *compound key*. For example, a student identifier and year might be a compound key.

There are two rows and three columns in the exhibits table and five rows and three columns in the names table. You do need to know about rows and columns for the exam.

Writing Basic SQL Statements

The most important thing that you need to know about SQL for the exam is that there are four types of statements for working with the data in tables. They are referred to as CRUD (Create, Read, Update, Delete). The SQL keywords don't match the acronym, so pay attention to the SQL keyword for each in Table 15.1.

TABLE 15.1 CRUD operations

Operation	SQL keyword	Description
Create	INSERT	Adds new row to table
Read	SELECT	Retrieves data from table
Update	UPDATE	Changes zero or more rows in table
Delete	DELETE	Removes zero or more rows from table

That's it. You are not expected to determine whether SQL statements are correct. You are not expected to spot syntax errors in SQL statements. You are not expected to write SQL statements. Notice a theme?

Unlike Java, SQL keywords are case insensitive. This means select, SELECT, and Select are all equivalent. Like Java primitive types, SQL has a number of data types. Most are self-explanatory, like INTEGER. There's also DECIMAL, which functions a lot like a double in Java. The strangest one is VARCHAR, standing for "variable character," which is like a String in Java. The *variable* part means that the database should use only as much space as it needs to store the value.

While you don't have to know how to write them, we present the basic four SQL statements in Table 15.2 since they appear in many questions.

TABLE 15.2 SQL

SQL keyword	Explanation
INSERT INTO exhibits VALUES (3, 'Asian Elephant', 7.5);	Adds new row with provided values. Defaults to order in which columns were defined in table.
SELECT * FROM</line><line xml:id="c15-line-0004"> exhibits WHERE ID = 3;	Reads data from table with optional WHERE clause to limit data returned. In SELECT, can use * to return all columns, list specific ones to return, or even call functions like COUNT(*) to return number of matching rows.
UPDATE exhibits SET num:acres = num:acres + .5 WHERE name = 'Asian Elephant';	Sets column's value with optional WHERE clause to limit rows updated.
DELETE FROM exhibits WHERE name = 'Asian Elephant';	Deletes rows with optional WHERE clause to limit rows deleted.

Introducing the Interfaces of JDBC

For the exam, you need to know five key interfaces of JDBC. The interfaces are declared in the JDK. They are just like all of the other interfaces and classes that you've seen in this book. For example, in Chapter 9, "Collections and Generics," you worked with the interface List and the concrete class ArrayList.

With JDBC, the concrete classes come from the JDBC driver. Each database has a different JAR file with these classes. For example, PostgreSQL's JAR is called something like postgresql-9.4–1201.jdbc4.jar. MySQL's JAR is called something like mysql-connector-java-5.1.36.jar. The exact name depends on the vendor and version of the driver JAR.

This driver JAR contains an implementation of these key interfaces along with a number of other interfaces. The key is that the provided implementations know how to communicate with a database. There are also different types of drivers; luckily, you don't need to know about this for the exam.

Figure 15.2 shows the five key interfaces that you need to know. It also shows that the implementation is provided by an imaginary Foo driver JAR. They cleverly stick the name Foo in all classes.

You've probably noticed that we didn't tell you what the implementing classes are called in any real database. The main point is that you shouldn't know. With JDBC, you use only the interfaces in your code and never the implementation classes directly. In fact, they might not even be public classes.

What do these five interfaces do? On a very high level, we have the following:

- Driver: Establishes a connection to the database
- Connection: Sends commands to a database
- PreparedStatement: Executes a SQL query
- CallableStatement: Executes commands stored in the database
- ResultSet: Reads the results of a query

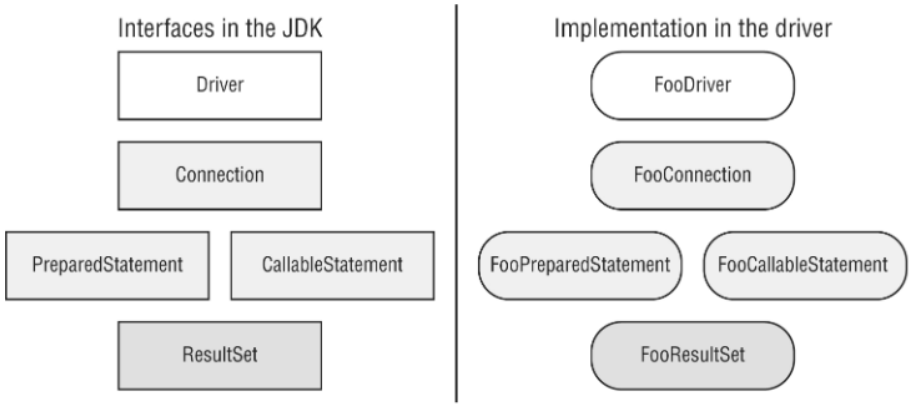


FIGURE 15.2 Key JDBC interfaces

All database interfaces are in the package java.sql, so we often omit the imports throughout this chapter.

In this next example, we show you what JDBC code looks like, end to end. If you are new to JDBC, just notice that three of the five interfaces are in the code. If you are experienced, remember that the exam uses the DriverManager class instead of the DataSource interface.

```
public class MyFirstDatabaseConnection {
  public static void main(String[] args) throws SQLException {
    String url = "jdbc:hsqldb:file:zoo";
    try (Connection conn = DriverManager.getConnection(url);
      PreparedStatement ps = conn.prepareStatement(
        "SELECT name FROM exhibits");
      ResultSet rs = ps.executeQuery()) {
      while (rs.next())
        System.out.println(rs.getString(1));
    }}}
```

If the URL were using our imaginary Foo driver, DriverManager would return an instance of FooConnection. Calling prepareStatement() would then return an instance of FooPreparedStatement, and calling executeQuery() would return an instance of FooResultSet. Since the URL uses hsqldb instead, it returns the implementations that HyperSQL has provided for these interfaces. You don't need to know their names. In the rest of the chapter, we explain how to use all five of the interfaces and go into more detail about what they do. By the end of the chapter, you'll be writing code like this yourself.

Compiling with Modules

Almost all the packages on the exam are in the java.base module. As you may recall from Chapter 12, "Modules," this module is included automatically when you run your application as a module.

In contrast, the JDBC classes are all in the module java.sql. They are also in the package java.sql. The names are the same, so they should be easy to remember. When working with SQL, you need the java.sql module and import java.sql.*.

