Chapter 21
JDBC

THE OCP EXAM TOPICS COVERED IN THIS CHAPTER INCLUDE THE FOLLOWING:

- Database Applications with JDBC
  - Connect to databases using JDBC URLs and DriverManager
  - Use PreparedStatement to perform CRUD operations
  - Use PreparedStatement and CallableStatement APIs to perform database operations

---

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

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, Allen G. Taylor (Wiley, 2018) and *Practical Database Programming with Java*, 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 topics:

- 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 actually have to understand this rather than relying 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 Language (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 ob-

jects. 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*. This is for databases that 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 will also cover some vocabulary that you need to know.

PICKING DATABASE SOFTWARE

In all of the other 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.

In this book we will be using Derby (http://db.apache.org/derby) for most of the examples. It is a small in-memory database. In fact, you need only one JAR file to run it. While the download is really easy, we've still provided instructions on what to do. They are linked from the book page.

www.selikoff.net/ocp11-2

There are also stand-alone databases that you can choose from if you want to install a full-fledged database. We like MySQL (www.mysql.com) or PostgreSQL (www.postgresql.org), both of which are open source and have been around for more than 20 years.

While the major databases all have many similarities, they do have important differences and advanced features. Choosing the correct database for use in your job is an important decision that you need to spend much time researching. For the exam, any database is fine for practice.

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 21.1 shows the structure of our database.

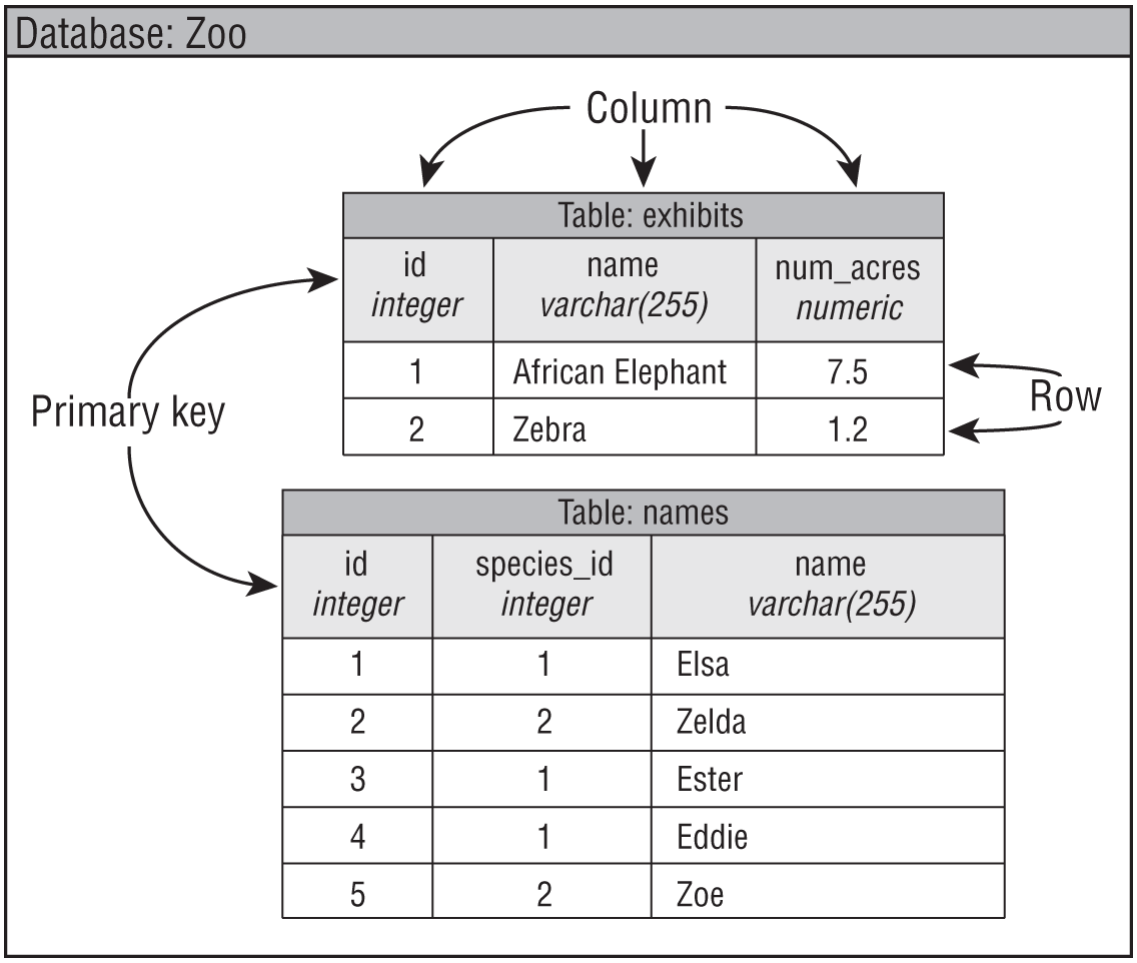

FIGURE 21.1 Tables in our relational database

As you can see in Figure 21.1, 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 it 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.

THE CODE TO SET UP THE DATABASE

We provide a program to download, install, and set up Derby to run the examples in this chapter. You don't have to understand it for the exam. Parts of SQL called database definition language (DDL) and database manipulation language (DML) are used to do so. Don't worry—knowing how to read or write SQL is not on the exam!

