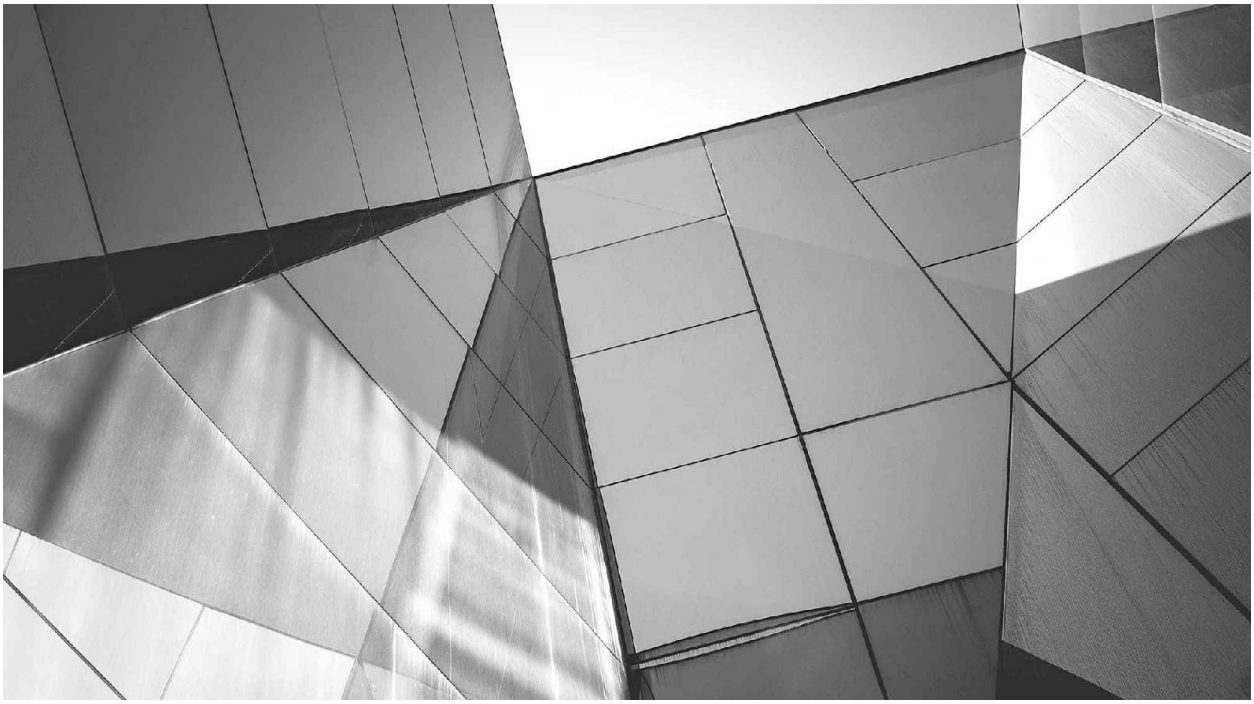


12

JDBC

CERTIFICATION OBJECTIVES

- Describe the interfaces that Make Up the Core of the JDBC API Including the Driver, Connection, Statement, and ResultSet Interfaces and Their Relationship to Provider Implementations
- Identify the Components Required to Connect to a Database Using the DriverManager Class Including the JDBC URL
- Submit Queries and Read Results from the Database Including Creating Statements, Returning Result Sets, Iterating Through the Results, and Properly Closing Result Sets, Statements, and Connections

✓ Two-Minute Drill

Q&A Self Test

This chapter covers the JDBC API that was added for the Java SE 7 and 8 exams. The exam developers have long felt that this API is truly a core feature of the language, and being able to demonstrate proficiency with JDBC goes a long way toward demonstrating your skills as a Java programmer.

Interestingly, JDBC has been a part of the language since JDK version 1.1 (1997) when JDBC 1.0 was introduced. Since then, there has been a steady progression of updates to the API, roughly one major release for each even-numbered JDK release, with the last major update being JDBC 4.0, released

in 2006 with Java SE 6. In Java SE 7 and 8, JDBC got some minor updates and is now at version 4.2. While the focus of the exam is on JDBC 4.*x*, there may be questions about the differences between loading a driver with a JDBC 3.0 and JDBC 4.*x* implementation, so we'll talk about that as well.

The good news is that the exam is not going to test your ability to write SQL statements. That would be an exam all by itself (maybe even more than one—SQL is a BIG topic!). But you will need to recognize some basic SQL syntax and commands, so we'll start by spending some time covering the basics of relational database systems and give you enough SQL to make you popular at database parties. If you feel you have experience with SQL and understand database concepts, you might just skim the first section or skip right to the first exam objective and dive right in.

Starting Out: Introduction to Databases and JDBC

When you think of organizing information and storing it in some easily understood way, a spreadsheet or a table is often the first approach you might take. A spreadsheet or a table is a natural way of categorizing information: The first row of a table defines the sort of information that the table will hold, and each subsequent row contains a set of data that is related to the key we create on the left. For example, suppose you wanted to chart your monthly spending for several types of expenses (Table 12-1).

TABLE 12-1 Methods to Map, Filter, and Reduce

Month	Gas	EatingOut	Utilities	Phone
January	$200.25	$109.87	$97.00	$45.08
February	$225.34	$121.08	$97.00	$23.36
March	$254.78	$130.45	$97.00	$56.09

From the data in the chart, we can determine that your overall expenses are increasing month to month in the first three months of this year. But notice that without the table, without a relationship between the month and the data in the columns, you would just have a pile of receipts with no way to draw out important conclusions, such as

- Assuming you drove the same number of miles per month, gas is getting pricey—maybe it is time to get a Prius.
- You are eating out more month to month (or the price of eating out is going up)—maybe it's time to start doing some meal planning.
- And maybe you need to be a little less social—that phone bill is high.

The point is that this small sample of data is the key to understanding a relational database system. A relational database is really just a software application designed to store and manipulate data in tables. The software itself is actually called a Relational Database Management System (RDBMS), but many people shorten that to just "database"—so know that going forward, when we refer to a database, we are actually talking about an RDBMS (the whole system). What the relational management system adds to a database is the ability to define relationships between tables. It also provides a language to get data in and out in a meaningful way.

Looking at the simple table in Table 12-1, we know that the data in the columns, Gas, EatingOut, Utilities, and Phone, are grouped by the months January, February, and so on. The month is unique to each row and identifies this row of data. In database parlance, the month is a "primary key." A primary key is generally required for a database table to identify which row of the table you want and to make sure that there are no duplicate rows.

Extending this a little further, if the data in Table 12-1 were stored in a database, I could ask the database (write a query) to give me all of the data for the month of January (again, my primary key is "month" for this table). I might write something like:

"Give me all of my expenses for January."

The result would be something like:

```
January: Gas: $200.25, EatingOut: $109.87, Utilities: $97.00, Phone: $45.08
```

This kind of query is what makes a database so powerful. With a relatively simple language, you can construct some really powerful queries in order to manipulate your data to tell a story. In most RDBMSs, this language is called the Structured Query Language (SQL). The same query we wrote out in a sentence earlier would be expressed like this in SQL:

```
SELECT * FROM Expenses WHERE Month = 'January'
```

which can be translated to “select all of the columns (`*`) from my table named ‘Expenses’ where the month column is equal to the string `'January'`.” Let’s look a bit more at how we “talk” to a database and what other sorts of queries we can make with tables in a relational database.

Talking to a Database

There are three important concepts when working with a database:

- Creating a connection to the database
- Creating a statement to execute in the database
- Getting back a set of data that represents the results

Let’s look at these concepts in more detail.

Before we can communicate with the software that manages the database, before we can send it a query, we need to make a connection with the RDBMS itself. There are many different types of connections, and a lot of underlying technology to describe the connection itself, but in general, to communicate with an RDBMS, we need to open a connection using an IP address and port number to the database. Once we have established the connection, we need to send it some parameters (such as a username and password) to authenticate ourselves as a valid user of the RDBMS. Finally, assuming all went well, we can send queries through the connection. This is like logging into your online account at a bank. You provide some credentials, a username and password, and a connection is established and opened between you and the bank. Later in the chapter, when we start writing code, we’ll open a connection using a Java class called the `DriverManager`, and in one request, pass in the database name, our username, and password.

Once we have established a connection, we can use some type of

application (usually provided by the database vendor) to send query statements to the database, have them executed in the database, and get a set of results returned. A set of results can be one row, as we saw before when we asked for the data from the month of January, or several rows. For example, suppose we wanted to see all of the Gas expenses from our Expenses table. We might query the database like this:

```
"Show me all of my Gas Expenses"
```

Or as a SQL query:

```
SELECT Gas FROM Expenses
```

The set of results that would "return" from my query would be three rows, and each row would contain one column.

$200.25
$225.34
$254.78

An important aspect of a database is that the data is presented back to you exactly the same way that it is stored. Since Gas expense is a column, the query will return three rows (one for January, one for February, and one for March). Note that because we did not ask the database to include the Month column in the results, all we got was the Gas column. The results do preserve the fact that Gas is a column and not a row and, in general, present the data in the same row-and-column order in which it is stored in the database.

SQL Queries

Let's look a bit more at the syntax of SQL, the language used to write queries in a database. There are really four basic SQL queries that we are going to use in this chapter and that are common to manipulating data in a database. In summary, the SQL commands we are interested in are used to perform

CRUD operations.

Like most terms presented in all caps, CRUD is an acronym and means *Create*, *Read*, *Update*, and *Delete*. These are the four basic operations for data in a database. They are represented by four distinct SQL commands, detailed in Table 12-2.

TABLE 12-2 Example SQL CRUD Commands

"CRUD"	SQL Command	Example SQL Query	Expressed in English
Create	INSERT	`INSERT INTO Expenses VALUES ('April', 231.21, 29.87, 97.00, 45.08)`	Add a new row (April) to expenses with the following values....
Read (or Find)	SELECT	`SELECT * FROM Expenses WHERE Month="February"`	Get me all of the columns in the Expenses table for February.
Read All	SELECT	`SELECT * FROM Expenses`	Get me all of the columns in the Expenses table.
Update	UPDATE	`UPDATE Expenses SET Phone=32.36, EatingOut=111.08 WHERE Month='February'`	Change my Phone expense and EatingOut expense for February to....
Delete	DELETE	`DELETE FROM Expenses WHERE Month='April'`	Remove the row of expenses for April.

Here is a quick explanation for the examples in Table 12-2:

- **INSERT** Add a row to the table Expenses, and set each of the columns in the table to the values expressed in the parentheses.
- **SELECT with WHERE** You have already seen the SELECT clause with a WHERE clause, so you know that this SQL statement returns a single row identified by the primary key—the Month column. Think of this statement as a refinement to Read—more like a Find or Find by primary key.
- **SELECT** When the SELECT clause does not have a WHERE clause, we are asking the database to return every row. Further, because we are using an asterisk (*) following the SELECT, we are asking for every column. Basically, it is a dump of the data shown in Table 15-1. Think of this statement as a Read All.
- **UPDATE** Change the data in the Phone and EatingOut cells to the new data provided for February.
- **DELETE** Remove a row altogether from the database where the Month is April.

Really, this is all the SQL you need to know for this chapter. There are many other SQL commands, but this is the core set. If we need to go beyond this set of four commands in the chapter, we will cover them as they come up. Now, let's look at a more detailed database example that we will use as the example set of tables for this chapter, using the data requirements of a small bookseller, Bob's Books.

SQL commands, like SELECT, INSERT, UPDATE, and so on, are case-insensitive. So it is largely by convention (and one we will use in this chapter) that we use all capital letters for SQL commands and key words, such as WHERE, FROM, LIKE, INTO, SET, and VALUES. SQL table names and column names, also called identifiers, can be case-sensitive or case-insensitive, depending on the database. The example code shown in

this chapter uses a case-insensitive database, so again, just for convention, we will use upper camel case, that is, the first letter of each noun capitalized and the rest in lowercase.

One final note about case—all databases preserve case when a string is delimited—that is, when it is enclosed in quotes. So a SQL clause that uses single or double quotation marks to delimit an identifier will preserve the case of the identifier.

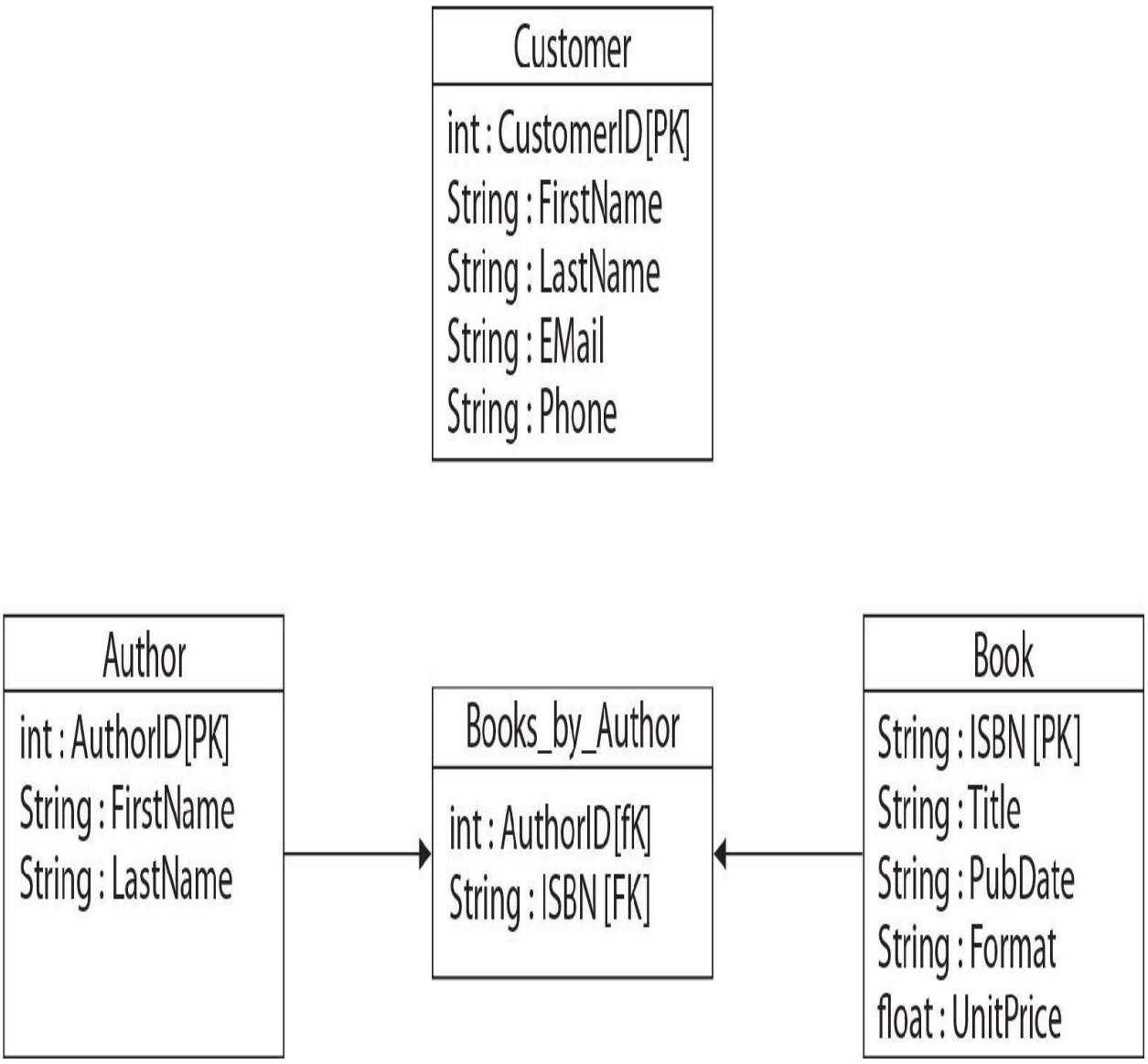
Bob's Books, Our Test Database

In this section, we'll describe a small database with a few tables and a few rows of data. As we work through the various JDBC topics in this chapter, we'll work with this database.