We recommend separating your studies for JDBC and modules. You can use the classpath when working with JDBC and reserve your practice with the module path for when you are studying modules.

That said, if you do want to use JDBC code with modules, remember to update your module-info file to include the following:

```
requires java.sql;
```

Connecting to a Database

The first step in doing anything with a database is connecting to it. First we show you how to build the JDBC URL. Then we show you how to use it to get a Connection to the database.

Building a JDBC URL

To access a website, you need to know its URL. To access your email, you need to know your username and password. JDBC is no different. To access a database, you need to know this information about it.

Unlike web URLs, JDBC URLs have a variety of formats. They have three parts in common, as shown in Figure 15.3. The first piece is always the same. It is the protocol jdbc. The second part is the *subprotocol*, which is the name of the database, such as hsqldb, mysql, or postgres. The third part is the *subname*, which is a database-specific format. Colons (:) separate the three parts.

The subname typically contains information about the database such as its location and/or name. The syntax varies. You need to know about the three main parts. You don't need to memorize the subname formats. Phew! You've already seen one such URL:

```
jdbc:hsqldb:file:zoo
```

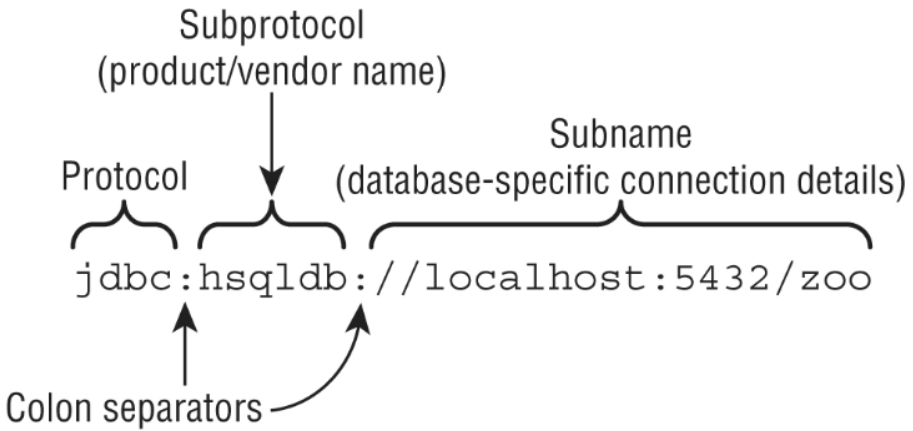


FIGURE 15.3 The JDBC URL format

Notice the three parts. It starts with jdbc, and then comes the subprotocol hsqldb. It ends with the subname, which tells us we are using the file system. The location is then the database name.

Other examples of subnames are shown here:

jdbc:postgresql://**localhost/zoo**
jdbc:oracle:thin:**@123.123.123.123:1521:zoo**
jdbc:mysql:**//localhost:3306**
jdbc:mysql:**//localhost:3306/zoo?profileSQL=true**

You can see that each of these JDBC URLs begins with jdbc, followed by a colon, followed by the vendor/product name. After that, the URLs vary. Notice how all of them include the location of the database: localhost, 123.123.123.123:1521, and localhost:3306. Also, notice that the port is optional when using the default location.

Getting a Database *Connection*

There are two main ways to get a Connection: DriverManager and DataSource. DriverManager is the one covered on the exam. Do not use a DriverManager in code someone is paying you to write. A DataSource has more features than DriverManager. For example, it can pool connections or store the database connection information outside the application.

The DriverManager class is in the JDK, as it is an API that comes with Java. It uses the factory pattern, which means that you call a static method to get a Connection rather than calling a constructor. As you saw in Chapter 11, "Exceptions and Localization," the factory pattern means that you can get any implementation of the interface when calling the method. The good news is that the method has an easy-to-remember name: getConnection().

To get a Connection from the HyperSQL database, you write the following:

```
import java.sql.*;
public class TestConnect {
```

```
public static void main(String[] args) throws SQLException {
  try (Connection conn =
    DriverManager.getConnection("jdbc:hsqldb:file:zoo")) {
    System.out.println(conn);
  }}}
```

As in Chapter 11, we use a try-with-resources statement to ensure that database resources are closed. We cover closing database resources in more detail later in the chapter. We also throw the checked SQLException, which means something went wrong. For example, you might have forgotten to set the location of the database driver in your classpath.

Assuming the program runs successfully, it prints something like this:

```
org.hsqldb.jdbc.JDBCConnection@3dfc5fb8
```

The details of the output aren't important. Just notice that the class is not Connection. It is a vendor implementation of Connection.

There is also a signature that takes a username and password.

```
import java.sql.*;
public class TestExternal {
  public static void main(String[] args) throws SQLException {
    try (Connection conn = DriverManager.getConnection(
      "jdbc:postgresql://localhost:5432/ocp-book",
      "username",
      "Password20182")) {
      System.out.println(conn);
  }}}
```

Notice the three parameters that are passed to getConnection(). The first is the JDBC URL that you learned about in the previous section. The second is the username for accessing the database, and the third is the password for accessing the database. It should go without saying that our password is not Password20182. Also, don't put your password in real code. It's a horrible practice. Always load it from some kind of configuration, ideally one that keeps the stored value encrypted.

If you were to run this with the Postgres driver JAR, it would print something like this:

```
org.postgresql.jdbc4.Jdbc4Connection@eed1f14
```

Again, notice that it is a driver-specific implementation class. You can tell from the package name. Since the package is org.postgresql.jdbc4, it is part of the PostgreSQL driver.

Unless the exam specifies a command line, you can assume that the correct JDBC driver JAR is in the classpath. The exam creators explicitly ask about the driver JAR if they want you to think about it.

The nice thing about the factory pattern is that it takes care of the logic of creating a class for you. You don't need to know the name of the class that implements Connection, and you don't need to know how it is created. You are probably a bit curious, though.

DriverManager looks through any drivers it can find to see whether they can handle the JDBC URL. If so, it creates a Connection using that Driver. If not, it gives up and throws a SQLException.

You might see Class.forName() in code. It was required with older drivers (that were designed for older versions of JDBC) before getting a Connection.

Working with a *PreparedStatement*

In Java, you have a choice of working with a Statement, PreparedStatement, or CallableStatement. The latter two are subinterfaces of Statement, as shown in Figure 15.4.