Before running the following code, you need to add a `.jar` file to your classpath. Add `<PATH TO DERBY>/derby.jar` to your classpath. Just make sure to replace `<PATH TO DERBY>` with the actual path on your file system. You run the program like this:

```
java -cp "<path_to_derby>/derby.jar" SetupDerbyDatabase.java
```

You can also find the code along with more details about setup here:

www.selikoff.net/ocp11-2

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. Then it loads data into those tables. By the end of this chapter, you should understand how to create a `Connection` and `PreparedStatement` in this manner.

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 ta-

bles. 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 of each in Table 21.1.

TABLE 21.1 CRUD operations

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

If you already know SQL, you can skip to the section on JDBC. We are covering the basics so that newer developers know what is going on, at least at a high level. We promise there is nothing else in this section on SQL that you need to know. In fact, you probably know a lot that isn't covered here. As far as the exam is concerned, joining two tables is a concept that doesn't exist!

Unlike Java, SQL keywords are case insensitive. This means `select`, `SELECT`, and `Select` are all equivalent. Many people use uppercase for the database keywords so that they stand out. It's also common practice to use *snake case* (underscores to separate "words") in column names. We

follow these conventions. Note that in some databases, table and column names may be case sensitive.

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.

Now it's time to write some code. The `INSERT` statement is usually used to create one new row in a table; here's an example:

```
INSERT INTO exhibits
VALUES (3, 'Asian Elephant', 7.5);
```

If there are two rows in the table before this command is run successfully, then there are three afterward. The `INSERT` statement lists the values that we want to insert. By default, it uses the same order in which the columns were defined. `String` data is enclosed in single quotes.

The `SELECT` statement reads data from the table.

```
SELECT *
FROM exhibits
WHERE ID = 3;
```

The `WHERE` clause is optional. If you omit it, the contents of the entire table are returned. The `*` indicates to return all of the columns in the order in which they were defined. Alternatively, you can list the columns that you want returned.

```
SELECT name, num_acres
FROM exhibits
WHERE id = 3;
```

It is preferable to list the column names for clarity. It also helps in case the table changes in the database.

You can also get information about the whole result without returning individual rows using special SQL functions.

```
SELECT COUNT(*), SUM(num_acres)
FROM exhibits;
```

This query tells us how many species we have and how much space we need for them. It returns only one row since it is combining information. Even if there are no rows in the table, the query returns one row that contains zero as the answer.

The `UPDATE` statement changes zero or more rows in the database.

```
UPDATE exhibits
SET num_acres = num_acres + .5
WHERE name = 'Asian Elephant';
```

Again, the `WHERE` clause is optional. If it is omitted, all rows in the table will be updated. The `UPDATE` statement always specifies the table to update and the column to update.

The `DELETE` statement deletes one or more rows in the database.

```
DELETE FROM exhibits
WHERE name = 'Asian Elephant';
```

And yet again, the `WHERE` clause is optional. If it is omitted, the entire table will be emptied. So be careful!

All of the SQL shown in this section is common across databases. For more advanced SQL, there is variation across databases.

Introducing the Interfaces of JDBC

For the exam you need to know five key interfaces of JDBC. The interfaces are declared in the JDK. This is just like all of the other interfaces and classes that you've seen in this book. For example, in Chapter 14, "Generics and Collections," 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 21.2 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.

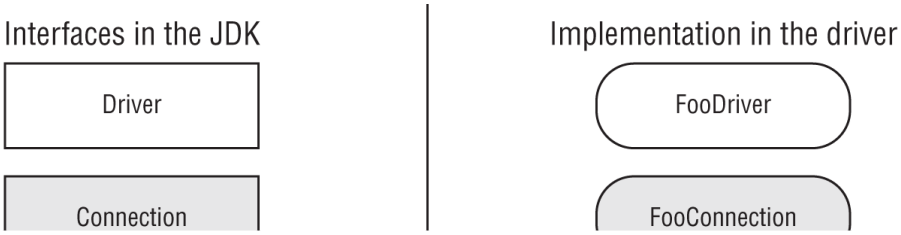

FIGURE 21.2 Key JDBC interfaces

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 results of a query

All database interfaces are in the package `java.sql` , so we will often omit the imports.