Bob is a small bookseller who specializes in children's books. Bob has designed his data around the need to sell his books online using a database (which one doesn't really matter) and a Java application. Bob has decided to use the JDBC API to allow him to connect to a database and perform queries through a Java application.

To start, let's look at the organization of Bob's data. In a database, the organization and specification of the tables is called the database schema (Figure 12-1). Bob's is a relatively simple schema, and again, for the purposes of this chapter, we are going to concentrate on just four tables from Bob's schema.

FIGURE 12-1 Bob's BookSeller database schema



This is a relatively simple schema that represents a part of the database for a small bookstore. In the schema shown, there is a table for Customer (Table 12-3). This table stores data about Bob's customers—a customer ID, first name and last name, an e-mail address, and phone number. Postal addresses and other information could be stored in another table.

TABLE 12-3 Bob's Books Customer Table Sample Data

e

CustomerID	FirstName	LastName	Email	Phone
5000	John	Smith	john.smith@verizon.net	555-340-1230
5001	Mary	Johnson	mary.johnson@comcast.net	555-123-4567
5002	Bob	Collins	bob.collins@yahoo.com	555-012-3456
5003	Rebecca	Mayer	rebecca.mayer@gmail.com	555-205-8212
5006	Anthony	Clark	anthony.clark@gmail.com	555-256-1901
5007	Judy	Sousa	judy.sousa@verizon.net	555-751-1207
5008	Christopher	Patriquin	patriquinc@yahoo.com	555-316-1803
5009	Deborah	Smith	debsmith@comcast.net	555-256-3421
5010	Jennifer	McGinn	jmcginn@comcast.net	555-250-0918

The next three tables we will look at represent the data required to store information about books that Bob sells. Because a book is a more complex set of data than a customer, we need to use one table for information about books, one for information about authors, and a third to create a relationship between books and authors.

Suppose that you tried to store a book in a single table with a column for the ISBN (International Standard Book Number), title, and author name. For many books, this would be fine. But what happens if a book has two authors? Or three authors? Remember that one requirement for a database table is a unique primary key, so you can’t simply repeat the ISBN in the table. In fact, having two rows with the same primary key will violate a key constraint in relational database design: the primary key of every row must be unique.

ISBN	Title	Author
ABCD	The Wonderful Life	Fred Smith
~~ABCD~~	~~The Wonderful Life~~	~~Tom Jones~~
1234	Some Enchanted Night	Paula Fredrick

Instead, there needs to be a way to have a separate table of books and authors and some way to link them together. Bob addressed this issue by placing Books in one table (Table 12-4) and Authors (Table 12-5) in another. The primary key for Books is the ISBN number, and therefore, each Book entry will be unique. For the Author table, Bob is creating a unique AuthorID for each author in the table.

TABLE 12-4 Bob's Books Sample Data for the "Books" Table

ISBN	Title	PubDate	Format	Price
142311339X	The Lost Hero (Heroes of Olympus, Book 1)	2010-10-12	Hardcover	10.95
0689852223	The House of the Scorpion	2002-01-01	Hardcover	16.95
0525423656	Crossed (Matched Trilogy, Book 2)	2011-11-01	Hardcover	12.95
1423153627	The Kane Chronicles Survival Guide	2012-03-01	Hardcover	13.95
0439371112	Howliday Inn	2001-11-01	Paperback	14.95
0439861306	The Lightning Thief	2006-03-12	Paperback	11.95
031673737X	How to Train Your Dragon	2010-02-01	Hardcover	10.95
0545078059	The White Giraffe	2008-05-01	Paperback	6.95
0803733428	The Last Leopard	2009-03-05	Hardcover	13.95
9780545236	Freaky Monday	2010-01-15	Paperback	12.95

TABLE 12-5 Bob’s Books Author Table Sample Data for the “Authors” Table

AuthorID	FirstName	LastName
1000	Rick	Riordan
1001	Nancy	Farmer
1002	Ally	Condie
1003	Cressida	Cowell
1004	Lauren	St. John
1005	Eoin	Colfer
1006	Esther	Freisner
1007	Chris	D'lacey
1008	Mary	Rodgers
1009	Heather	Hatch

To tie Authors to Books and Books to Authors, Bob has created a third table called Books_by_Author. This is a unique table type in a relational database. This table is called a *join* table. In a join table, there are no primary keys—instead, all the columns represent data that can be used by other tables to create a relationship. These columns are referred to as foreign keys—they represent a primary key in another table. Looking at the last two rows of this table, you can see that the Book with the ISBN 9780545236 has two authors: author id 1008 (Mary Rodgers) and 1009 (Heather Hatch). Using this join table, we can combine the two sets of data without needing duplicate entries in either table. We'll return to the concept of a join table later in the chapter.

TABLE 12-6 Bob's Books Books by Author Sample Data

AuthorID	ISBN
1000	142311339X
1001	0689852223
1002	0525423656
1000	1423153627
1003	031673737X
1004	0545078059
1004	0803733428
1008	9780545236
1009	9780545236

A complete Bob's Books database schema would include tables for publishers, addresses, stock, purchase orders, and other data that the store needs to run its business. But for our purposes, this part of the schema is sufficient. Using this schema, we can write SQL queries using the SQL CRUD commands you learned earlier.

To summarize, before looking at JDBC, you should now know about connections, statements, and result sets:

- A connection is how an application communicates with a database.
- A statement is a SQL query that is executed on the database.

- A result set is the data that is returned from a SELECT statement.

Having these concepts down, we can use Bob's Books simple schema to frame some common uses of the JDBC API to submit SQL queries and get results in a Java application.

CERTIFICATION OBJECTIVE

Core Interfaces of the JDBC API (OCP Objective 11.1)

11.1 Describe the interfaces that make up the core of the JDBC API including the Driver, Connection, Statement, and ResultSet interfaces and their relationship to provider implementations.

As we mentioned in the previous section, the purpose of a relational database is really threefold:

- To provide storage for data in tables
- To provide a way to create relationships between the data—just as Bob did with the Authors, Books, and Books_by_Author tables
- To provide a language that can be used to get the data out, update the data, remove the data, and create new data

The purpose of JDBC is to provide an application programming interface (API) for Java developers to write Java applications that can access and manipulate relational databases and use SQL to perform CRUD operations.

Once you understand the basics of the JDBC API, you will be able to access a huge list of databases. One of the driving forces behind JDBC was to provide a standard way to access relational databases, but JDBC can also be used to access file systems and object-oriented data sources. The key is that the API provides an abstract view of a database connection, statements, and result sets. These concepts are represented in the API as interfaces in the `java.sql` package: `Connection`, `Statement`, and `ResultSet`, respectively. What these interfaces define are the *contracts* between you and the implementing class. In truth, you may not know (nor should you care) *how* the implementation class works. As long as the implementation class implements the interface you need, you are assured that the methods defined

by the interface exist and you can invoke them.

The `java.sql.Connection` interface defines the contract for an object that represents the connection with a relational database system. Later, we will look at the methods of this contract, but for now, an instance of a `Connection` is what we need to communicate with the database. How the `Connection` interface is implemented is vendor dependent, and again, we don't need to worry so much about the how—as long as the vendor follows the contract, we are assured that the object that represents a `Connection` will allow us to work with a database connection.

The `Statement` interface provides an abstraction of the functionality needed to get a SQL statement to execute on a database, and a `ResultSet` interface is an abstraction functionality needed to process a result set (the table of data) that is returned from the SQL query when the query involves a SQL SELECT statement.

The classes that implement `Connection`, `Statement`, `ResultSet`, and a number of other interfaces we will look at shortly are created by the vendor of the database we are using. The vendor understands their database product better than anyone else, so it makes sense that they create these classes. And it allows the vendor to optimize or hide any special characteristics of their product. The collection of the implementation classes is called the JDBC driver. A JDBC driver (lowercase "d") is the collection of classes required to support the API, whereas Driver (uppercase "D") is one of the implementations required in a driver.

A JDBC driver is typically provided by the vendor in a JAR or ZIP file. The implementation classes of the driver must meet a minimum set of requirements in order to be JDBC compliant. The JDBC specification provides a list of the functionality that a vendor must support and what functionality a vendor may optionally support.

Here is a partial list of the requirements for a JDBC driver. For more details, please read the specification (JSR-221). Note that the details of implementing a JDBC driver are NOT on the exam.

- Fully implement the interfaces: `java.sql.Driver`, `java.sql.DatabaseMetaData`, `java.sql.ResultSetMetaData`.
- Implement the `java.sql.Connection` interface. (Note that some methods are optional depending on the SQL version the database supports—more on SQL versions later in the chapter.)

- Implement `java.sql.Statement`, and `java.sql.PreparedStatement`.
- Implement the `java.sql.CallableStatement` interfaces if the database supports stored procedures. Again, more on this interface later in the chapter.
- Implement the `java.sql.ResultSet` interface.

CERTIFICATION OBJECTIVE

Connect to a Database Using DriverManager (OCP Objective 11.2)

11.2 Identify the components required to connect to a database using the DriverManager class including the JDBC URL

Not all of the types defined in the JDBC API are interfaces. One important class for JDBC is the `java.sql.DriverManager` class. This concrete class is used to interact with a JDBC driver and return instances of `Connection` objects to you. Conceptually, the way this works is by using a Factory design pattern. Next, we'll look at `DriverManager` in more detail.

Let's take this opportunity to see the factory design pattern in use. In a factory pattern, a concrete class with static methods is used to create instances of objects that implement an interface. For example, suppose we wanted to create an instance of a `Vehicle` object:

```
public interface Vehicle {
  public void start();  // Methods we think all vehicles should
  public void stop();   // support.
}
```

We need an implementation of `Vehicle` in order to use this contract. So

we design a `Car`:

```
package com.us.automobile;
public class Car implements Vehicle {
  public void start() { }  // ... do start things
  public void stop() { }   // ... do stop things
}
```

In order to use the `Car`, we could create one:

```
public class MyClass {
  public static void main(String args[]) {
    Vehicle ferrari =
      new com.us.automobile.Car(); // Create a Ferrari
    ferrari.start();               // Start the Ferrari
  }
}
```

However, here it would be better to use a factory—that way, we need not know anything about the actual implementation, and, as we will see later with `DriverManager`, we can use methods of the factory to dynamically determine which implementation to use at runtime.

```
public class MyClass {
  public static void main(String args[]) {
    Vehicle ferrari =
      CarFactory.getVehicle("Ferrari");       // Use a factory to
                                              // create a Ferrari
    ferrari.start();
  }
}
```

The factory in this case could create a different car based on the string passed to the static `getVehicle()` method—something like this:

```
public class CarFactory {
  public static Vehicle getVehicle(String type) {
    // ... create an instance of an object that represents the
    // type of car passed as the argument
  }
}
```

`DriverManager` uses this factory pattern to "construct" an instance of a `Connection` object by passing a string to its `getConnection()` method.

The DriverManager Class

The `DriverManager` class is a concrete, utility class in the JDBC API with static methods. You will recall that static or class methods can be invoked by other classes using the class name. One of those methods is `getConnection()`, which we look at next.

The `DriverManager` class is so named because it manages which JDBC driver implementation you get when you request an instance of a `Connection`

through the `getConnection()` method.

There are several overloaded `getConnection` methods, but they all share one common parameter: a `String` URL. One pattern for `getConnection` is

```
DriverManager.getConnection(String url, String username, String password);
```

For example:

```
String url
    = "jdbc:derby://localhost:1521/BookSellerDB";   // JDBC URL
String user = "bookguy";  // BookSellerDB user name
String pwd = "$3lleR";    // BookSellerDB password
try {
  Connection conn
    = DriverManager.getConnection(url, user, pwd);  // Get an
                                                    // instance of a
                                                    // Connection
                                                    // object
} catch (SQLException se) { }
```

In this example, we are creating a connection to a Derby database, on a network, at a localhost address (on the local machine), at port number 1521, to a database called `"BookSellerDB"`, and we are using the credentials `"bookguy"` as the user id, and `"$3lleR"` as the password. Don't worry too much about the syntax of the URL right now—we'll cover that soon.

It's a horrible idea to hard-code a username and password in the `getConnection()` method. Obviously, anyone reading the code would then

know the username and password to the database. A more secure way to handle database credentials would be to separate the code that produces the credentials from the code that makes the connection. In some other class, you would use some type of authentication and authorization code to produce a set of credentials to allow access to the database. For simplicity in the examples in the chapter, we'll hard-code the username and password, but just keep in mind that on the job, this is not a best practice.

