Chapter 21 - JDBC

Introducing Relational Databases and SQL

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 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.

Writing Basic SQL Statements

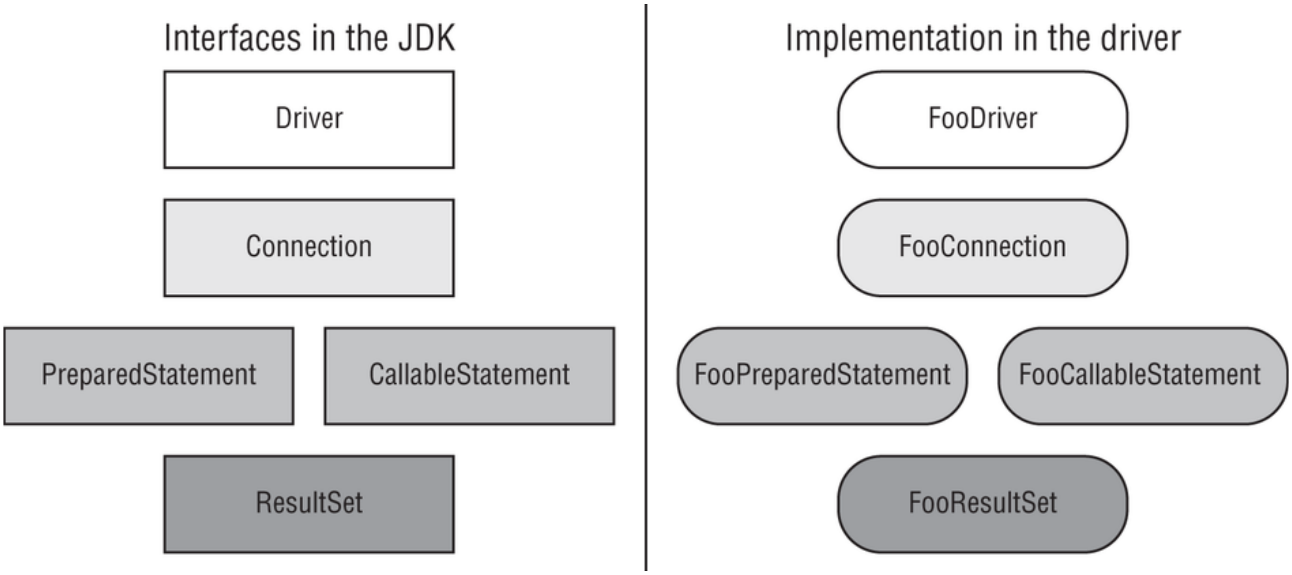
TABLE 21.1 CRUD operations

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

Introducing the Interfaces of JDBC

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.



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

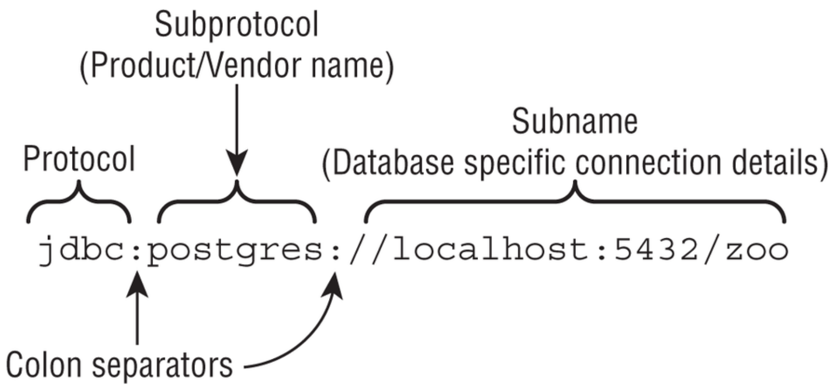
- `Driver`: Establishes a connection to the database
- `Connection`: Sends commands to a database
- `PreparedStatement`: Executes a SQL query
- `CallableStatement`: Executes commands stored in the database
- `ResultSet`: Reads results of a query

All database interfaces are in the package `java.sql`

```
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));
      }
   }
}
```

Building a JDBC URL

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.


Subprotocol
(Product/Vendor name)
Protocol
Subname
(Database specific connection details)
jdbc:postgres://localhost:5432/zoo
Colon separators

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
```

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.

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.

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

```java
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);
   }
}
```

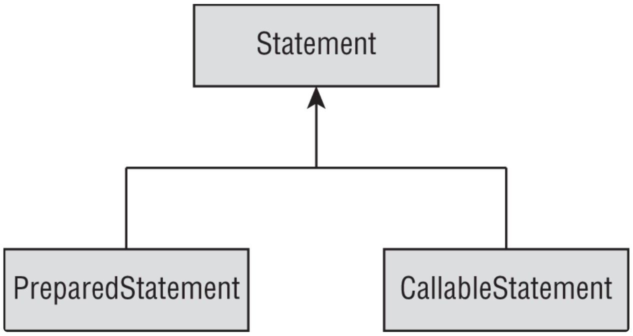
There is also a signature that takes a username and password.

```java
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);
   }
}
```

`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`.

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.



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.

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
}
```

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.

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.

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.

```
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: }
```

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`

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.
```

Reviewing *PreparedStatement* Methods

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

Working with Parameters

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 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: }
```

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:

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 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.

Getting Data from a ResultSet

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: }
```

A `ResultSet` has a *cursor*, which points to the current location in the data. Figure 21.5 shows the position as we loop through.

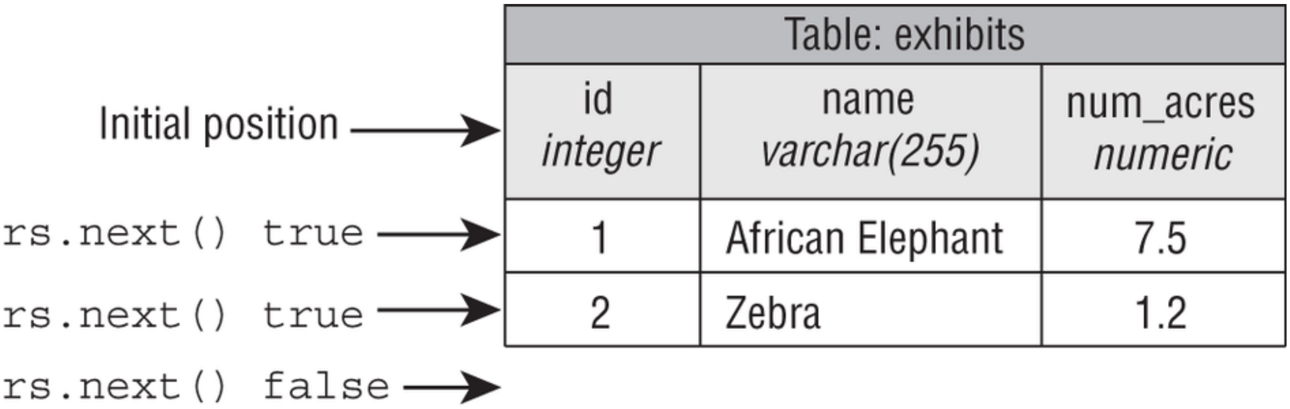


Table: exhibits

id *integer*	name *varchar(255)*	num_acres *numeric*
1	African Elephant	7.5
2	Zebra	1.2

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: }
```

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`

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()` wo

```
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.

TABLE 21.5 `ResultSet get` methods

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

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.

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 (`{}`).

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 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.

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`.

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

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`.

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.

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());

}
```

```
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.