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 `DriverManager` instead of `DataSource` .

```
public class MyFirstDatabaseConnection {
   public static void main(String[] args) throws SQLException {
      String url = "jdbc:derby:zoo";
      try (Connection conn = DriverManager.getConnection(url);
         PreparedStatement ps = conn.prepareStatement(
            "SELECT name FROM animal");
         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 `derby` instead, it returns the implementations that `derby` has provided for these interfaces. You don't need to know their names. In the rest of the chapter, we will 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 11, "Modules," this module is included automatically when you run your application as a module.

By 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 it 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 will show you how to build the JDBC URL. Then, we will 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 the URL of the website. 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, a JDBC URL has a variety of formats. They have three parts in common, as shown in Figure 21.3. 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 `derby`, `mysql`, or `postgres`. The third part is the *subname*, which is a database-specific format. Colons (`:`) separate the three parts.

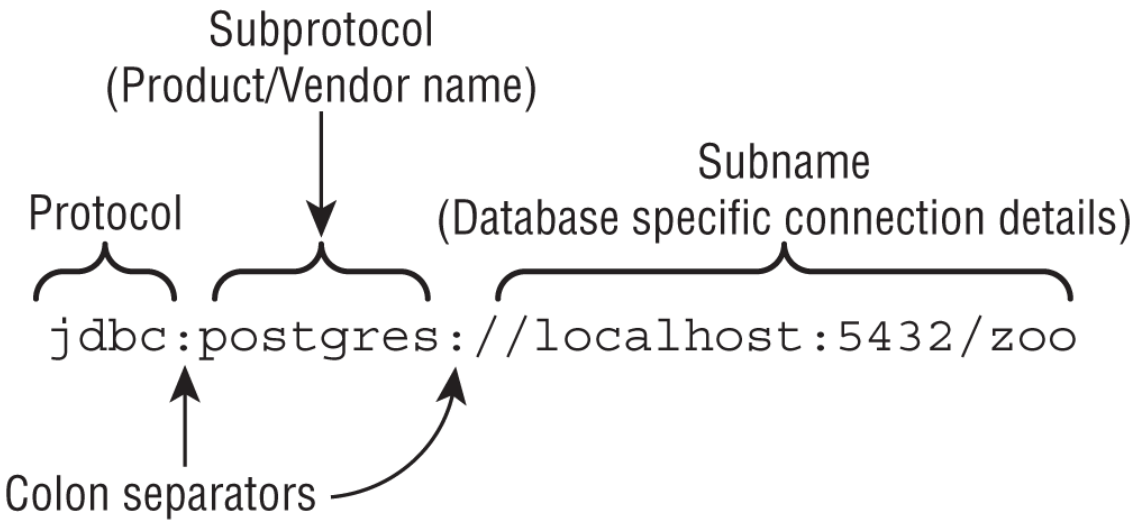

FIGURE 21.3 The JDBC URL format

The subname typically contains information about the database such as the location and/or name of the database. 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:derby:zoo
```

Notice the three parts. It starts with `jdbc` and then comes the subprotocol `derby`, and it ends with the subname, which is the database name. The location is not required, because Derby is an in-memory database.

Other examples of subname 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, and then followed by the vendor/product name. After that it varies. Notice how all of them include the location of the database, which are `localhost`, `123.123.123.123:1521`, and `localhost:3306`. Also, notice that the port is optional when using the default location.

To make sure you get this, do you see what is wrong with each of the following?

```
jdbc:postgresql://local/zoo
jdbc:mysql://123456/zoo
jdbc;oracle;thin;/localhost/zoo
```

The first one uses `local` instead of `localhost`. The literal `localhost` is a specially defined name. You can't just make up a name. Granted, it is possible for our database server to be named *local*, but the exam won't have you assume names. If the database server has a special name, the question will let you know it. The second one says that the location of the database is `123456`. This doesn't make sense. A location can be `localhost` or an IP address or a domain name. It can't be any random number. The third one is no good because it uses semicolons (`;`) instead of colons (`:`).

Getting a Database *Connection*

There are two main ways to get a `Connection`: `DriverManager` or `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.

Real World Scenario

USING A *DATASOURCE*

In real applications, you should use a `DataSource` rather than `DriverManager` to get a `Connection`. For one thing, there's no reason why you should have to know the database password. It's far better if the database team or another team can set up a data source that you can reference.

Another reason is that a `DataSource` maintains a connection pool so that you can keep reusing the same connection rather than needing to get a new one each time. Even the Javadoc says `DataSource` is preferred over `DriverManager`. But `DriverManager` is in the exam objectives, so you still have to know it.

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. 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 Derby database, you write the following:

```
import java.sql.*;
public class TestConnect {
   public static void main(String[] args) throws SQLException {
      Connection conn =
         DriverManager.getConnection("jdbc:derby:zoo");
      System.out.println(conn);
   }
}
```

Running this example as `java TestConnect.java` will give you an error that looks like this:

```
Exception in thread "main" java.sql.SQLException:
No suitable driver found for jdbc:derby:zoo
   at java.sql/java.sql.DriverManager.getConnection(
      DriverManager.java:702)
   at java.sql/java.sql.DriverManager.getConnection(
      DriverManager.java:251)
   at connection.TestConnect.main(TestConnect.java:9)
```

Seeing `SQLException` means "something went wrong when connecting to or accessing the database." You will need to recognize when a `SQLException` is thrown, but not the exact message. As you learned in Chapter 16, "Exceptions, Assertions, and Localization," `SQLException` is a checked exception.

If code snippets aren't in a method, you can assume they are in a context where checked exceptions are handled or declared.

In this case, we didn't tell Java where to find the database driver JAR file. Remember that the implementation class for `Connection` is found inside a driver JAR.

We try this again by adding the classpath with the following:

```
java -cp "<path_to_derby>/derby.jar" TestConnect.java
```

Remember to substitute the location of where the Derby JAR is located.

Notice that we are using single-file source-code execution rather than compiling the code first. This allows us to use a simpler classpath since it has only one element.

This time the program runs successfully and prints something like the following:

```
org.apache.derby.impl.jdbc.EmbedConnection40@1372082959
(XID = 156), (SESSIONID = 1), (DATABASE = zoo), (DRDAID = null)
```

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 {
      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`.