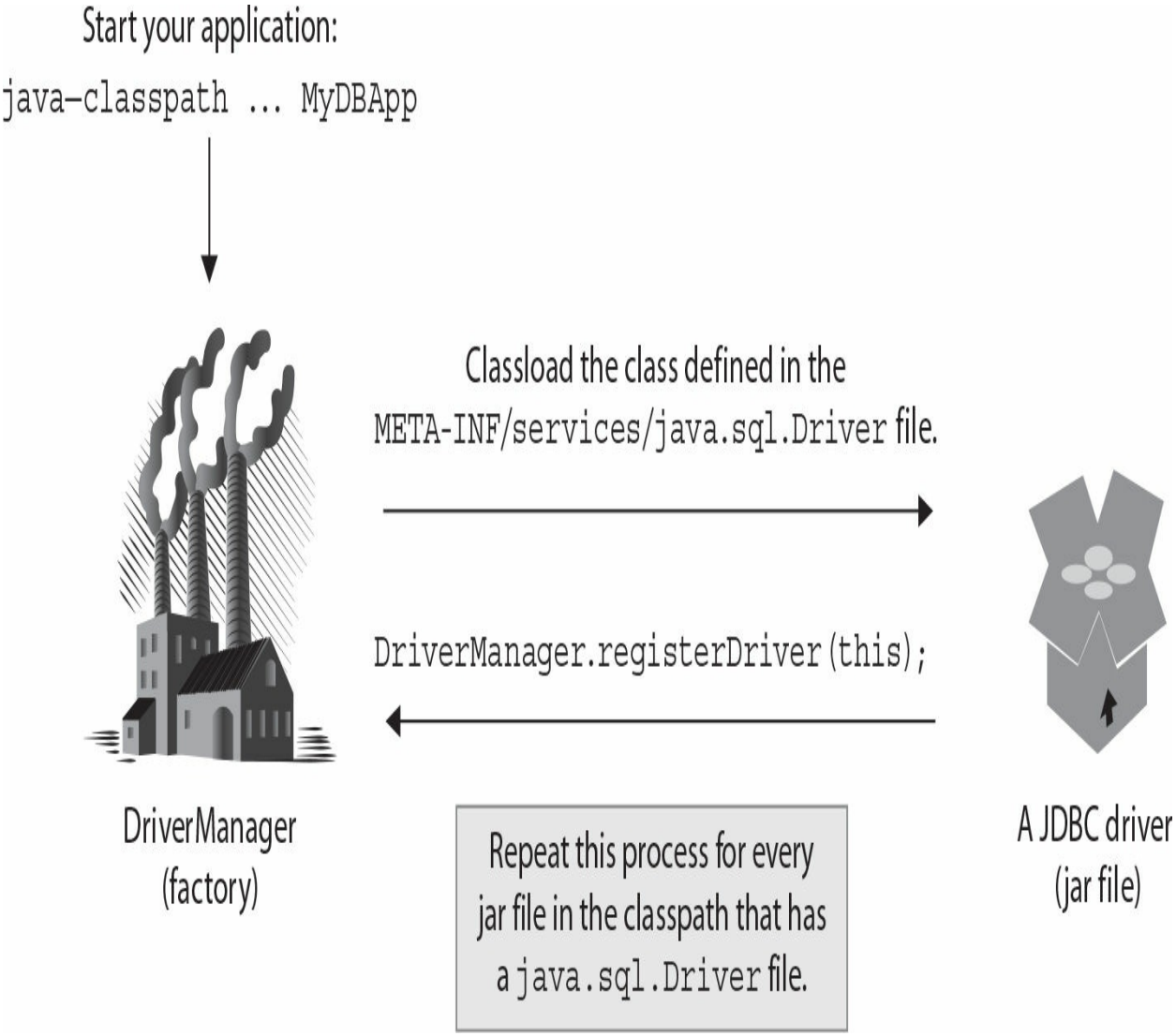
When you invoke the `DriverManager`'s `getConnection()` method, you are asking the `DriverManager` to try passing the first string in the statement, the driver URL, along with the username and password to each of the driver classes registered with the `DriverManager` in turn. If one of the driver classes recognizes the URL string, and the username and password are accepted, the driver returns an instance of a `Connection` object. If, however, the URL is incorrect, or the username and/or password are incorrect, then the method will throw a `SQLException`. We'll spend some time looking at `SQLException` later in this chapter.

How JDBC Drivers Register with the DriverManager

Because this part of the JDBC process is important to understand, and it involves a little Java magic, let's spend some time diagramming how driver classes become "registered" with the `DriverManager`, as shown in Figure 12-2.

FIGURE 12-2

How JDBC drivers self-register with `DriverManager`



First, one or more JDBC drivers, in a JAR or ZIP file, are included in the classpath of your application. The `DriverManager` class uses a service provider mechanism to search the classpath for any JAR or ZIP files that contain a file named `java.sql.Driver` in the META-INF/services folder of the driver jar or zip. This is simply a text file that contains the full name of the class that the vendor used to implement the `jdbc.sql.Driver` interface. For example, for a Derby driver, the full name is `org.apache.derby.jdbc.ClientDriver`.

The `DriverManager` will then attempt to load the class it found in the `java.sql.Driver` file using the class loader:

```
Class.forName("org.apache.derby.jdbc.ClientDriver");
```

When the driver class is loaded, its static initialization block is executed. Per the JDBC specification, one of the first activities of a driver instance is to "self-register" with the `DriverManager` class by invoking a static method on `DriverManager`. The code (minus error handling) looks something like this:

```
public class ClientDriver implements java.sql.Driver{
  static {
    ClientDriver driver = new ClientDriver();
    DriverManager.registerDriver(driver);
  }
  //...
}
```

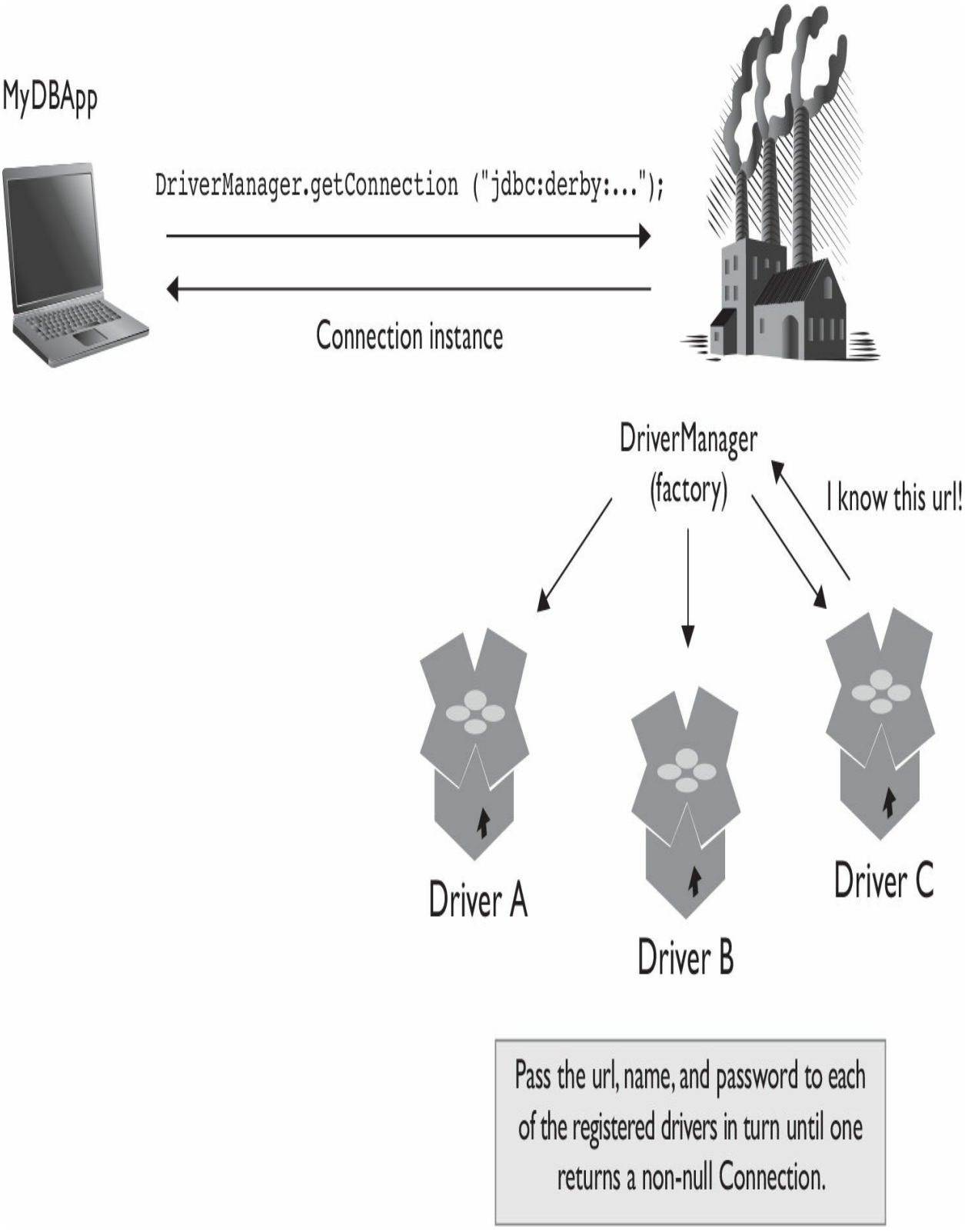
This registers (stores) an instance of the `Driver` class into the `DriverManager`.

Now, when your application invokes the `DriverManager.getConnection()` method and passes a JDBC URL, username, and password to the method, the `DriverManager` simply invokes the `connect()` method on the registered `Driver`. If the connection was successful, the method returns a `Connection` object instance to `DriverManager`, which, in turn, passes that back to you.

If there is more than one registered driver, the `DriverManager` calls each of the drivers in turn and attempts to get a `Connection` object from them, as shown in Figure 12-3.

FIGURE 12-3

How the `DriverManager` gets a `Connection`



The first driver that recognizes the JDBC URL and successfully creates a connection using the username and password will return an instance of a

`Connection` object. If no drivers recognize the URL, username, and password combination, or if there are no registered drivers, then a `SQLException` is thrown instead.

To summarize:

- The JVM loads the `DriverManager` class, a concrete class in the JDBC API.
- The `DriverManager` class loads any instances of classes it finds in the META-INF/services/java.sql.Driver file of JAR/ZIP files on the classpath.
- Driver classes call `DriverManager.register(this)` to self-register with the `DriverManager`.
- When the `DriverManager.getConnection(String url)` method is invoked, `DriverManager` invokes the `connect()` method of each of these registered `Driver` instances with the URL string.
- The first `Driver` that successfully creates a connection with the URL returns an instance of a `Connection` object to the `DriverManager.getConnection` method invocation.

Let's look at the JDBC URL syntax next.

The JDBC URL

The JDBC URL is what is used to determine which driver implementation to use for a given `Connection`. Think of the JDBC URL (uniform resource locator) as a way to narrow down the universe of possible drivers to one specific connection. For example, suppose you need to send a package to someone. In order to narrow the universe of possible addresses down to a single unique location, you would have to identify the country, the state, the city, the street, and perhaps a house or address number on your package:

```
USA:California://SanJose:FirstStreet/15
```

This string indicates that the address you want is in the United States, California State, San Jose city, First Street, number 15.

JDBC URLs follow this same idea. To access Bob's Books, we might write the URL like this:

```
jdbc:derby://localhost:1521/BookSellerDB
```

The first part, `jdbc`, simply identifies that this is a JDBC URL (versus HTTP or something else). The second part indicates that driver vendor is `derby` driver. The third part indicates that the database is on the `localhost` of this machine (IP address 127.0.0.1), at port `1521`, and the final part indicates that we are interested in the `BookSellerDB` database.

Just like street addresses, the reason we need this string is because JDBC was designed to work with multiple databases at once. Each of the JDBC database drivers will have a different URL, so we need to be able to pass the JDBC URL string to the `DriverManager` and ensure that the `Connection` returned was for the intended database instance.

Unfortunately, other than a requirement that the JDBC URL begin with "jdbc," there is very little standard about a JDBC URL. Vendors may modify the URL to define characteristics for a particular driver implementation. The format of the JDBC URL is

```
jdbc:<subprotocol>:<subname>
```

In general, the subprotocol is the vendor name; for example:

```
jdbc:derby
jdbc:mysql
jdbc:oracle
```

There are two ways to establish a connection in JDBC. The first way is using one of the few concrete classes in the `java.sql` package, `DriverManager`. The `java.sql.DriverManager` class has been a part of the JDBC implementation since the beginning, and is the easiest way to obtain a connection from a Java SE application. The alternative way is with an instance of a class that implements `javax.sql.DataSource`, introduced in JDBC 2.0.

Since a `DataSource` instance is typically obtained through a Java

Naming and Directory Interface (JNDI) lookup, it is more often used in Java applications where there is a container that supports JNDI—for example, a Java EE application server. For the purposes of this chapter (and because `DataSource` is not on the exam), we'll focus on using `DriverManager` to obtain a connection, but in the end, both ways serve to give you an instance of a `Connection` object.

To summarize, `DriverManager` is on the exam and `DataSource` is not.

The subname field is where things get a bit more vendor specific. Some vendors use the subname to identify the hostname and port, followed by a database name. For example:

```
jdbc:derby://localhost:1521/MyDB
jdbc:mysql://localhost:3306/MyDB
```

Other vendors may use the subname to identify additional context information about the driver. For example:

```
jdbc:oracle:thin:@//localhost:1527/MyDB
```

In any case, it is best to consult the documentation for your specific database vendor's JDBC driver to determine the syntax of the URL.

JDBC Driver Implementation Versions

We talked about how the `DriverManager` will scan the classpath for JAR files that contain the `META-INF/services/java.sql.Driver` file and use a classloader to load those drivers. This feature was introduced in the JDBC 4.0 specification. Prior to that, JDBC drivers were loaded manually by the application.

If you are using a JDBC driver that is an earlier version, say, a JDBC 3.0 driver, then you must explicitly load the class provided by the database vendor that implements the `java.sql.Driver` interface. Typically, the database vendor's documentation would tell you what the driver class is. For example, if our Apache Derby JDBC driver were a 3.0 driver, you would

manually load the `Driver` implementation class before calling the `getConnection()` method:

```
Class.forName("org.apache.derby.jdbc.ClientDriver");  // Class loads
                                                      // ClientDriver
try {
  Connection conn
    = DriverManager.getConnection(url, user, pwd);
```

Note that using the `Class.forName()` method is compatible with both JDBC 3.0 and JDBC 4.0 drivers. It is simply not needed when the driver supports 4.0.

Here is a quick summary of what we have discussed so far:

- Before you can start working with JDBC, creating queries and getting results, you must first establish a connection.
- In order to establish a connection, you must have a JDBC driver.
- If your JDBC driver is a JDBC 3.0 driver, then you are required to explicitly load the driver in your code using `Class.forName()` and the fully qualified path of the `Driver` implementation class.
- If your JDBC driver is a JDBC 4.0 driver, then simply include the driver (jar or zip) in the classpath.

Although the certification exam covers up through Java SE 8, the exam developers felt they ought to include some questions about obtaining a connection using both JDBC 3.0 and JDBC 4.0 drivers. So keep in mind that for JDBC 3.0 drivers (and earlier), you are responsible for loading the class using the static `forName()` method from `java.lang.Class`.

CERTIFICATION OBJECTIVE

Submit Queries and Read Results from the Database (OCP Objective 11.3)

11.3 Submit queries and read results from the database including creating statements, returning result sets, iterating through the results, and properly closing result sets, statements, and connections.