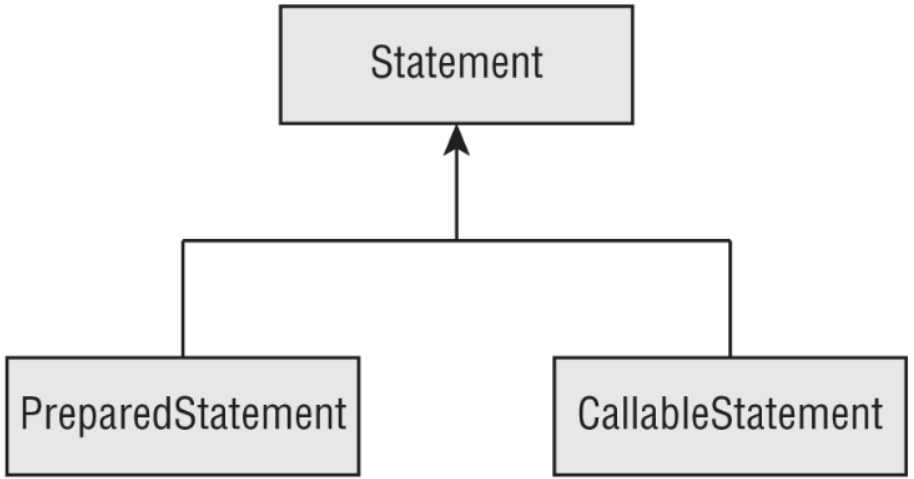


FIGURE 15.4 Types of statements

Later in the chapter, you learn about using CallableStatement for queries that are inside the database. In this section, we look at PreparedStatement.

What about Statement, you ask? It is an interface that both PreparedStatement and CallableStatement extend. A Statement and a PreparedStatement are similar to each other, except that a PreparedStatement takes parameters, while a Statement does not. A Statement just executes whatever SQL query you give it.

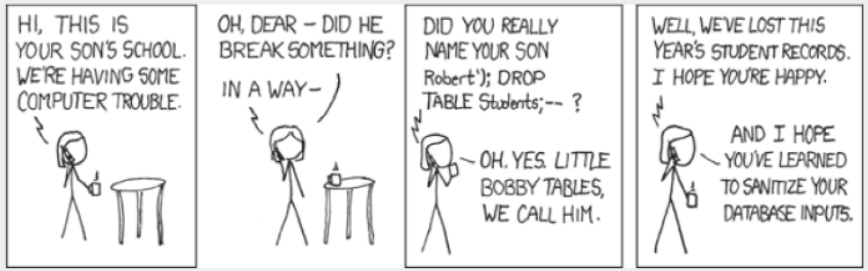
While it is possible to run SQL directly with Statement, you shouldn't. PreparedStatement is far superior for the following reasons:

- **Performance**: In most programs, you run similar queries multiple times. When you use PreparedStatement, the database software often devises a plan to run the query well and remembers it.
- **Security**: You are protected against an attack called SQL injection when using a PreparedStatement correctly.
- **Readability**: It's nice not to have to deal with string concatenation in building a query string with lots of parameters.
- **Future use:** Even if your query is being run only once or doesn't have any parameters, you should still use a PreparedStatement. That way, future editors of the code won't add a variable and have to remember to change to PreparedStatement then.

Little Bobby Tables

SQL injection is often caused by a lack of properly sanitized user input. The author of the popular xkcd.com, web-comic once asked the question, what would happen if someone's name contained a SQL statement?

"Exploits of a Mom" reproduced with permission from xkcd.com/327/



Oops! Guess the school should have used a PreparedStatement and bound each student's name to a variable. If it had, the entire String would have been properly escaped and stored in the database.

Using the Statement interface directly is not in scope for the JDBC exam, so we do not cover it in this book. In the following sections, we cover obtaining a PreparedStatement, executing one, working with parameters, and running multiple updates.

Obtaining a *PreparedStatement*

To run SQL, you need to tell a PreparedStatement about it. Getting a PreparedStatement from a Connection is easy.

```
try (PreparedStatement ps = conn.prepareStatement(
  "SELECT * FROM exhibits")) {
  // work with ps
}
```

An instance of a PreparedStatement represents a SQL statement that you want to run using the Connection. It does not execute the query yet! We get to that shortly.

Passing a SQL statement when creating the object is mandatory. You might see a trick on the exam.

```
try (var ps = conn.prepareStatement()) { // DOES NOT COMPILE
}
```

The previous example does not compile, because SQL is not supplied at the time a PreparedStatement is requested. We also used var in this example. We write JDBC code both using var and the actual class names to get you used to both approaches.

There are overloaded signatures that allow you to specify a ResultSet type and concurrency mode. On the exam, you only need to know how to use the default options, which process the results in order.

Executing a *PreparedStatement*

Now that we have a PreparedStatement, we can run the SQL statement. The method for running SQL varies depending on what kind of SQL statement it is. Remember that you aren't expected to be able to read SQL, but you do need to know what the first keyword means.

Modifying Data with *executeUpdate()*

Let's start with statements that change the data in a table. Those are SQL statements that begin with DELETE, INSERT, or UPDATE. They typically use a method called executeUpdate(). The name is a little tricky because the SQL UPDATE statement is not the only statement that uses this method.

The method takes the SQL statement to run as a parameter. It returns the number of rows that were inserted, deleted, or changed. Here's an example of all three update types:

```
10: var insertSql = "INSERT INTO exhibits VALUES(10, 'Deer', 3)";
11: var updateSql = "UPDATE exhibits SET name = '' " +
12:   "WHERE name = 'None'";
13: var deleteSql = "DELETE FROM exhibits WHERE id = 10";
14:
15: try (var ps = conn.prepareStatement(insertSql)) {
```

```
16:   int result = ps.executeUpdate();
17:   System.out.println(result); // 1
18: }
19:
20: try (var ps = conn.prepareStatement(updateSql)) {
21:   int result = ps.executeUpdate();
22:   System.out.println(result); // 0
23: }
24:
25: try (var ps = conn.prepareStatement(deleteSql)) {
26:   int result = ps.executeUpdate();
27:   System.out.println(result); // 1
28: }
```

For the exam, you don't need to read SQL. The question will tell you how many rows are affected if you need to know. Notice how each distinct SQL statement needs its own prepareStatement() call.