Real World Scenario

USING *CLASS.FORNAME()*

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`. It looks like this:

```
public static void main(String[] args)
   throws SQLException, ClassNotFoundException {

   Class.forName("org.postgresql.Driver");
   Connection conn = DriverManager.getConnection(
      "jdbc:postgresql://localhost:5432/ocp-book",
      "username",
      "password");
}
```

`Class.forName()` loads a class before it is used. With newer drivers, `Class.forName()` is no longer required.

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 21.4.

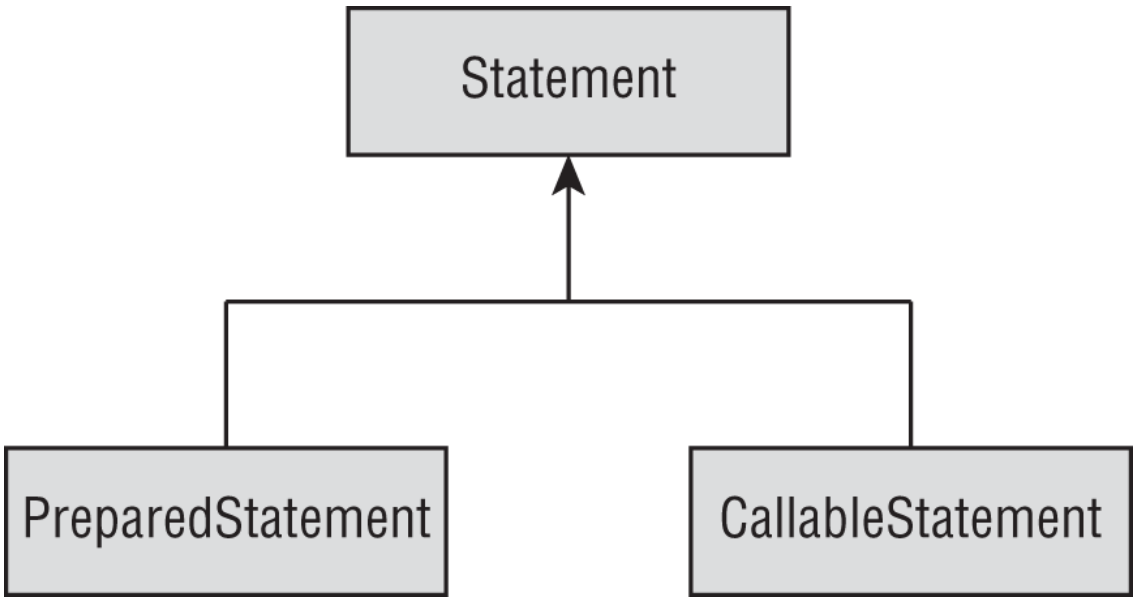

FIGURE 21.4 Types of statements

Later in the chapter, you'll learn about using `CallableStatement` for queries that are inside the database. In this section, we will be looking at `PreparedStatement`.

What about `Statement` you ask? It is an interface that both `PreparedStatement` and `CallableStatement` extend. A `Statement` and `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. A `PreparedStatement` figures out a plan to run the SQL well and remembers it.
- **Security**: As you will see in Chapter 22, "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.

Using the `Statement` interface is also no longer in scope for the JDBC exam, so we do not cover it in this book. In the following sections, we will 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 actually execute the query yet! We'll 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 will 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 processes the results in order.

Executing a *PreparedStatement*

Now that we have a `PreparedStatement`, we can run the SQL statement. The way you run 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 out with statements that change the data in a table. That would be 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 actually run it. Since we are running a query to get a result, the return type is `ResultSet`. In the next section, we will 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.

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 conn = DriverManager.getConnection("jdbc:derby:zoo");
   var ps = conn.prepareStatement(sql)) {

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

This throws a `SQLException` similar to the following:

```
Statement.executeUpdate() cannot be called with a statement
that returns a ResultSet.
```

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.

```
Statement.executeQuery() cannot be called with a statement
that returns a row count.
```

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 21.2 and Table 21.3 well. Table 21.2 shows which SQL statements can be run by each of the three key methods on `PreparedStatement`.

TABLE 21.2 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 21.3 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`	The rows and columns returned	n/a
`ps.executeUpdate()`	`int`	n/a	Number of rows added/changed/removed

Table 21.3 shows what is returned by each method.

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. However, we don't want the caller of this method to

need to write SQL or know the exact details of our database.

Luckily, a `PreparedStatement` allows us to set parameters. Instead of specifying the three values in the SQL, we can use a question mark (`?`) instead. 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 re-write 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, 21, and 22 set those variables. You can think of the bind variables as a list of parameters where each one gets set in turn. Notice how the bind variables can be set in any order. Line 23 actually 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 (and interview) 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.

```
At least one parameter to the current statement is uninitialized.
```

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 Derby, that message was as follows:

```
The number of values assigned is not the same as the
number of specified or implied columns.
```

Table 21.4 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 getting. There are many others, like dates, that are out of scope for the exam.

TABLE 21.4 `PreparedStatement` methods

Method name	Parameter type	Example database type
`setBoolean`	`Boolean`	`BOOLEAN`
`setDouble`	`Double`	`DOUBLE`
`setInt`	`Int`	`INTEGER`
`setLong`	`Long`	`BIGINT`
`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.