In this section, we'll explore the JDBC API in much greater detail. We will start by looking at a simple example using the `Connection`, `Statement`, and `ResultSet` interfaces to pull together what we've learned so far in this chapter. Then we'll do a deep dive into `Statements` and `ResultSets`.

All of Bob's Customers

Probably one of the most used SQL queries is SELECT * FROM <Table name>, which is used to print out or see all of the records in a table. Assume that we have a Java DB (Derby) database populated with data from Bob's Books. To query the database and return all of the Customers in the database, we would write something like the example shown next.

Note that to make the code listing a little shorter, going forward, we will use `out.println` instead of `System.out.println`. Just assume that means we have included a static import statement, like the one at the top of this example:

```
import static java.lang.System.*;  // Static import of the
                                   // System class methods.
                                   // Now we can use just 'out'
                                   // instead of System.out.
String url = "jdbc:derby://localhost:1521/BookSellerDB";
String user = "bookguy";
String pwd = "$3lleR";
try {
  Connection conn =
    DriverManager.getConnection(url, user, pwd); // Get Connection
  Statement stmt = conn.createStatement();       // Create Statement
  String query = "SELECT * FROM Customer";
  ResultSet rs = stmt.executeQuery(query);       // Execute Query
  while (rs.next()) {                            // Process Results
    out.print(rs.getInt("CustomerID") + "  ");   // Print Columns
    out.print(rs.getString("FirstName") + "  ");
    out.print(rs.getString("LastName") + "  ");
    out.print(rs.getString("EMail") + "  ");
    out.println(rs.getString("Phone"));
  }
} catch (SQLException se) { }                    // Catch SQLException
```

Again, we'll dive into all of the parts of this example in greater detail, but here is what is happening:

- **Get connection** We are creating a `Connection` object instance using the information we need to access Bob's Books Database (stored on a

Java DB Relational database, `BookSellerDB`, and accessed via the credentials `"bookguy"` with a password of `"$3lleR"`).

- **Create statement** We are using the `Connection` to create a `Statement` object. The `Statement` object handles passing `Strings` to the database as queries for the database to execute.
- **Execute query** We are executing the query string on the database and returning a `ResultSet` object.
- **Process results** We are iterating through the result set rows—each call to `next()` moves us to the next row of results.
- **Print columns** We are getting the values of the columns in the current result set row and printing them to standard out.
- **Catch `SQLException`** All of the JDBC API method invocations throw `SQLException`. A `SQLException` can be thrown when a method is used improperly or if the database is no longer responding. For example, a `SQLException` is thrown if the JDBC URL, username, or password is invalid. Or we attempted to query a table that does not exist. Or the database is no longer reachable because the network went down or the database went offline. We will look at `SQLException` in greater detail later in the chapter.

The output of the previous code will look something like this:

```
5000  John  Smith  John.Smith@comcast.net  555-340-1230
5001  Mary  Johnson  mary.johnson@comcast.net  555-123-4567
5002  Bob  Collins  bob.collins@yahoo.com  555-012-3456
5003  Rebecca  Mayer  rebecca.mayer@gmail.com  555-205-8212
5006  Anthony  Clark  anthony.clark@gmail.com  555-256-1901
5007  Judy  Sousa  judy.sousa@verizon.net  555-751-1207
5008  Christopher  Patriquin  patriquinc@yahoo.com  555-316-1803
5009  Deborah  Smith  debsmith@comcast.net  555-256-3421
5010  Jennifer McGinn  jmcginn@comcast.net  555-250-0918
```

We'll take a detailed look at the `Statement` and `ResultSet` interfaces and methods in the next two sections.

Statements

Once we have successfully connected to a database, the fun can really start. From a `Connection` object, we can create an instance of a `Statement` object (or, to be precise, using the `Connection` instance we received from the `DriverManager`, we can get an instance of an object that implements the `Statement` interface). For example:

```
String url = "jdbc:derby://localhost:1521/BookSellerDB";
String user = "bookguy";
String pwd = "$3lleR";
try {
  Connection conn = DriverManager.getConnection(url, user, pwd);
  Statement stmt = conn.createStatement();
  // do stuff with SQL statements
} catch (SQLException se) { }
```

The primary purpose of a `Statement` is to execute a SQL statement using a method and return some type of result. There are several forms of `Statement` methods: those that return a result set, and those that return an integer status. The most commonly used `Statement` method performs a SQL query that returns some data, like the SELECT call we used earlier to fetch all the `Customer` table rows.

Constructing and Using Statements

To start, let's look at the base `Statement`, which is used to execute a static SQL query and return a result. You'll recall that we get a `Statement` from a `Connection` and then use the `Statement` object to execute a SQL statement, like a query on the database. For example:

```
Connection conn = DriverManager.getConnection(url, user, pwd);
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery("SELECT * FROM Customer");
```

Because not all SQL statements return results, the `Statement` object provides several different methods to execute SQL commands. Some SQL commands do not return a result set, but instead return an integer status. For example, SQL INSERT, UPDATE, and DELETE commands, or any of the SQL Data Definition Language (DDL) statements like CREATE TABLE, return either the number of rows affected by the query or 0.

Let's look at each of the execute methods in detail.

public ResultSet executeQuery(String sql) throws SQLException This is the most commonly executed `Statement` method. This method is used when we know that we want to return results—we are querying the database for one or more rows of data. For example:

```
ResultSet rs = stmt.executeQuery("SELECT * from Customer");
```

Assuming there is data in the `Customer` table, this statement should return all of the rows from the `Customer` table into a `ResultSet` object—we'll look at `ResultSet` in the next section. Notice that the method declaration includes "throws `SQLException`." This means that this method must be called in a `try-catch` block or must be called in a method that also throws `SQLException`. Again, one reason that these methods all throw `SQLException` is that a connection to the database is likely **to a database on a network**. As with all things on the network, availability is not guaranteed, so one possible reason for `SQLException` is the lack of availability of the database itself.

public int executeUpdate(String sql) throws SQLException This method is used for a SQL operation that affects one or more rows and does not return results—for example, SQL INSERT, UPDATE, DELETE, and DDL queries. These statements do not return results, but do return a count of the number of rows affected by the SQL query. For example, here is an example method invocation where we want to update the Book table, increasing the price of every book that is currently priced less than 8.95 and is a hardcover book:

```
String q = "UPDATE Book SET UnitPrice=8.95
           WHERE UnitPrice < 8.95 AND Format='Hardcover'";
int numRows = stmt.executeUpdate(q);
```

When this query executes, we are expecting some number of rows will be affected. The integer that returns is the number of rows that were updated.

Note that this `Statement` method can also be used to execute SQL queries that do not return a row count, such as CREATE TABLE or DROP TABLE and other DDL queries. For DDL queries, the return value is 0.

public boolean execute(String sql) throws SQLException This method is used when you are not sure what the result will be—perhaps the query will return a result set and perhaps not. This method can be used to execute a query whose type may not be known until runtime—for example, one constructed in code. The return value is true if the query resulted in a result set and false if the query resulted in an update count or no results.

However, more often, this method is used when invoking a stored procedure (using the `CallableStatement`, which we'll talk about later in the chapter). A stored procedure can return a single result set or row count, or multiple result sets and row counts, so this method was designed to handle what happens when a single database invocation produces more than one result set or row count.

You might also use this method if you wrote an application to test queries —something that reads a `String` from the command line and then runs that `String` against the database as a query. For example:

```
ResultSet rs;
int numRows;
boolean status = stmt.execute("");  // True if there is a ResultSet
if (status) {                       // True
  rs = stmt.getResultSet();         // Get the ResultSet
  // Process the result set...
} else {                            // False
  numRows = stmt.getUpdateCount();  // Get the update count
  if (numRows == -1) {              // If -1, there are no results
    out.println("No results");
  } else {                           // else, print the number of
                                     // rows affected
    out.println(numRows + " rows affected.");
  }
}
```

Because this statement may return a result set or may simply return an integer row count, there are two additional statement commands you can use to get the results or the count based on whether the `execute()` method returned true (there is a result set) or false (there is an update count or there was no result). The `getResultSet()` is used to retrieve results when the `execute()` method returns true, and the `getUpdateCount()` is used to retrieve the count when the `execute()` method returns false. Let's look at these methods next.

It is generally a very bad idea to allow a user to enter a query string directly in an input field or allow a user to pass a string to construct a query directly. The reason is that if a user can construct a query or even include a freeform string into a query, he or she can use the query to return more data than you intended or alter the database table permissions.

For example, assume that we have a query where the user enters his e-mail address and the string the user enters is inserted directly to the query:

```
String s = System.console().readLine("Enter your e-mail address: ");
ResultSet rs = stmt.executeQuery("SELECT * FROM Customer
                                  WHERE EMail='" + s + "'");
```

The user of this code could enter a string like this:

```
tom@trouble.com' OR 'x'='x
```

The resulting query executed by the database becomes:

```
SELECT * FROM Customer WHERE Email='tom@trouble.com' OR 'x'='x'
```

Because the OR statement will always return `true`, the result is that the query will return ALL of the customer rows, effectively the same as the query:

```
SELECT * FROM Customer
```

And now this user of your code has a list of the e-mail addresses of every customer in the database.

This type of attack is called a SQL injection attack. It is easy to prevent by carefully sanitizing any string input used in a query to the database and/or by using one of the other `Statement` types: `PreparedStatement` and `CallableStatement`. Despite how easy it is to prevent, it happens frequently, even to large, experienced companies like Yahoo!

public ResultSet getResultSet() throws SQLException If the boolean value from the `execute()` method returns true, then there is a result set. To get the result set, as shown earlier, call the `getResultSet()` method on the `Statement` object. Then you can process the `ResultSet` object (which we will

cover in the next section). This method is basically foolproof—if, in fact, there are no results, the method will return a null.

```
ResultSet rs = stmt.getResultSet();
```

public int getUpdateCount() throws SQLException If the boolean value from the `execute()` method returns false, then there is a row count, and this method will return the number of rows affected. A return value of –1 indicates that there are no results.

```
int numRows = stmt.getUpdateCount();
if (numRows == -1) {
  out.println("No results");
} else {
  out.println(numRows + " rows affected.");
}
```

Table 12-7 summarizes the `Statement` methods we just covered.

TABLE 12-7 Important `Statement` Methods

Method (Each Throws SQLException)	Description
`ResultSet executeQuery(String sql)`	Execute a SQL query and return a `ResultSet` object, i.e., SELECT commands.
`int executeUpdate(String sql)`	Execute a SQL query that will only modify a number of rows, i.e. INSERT, DELETE, or UPDATE commands.
`boolean execute(String sql)`	Execute a SQL query that may return a result set OR modify a number of rows (or do neither). The method will return true if there is a result set or false if there may be a row count of affected rows.
`ResultSet getResultSet()`	If the return value from the `execute()` method was true, you can use this method to retrieve the result set from the query.
`int getUpdateCount()`	If the return value from the `execute()` method was false, you can use this method to get the number of rows affected by the SQL command.

ResultSets

When a query returns a result set, an instance of a class that implements the ResultSet interface is returned. The `ResultSet` object represents the results of the query—all of the data in each row on a per-column basis. Again, as a reminder, *how* data in a `ResultSet` are stored is entirely up to the JDBC driver vendor. It is possible that the JDBC driver caches the entire set of

results in memory all at once, or that it uses internal buffers and gets only a few rows at a time. From your point of view as the user of the data, it really doesn't matter much. Using the methods defined in the `ResultSet` interface, you can read and manipulate the data and that's all that matters.

One important thing to keep in mind is that a `ResultSet` is a *copy* of the data from the database from the instance in time when the query was executed. Unless you are the only person using the database, you need to always assume that the underlying database table or tables that the `ResultSet` came from could be changed by some other user or application.

Because `ResultSet` is such a comprehensive part of the JDBC API, we are going to tackle it in sections. Table 12-8 summarizes each section so you can reference these later.

TABLE 12-8 `ResultSet` Sections

Section Title	Description
"Moving Forward in a ResultSet"	How to access each "row" of the result of a query.
"Reading Data from a ResultSet"	How to use `ResultSet` methods to access the individual columns of each "row" in the result set.
"Getting Information about a ResultSet"	How to use a `ResultSetMetaData` object to retrieve information about the result set: the number of columns returned in the results, the names of each column, and the Java type of each column.
"Printing a Report"	How to use the `ResultSetMetaData` methods to print a nicely formatted set of results to the console.
"Moving Around in ResultSets"	How to change the cursor type and concurrency settings on a `Statement` object to create a `ResultSet` that allows the row cursor to be positioned and allows the data to be modified.
"Updating ResultSets"	How to use the concurrency settings on a `Statement` object to create a `ResultSet` that allows you to update the results returned and later synchronize those results with the database.
"Inserting New Rows into a ResultSet"	How to manipulate a `ResultSet` further by deleting and inserting rows.
"Getting Information about a Database Using DatabaseMetaData"	How to use the `DatabaseMetaData` object to retrieve information about a database.