Line 15 creates the insert statement, and line 16 runs that statement to insert one row. The result is 1 because one row was affected. Line 20 creates the update statement, and line 21 checks the whole table for matching records to update. Since no records match, the result is 0. Line 25 creates the delete statement, and line 26 deletes the row created on line 16. Again, one row is affected, so the result is 1.

Reading Data with *executeQuery()*

Next, let's look at a SQL statement that begins with SELECT. This time, we use the executeQuery() method.

```
30: var sql = "SELECT * FROM exhibits";
31: try (var ps = conn.prepareStatement(sql);
32:   ResultSet rs = ps.executeQuery() ) {
33:
34:   // work with rs
35: }
```

On line 31, we create a PreparedStatement for our SELECT query. On line 32, we run it. Since we are running a query to get a result, the return type is ResultSet. In the next section, we show you how to process the ResultSet.

Processing Data with *execute()*

There's a third method called execute() that can run either a query or an update. It returns a boolean so that we know whether there is a ResultSet. That way, we can call the proper method to get more detail. The pattern looks like this:

```
boolean isResultSet = ps.execute();
if (isResultSet) {
  try (ResultSet rs = ps.getResultSet()) {
    System.out.println("ran a query");
  }
} else {
  int result = ps.getUpdateCount();
  System.out.println("ran an update");
}
```

If the PreparedStatement refers to sql that is a SELECT, the boolean is true, and we can get the ResultSet. If it is not a SELECT, we can get the number of rows updated.

Using the Correct Method

What do you think happens if we use the wrong method for a SQL statement? Let's take a look:

```
var sql = "SELECT * FROM names";
try (var ps = conn.prepareStatement(sql)) {

  var result = ps.executeUpdate();
}
```

This throws a SQLException similar to the following:

```
Exception in thread "main" java.sql.SQLException:
statement does not generate a row count
```

We can't get a compiler error since the SQL is a String. We can get an exception, though, and we do. We also get a SQLException when using executeQuery() with SQL that changes the database.

```
Exception in thread "main" java.sql.SQLException:
statement does not generate a result set
```

Again, we get an exception because the driver can't translate the query into the expected return type.

Reviewing *PreparedStatement* Methods

To review, make sure that you know Table 15.3 and Table 15.4 well. Table 15.3 shows which SQL statements can be run by each of the three key methods on PreparedStatement. Table 15.4 shows what is returned by each method.

TABLE 15.3 SQL runnable by the execute method

Method	DELETE	INSERT	SELECT	UPDATE
ps.execute()	Yes	Yes	Yes	Yes
ps.executeQuery()	No	No	Yes	No
ps.executeUpdate()	Yes	Yes	No	Yes

TABLE 15.4 Return types of execute methods

Method	Return type	What is returned for SELECT	What is returned for DELETE/ INSERT/UPDATE
ps.execute()	boolean	true	false
ps.executeQuery()	ResultSet	Rows and columns returned	n/a
ps.executeUpdate()	int	n/a	Number of rows added/changed/ removed

Working with Parameters

Suppose our zoo acquires a new elephant and we want to register it in our names table. We've already learned enough to do this.

```
public static void register(Connection conn) throws SQLException {
  var sql = "INSERT INTO names VALUES(6, 1, 'Edith')";

  try (var ps = conn.prepareStatement(sql)) {
    ps.executeUpdate();
  }
}
```

However, everything is hard-coded. We want to be able to pass in the values as parameters. Luckily, a PreparedStatement allows us to set parameters. Instead of specifying the three values in the SQL, we can use a question mark (?). A *bind variable* is a placeholder that lets you specify the actual values at runtime. A bind variable is like a parameter, and you will see bind variables referenced as both variables and parameters. We can rewrite our SQL statement using bind variables.

```
String sql = "INSERT INTO names VALUES(?, ?, ?)";
```

Bind variables make the SQL easier to read since you no longer need to use quotes around String values in the SQL. Now we can pass the parameters to the method itself.

```
14: public static void register(Connection conn, int key,
15:   int type, String name) throws SQLException {
```

```
16:
17:   String sql = "INSERT INTO names VALUES(?, ?, ?)";
18:
19:   try (PreparedStatement ps = conn.prepareStatement(sql)) {
20:     ps.setInt(1, key);
21:     ps.setString(3, name);
22:     ps.setInt(2, type);
23:     ps.executeUpdate();
24:   }
25: }
```

Line 19 creates a PreparedStatement using our SQL that contains three bind variables. Lines 20–22 set those variables. You can think of the bind variables as a list of parameters, where each is set in turn. Notice how the bind variables can be set in any order. Line 23 executes the query and runs the update.

Notice how the bind variables are counted starting with 1 rather than 0. This is really important, so we will repeat it.

Remember that JDBC starts counting columns with 1 rather than 0. A common exam question tests that you know this!

In the previous example, we set the parameters out of order. That's perfectly fine. The rule is only that they are each set before the query is executed. Let's see what happens if you don't set all the bind variables.

```
var sql = "INSERT INTO names VALUES(?, ?, ?)";
try (var ps = conn.prepareStatement(sql)) {
  ps.setInt(1, key);
  ps.setInt(2, type);
  // missing the set for parameter number 3
  ps.executeUpdate();
}
```

The code compiles, and you get a SQLException. The message may vary based on your database driver.

```
Exception in thread "main" java.sql.SQLException: Parameter not set
```

What about if you try to set more values than you have as bind variables?

```
var sql = "INSERT INTO names VALUES(?, ?)";
try (var ps = conn.prepareStatement(sql)) {
  ps.setInt(1, key);
```

```
  ps.setInt(2, type);
  ps.setString(3, name);
  ps.executeUpdate();
}
```

Again, you get a SQLException, this time with a different message. On HyperSQL, that message was as follows:

```
Exception in thread "main" java.sql.SQLException:
row column count mismatch in statement [INSERT INTO names
VALUES(?, ?)]
```

Table 15.5 shows the methods you need to know for the exam to set bind variables. The ones that you need to know for the exam are easy to remember since they are called set followed by the name of the type you are setting. There are many others, like dates, that are out of scope for the exam.

TABLE 15.5 PreparedStatement methods

Method	Parameter type	Example database type
setBoolean	boolean	BOOLEAN
setDouble	double	DOUBLE
setInt	int	INTEGER
setLong	long	BIGINT
setNull	int	Any type
setObject	Object	Any type
setString	String	CHAR, VARCHAR

The first column shows the method name, and the second column shows the type that Java uses. The third column shows the type name that could be in the database. There is some variation by databases, so check your specific database documentation. You need to know only the first two columns for the exam.