Notice 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.

Real World Scenario

COMPILE VS. RUNTIME ERROR WHEN EXECUTING

The following code is incorrect. Do you see why?

```
ps.setObject(1, key);
ps.setObject(2, type);
ps.setObject(3, name);
ps.executeUpdate(sql);  // INCORRECT
```

The problem is that the last line passes a SQL statement. With a `PreparedStatement`, we pass the SQL in when creating the object.

More interesting is that this does not result in a compiler error. Remember that `PreparedStatement` extends `Statement`. The `Statement` interface does accept a SQL statement at the time of execution, so the code compiles. Running this code gives `SQLException`. The text varies by database.

Updating Multiple Times

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 Scenario

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, then 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 the array has two elements since there are two different items in the batch. Each one added one row in the database.

```
[1, 1]
```

When using batching, you should call `executeBatch()` at a set interval, such as every 10,000 records (rather than after ten million). Waiting too long to send the batch to the database could produce operations that are so large that they freeze the client (or even worse the database!).

Getting Data from a ResultSet

A database isn't useful if you can't get your data. We will start by showing you how to go through a `ResultSet`. Then we will 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: Map<Integer, String> idToNameMap = new HashMap<>();
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 21.5 shows the position as we loop through.

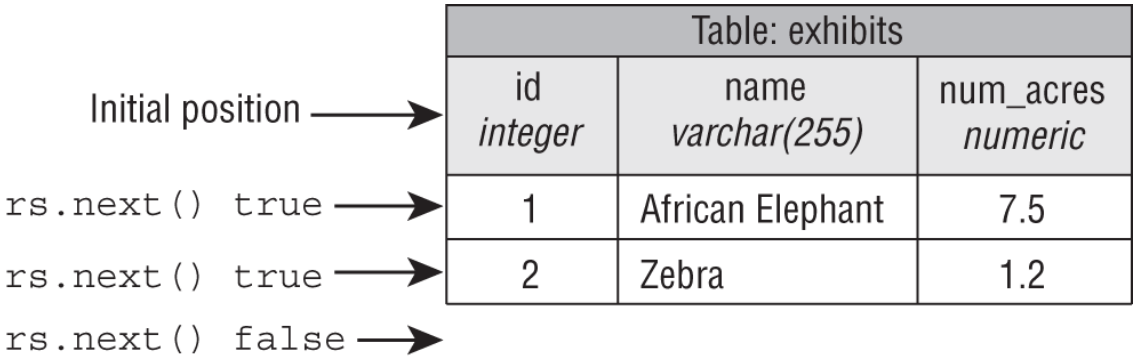

FIGURE 21.5 The `ResultSet` cursor

At line 24, the cursor starts out 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.” There is another way to access the columns. You can use an index instead of a column name. 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.

Rewriting this same example with column numbers looks like the following:

```
20: String sql = "SELECT id, name FROM exhibits";
21: Map<Integer, String> idToNameMap = new HashMap<>();
22:
23: try (var ps = conn.prepareStatement(sql);
24:    ResultSet rs = ps.executeQuery()) {
25:
26:    while (rs.next()) {
27:       int id = rs.getInt(1);
28:       String name = rs.getString(2);
29:       idToNameMap.put(id, name);
30:    }
31:    System.out.println(idToNameMap);
32: }
```

This time, you can see the column positions on lines 27 and 28. 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`.

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

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

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

```
Column 'total' not found.
```

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 here 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 does throw a `SQLException` because the result set cursor does not point to a valid position. If there actually were a match 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. How about this one?

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

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

   if (rs.next())
      rs.getInt(0); // SQLException
}
```

Since column indexes begin with 1, there is no column 0 to point to and a `SQLException` is thrown. Let's try one more example. What is wrong with this one?

```
var sql = "SELECT name FROM exhibits";

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

   if (rs.next())
   rs.getInt("badColumn"); // SQLException
}
```

Trying to get a column that isn't in the `ResultSet` is just as bad as an invalid column index, and it also throws a `SQLException`.

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 21.5 shows the `get` methods that you need to know. These are the getter equivalents of the setters in Table 21.4.

TABLE 21.5 `ResultSet get` methods

Method name	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 21.5. 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) {
24:          int id = (Integer) idField;
25:          System.out.println(id);
26:       }
27:       if (nameField instanceof String) {
28:          String name = (String) nameField;
29:          System.out.println(name);
30:       }
31:    }
32: }
```

Lines 21 and 22 get the column as whatever type of `Object` is most appropriate. Lines 23–26 show you how to confirm that the type is `Integer` before casting and unboxing it into an `int`. Lines 27–30 show you how to confirm that the type is `String` and cast it as well. 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*

Sometimes you want your SQL to be directly in the database instead of packaged with your Java code. This is particularly useful when you have many queries and they are complex. 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 of 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. If you want to try the examples, the setup procedure and source code is linked from here:

www.selikoff.net/ocp11-2

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 21.6 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.

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

TABLE 21.6 Sample stored procedures

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

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 included 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 will 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 50 is required since it passes the parameter. Similarly, line 52 is required since it sets the parameter. For an `OUT` parameter, line 53 is required to register the parameter. 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 21.7 reviews the different types of parameters. You need to know this well for the exam.