Moving Forward in a ResultSet

The best way to think of a `ResultSet` object is visually. Assume that in our BookSellerDB database we have several customers whose last name begins with the letter "C." We could create a query to return those rows "like" this:

```
String query = "SELECT FirstName, LastName, EMail from Customer
                WHERE LastName LIKE 'C%'";
```

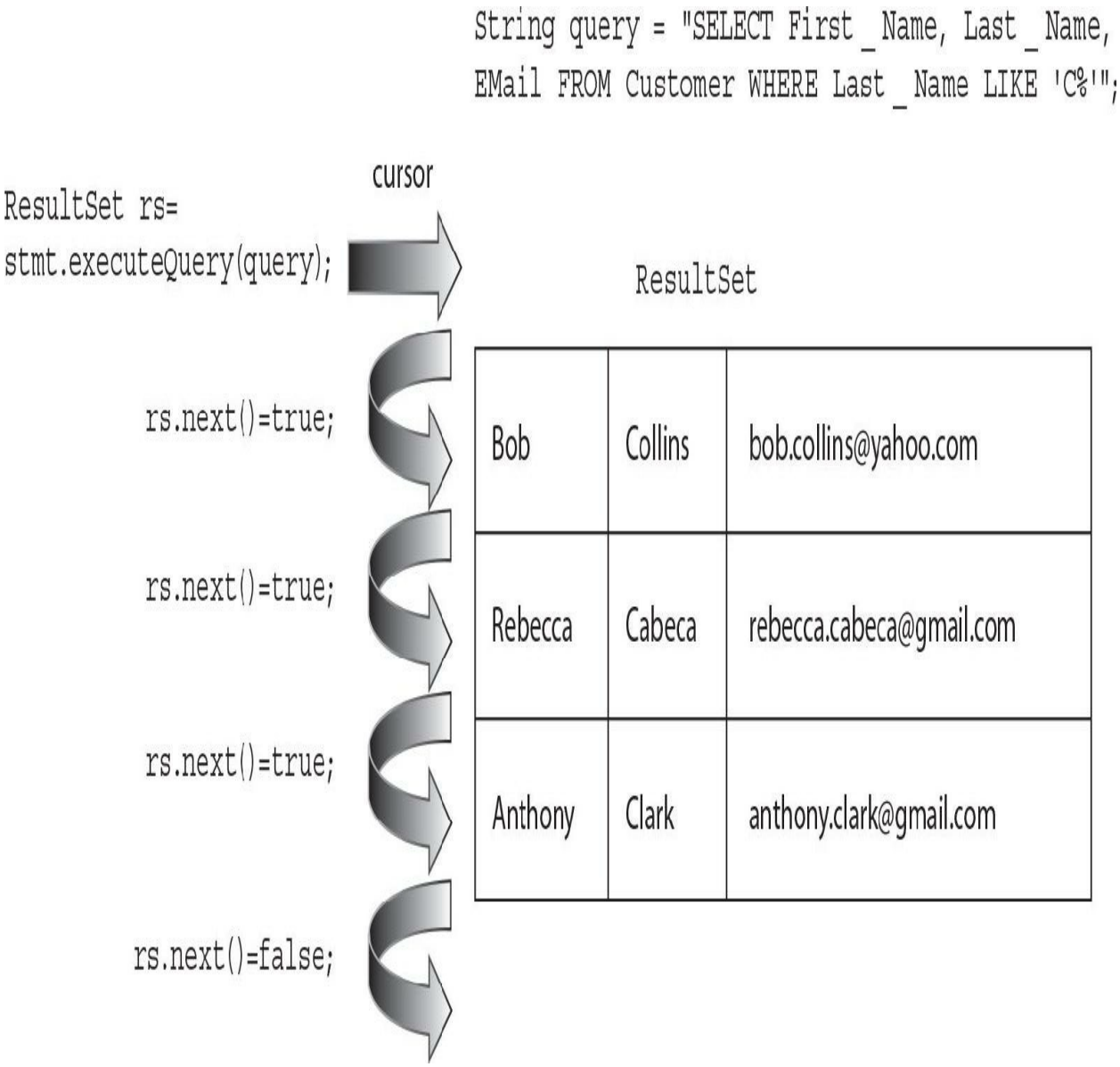
The SQL operator LIKE treats the string that follows as a pattern to match, where the `%` indicates a wildcard. So, `LastName LIKE 'C%'` means "any `LastName` with a `C`, followed by any other character(s)."

When we execute this query using the `executeQuery()` method, the `ResultSet` returned will contain the `FirstName`, `LastName`, and `EMail` columns where the customer's `LastName` starts with the capital letter "C":

```
ResultSet rs = stmt.executeQuery (query);
```

The `ResultSet` object returned contains the data from the query as shown in Figure 12-4.

FIGURE 12-4 A `ResultSet` after the `executeQuery`



Note in Figure 12-4 that the `ResultSet` object maintains a cursor, or a pointer, to the current row of the results. When the `ResultSet` object is first returned from the query, the cursor is not yet pointing to a row of results—the cursor is pointing above the first row. In order to get the results of the table, you must always call the `next()` method on the `ResultSet` object to move the cursor forward to the first row of data. By default, a `ResultSet` object is read-only (the data in the rows cannot be updated), and you can only move the cursor forward. We'll look at how to change this behavior a little later on.

So the first method you will need to know for `ResultSet` is the `next()` method.

public boolean next() The `next()` method moves the cursor forward one row and returns true if the cursor now points to a row of data in the `ResultSet`. If the cursor points beyond the last row of data as a result of the `next()` method (or if the `ResultSet` contains no rows), the return value is false.

So in order to read the three rows of data in the table shown in Figure 12-4, we need to call the `next()` method, read the row of data, and then call `next()`again twice more. When the `next()`method is invoked the fourth time, the method will return false. The easiest way to read all of the rows from first to last is in a `while` loop:

```
String query = "SELECT FirstName, LastName, EMail FROM Customer
               WHERE LastName LIKE 'C%'";
ResultSet rs = stmt.executeQuery(query);
while (rs.next()) {  // Move the cursor from the current position
                     // to the next row of data - return true if the
                     // next row is valid data and false if the
                     // cursor has moved past the last row
  // ...
}
```

Because the cursor is such a fundamental concept in JDBC, the exam will test you on the status of the cursor in a `ResultSet`. As long as you keep in mind that you must call the `next()` method before processing even one row of data in a `ResultSet`, then you'll be fine. Maybe you could use a memory device like this one: "When getting results, don't vex, always call next!" Okay, maybe not.

Reading Data from a ResultSet

Moving the cursor forward through the `ResultSet` is just the start of reading data from the results of the query. Let's look at the two ways to get the data from each row in a result set.

When a `ResultSet` is returned and you have dutifully called `next()` to move the cursor to the first actual row of data, you can now read the data in each column of the current row. As illustrated in Figure 12-4, a result set from a database query is like a table or a spreadsheet. Each row contains (typically) one or more columns, and the data in each column is one of the SQL data types. In order to bring the data from each column into your Java application, you must use a `ResultSet` method to retrieve each of the SQL column values into an appropriate Java type. So SQL INTEGER, for example, can be read as a Java int primitive, SQL VARCHAR can be read as a Java String, SQL DATE can be read as a `java.sql.Date` object, and so on. `ResultSet` defines several other types as well, but whether or not the database or the driver supports all of the types defined by the specification depends on the database vendor. For the exam, we recommend you focus on the most common SQL data types and the `ResultSet` methods shown in Table 12-9.

TABLE 12-9 SQL Types and JDBC Types

SQL Type	Java Type	ResultSet get methods
BOOLEAN	boolean	getBoolean(String columnName) getBoolean(int columnIndex)
INTEGER	int	getInt(String columnName) getInt(int columnIndex)
DOUBLE, FLOAT	double	getDouble(String columnName) getDouble(int columnIndex)
REAL	float	getFloat(String columnName) getFloat(int columnIndex)
BIGINT	long	getLong(String columnName) getLong(int columnIndex)
CHAR, VARCHAR, LONGVARCHAR	String	getString(String columnName) getString(int columnIndex)
DATE	java.sql.Date	getDate(String columnName) getDate(int columnIndex)
TIME	java.sql.Time	getTime(String columnName) getTime(int columnIndex)
TIMESTAMP	java.sql.Timestamp	getTimestamp(String columnName) getTimestamp(int columnIndex)
Any of the above	java.lang.Object	getObject(String columnName) getObject(int columnIndex)

SQL has been around for a long time. The first formalized American National Standards Institute (ANSI)–approved version was adopted in 1986 (SQL-86). The next major revision was in 1992, SQL-92, which is widely considered the "base" release for every database. SQL-92 defined a number of new data types, including DATE, TIME, TIMESTAMP, BIT, and VARCHAR strings. SQL-92 has multiple levels; each level adds a bit more functionality to the previous level. JDBC drivers recognize three ANSI SQL-92 levels: Entry, Intermediate, and Full.

SQL-1999, also known as SQL-3, added LARGE OBJECT types, including BINARY LARGE OBJECT (BLOB) and CHARACTER LARGE OBJECT (CLOB). SQL-1999 also introduced the BOOLEAN type and a composite type, ARRAY and ROW, to store collections directly into the database. In addition, SQL-1999 added a number of features to SQL, including triggers, regular expressions, and procedural and flow control.

SQL-2003 introduced XML to the database, and importantly, added columns with auto-generated values, including columns that support identity, like the primary key and foreign key columns. Believe it or not, other standards have been proposed, including SQL-2006, SQL-2008, and SQL-2011.

The reason this matters is because the JDBC specification has attempted to be consistent with features from the most widely adopted specification at the time. Thus, JDBC 3.0 supports SQL-92 and a part of the SQL-1999 specification, and JDBC 4.0 supports parts of the SQL-2003 specification. In this chapter, we'll try to stick to the most widely used SQL-92 features and the most commonly supported SQL-1999 features that JDBC also supports.

One way to read the column data is by using the names of the columns themselves as string values. For example, using the column names from Bob's Book table (Table 12-4), in these `ResultSet` methods, the `String` name of the column from the Book table is passed to the method to read the column data type:

```
String query = "SELECT Title, PubDate, Price FROM Book";
ResultSet rs = stmt.executeQuery(query);
while (rs.next()) {
    String title = rs.getString("Title");     // Read the data in the
                                              // column named "Title"
                                              // into a String
    Date PubDate = rs.getDate("PubDate");     // Read the data in the
                                              // "PubDate" column into
                                              // a java.sql.Date object
    float price = rs.getFloat("Price");       // Read the data in the
                                              // column "Price"
                                              // into a float
    // ....
}
```

Note that although here the column names were retrieved from the `ResultSet` row in the order they were requested in the SQL query, they could have been processed in any order.

`ResultSet` also provides an overloaded method that takes an integer index value for each of the SQL types. This value is the integer position of the column in the result set, numbered from 1 to the number of columns returned. So we could write the same statements earlier like this:

```
String title = rs.getString(1); // Title is first column
Date PubDate = rs.getDate(2);   // PubDate is second column
float price = rs.getFloat(3);   // Price is third column
```

Using the positional methods shown earlier, the order of the column in the `ResultSet` does matter. In our query, Title is in position 1, PubDate is in position 2, and Price is in position 3.

Remember: Column indexes start with 1.

It is important to keep in mind that when you are accessing columns using integer index values, the column indexes always start with 1, not 0 as in traditional arrays. If you attempt to access a column with an index of less than 1 or greater than the number of columns returned, a `SQLException` will be thrown. You can get the number of columns returned in a `ResultSet` through the result set's metadata object. See the section on `ResultSetMetaData` to learn more.

What the database stores as a type, the SQL type, and what JDBC returns as a type are often two different things. It is important to understand that the JDBC specification provides a set of standard mappings—the best match between what the database provides as a type and the Java type a programmer should use with that type. Rather than repeating what is in the specification, we encourage you to look at Appendix B of the JDBC (JSR-221) specification.

The most commonly used `ResultSet get` methods are listed next. Let's look at these methods in detail.

public boolean getBoolean(String columnLabel) This method retrieves the value of the named column in the `ResultSet` as a Java boolean. Boolean values are rarely returned in SQL queries, and some databases may not support a SQL BOOLEAN type, so check with your database vendor. In this contrived example here, we are returning employment status:

```
if (rs.getBoolean("CURR_EMPLOYEE")) {
    // Now process the remaining columns
}
```

public double getDouble(String columnLabel) This method retrieves the value of the column as a Java double. This method is recommended for returning the value stored in the database as SQL DOUBLE and SQL FLOAT types.

```
double cartTotal = rs.getDouble("CartTotal");
```

public int getInt(String columnLabel) This method retrieves the value of the column as a Java int. Integers are often a good choice for primary keys. This method is recommended for returning values stored in the database as SQL INTEGER types.

```
int authorID = rs.getInt("AuthorID");
```

public float getFloat(String columnLabel) This method retrieves the value of the column as a Java float. It is recommended for SQL REAL types.

```
float price = rs.getFloat("UnitPrice");
```

public long getLong(String columnLabel) This method retrieves the value of the column as a Java long. It is recommended for SQL BIGINT types.

```
long socialSecurityNumber = rs.get("SocSecNum");
```

public java.sql.Date getDate(String columnLabel) This method retrieves the value of the column as a Java `Date` object. Note that `java.sql.Date` extends `java.util.Date`. One difference between the two is that the `toString()` method of `java.sql.Date` returns a date string in the form: "yyyy mm dd." This method is recommended for SQL DATE types.

```
java.sql.Date pubDate = rs.getDate("PubDate");
```

public java.lang.String getString(String columnLabel) This method

retrieves the value of the column as a Java `String` object. It is good for reading SQL columns with CHAR, VARCHAR, and LONGVARCHAR types.

```
String lastName = rs.getString("LastName");
```

public java.sql.Time getTime(String columnLabel) This method retrieves the value of the column as a Java `Time` object. Like `java.sql.Date`, this class extends `java.util.Date`, and its `toString()` method returns a time string in the form: "hh:mm:ss." TIME is the SQL type that this method is designed to read.

```
java.sql.Time time = rs.getTime("FinishTime");
```

public java.sql.Timestamp getTimestamp(String columnLabel) This method retrieves the value of the column as a `Timestamp` object. Its `toString()` method formats the result in the form: yyyy-mm-dd hh:mm:ss.fffffffff, where fffffffff is nanoseconds. This method is recommended for reading SQL TIMESTAMP types.

```
java.sql.Timestamp timestamp = rs.getTimestamp("ClockInTime");
```

public java.lang.Object getObject(String columnLabel) This method retrieves the value of the column as a Java `Object. It can be used as a general-purpose method for reading data in a column. This method works by reading the value returned as the appropriate Java wrapper class for the type and returning that as a Java Object object. So, for example, reading an integer (SQL INTEGER type) using this method returns an object that is a java.lang.Integer type. We can use instanceof to check for an Integer` and get the `int` value:

```
Object o = rs.getObject("AuthorID");
if (o instanceof java.lang.Integer) {
    int id = ((Integer)o).intValue();
}
```

Table 12-9 lists the most commonly used methods to retrieve specific data

from a `ResultSet`. For the complete and exhaustive set of `ResultSet` get methods, see the Java documentation for `java.sql.ResultSet`.

The exam is not going to test your knowledge of all of the `ResultSet` `get` and `set` methods for SQL types. For the exam, just remember the basic Java types: `String` and `int`. Each `ResultSet` getter method is named by its closest Java type, so, for example, to read a database column that holds an integer into a Java `int` type, you invoke the `getInt()` method with either the `String` column or the column index of the column you wish to read.

Getting Information about a ResultSet

When you write a query using a string, as we have in the examples so far, you know the name and type of the columns returned. However, what happens when you want to allow your users to dynamically construct the query? You may not always know in advance how many columns are returned and the type and name of the columns returned.