The setNull() method takes an int parameter representing the column type in the database. You do not need to know these types. Notice that the setObject() method works with any Java type. If you pass a primitive, it will be autoboxed into a wrapper type. That means we can rewrite our example as follows:

```
String sql = "INSERT INTO names VALUES(?, ?, ?)";
try (PreparedStatement ps = conn.prepareStatement(sql)) {
  ps.setObject(1, key);
  ps.setObject(2, type);
  ps.setObject(3, name);
```

```
  ps.executeUpdate();
}
```

Java will handle the type conversion for you. It is still better to call the more specific setter methods since that will give you a compile-time error if you pass the wrong type instead of a runtime error.

Updating Multiple Records

Suppose we get two new elephants and want to add both. We can use the same PreparedStatement object.

```
var sql = "INSERT INTO names VALUES(?, ?, ?)";

try (var ps = conn.prepareStatement(sql)) {

  ps.setInt(1, 20);
  ps.setInt(2, 1);
  ps.setString(3, "Ester");
  ps.executeUpdate();

  ps.setInt(1, 21);
  ps.setString(3, "Elias");
  ps.executeUpdate();
}
```

Note that we set all three parameters when adding Ester but only two for Elias. The PreparedStatement is smart enough to remember the parameters that were already set and retain them. You only have to set the ones that are different.

Real World Scenaria

Batching Statements

JDBC supports batching so you can run multiple statements in fewer trips to the database. Often the database is located on a different machine than the Java code runs on. Saving trips to the database saves time because network calls can be expensive. For example, if you need to insert 1,000 records into the database, inserting them as a single network call (as opposed to 1,000 network calls) is usually *a lot* faster.

You don't need to know the addBatch() and executeBatch() methods for the exam, but they are useful in practice.

```
  public static void register(Connection conn, int firstKey,
int type, String... names) throws SQLException {

var sql = "INSERT INTO names VALUES(?, ?, ?)";
var nextIndex = firstKey;

try (var ps = conn.prepareStatement(sql)) {
  ps.setInt(2, type);
  for(var name: names) {
    ps.setInt(1, nextIndex);
    ps.setString(3, name);
    ps.addBatch();

    nextIndex++;
  }
  int[] result = ps.executeBatch();
  System.out.println(Arrays.toString(result));
 }
}
```

Now we call this method with two names:

```
register(conn, 100, 1, "Elias", "Ester");
```

The output shows that the array has two elements since there are two different items in the batch. Each added one row in the database.

```
[1, 1]
```

You can use batching to break up large operations, such as inserting 10 million records in groups of 100. In practice, it takes a bit of work to

determine an appropriate batch size, but the performance of using batch is normally far better than inserting one row at a time (or all ten million at once).

Getting Data from a *ResultSet*

A database isn't useful if you can't get your data. We start by showing you how to go through a ResultSet. Then we go through the different methods to get columns by type.

Reading a *ResultSet*

When working with a ResultSet, most of the time, you will write a loop to look at each row. The code looks like this:

```
20: String sql = "SELECT id, name FROM exhibits";
21: var idToNameMap = new HashMap<Integer, String>();
22:
23: try (var ps = conn.prepareStatement(sql);
24:   ResultSet rs = ps.executeQuery()) {
25:
26:   while (rs.next()) {
27:     int id = rs.getInt("id");
28:     String name = rs.getString("name");
29:     idToNameMap.put(id, name);
30:   }
31:   System.out.println(idToNameMap);
32: }
```

It outputs this:

```
{1=African Elephant, 2=Zebra}
```

There are a few things to notice here. First, we use the executeQuery() method on line 24, since we want to have a ResultSet returned. On line 26, we loop through the results. Each time through the loop represents one row in the ResultSet. Lines 27 and 28 show you the best way to get the columns for a given row.

A ResultSet has a *cursor*, which points to the current location in the data. Figure 15.5 shows the position as we loop through.

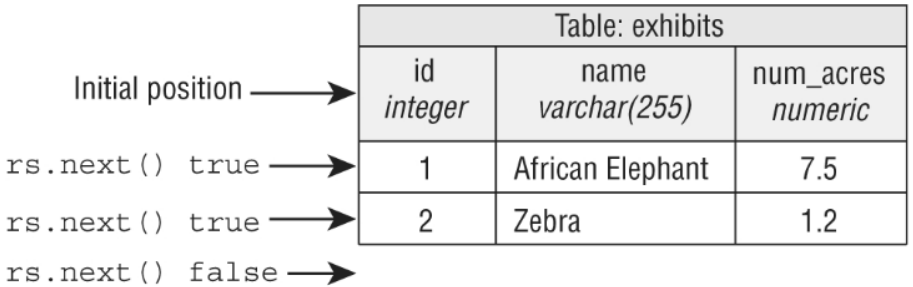


FIGURE 15.5 The ResultSet cursor

At line 24, the cursor starts by pointing to the location before the first row in the ResultSet. On the first loop iteration, rs.next() returns true, and the cursor moves to point to the first row of data. On the second loop iteration, rs.next() returns true again, and the cursor moves to point to the second row of data. The next call to rs.next() returns false. The cursor advances past the end of the data. The false signifies that there is no more data available to get.

We did say the "best way" to get data was with column names. There is another way to access the columns. You can use an index, counting from 1 instead of a column name.

```
27:    int id = rs.getInt(1);
28:    String name = rs.getString(2);
```

Now you can see the column positions. Notice how the columns are counted starting with 1 rather than 0. Just like with a PreparedStatement, JDBC starts counting at 1 in a ResultSet.

The column name is better because it is clearer what is going on when reading the code. It also allows you to change the SQL to reorder the columns.

On the exam, either you will be told the names of the columns in a table, or you can assume that they are correct. Similarly, you can assume that all SQL is correct.