TABLE 21.7 Stored procedure parameter types

	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

Closing Database Resources

As you saw in Chapter 19, “I/O,” and Chapter 20, “NIO.2,” 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 down your program.

Throughout the chapter, we've been using the try-with-resources syntax from Chapter 16. 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 16, 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:derby: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`. How many resources are closed in this code?

```
14: var url = "jdbc:derby:zoo";
15: var sql = "SELECT count(*) FROM names where id = ?";
16: try (var conn = DriverManager.getConnection(url);
17:    var ps = conn.prepareStatement(sql)) {
18:
19:    ps.setInt(1, 1);
20:
```

```
21:    var rs1 = ps.executeQuery();
22:    while (rs1.next()) {
23:       System.out.println("Count: " + rs1.getInt(1));
24:    }
25:
26:    ps.setInt(1, 2);
27:
28:    var rs2 = ps.executeQuery();
29:    while (rs2.next()) {
30:       System.out.println("Count: " + rs2.getInt(1));
31:    }
32:    rs2.close();
33: }
```

The correct answer is four. On line 28, `rs1` is closed because the same `PreparedStatement` runs another query. On line 32, `rs2` is closed in the method call. Then the try-with-resources statement runs and closes the `PreparedSatement` and `Connection` objects.

Real World Scenario

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 actually 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:derby: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 as to 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. By 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.

For the exam, you should be familiar with five JDBC interfaces: `Driver` , `Connection` , `PreparedStatement` , `CallableStatement` , and `ResultSet` . The interfaces are part of the Java API. A database-specific JAR file provides the implementations.

To connect to a database, you need the JDBC URL. A JDBC URL has three parts separated by colons. The first part is `jdbc` . The second part is the name of the vendor/product. The third part varies by database, but it includes the location and/or name of the database. The location is either `localhost` or an IP address followed by an optional port.

The `DriverManager` class provides a factory method called `getConnection()` to get a `Connection` implementation. You create a `PreparedStatement` or `CallableStatement` using `prepareStatement()` and `prepareCall()` , respectively. A `PreparedStatement` is used when the SQL is specified in your application, and a `CallableStatement` is used when the SQL is in the database. A `PreparedStatement` allows you to set the values of bind variables. A `CallableStatement` also allows you to set `IN` , `OUT` , and `INOUT` parameters.

When running a `SELECT` SQL statement, the `executeQuery()` method returns a `ResultSet` . When running a `DELETE` , `INSERT` , or `UPDATE` SQL statement, the `executeUpdate()` method returns the number of rows that were affected. There is also an `execute()` method that returns a `boolean` to indicate whether the statement was a query.

You call `rs.next()` from an `if` statement or `while` loop to advance the cursor position. To get data from a column, call a method like `getString(1)` or `getString("a")`. Column indexes begin with `1`, not `0`. In addition to getting a `String` or primitive, you can call `getObject()` to get any type.

It is important to close JDBC resources when finished with them to avoid leaking resources. Closing a `Connection` automatically closes the `Statement` and `ResultSet` objects. Closing a `Statement` automatically closes the `ResultSet` object. Also, running another SQL statement closes the previous `ResultSet` object from that `Statement`.

Exam Essentials

- **Name the core five JDBC interfaces that you need to know for the exam and where they are defined.** The five key interfaces are `Driver`, `Connection`, `PreparedStatement`, `CallableStatement`, and `ResultSet`. The interfaces are part of the core Java APIs. The implementations are part of a database driver JAR file.
- **Identify correct and incorrect JDBC URLs.** A JDBC URL starts with `jdbc:`, and it is followed by the vendor/product name. Next comes another colon and then a database-specific connection string. This database-specific string includes the location, such as `localhost` or an IP address with an optional port. It may also contain the name of the database.
- **Describe how to get a *`Connection`* using *`DriverManager`*.** After including the driver JAR in the classpath, call `DriverManager.getConnection(url)` or `DriverManager.getConnection(url, username, password)` to get a driver-specific `Connection` implementation class.
- **Run queries using a *`PreparedStatement`*.** When using a `PreparedStatement`, the SQL contains question marks (`?`) for the parameters or bind variables. This SQL is passed at the time the

`PreparedStatement` is created, not when it is run. You must call a setter for each of these with the proper value before executing the query.

- **Run queries using a *`CallableStatement`*.** When using a `CallableStatement`, the SQL looks like `{ call my_proc(?)}`. If you are returning a value, `{?= call my_proc(?)}` is also permitted. You must set any parameter values before executing the query. Additionally, you must call `registerOutParameter()` for any `OUT` or `INOUT` parameters.
- **Choose which method to run given a SQL statement.** For a `SELECT` SQL statement, use `executeQuery()` or `execute()`. For other SQL statements, use `executeUpdate()` or `execute()`.
- **Loop through a *`ResultSet`*.** Before trying to get data from a `ResultSet`, you call `rs.next()` inside an `if` statement or `while` loop. This ensures that the cursor is in a valid position. To get data from a column, call a method like `getString(1)` or `getString("a")`. Remember that column indexes begin with 1.
- **Identify when a resource should be closed.** If you're closing all three resources, the `ResultSet` must be closed first, followed by the `PreparedStatement`, `CallableStatement`, and then followed by the `Connection`.

Review Questions

The answers to the chapter review questions can be found in the Appendix.