Fortunately, the `ResultSetMetaData` class was designed to provide just that information. Using `ResultSetMetaData`, you can get important information about the results returned from the query, including the number of columns, the table name, the column name, and the column class name—the Java class that is used to represent this column when the column is returned as an `Object`. Here is a simple example, and then we'll look at these methods in more detail:

```
String query = "SELECT AuthorID FROM Author";
ResultSet rs = stmt.executeQuery(query);
ResultSetMetaData rsmd = rs.getMetaData();
rs.next();
int colCount = rsmd.getColumnCount();  // How many columns in this
                                       // ResultSet?
out.println("Column Count: " + colCount);
for (int i = 1; i <= colCount; i++) {
  out.println("Table Name:   " + rsmd.getTableName(i));
  out.println("Column Name:  " + rsmd.getColumnName(i));
  out.println("Column Size:  " + rsmd.getColumnDisplaySize(i));
}
```

Running this code using the BookSeller database (Bob's Books) produces the following output:

```
Column Count: 1
Table Name:   AUTHOR
Column Name:  AUTHORID
Column Size:  11
```

`ResultSetMetaData` is often used to generate reports, so here are the most commonly used methods. For more information and more methods, check out the JavaDocs.

public int getColumnCount() throws SQLException This method is probably the most used `ResultSetMetaData` method. It returns the integer count of the number of columns returned by the query. With this method, you can iterate through the columns to get information about each column.

```
try {
   conn = DriverManager.getConnection(...);
   stmt = conn.createStatement();
   String query = "SELECT * FROM Author";
  ResultSet rs = stmt.executeQuery(query);
  ResultSetMetaData rsmd = rs.getMetaData(); // Get the meta data
                                             // for this ResultSet
  int columnCount = rsmd.getColumnCount();   // Get the number
                                             // of columns in this
  ...                                        // ResultSet
} catch (SQLException se) { }
```

The value of `columnCount` for the `Author` table is 3. We can use this value to iterate through the columns using the methods illustrated next.

public String getColumnName(int column) throws SQLException This method returns the `String` name of this column. Using the `columnCount`, we can create an output of the data from the database in a report-like format. For example:

```
String colData;
ResultSet rs = stmt.executeQuery(query);
ResultSetMetaData rsmd = rs.getMetaData();
int cols = rsmd.getColumnCount();
for (int i = 1; i <= cols; i++) {
  out.print(rsmd.getColumnName(i)+ " ");     // Print each column name
}
out.println();
while (rs.next()) {
  for (int i = 1; i <= cols; i++) {
    if (rs.getObject(i) != null) {
      colData = rs.getObject(i).toString();  // Get the String value
                                             // of the column object
    } else {
      colData = "NULL";                      // or NULL for a null
    }
    out.print(colData);                      // Print the column data
  }
  out.println();
}
```

This example is somewhat rudimentary, as we probably need to do some better formatting on the data, but it will produce a table of output:

```
AUTHORID FIRSTNAME LASTNAME
1000 Rick Riordan
1001 Nancy Farmer
1002 Ally Condie
1003 Cressida Cowell
1004 Lauren St. John
1005 Eoin Colfer
...
```

public String getTableName(int column) throws SQLException The method returns the `String` name of the table that this column belongs to. This method is useful when the query is a join of two or more tables and we need to know which table a column came from. For example, suppose that we want to get a list of books by author's last name:

```
String query = "SELECT Author.LastName, Book.Title
               FROM Author, Book, Books_By_Author
               WHERE Author.AuthorID = Books_By_Author.AuthorID
               AND Book.isbn = Books_By_Author.isbn"
```

With a query like this, we might want to know which table the column data came from:

```
ResultSetMetaData rsmd = rs.getMetaData();
int cols = rsmd.getColumnCount();
for (int i = 1; i <= cols; i++) {
  out.print(rsmd.getTableName(i) + ":" +
                    rsmd.getColumnName(i) + " ");
}
```

This code will print the name of the table, a colon, and the column name. The output might look something like this:

```
AUTHOR:LASTNAME BOOK:TITLE
```

public int getColumnDisplaySize(int column) throws SQLException This method returns an integer of the size of the column. This information is useful for determining the maximum number of characters a column can hold (if it is a VARCHAR type) and the spacing that is required between columns for a report.

Printing a Report

To make a prettier report than the one in the `getColumnName` method earlier, for example, we could use the display size to pad the column name and data with spaces. What we want is a table with spaces between the columns and headings that looks something like this when we query the Author table:

```
AUTHORID   FIRSTNAME              LASTNAME
1000         Rick                   Riordan
1001         Nancy                  Farmer
1002         Ally                   Condie
1003         Cressida               Cowell
1004         Lauren                 St. John
1005         Eoin                   Colfer
...
```

Using the methods we have discussed so far, here is code that produces a pretty report from a query:

```
ResultSet rs = stmt.executeQuery(query);
ResultSetMetaData rsmd = rs.getMetaData();
int cols = rsmd.getColumnCount();
String col, colData;
for (int i = 1; i <= cols; i++) {
  col = leftJustify(rsmd.getColumnName(i),          // Left justify
                    rsmd.getColumnDisplaySize(i));  // column name
  out.print(col);                                   // padded with
}                                                   // size spaces
out.println(); // Print a linefeed
while (rs.next()) {
  for (int i = 1; i <= cols; i++) {
    if (rs.getObject(i) != null) {
      colData = rs.getObject(i).toString(); // Get the data in the
                                            // column as a String
    } else {
      colData = "NULL";                     // If the column is null
                                            // use "NULL"
    }
    col = leftJustify(colData,
                      rsmd.getColumnDisplaySize(i)));
    out.print(col);
  }
  out.println();
}
```

A couple of things to note about the example code: first, the `leftJustify` method, which takes a string to print left-justified and an integer for the total number of characters in the string. The difference between the actual string length and the integer value will be filled with spaces. This method uses the `String format()` method and the `"-"` (dash) flag to return a `String` that is left-justified with spaces. The `%1$` part indicates the flag should be applied to the first argument. What we are building is a format string dynamically. If the column display size is 20, the format string will be `%1$-20s`, which says "print the argument passed (the first argument) on the left with a width of 20 and use a string conversion."

Note that if the length of the string passed in and the integer field length (`n`) are the same, we add one space to the length to make it look pretty:

```
public static String leftJustify(String s, int n) {
  if (s.length() <= n) n++;  // Add an extra space if the length of
                             // the String s is less than or equal to
                             // the length of the column n
  return String.format("%1$-" + n + "s", s);  // Pad to the right of
                                              // the String by n
                                              // spaces
}
```

Second, databases can store NULL values. If the value of a column is NULL, the object returned in the `rs.getObject()` method is a Java null. So we have to test for null to avoid getting a null pointer exception when we execute the `toString()` method.

Notice that we don't have to use the `next()` method before reading the `ResultSetMetaData`—we can do that at any time after obtaining a valid result set. Running this code and passing it a query like "SELECT * FROM Author" returns a neatly printed set of authors:

```
AUTHORID    FIRSTNAME           LASTNAME
1000        Rick                Riordan
1001        Nancy               Farmer
1002        Ally                Condie
1003        Cressida            Cowell
1004        Lauren              St. John
1005        Eoin                Colfer
...
```

Moving Around in ResultSets

So far, for all the result sets we looked at, we simply moved the cursor forward by calling `next()`. The default characteristics of a `Statement` are cursors that only move forward and result sets that do not support changes. The `ResultSet` interface actually defines these characteristics as static int variables: TYPE_FORWARD_ONLY and CONCUR_READ_ONLY. However, the JDBC specification defines additional static `int` types (shown next) that allow a developer to move the cursor forward, backward, and to a specific position in the result set. In addition, the result set can be modified while open and the changes written to the database. Note that support for cursor movement and updatable result sets is not a requirement on a driver, but most drivers provide this capability. In order to create a result set that uses positionable cursors and/or supports updates, you must create a `Statement` with the appropriate scroll type and concurrency setting, and then use that `Statement` to create the `ResultSet` object.

The ability to move the cursor to a particular position is the key to being able to determine how many rows are returned from a result set—something we will look at shortly. The ability to modify an open result set may seem odd, particularly if you are a seasoned database developer. After all, isn't that what a SQL UPDATE command is for?

Consider a situation in which you want to perform a series of calculations using the data from the result set rows, then write a change to each row based on some criteria, and finally write the data back to the database. For example, imagine a database table that contains customer data, including the date they

joined as a customer, their purchase history, and the total number of orders in the last two months. After reading this data into a result set, you could iterate over each customer record and modify it based on business rules: set their minimum discount higher if they have been a customer for more than a year with at least one purchase per year or set their preferred credit status if they have been purchasing more than $100 per month. With an updatable result set, you can modify several customer rows, each in a different way, and commit the rows to the database without having to write a complex SQL query or a set of SQL queries—you simply commit the updates on the open result set.

Let's look at how to modify a result set in more detail. There are three `ResultSet` cursor types:

- **TYPE_FORWARD_ONLY** The default value for a `ResultSet`—the cursor moves forward only through a set of results.
- **TYPE_SCROLL_INSENSITIVE** A cursor position can be moved in the result forward or backward, or positioned to a particular cursor location. Any changes made to the underlying data—the database itself—are not reflected in the result set. In other words, the result set does not have to "keep state" with the database. This type is generally supported by databases.
- **TYPE_SCROLL_SENSITIVE** A cursor can be moved in the results forward or backward, or positioned to a particular cursor location. Any changes made to the underlying data are reflected in the open result set. As you can imagine, this is difficult to implement and is, therefore, not implemented in a database or JDBC driver very often.

JDBC provides two options for data concurrency with a result set:

- **CONCUR_READ_ONLY** This is the default value for result set concurrency. Any open result set is read-only and cannot be modified or changed.
- **CONCUR_UPDATABLE** A result set can be modified through the `ResultSet` methods while the result set is open.

Because a database and JDBC driver are not required to support cursor movement and concurrent updates, the JDBC provides methods to query the

database and driver using the `DatabaseMetaData` object to determine if your driver supports these capabilities. For example:

```
Connection conn = DriverManager.getConnection(...);
DatabaseMetaData dbmd = conn.getMetaData();
if (dbmd.supportsResultSetType(ResultSet.TYPE_FORWARD_ONLY)) {
  out.print("Supports TYPE_FORWARD_ONLY");
  if (dbmd.supportsResultSetConcurrency(
      ResultSet.TYPE_FORWARD_ONLY,
      ResultSet.CONCUR_UPDATABLE)) {
    out.println(" and supports CONCUR_UPDATABLE");
  }
}

if (dbmd.supportsResultSetType(ResultSet.TYPE_SCROLL_INSENSITIVE)) {
  out.print("Supports TYPE_SCROLL_INSENSITIVE");
  if (dbmd.supportsResultSetConcurrency(
      ResultSet.TYPE_SCROLL_INSENSITIVE,
      ResultSet.CONCUR_UPDATABLE)) {
    out.println(" and supports CONCUR_UPDATABLE");
  }
}
if (dbmd.supportsResultSetType(ResultSet.TYPE_SCROLL_SENSITIVE)) {
  out.print("Supports TYPE_SCROLL_SENSITIVE");
  if (dbmd.supportsResultSetConcurrency(
      ResultSet.TYPE_SCROLL_SENSITIVE,
      ResultSet.CONCUR_UPDATABLE)) {
    out.println("Supports CONCUR_UPDATABLE");
  }
}
```

Running this code on the Java DB (Derby) database, these are the results:

```
Supports TYPE_FORWARD_ONLY and supports CONCUR_UPDATABLE
Supports TYPE_SCROLL_INSENSITIVE and supports CONCUR_UPDATABLE
```

In order to create a `ResultSet` with TYPE_SCROLL_INSENSITIVE and CONCUR_UPDATABLE, the `Statement` used to create the `ResultSet` must be created (from the `Connection`) with the cursor type and concurrency you want. You can determine what cursor type and concurrency the `Statement` was created with, but once created, you can't change the cursor type or concurrency of an existing `Statement` object. Also, note that just because you set a cursor type or concurrency setting, that doesn't mean you will get those settings. As you will see in the section on exceptions, the driver can determine that the database doesn't support one or both of the settings you chose and it will throw a warning and (silently) revert to its default settings, if they are not supported. You will see how to detect these JDBC warnings in the section on exceptions and warnings.

```
Connection conn = DriverManager.getConnection(...);
Statement stmt =
  conn.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
                       ResultSet.CONCUR_UPDATABLE);
```

Besides being able to use a `ResultSet` object to update results, which we'll look at next, being able to manipulate the cursor provides a side benefit —we can use the cursor to determine the number of rows returned in a query. Although it would seem like there ought to be a method in `ResultSet` or `ResultSetMetaData` to do this, this method does not exist.

In general, you should not need to know how many rows are returned, but during debugging, you may want to diagnose your queries with a stand-alone database and use cursor movement to read the number of rows returned.

Something like this would work:

```
ResultSet rs = stmt.executeQuery(query);  // Get a ResultSet
if (rs.last()) {                          // Move the very last row
    int rowCount = rs.getRow();           // Get row number (the count)
    rs.beforeFirst();                     // Move to before the 1st row
}
```

Of course, you may also want to have a more sophisticated method that preserves the current cursor position and returns the cursor to that position, regardless of when the method was called. Before we look at that code, let's look at the other cursor movement methods and test methods (besides `next`) in `ResultSet`. As a quick summary, Table 12-10 lists the methods you use to change the cursor position in a `ResultSet`.

TABLE 12-10 `ResultSet` Cursor Positioning Methods

Method	Effect on the Cursor and Return Value
`boolean next()`	Moves the cursor to the next row in the `ResultSet`. Returns `false` if the cursor is positioned beyond the last row.
`boolean previous()`	Moves the cursor backward one row. Returns `false` if the cursor is positioned before the first row.
`boolean absolute(int row)`	Moves the cursor to an absolute position in the `ResultSet`. Rows are numbered from 1. Moving to row 0 moves the cursor to before the first row. Moving to negative row numbers starts from the last row and works backward. Returns `false` if the cursor is positioned beyond the last row or before the first row.
`boolean relative(int row)`	Moves the cursor to a position relative to the current position. Invoking `relative(1)` moves forward one row; invoking `relative(-1)` moves backward one row. Returns `false` if the cursor is positioned beyond the last row or before the first row.
`boolean first()`	Moves the cursor to the first row in the `ResultSet`. Returns `false` if there are no rows in the `ResultSet` (empty result set).
`boolean last()`	Moves the cursor to the last row in the `ResultSet`. Returns `false` if there are no rows in the `ResultSet` (empty result set).
`void beforeFirst()`	Moves the cursor to before the first row in the `ResultSet`.
`void afterLast()`	Moves the cursor to after the last row in the `ResultSet`.