Sometimes you want to get only one row from the table. Maybe you need only one piece of data. Or maybe the SQL is just returning the number of rows in the table. When you want only one row, you use an if statement rather than a while loop.

```
var sql = "SELECT count(*) FROM exhibits";

try (var ps = conn.prepareStatement(sql);
  var rs = ps.executeQuery()) {
```

```
  if (rs.next()) {
    int count = rs.getInt(1);
    System.out.println(count);
  }
}
```

It is important to check that rs.next() returns true before trying to call a getter on the ResultSet. If a query didn't return any rows, it would throw a SQLException, so the if statement checks that it is safe to call. Alternatively, you can use the column name.

```
var count = rs.getInt("count");
```

Let's try to read a column that does not exist.

```
var sql = "SELECT count(*) AS count FROM exhibits";

try (var ps = conn.prepareStatement(sql);
  var rs = ps.executeQuery()) {

  if (rs.next()) {
    var count = rs.getInt("total");
    System.out.println(count);
  }
}
```

This throws a SQLException with a message like this:

```
Exception in thread "main" java.sql.SQLException: Column not found: total
```

Attempting to access a column name or index that does not exist throws a SQLException, as does getting data from a ResultSet when it isn't pointing at a valid row. You need to be able to recognize such code. Here are a few examples to watch out for. Do you see what is wrong when no rows match?

```
var sql = "SELECT * FROM exhibits where name='Not in table'";

try (var ps = conn.prepareStatement(sql);
  var rs = ps.executeQuery()) {

  rs.next();
  rs.getInt(1); // SQLException
}
```

Calling rs.next() works. It returns false. However, calling a getter afterward throws a SQLException because the result set cursor does not point to a valid

position. If a match were returned, this code would have worked. Do you see what is wrong with the following?

```
var sql = "SELECT count(*) FROM exhibits";

try (var ps = conn.prepareStatement(sql);
  var rs = ps.executeQuery()) {

  rs.getInt(1); // SQLException
}
```

Not calling rs.next() at all is a problem. The result set cursor is still pointing to a location before the first row, so the getter has nothing to point to.

To sum up this section, it is important to remember the following:

- Always use an if statement or while loop when calling rs.next().
- Column indexes begin with 1.

Getting Data for a Column

There are lots of get methods on the ResultSet interface. Table 15.6 shows the get methods that you need to know. These are the getter equivalents of the setters in Table 15.5.

TABLE 15.6 ResultSet get methods

Method	Return type
getBoolean	boolean
getDouble	double
getInt	int
getLong	long
getObject	Object
getString	String

You might notice that not all of the primitive types are in Table 15.6. There are getByte() and getFloat() methods, but you don't need to know about them for the exam. There is no getChar() method. Luckily, you don't need to remember this. The exam will not try to trick you by using a get method name that doesn't exist for JDBC. Isn't that nice of the exam creators?

The getObject() method can return any type. For a primitive, it uses the wrapper class. Let's look at the following example:

```
16: var sql = "SELECT id, name FROM exhibits";
17: try (var ps = conn.prepareStatement(sql);
18:   var rs = ps.executeQuery()) {
```

```
19:
20:   while (rs.next()) {
21:     Object idField = rs.getObject("id");
22:     Object nameField = rs.getObject("name");
23:     if (idField instanceof Integer id)
24:       System.out.println(id);
25:     if (nameField instanceof String name)
26:       System.out.println(name);
27:   }
28: }
```

Lines 21 and 22 get the column as whatever type of Object is most appropriate. Lines 23–26 use pattern matching to get the actual types. You probably won't use getObject() when writing code for a job, but it is good to know about it for the exam.

Using Bind Variables

We've been creating the PreparedStatement and ResultSet in the same try-with-resources statement. This doesn't work if you have bind variables because they need to be set in between. Luckily, we can nest try-with-resources to handle this. This code prints out the ID for any exhibits matching a given name:

```
30: var sql = "SELECT id FROM exhibits WHERE name = ?";
31:
32: try (var ps = conn.prepareStatement(sql)) {
33:   ps.setString(1, "Zebra");
34:
35:   try (var rs = ps.executeQuery()) {
36:     while (rs.next()) {
37:       int id = rs.getInt("id");
38:       System.out.println(id);
39:     }
40:   }
41: }
```

Pay attention to the flow here. First we create the PreparedStatement on line 32. Then we set the bind variable on line 33. It is only after these are both done that we have a nested try-with-resources on line 35 to create the ResultSet.

Calling a *CallableStatement*

In some situations, it is useful to store SQL queries in the database instead of packaging them with the Java code. This is particularly useful when there are many complex queries. A *stored procedure* is code that is compiled in

advance and stored in the database. Stored procedures are commonly written in a database-specific variant of SQL, which varies among database software providers.

Using a stored procedure reduces network round trips. It also allows database experts to own that part of the code. However, stored procedures are database-specific and introduce complexity into maintaining your application. On the exam, you need to know how to call a stored procedure but not decide when to use one.

You don't need to know how to read or write a stored procedure for the exam. Therefore, we have not included any in the book. They are in the code from setting up the sample database if you are curious.

You do not need to learn anything database-specific for the exam. Since studying stored procedures can be quite complicated, we recommend limiting your studying on CallableStatement to what is in this book.

We will be using four stored procedures in this section. Table 15.7 summarizes what you need to know about them. In the real world, none of these would be good implementations since they aren't complex enough to warrant being stored procedures. As you can see in the table, stored procedures allow parameters to be for input only, output only, or both.

TABLE 15.7 Sample stored procedures

Name	Parameter name	Parameter type	Description
read_e_names()	n/a	n/a	Returns all rows in names table that have name beginning with e or E
read_names_by_letter()	prefix	IN	Returns all rows in names table that have name beginning with specified parameter (case insensitive)
magic_number()	num	OUT	Returns number 42
double_number()	num	INOUT	Multiplies parameter by two and returns that number

In the next four sections, we look at how to call each of these stored procedures.

Calling a Procedure without Parameters

Our read_e_names() stored procedure doesn't take any parameters. It does return a ResultSet. Since we worked with a ResultSet in the PreparedStatement section, here we can focus on how the stored procedure is called.

```
12: String sql = "{call read_e_names()}";
13: try (CallableStatement cs = conn.prepareCall(sql);
14:   ResultSet rs = cs.executeQuery()) {
15:
16:   while (rs.next()) {
17:     System.out.println(rs.getString(3));
18:   }
19: }
```

Line 12 introduces a new bit of syntax. A stored procedure is called by putting the word call and the procedure name in braces ({}). Line 13 creates a CallableStatement object. When we created a PreparedStatement, we used the prepareStatement() method. Here, we use the prepareCall() method instead.

Lines 14–18 should look familiar. They are the standard logic we have been using to get a ResultSet and loop through it. This stored procedure returns the underlying table, so the columns are the same.