1. Which interfaces or classes are in a database-specific JAR file? (Choose all that apply.)
   A. `Driver`
   B. `Driver`'s implementation
   C. `DriverManager`
   D. `DriverManager`'s implementation
   E. `PreparedStatement`
   F. `PreparedStatement`'s implementation

2. Which are required parts of a JDBC URL? (Choose all that apply.)
   A. Connection parameters
   B. IP address of database
   C. `jdbc`
   D. Password
   E. Port
   F. Vendor-specific string
3. Which of the following is a valid JDBC URL?
   A. `jdbc:sybase:localhost:1234/db`
   B. `jdbc::sybase::localhost::/db`
   C. `jdbc::sybase:localhost::1234/db`
   D. `sybase:localhost:1234/db`
   E. `sybase::localhost::/db`
   F. `sybase::localhost::1234/db`
4. Which of the options can fill in the blank to make the code compile and run without error? (Choose all that apply.)

```
var sql = "UPDATE habitat WHERE environment = ?";
try (var ps = conn.prepareStatement(sql)) {

   _________

   ps.executeUpdate();
}
```

   A. `ps.setString(0, "snow");`
   B. `ps.setString(1, "snow");`
   C. `ps.setString("environment", "snow");`
   D. `ps.setString(1, "snow"); ps.setString(1, "snow");`
   E. `ps.setString(1, "snow"); ps.setString(2, "snow");`
   F. `ps.setString("environment", "snow");ps.setString("environment", "snow");`
5. Suppose that you have a table named `animal` with two rows. What is the result of the following code?

```
 6:  var conn = new Connection(url, userName, password);
 7:  var ps = conn.prepareStatement(
 8:     "SELECT  count(*) FROM animal");
 9:  var rs = ps.executeQuery();
10: if (rs.next()) System.out.println(rs.getInt(1));
```

A. `0`
B. `2`
C. There is a compiler error on line 6.
D. There is a compiler error on line 10.
E. There is a compiler error on another line.
F. A runtime exception is thrown.

6. Which of the options can fill in the blanks in order to make the code compile?

```
boolean bool = ps.();
int num = ps.();
ResultSet rs = ps.();
```

A. `execute`, `executeQuery`, `executeUpdate`
B. `execute`, `executeUpdate`, `executeQuery`
C. `executeQuery`, `execute`, `executeUpdate`
D. `executeQuery`, `executeUpdate`, `execute`
E. `executeUpdate`, `execute`, `executeQuery`
F. `executeUpdate`, `executeQuery`, `execute`

7. Which of the following are words in the CRUD acronym? (Choose all that apply.)
A. Create
B. Delete
C. Disable
D. Relate
E. Read
F. Upgrade

8. Suppose that the table `animal` has five rows and the following SQL statement updates all of them. What is the result of this code?

```
public static void main(String[] args) throws SQLException {
    var sql = "UPDATE names SET name = 'Animal'";
    try (var conn = DriverManager.getConnection("jdbc:derby:zoo");
       var ps = conn.prepareStatement(sql)) {

       var result = ps.executeUpdate();
       System.out.println(result);
    }
 }
```

A. `0`

B. `1`

C. `5`

D. The code does not compile.

E. A `SQLException` is thrown.

F. A different exception is thrown.

9. Suppose `learn()` is a stored procedure that takes one `IN` parameter. What is wrong with the following code? (Choose all that apply.)

```
18: var sql = "call learn()";
19: try (var cs = conn.prepareCall(sql)) {
20:    cs.setString(1, "java");
21:    try (var rs = cs.executeQuery()) {
22:       while (rs.next()) {
23:          System.out.println(rs.getString(3));
24:       }
25:    }
26: }
```

A. Line 18 is missing braces.

B. Line 18 is missing a `?`.

C. Line 19 is not allowed to use `var`.

D. Line 20 does not compile.

E. Line 22 does not compile.

F. Something else is wrong with the code.

G. None of the above. This code is correct.

10. Suppose that the table `enrichment` has three rows with the animals `bat`, `rat`, and `snake`. How many lines does this code print?

```
var sql = "SELECT toy FROM enrichment WHERE animal = ?";
try (var ps = conn.prepareStatement(sql)) {
   ps.setString(1, "bat");

   try (var rs = ps.executeQuery(sql)) {
      while (rs.next())
         System.out.println(rs.getString(1));
   }
}
```

A. 0

B. 1

C. 3

D. The code does not compile.

E. A `SQLException` is thrown.

F. A different exception is thrown.

11. Suppose that the table `food` has five rows and this SQL statement updates all of them. What is the result of this code?

```
public static void main(String[] args) {
   var sql = "UPDATE food SET amount = amount + 1";
   try (var conn = DriverManager.getConnection("jdbc:derby:zoo");
      var ps = conn.prepareStatement(sql)) {

      var result = ps.executeUpdate();
      System.out.println(result);
   }
}
```

A. `0`

B. `1`

C. `5`

D. The code does not compile.

E. A `SQLException` is thrown.

F. A different exception is thrown.

12. Suppose we have a JDBC program that calls a stored procedure, which returns a set of results. Which is the correct order in which to close database resources for this call?