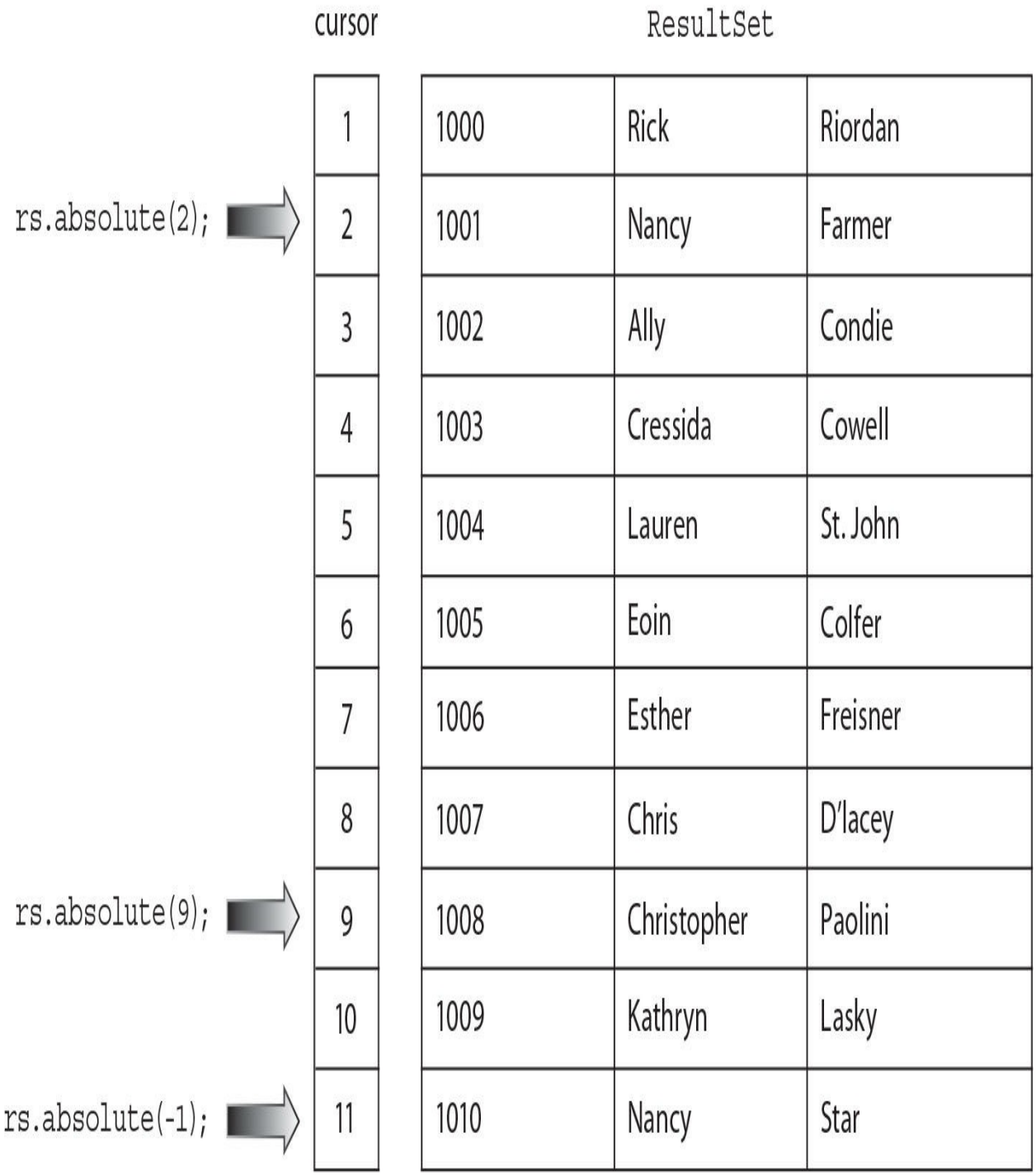
Let’s look at each of these methods in more detail.

public boolean absolute(int row) throws SQLException This method positions the cursor to an absolute row number. The contrasting method is relative. Passing `0` as the row argument positions the cursor to before the first row. Passing a negative value, like `-1`, positions the cursor to the position after the last row minus one—in other words, the last row. If you attempt to position the cursor beyond the last row, say at position 22 in a 19-row result set, the cursor will be positioned beyond the last row, the implications of which we’ll discuss next. Figure 12-5 illustrates how invocations of `absolute()` position the cursor.

FIGURE 12-5

Absolute cursor positioning

```
String query = "SELECT * FROM Author";
```



The `absolute()` method returns `true` if the cursor was successfully positioned within the `ResultSet` and `false` if the cursor ended up before the

first or after the last row. For example, suppose you wanted to process only every other row:

```
ResultSet rs = stmt.executeQuery(query);
for (int i = 1; ; i += 2) {
  if (rs.absolute(i)) {  // The absolute method moves to the row
                         // passed as the integer value and returns
                         // true if the move was successful
                         // ... process the odd row
  } else {
    break;
  }
}
```

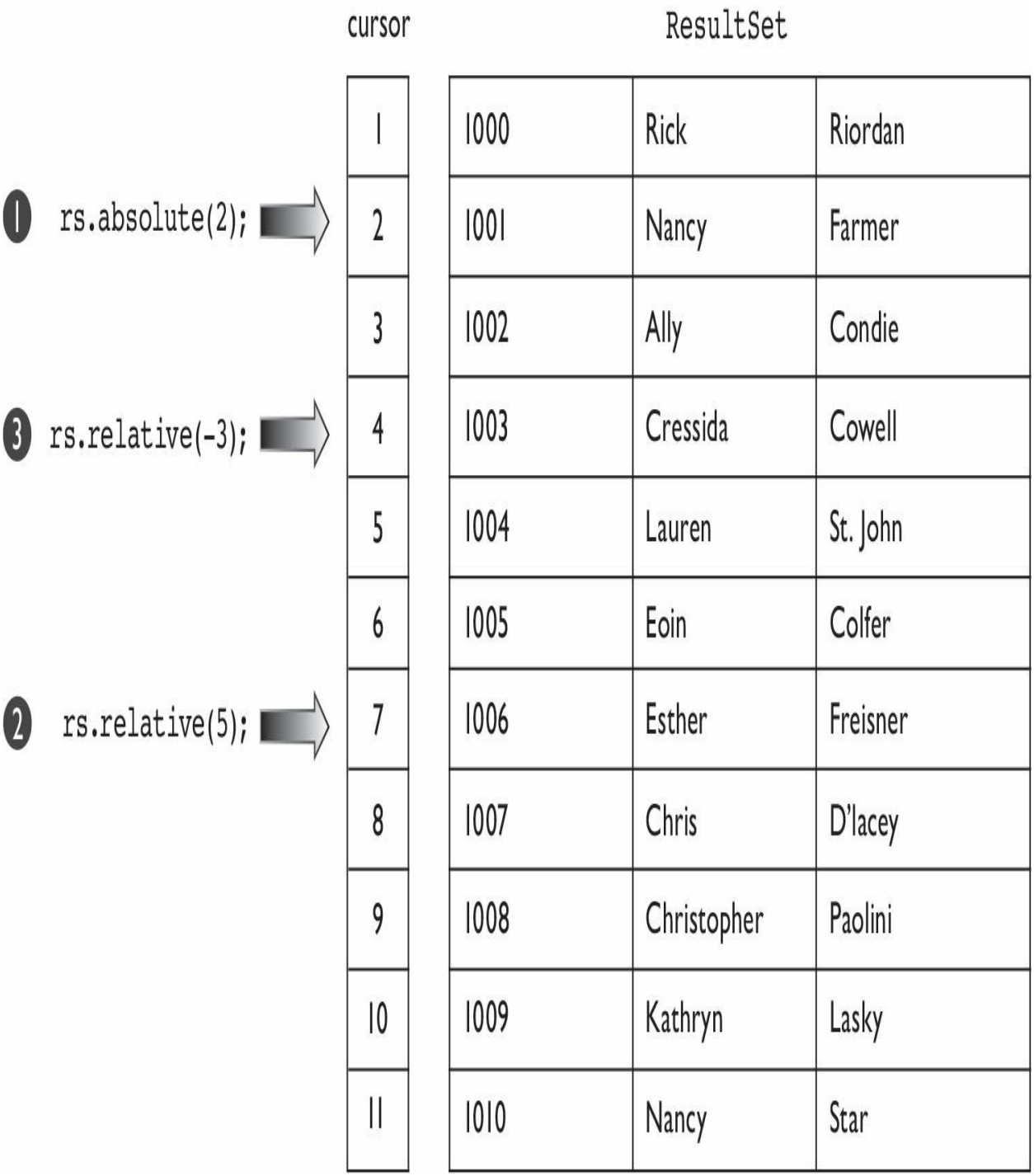
public int getRow() throws SQLException This method returns the current row position as a positive integer (1 for the first row, 2 for the second, and so on) or 0 if there is no current row—the cursor is either before the first row or after the last row. This is the only method of this set of cursor methods that is optionally supported for TYPE_FORWARD_ONLY `ResultSets`.

public boolean relative(int rows) throws SQLException The `relative()` method is the cousin to `absolute`. Get it, cousin? Okay, anyway, `relative()` will position the cursor either before or after the current position of the number of rows passed in to the method. So if the cursor is on row 15 of a 30-row `ResultSet`, calling `relative(2)` will position the cursor to row 17, and then calling `relative(-5)` positions the cursor to row 12. Figure 12-6 shows how the cursor is moved based on calls to `absolute()` and `relative()`.

FIGURE 12-6

Relative cursor positioning (Circled numbers indicate order of invocation.)

```
String query = "SELECT * FROM Author";
```



Like absolute positioning, attempting to position the cursor beyond the last row or before the first row simply results in the cursor being after the last row

or before the first row, respectively, and the method returns false. Also, calling relative with an argument of 0 does exactly what you might expect—the cursor remains where it is. Why would you use relative? Let's assume you are displaying a fairly long database table on a web page using an HTML table. You might want to allow your user to be able to page forward or backward relative to the currently selected row—maybe something like this:

```
public boolean getNextPageOfData (ResultSet rs, int pageSize) throws
SQLException{
  return rs.relative(pageSize);
}
```

public boolean previous() throws SQLException The `previous()` method works exactly the same as the `next()` method, only it backs up through the `ResultSet`. Using this method with the `afterLast()` method described next, you can move through a `ResultSet` in reverse order (from last row to first).

public void afterLast() throws SQLException This method positions the cursor after the last row. Using this method and then the `previous()` method, you can iterate through a `ResultSet` in reverse. For example:

```
public void showFlippedResultSet(ResultSet rs) throws SQLException {
  rs.afterLast();          // Position the cursor after the last row
  while (rs.previous()) {  // Back up through the ResultSet
                           // process the result set
  }
}
```

Just like `next()`, when `previous()` backs up all the way to before the first row, the method returns `false`.

public void beforeFirst() throws SQLException This method will return the cursor to the position it held when the `ResultSet` was first created and returned by a `Statement` object.

```
rs.beforeFirst(); // Position the cursor before the first row
```

public boolean first() throws SQLException The `first()` method positions the cursor on the first row. It is the equivalent of calling `absolute(1)`. This method returns `true` if the cursor was moved to a valid row and `false` if the `ResultSet` has no rows.

```
if (!rs.first()) {
  out.println("No rows in this result set");
}
```

public boolean last() throws SQLException The `last()` method positions the cursor on the last row. This method is the equivalent of calling `absolute(-1)`. This method returns `true` if the cursor was moved to a valid row and `false` if the `ResultSet` has no rows.

```
if (!rs.last()) {
  out.println("No rows in this result set");
}
```

A couple of notes on the exceptions thrown by all of these methods:

- A `SQLException` will be thrown by these methods if the type of the `ResultSet` is TYPE_FORWARD_ONLY, if the `ResultSet` is closed (we will look at how a result set is closed in an upcoming section), or if a database error occurs.
- A `SQLFeatureNotSupportedException` will be thrown by these methods if the JDBC driver does not support the method. This exception is a subclass of `SQLException`.
- Most of these methods have no effect if the `ResultSet` has no rows—for example, a `ResultSet` returned by a query that returned no rows.

The following methods return a boolean to allow you to “test” the current cursor position without moving the cursor. Note that these are not on the

exam but are provided to you for completeness:

- **`isBeforeFirst()`** True if the cursor is positioned before the first row
- **`isAfterLast()`** True if the cursor is positioned after the last row
- **`isFirst()`** True if the cursor is on the first row
- **`isLast()`** True if the cursor is on the last row

So now that we have looked at the cursor positioning methods, let's revisit the code to calculate the row count. We will create a general-purpose method to allow the row count to be calculated at any time and at any current cursor position. Here is the code:

```
public static int getRowCount(ResultSet rs) throws SQLException {
  int rowCount = -1;
  int currRow = 0;

  if (rs != null) {              // make sure the ResultSet is not null
    currRow = rs.getRow();       // Save the current row position:
                                 // zero indicates that there is no
                                 // current row position - could be
                                 // beforeFirst or afterLast
    if (rs.isAfterLast()) {      // afterLast, so set the currRow negative
        currRow = -1;
    }
    if (rs.last()) {             // move to the last row and get the position
                                 // if this method returns false, there are no
                                 // results
      rowCount = rs.getRow();    // Get the row count
                                 // Return the cursor to the position it
                                 // was in before the method was called.
      if (currRow == -1) {       // if the currRow is negative, the cursor
                                 // position was after the last row, so
                                 // return the cursor to the last row
        rs.afterLast();
      } else if (currRow == 0) { // else if the cursor is zero, move
                                 // the cursor to before the first row
        rs.beforeFirst();
      } else {                   // else return the cursor to its last position
        rs.absolute(currRow);
      }
    }
  }
  return rowCount;
}
```

Looking through the code, you notice that we took special care to preserve the current position of the cursor in the `ResultSet`. We called `getRow()` to get the current position, and if the value returned was 0, the current position of the `ResultSet` could be either before the first row or after the last row, so we used the `isAfterLast()` method to determine where the cursor was. If the cursor was after the last row, then we stored a `-1` in the `currRow` integer.

We then moved the cursor to the last position in the `ResultSet`, and if that move was successful, we get the current position and save it as the `rowCount` (the last row and, therefore, the count of rows in the `ResultSet`). Finally, we use the value of `currRow` to determine where to return the cursor. If the value of the cursor is `-1`, we need to position the cursor after the last row. Otherwise, we simply use `absolute()` to return the cursor to the appropriate position in the `ResultSet`.

While this may seem like several extra steps, we will look at why preserving the cursor can be important when we look at updating `ResultSets` next.

Updating ResultSets

Please note that you might not get any questions on the real exam for this section and the subsections that follow.

If you have casually used JDBC, or are new to JDBC, you may be surprised to know that a `ResultSet` object can do more than just provide the results of a query to your application. Besides just returning the results of a query, a `ResultSet` object may be used to modify the contents of a database table, including update existing rows, delete existing rows, and add new rows.

In a traditional SQL application, you might perform the following SQL queries to raise the price of all of the hardcover books in inventory that are currently 10.95 to 11.95 in price:

```
UPDATE Book SET UnitPrice = 11.95 WHERE UnitPrice = 10.95
                                  AND Format = 'Hardcover'
```

Hopefully, by now you feel comfortable creating a `Statement` to perform this query using a SQL UPDATE:

```
// We have a connection and we are in a try-catch block...
Statement stmt = conn.createStatement();
String query = "UPDATE Book SET UnitPrice = 11.95 " +
               "WHERE UnitPrice = 10.95 AND Format = 'Hardcover'";
int rowsUpdated = stmt.executeUpdate(query);
```

But what if you wanted to do the updates on a book-by-book basis? What if you only want to increase the price of your bestsellers, rather than every single book?