Passing an *IN* Parameter

A stored procedure that always returns the same thing is only somewhat useful. We've created a new version of that stored procedure that is more generic. The read_names_by_letter() stored procedure takes a parameter for the prefix or first letter of the stored procedure. An IN parameter is used for input.

There are two differences in calling it compared to our previous stored procedure.

```
25: var sql = "{call read_names_by_letter(?)}";
26: try (var cs = conn.prepareCall(sql)) {
27:   cs.setString("prefix", "Z");
28:
29:   try (var rs = cs.executeQuery()) {
30:     while (rs.next()) {
31:       System.out.println(rs.getString(3));
32:     }
```

```
33:   }
34: }
```

On line 25, we have to pass a ? to show we have a parameter. This should be familiar from bind variables with a PreparedStatement.

On line 27, we set the value of that parameter. Unlike with PreparedStatement, we can use either the parameter number (starting with 1) or the parameter name. That means these two statements are equivalent:

```
cs.setString(1, "Z");
cs.setString("prefix", "Z");
```

Returning an *OUT* Parameter

In our previous examples, we returned a ResultSet. Some stored procedures return other information. Luckily, stored procedures can have OUT parameters for output. The magic_number() stored procedure sets its OUT parameter to 42. There are a few differences here:

```
40: var sql = "{?= call magic_number(?) }";
41: try (var cs = conn.prepareCall(sql)) {
42:   cs.registerOutParameter(1, Types.INTEGER);
43:   cs.execute();
44:   System.out.println(cs.getInt("num"));
45: }
```

On line 40, we include two special characters (?=) to specify that the stored procedure has an output value. This is optional since we have the OUT parameter, but it does aid in readability.

On line 42, we register the OUT parameter. This is important. It allows JDBC to retrieve the value on line 44. Remember to always call registerOutParameter() for each OUT or INOUT parameter (which we cover next).

On line 43, we call execute() instead of executeQuery() since we are not returning a ResultSet from our stored procedure.

> **Database-Specific Behavior**
>
> Some databases are lenient about certain things this chapter says are required. For example, some databases allow you to omit the following:
>
> - Braces ({})
> - Bind variable (?) if it is an OUT parameter
> - Call to registerOutParameter()

For the exam, you need to answer according to the full requirements, which are described in this book. For example, you should answer exam questions as if braces are required.

Working with an *INOUT* Parameter

Finally, it is possible to use the same parameter for both input and output. As you read this code, see whether you can spot which lines are required for the IN part and which are required for the OUT part:

```
50: var sql = "{call double_number(?)}";
51: try (var cs = conn.prepareCall(sql)) {
52:   cs.setInt(1, 8);
53:   cs.registerOutParameter(1, Types.INTEGER);
54:   cs.execute();
55:   System.out.println(cs.getInt("num"));
56: }
```

For an IN parameter, line 52 is required since it sets the value. For an OUT parameter, line 53 is required to register it. Line 54 uses execute() again because we are not returning a ResultSet.

Remember that an INOUT parameter acts as both an IN parameter and an OUT parameter, so it has all the requirements of both.

Comparing Callable Statement Parameters

Table 15.8 reviews the different types of parameters. You need to know this well for the exam.

TABLE 15.8 Stored procedure parameter types

Parameter type	IN	OUT	INOUT
Used for input	Yes	No	Yes
Used for output	No	Yes	Yes
Must set parameter value	Yes	No	Yes
Must call registerOutParameter()	No	Yes	Yes
Can include ?=	No	Yes	Yes

Using Additional Options

So far, we've been creating PreparedStatement and CallableStatement with the default options. Both support ResultSet type and concurrency options. Not all options are available on all databases. Luckily, you just have to be able to recognize them as valid on the exam.

There are three ResultSet integer type values:

- ResultSet.TYPE_FORWARD_ONLY: Can go through the ResultSet only one row at a time
- ResultSet.TYPE_SCROLL_INSENSITIVE: Can go through the ResultSet in any order but will not see changes made to the underlying database table
- ResultSet.TYPE_SCROLL_SENSITIVE: Can go through the ResultSet in any order and will see changes made to the underlying database table

There are two ResultSet integer concurrency mode values:

- ResultSet.CONCUR_READ_ONLY: The ResultSet cannot be updated.
- ResultSet.CONCUR_UPDATABLE: The ResultSet can be updated.

These options are integer values, not enum values, which means you pass both as additional parameters after the SQL.

```
conn.prepareCall(sql, ResultSet.TYPE_FORWARD_ONLY,
  ResultSet.CONCUR_READ_ONLY);

conn.prepareStatement(sql, ResultSet.TYPE_SCROLL_INSENSITIVE,
  ResultSet.CONCUR_UPDATABLE);
```

If you see these options on the exam, pay attention to how they are used. Remember that type always comes first. Also, the methods that take type also take concurrency mode, so be wary of any question that only passes one option.

Controlling Data with Transactions

Until now, any changes we made to the database took effect right away. A *commit* is like saving a file. On the exam, changes commit automatically unless otherwise specified. However, you can change this behavior to control commits yourself. A *transaction* is when one or more statements are grouped with the final results committed or rolled back. *Rollback* is like closing a file without saving. All the changes from the start of the transaction are discarded. First, we look at writing code to commit and roll back. Then we look at how to control your rollback points.

Committing and Rolling Back

Our zoo is renovating and has decided to give more space to the elephants.

However, we only have so much space, so the zebra exhibit will need to be made smaller. Since we can't invent space out of thin air, we want to ensure that the total amount of space remains the same. If either adding space for the elephants or removing space for the zebras fails, we want our transaction to roll back. In the interest of simplicity, we assume that the database table is in a valid state before we run this code. Now, let's examine the code for this scenario:

```
5:  public static void main(String[] args) throws SQLException {
6:    try (Connection conn =
7:      DriverManager.getConnection("jdbc:hsqldb:file:zoo")) {
8:
9:      conn.setAutoCommit(false);
10:
11:     var elephantRowsUpdated = updateRow(conn, 5, "African Elephant");
12:     var zebraRowsUpdated = updateRow(conn, -5, "Zebra");
13:
14:     if (! elephantRowsUpdated || ! zebraRowsUpdated)
15:       conn.rollback();
16:     else {
17:       String selectSql = """
18:         SELECT COUNT(*)
19:         FROM exhibits
20:         WHERE num:acres <= 0""";
21:       try (PreparedStatement ps = conn.prepareStatement(selectSql);
22:         ResultSet rs = ps.executeQuery()) {
23:
24:         rs.next();
25:         int count = rs.getInt(1);
26:         if (count == 0)
27:           conn.commit();
28:         else
29:           conn.rollback();
30:       } } } }
31:
32: private static boolean updateRow(Connection conn,
33:   int numToAdd, String name)
34:
35:   throws SQLException {
36:
37:   String updateSql = """
38:     UPDATE exhibits
39:     SET num:acres = num:acres + ?
40:     WHERE name = ?""";
41:
42:   try (PreparedStatement ps = conn.prepareStatement(updateSql)) {
```

```
43:     ps.setInt(1, numToAdd);
44:     ps.setString(2, name);
45:     return ps.executeUpdate()> 0;
46:   }}
```