A. `Connection`, `ResultSet`, `CallableStatement`

B. `Connection`, `CallableStatement`, `ResultSet`

C. `ResultSet`, `Connection`, `CallableStatement`

D. `ResultSet`, `CallableStatement`, `Connection`

E. `CallableStatement`, `Connection`, `ResultSet`

F. `CallableStatement`, `ResultSet`, `Connection`

13. Suppose that the table `counts` has five rows with the numbers 1 to 5. How many lines does this code print?

```
var sql = "SELECT num FROM counts WHERE num> ?";
try (var ps = conn.prepareStatement(sql)) {
   ps.setInt(1, 3);

   try (var rs = ps.executeQuery()) {
      while (rs.next())
         System.out.println(rs.getObject(1));
   }

   ps.setInt(1, 100);

   try (var rs = ps.executeQuery()) {
      while (rs.next())
         System.out.println(rs.getObject(1));
   }
}
```

A. 0

B. 1

C. 2

D. 4

E. The code does not compile.

F. The code throws an exception.

14. Which of the following can fill in the blank correctly? (Choose all that apply.)

```
var rs = ps.executeQuery();
if (rs.next()) {
   _____________________________________;
}
```

A. `String s = rs.getString(0)`

B. `String s = rs.getString(1)`

C. `String s = rs.getObject(0)`

D. `String s = rs.getObject(1)`

E. `Object s = rs.getObject(0)`

F. `Object s = rs.getObject(1)`

15. Suppose `learn()` is a stored procedure that takes one `IN` parameter and one `OUT` parameter. What is wrong with the following code? (Choose all that apply.)

```
18: var sql = "{?= call learn(?)}";
19: try (var cs = conn.prepareCall(sql)) {
20:    cs.setInt(1, 8);
21:    cs.execute();
22:    System.out.println(cs.getInt(1));
23: }
```

A. Line 18 does not call the stored procedure properly.

B. The parameter value is not set for input.

C. The parameter is not registered for output.

D. The code does not compile.

E. Something else is wrong with the code.

F. None of the above. This code is correct.

16. Which of the following can fill in the blank? (Choose all that apply.)

```
var sql = "_____________________";
try (var ps = conn.prepareStatement(sql)) {
   ps.setObject(3, "red");
   ps.setInt(2, 8);
   ps.setString(1, "ball");
   ps.executeUpdate();
}
```

A. `{ call insert_toys(?, ?) }`
B. `{ call insert_toys(?, ?, ?) }`
C. `{ call insert_toys(?, ?, ?, ?) }`
D. `INSERT INTO toys VALUES (?, ?)`
E. `INSERT INTO toys VALUES (?, ?, ?)`
F. `INSERT INTO toys VALUES (?, ?, ?, ?)`

17. Suppose that the table `counts` has five rows with the numbers 1 to 5. How many lines does this code print?

```
var sql = "SELECT num FROM counts WHERE num> ?";
try (var ps = conn.prepareStatement(sql)) {
   ps.setInt(1, 3);

   try (var rs = ps.executeQuery()) {
      while (rs.next())
         System.out.println(rs.getObject(1));
   }

   try (var rs = ps.executeQuery()) {
      while (rs.next())
         System.out.println(rs.getObject(1));
   }
}
```

A. 0
B. 1

C. 2

D. 4

E. The code does not compile.

F. The code throws an exception.

18. There are currently 100 rows in the table `species` before inserting a new row. What is the output of the following code?

```
String insert = "INSERT INTO species VALUES (3, 'Ant', .05)";
String select = "SELECT count(*) FROM species";
try (var ps = conn.prepareStatement(insert)) {
   ps.executeUpdate();
}
try (var ps = conn.prepareStatement(select)) {
   var rs = ps.executeQuery();
   System.out.println(rs.getInt(1));
}
```

A. `100`

B. `101`

C. The code does not compile.

D. A `SQLException` is thrown.

E. A different exception is thrown.

19. Which of the options can fill in the blank to make the code compile and run without error? (Choose all that apply.)

```
var sql = "UPDATE habitat WHERE environment = ?";
try (var ps = conn.prepareCall(sql)) {

    ________

   ps.executeUpdate();
}
```

A. `ps.setString(0, "snow");`

B. `ps.setString(1, "snow");`

C. `ps.setString("environment", "snow");`

D. The code does not compile.

E. The code throws an exception at runtime.

20. Which of the following could be true of the following code? (Choose all that apply.)

```
var sql = "{call transform(?)}";
try (var cs = conn.prepareCall(sql)) {
   cs.registerOutParameter(1, Types.INTEGER);
   cs.execute();
   System.out.println(cs.getInt(1));
}
```

A. The stored procedure can declare an `IN` or `INOUT` parameter.

B. The stored procedure can declare an `INOUT` or `OUT` parameter.

C. The stored procedure must declare an `IN` parameter.

D. The stored procedure must declare an `INOUT` parameter.

E. The stored procedure must declare an `OUT` parameter.

21. Which is the first line containing a compiler error?

```
25: String url = "jdbc:derby:zoo";
26: try (var conn = DriverManager.getConnection(url);
27:    var ps = conn.prepareStatement();
28:    var rs = ps.executeQuery("SELECT * FROM swings")) {
29:    while (rs.next()) {
30:       System.out.println(rs.getInteger(1));
31:    }
32: }
```

A. Line 26

B. Line 27

C. Line 28

D. Line 29

E. Line 30

F. None of the above