The first interesting thing in this example is on line 9, where we turn off autocommit mode and declare that we will handle transactions ourselves. Most databases support disabling autocommit mode. If a database does not, it will throw a SQLException on line 9. We then attempt to update the number of acres allocated to each animal. If we are unsuccessful and no rows are updated, we roll back the transaction on line 15, causing the state of the database to remain unchanged.

Assuming at least one row is updated, we check exhibits and make sure none of the rows contain an invalid num:acres value. If this were a real application, we would have more logic to make sure the amount of space makes sense. On lines 26–30, we decide whether to commit the transaction to the database or roll back all updates made to the exhibits table.

Autocommit Edge Cases

You need to know two edge cases for the exam. First, calling setAutoCommit(true) will automatically trigger a commit when you are not already in autocommit mode. After that, autocommit mode takes effect, and each statement is automatically committed.

The other edge case is what happens if you have autocommit set to false and close your connection without rolling back or committing your changes. The answer is that the behavior is undefined. It may commit or roll back, depending solely on the driver. Don't depend on this behavior; remember to commit or roll back at the end of a transaction!

Bookmarking with Savepoints

So far, we have rolled back to the point where autocommit was turned off. You can use savepoints to have more control of the rollback point. Consider the following example:

```
20: conn.setAutoCommit(false);
21: Savepoint sp1 = conn.setSavepoint();
22: // database code
23: Savepoint sp2 = conn.setSavepoint("second savepoint");
24: // database code
25: conn.rollback(sp2);
26: // database code
27: conn.rollback(sp1);
```

Line 20 is important. You can only use savepoints when you are controlling the transaction. Lines 21 and 23 show how to create a Savepoint. The name is optional and typically included in the toString() if you print the savepoint reference.

Line 25 shows the first rollback. That gets rid of any changes made since that savepoint was created: in this case, the code on line 24. Then line 27 shows the second rollback getting rid of the code on line 22.

Order matters. If we reversed lines 25 and 27, the code would throw an exception. Rolling back to sp1 gets rid of any changes made after that, which includes the second savepoint! Similarly, calling conn.rollback() on line 25 would void both savepoints, and line 27 would again throw an exception.

Reviewing Transaction APIs

There aren't many methods for working with transactions, but you need to know all of the ones in Table 15.9.

TABLE 15.9 Connection APIs for transactions

Method	Description
setAutoCommit(boolean b)	Sets mode for whether to commit right away
commit()	Saves data in database
rollback()	Gets rid of statements already made
rollback(Savepoint sp)	Goes back to state at Savepoint
setSavepoint()	Creates bookmark
setSavepoint(String name)	Creates bookmark with name

Closing Database Resources

As you saw in Chapter 14, "I/O," it is important to close resources when you are finished with them. This is true for JDBC as well. JDBC resources, such as a Connection, are expensive to create. Not closing them creates a resource leak that will eventually slow your program.

Throughout the chapter, we've been using the try-with-resources syntax from Chapter 11. The resources need to be closed in a specific order. The ResultSet is closed first, followed by the PreparedStatement (or CallableStatement) and then the Connection.

While it is a good habit to close all three resources, it isn't strictly necessary. Closing a JDBC resource should close any resources that it created. In particular, the following are true:

- Closing a Connection also closes PreparedStatement (or CallableStatement) and ResultSet.
- Closing a PreparedStatement (or CallableStatement) also closes the ResultSet.

It is important to close resources in the right order. This avoids both resource leaks and exceptions.

Writing a Resource Leak

In Chapter 11, you learned that it is possible to declare a type before a try-with-resources statement. Do you see why this method is bad?

```
40: public void bad() throws SQLException {
41:   var url = "jdbc:hsqldb:zoo";
42:   var sql = "SELECT not_a_column FROM names";
43:   var conn = DriverManager.getConnection(url);
44:   var ps = conn.prepareStatement(sql);
45:   var rs = ps.executeQuery();
46:
47:   try (conn; ps; rs) {
48:     while (rs.next())
49:       System.out.println(rs.getString(1));
50:   }
51: }
```

Suppose an exception is thrown on line 45. The try-with-resources block is never entered, so we don't benefit from automatic resource closing. That means this code has a resource leak if it fails. Do not write code like this.

There's another way to close a ResultSet. JDBC automatically closes a ResultSet when you run another SQL statement from the same Statement. This could be a PreparedStatement or a CallableStatement.

Real World Scenaria

Dealing with Exceptions

In most of this chapter, we've lived in a perfect world. Sure, we mentioned that a checked SQLException might be thrown by any JDBC method—but we never caught it. We just declared it and let the caller deal with it. Now let's catch the exception.

```
    var sql = "SELECT not_a_column FROM names";
var url = "jdbc:hsqldb:zoo";
try (var conn = DriverManager.getConnection(url);
  var ps = conn.prepareStatement(sql);
  var rs = ps.executeQuery()) {
  while (rs.next())
    System.out.println(rs.getString(1));
} catch (SQLException e) {
  System.out.println(e.getMessage());
  System.out.println(e.getSQLState());
  System.out.println(e.getErrorCode());
}
```

The output looks like this:

```
    Column 'NOT_A_COLUMN' is either not in any table …
42X04
30000
```

Each of these methods gives you a different piece of information. The getMessage() method returns a human-readable message about what went wrong. We've only included the beginning of it here. The getSQLState() method returns a code as to what went wrong. You can Google the name of your database and the SQL state to get more information about the error. In comparison, getErrorCode() is a database-specific code. On this database, it doesn't do anything.

Summary

There are four key SQL statements you should know for the exam, one for each of the CRUD operations: create (INSERT) a new row, read (SELECT) data, update (UPDATE) one or more rows, and delete (DELETE) one or more rows.