You would have to get the values from the database using a SELECT, then store the values in an array indexed somehow (perhaps with the primary key), then construct the appropriate UPDATE command strings, and call `executeUpdate()` one row at a time. Another option is to update the `ResultSet` directly.

When you create a `Statement` with concurrency set to CONCUR_UPDATABLE, you can modify the data in a result set and then apply your changes back to the database without having to issue another query.

In addition to the `getXXXX` methods we looked at for `ResultSet`—methods that get column values as integers, `Date` objects, `Strings`, etc.—there is an equivalent `updateXXXX` method for each type. And, just like the `getXXXX` methods, the `updateXXXX` methods can take either a `String` column name or an integer column index.

Let’s rewrite the previous update example using an updatable `ResultSet`:

```
// We have a connection and we are in a try-catch block...
Statement stmt =                                        // Scrollable
  conn.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE, // and
                       ResultSet.CONCUR_UPDATABLE);     // updatable
String query = "SELECT UnitPrice from Book " +
               "WHERE Format = 'Hardcover'";
ResultSet rs = stmt.executeQuery(query);      // Populate the ResultSet
while (rs.next()) {
    if (rs.getFloat("UnitPrice") == 10.95f) { // Check each row: if
                                              // unitPrice = 10.95
        rs.updateFloat("UnitPrice", 11.95f);  // set it to 11.95
        rs.updateRow();                       // and update the row
                                              // in the database
    }
}
```

Notice that after modifying the value of `UnitPrice` using the `updateFloat()` method, we called the method `updateRow()`. This method writes the current row to the database. This two-step approach ensures that all of the changes are made to the row before the row is written to the database. And you can change your mind with a `cancelRowUpdates()` method call.

Table 12-11 summarizes methods that are commonly used with updatable `ResultSets` (whose concurrency type is set to CONCUR_UPDATABLE).

Let's look at the common methods used for altering database contents through the `ResultSet` in detail.

TABLE 12-11 Methods Used with Updatable `ResultSets`

Method	Purpose
`void updateRow()`	Updates the database with the contents of the current row of this `ResultSet`.
`void deleteRow()`	Deletes the current row from the `ResultSet` and the underlying database.
`void cancelRowUpdates()`	Cancels any updates made to the current row of this `ResultSet` object. This method will effectively undo any changes made to the `ResultSet` row. If the `updateRow()` method was called before `cancelRowUpdates`, this method will have no effect.
`void moveToInsertRow()`	Moves the cursor to a special row in the `ResultSet` set aside for performing an insert. You need to move to the insert row before updating the columns of the row with update methods and calling `insertRow()`.
`void insertRow()`	Inserts the contents of the insert row into the database. Note that this method does not change the current `ResultSet`, so the `ResultSet` should be read again if you want the `ResultSet` to be consistent with the contents of the database.
`void moveToCurrentRow()`	Moves the cursor back to the current row from the insert row. If the cursor was not on the insert row, this method has no effect.

public void updateRow() throws SQLException This method updates the database with the contents of the current row of the `ResultSet`. There are a couple of caveats for this method. First, the `ResultSet` must be from a SQL SELECT statement on a single table—a SQL statement that includes a JOIN or a SQL statement with two tables cannot be updated. Second, the `updateRow()` method should be called *before* moving to the next row. Otherwise, the updates to the current row may be lost.

So the typical use for this method is to update the contents of a row using the appropriate `updateXXXX()` methods and then update the database with the contents of the row using the `updateRow()` method. For example, in this fragment, we are updating the `UnitPrice` of a row to $11.95:

```
rs.updateFloat("UnitPrice", 11.95f); // Set the price to 11.95
rs.updateRow();                      // Update the row in the DB
```

public boolean rowUpdated() throws SQLException This method returns true if the current row was updated. Note that not all databases can detect updates. However, JDBC provides a method in `DatabaseMetaData` to determine if updates are detectable, `DatabaseMetaData.updatesAreDetected(int type)`, where the type is one of the `ResultSet` types—TYPE_SCROLL_INSENSITIVE, for example. We will cover the `DatabaseMetaData` interface and its methods a little later in this section.

```
if (rs.rowUpdated()) {            // Has this row been modified?
  out.println("Row: " + rs.getRow() + " updated.");
}
```

public void cancelRowUpdates() throws SQLException This method allows you to "back out" changes made to the row. This method is important, because the `updateXXXX` methods should not be called twice on the same column. In other words, if you set the value of `UnitPrice` to 11.95 in the previous example and then decided to switch the price back to 10.95, calling the `updateFloat()` method again can lead to unpredictable results. So the better approach is to call `cancelRowUpdates()` before changing the value of a column a second time.

```
boolean priceRollback = ...;  // Price rollback set somewhere else
while (rs.next()) {
  if (rs.getFloat("UnitPrice") == 10.95f) {
    rs.updateFloat("UnitPrice", 11.95f);
  }
  if (priceRollback) {        // If priceRollback is true
    rs.cancelRowUpdates();    // Rollback changes to this row
  } else {
    rs.updateRow();           // else, commit this row to the DB
  }
}
```

public void deleteRow() throws SQLException This method will remove the current row from the `ResultSet` and from the underlying database. The row in the database is removed (similar to the result of a DELETE statement).

```
rs.last();
rs.deleteRow();  // Delete the last row.
```

What happens to the `ResultSet` after a `deleteRow()` method depends on whether the `ResultSet` can detect deletions. And this ability depends on the JDBC driver.

The `DatabaseMetaData` interface can be used to determine if the `ResultSet` can detect deletions:

```
int type = ResultSet.TYPE_SCROLL_INSENSITIVE; // Scrollable ResultSet
DatabaseMetaData dbmd = conn.getMetaData();   // Get meta data about
                                              // the driver and DB
if (dbmd.deletesAreDetected(type)) {          // Returns false if deleted rows
                                              // are removed from the ResultSet
  while (rs.next()) {                         // Iterate through the ResultSet
    if (rs.rowDeleted()) {                    // Deleted rows are flagged, but
      continue;                               // not removed, so skip them
    } else {
     // process the row
    }
} else {
  // Close the ResultSet and re-run the query
}
```

In general, to maintain an up-to-date `ResultSet` after a deletion, the `ResultSet` should be re-created with a query.

Deleting the current row does not move the cursor—it remains on the current row—so if you deleted row 1, the cursor is still positioned at row 1. However, if the deleted row was the last row, then the cursor is positioned after the last row. Note that there is no undo for `deleteRow()`, at least, not by default.

public boolean rowDeleted() throws SQLException As described earlier, when a `ResultSet` can detect deletes, the `rowDeleted()` method is used to indicate a row has been deleted but remains as a part of the `ResultSet` object. For example, suppose that we deleted the second row of the Customer table. Printing the results (after the delete) to the console would look like Figure 12-7.

FIGURE 12-7

A `ResultSet` after `deleteRow()` is called on the second row

```
String query = "SELECT * FROM Customer";
ResultSet rs = stmt.executeQuery(query);
rs.next();
rs.next();
rs.deleteRow();
```

ResultSet

5000	John	Smith	john.smith@verizon.net	555-340-1230
null	null	null	null	null
5002	Bob	Collins	bob.collins@yahoo.com	555-012-3456
5003	Rebecca	Mayer	rebecca.mayer@gmail.com	555-205-8212
5006	Anthony	Clark	anthony.clark@gmail.com	555-256-1901
5007	Judy	Sousa	judy.sousa@verizon.net	555-751-1207
5008	Christopher	Patriquin	patriquinc@yahoo.com	555-316-1803
5009	Deborah	Smith	deb.smith@comcast.net	555-256-3421
5010	Jennifer	McGinn	jmcginn@comcast.net	555-250-0918

So if you are working with a `ResultSet` that is being passed around between methods and shared across classes, you might use `rowDeleted()` to

detect if the current row contains valid data.

Updating Columns Using Objects An interesting aspect of the `getObject()` and `updateObject()` methods is that they retrieve a column as a Java object. And because every Java object can be turned into a `String` using the object's `toString()` method, you can retrieve the value of any column in the database and print the value to the console as a `String`, as you saw in the section "Printing a Report."

Going the other way, toward the database, you can also use `String`s to update almost every column in a `ResultSet`. All of the most common SQL types—integer, float, double, long, and date—are wrapped by their representative Java object: `Integer`, `Float`, `Double`, `Long`, and `java.sql.Date`. Each of these objects has a method `valueOf()` that takes a `String`.

The `updateObject()` method takes two arguments: the first, a column name (`String`) or column index; and the second, an `Object`. We can pass a `String` as the `Object` type, and as long as the `String` meets the requirements of the `valueOf()` method for the column type, the `String` will be properly converted and stored in the database as the desired SQL type.

For example, suppose that we are going to update the publish date (`PubDate`) of one of our books:

```
// We have a connection and we are in a try-catch block...
Statement stmt =
  conn.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
                       ResultSet.CONCUR_UPDATABLE);
String query = "SELECT * FROM Book WHERE ISBN='142311339X'";
ResultSet rs = stmt.executeQuery(query);
rs.next();
rs.updateObject("PubDate", "2005-04-23"); // Update PubDate using
                                          // a String date
rs.updateRow();                           // Update this row
```

The `String` we passed meets the requirements for java.sql.Date, "yyyy-[m]m-[d]d," so the `String` is properly converted and stored in the database as the SQL Date value: 2005-04-23. Note this technique is limited to those SQL types that can be converted to and from a `String`, and if the `String` passed to the `valueOf()` method for the SQL type of the column is not properly formatted for the Java object, an `IllegalArgumentException` is thrown.

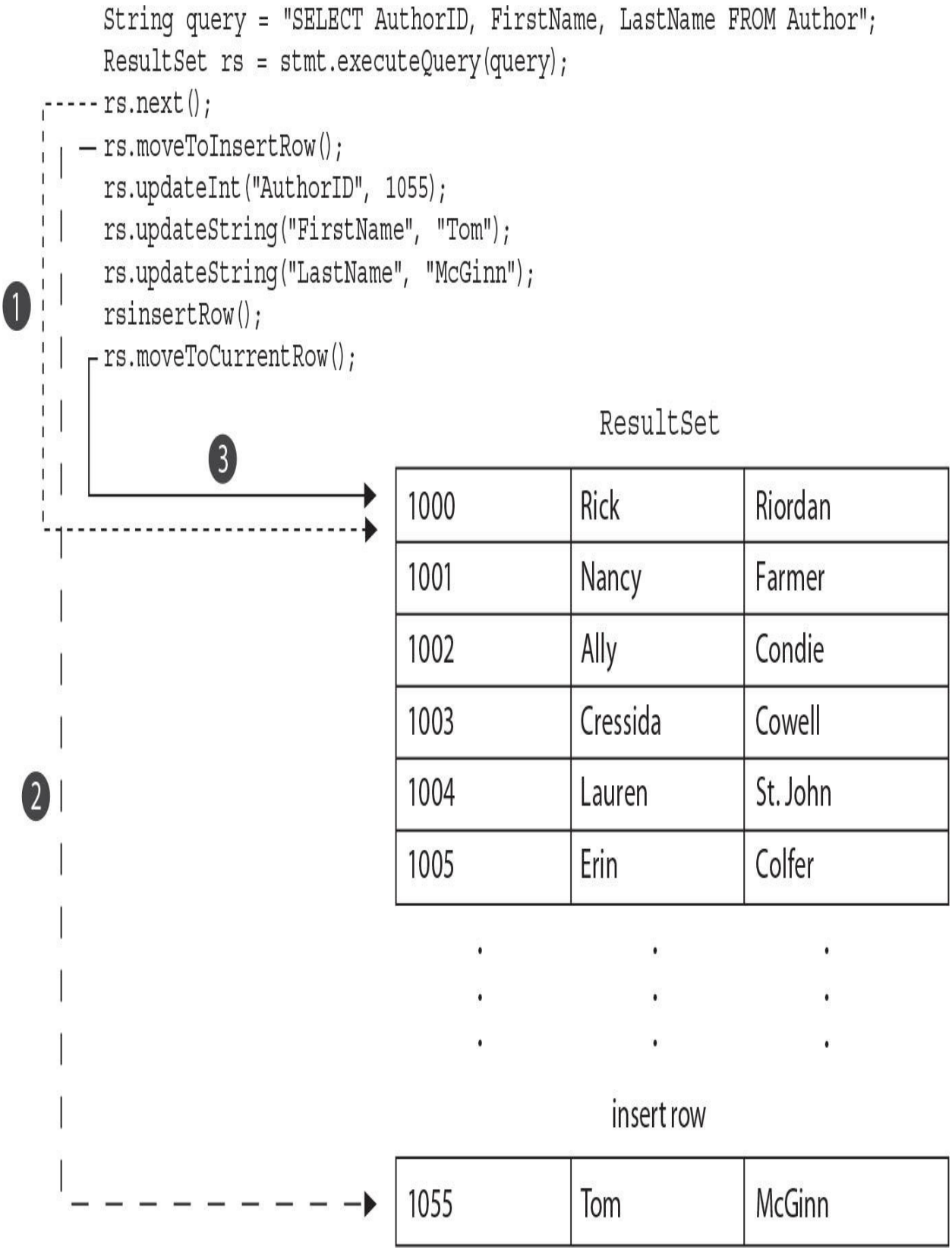
Inserting New Rows Using a ResultSet

In the last section, we looked at modifying the existing column data in a `ResultSet` and removing existing rows. In our final section on `ResultSets`, we'll look at how to create and insert a new row. First, you must have a valid `ResultSet` open, so typically, you have performed some query. `ResultSet` provides a special row, called the insert row, that you are actually modifying (updating) before performing the insert. Think of the insert row as a buffer where you can modify an empty row of your `ResultSet` with values.

Inserting a row is a three-step process, as shown in Figure 12-8: First (1) move to the special insert row, then (2) update the values of the columns for the new row, and finally (3) perform the actual insert (write to the underlying database). The existing `ResultSet` is not changed—you must rerun your query to see the underlying changes in the database. However, you can insert as many rows as you like. Note that each of these methods throws a `SQLException` if the concurrency type of the result set is set to CONCUR_READ_ONLY. Let's look at the methods before we look at example code.

FIGURE 12-8

The `ResultSet` insert row



public void moveToInsertRow() throws SQLException This method

moves the cursor to insert a row buffer. Wherever the cursor was when this method was called is remembered. After calling this method, the appropriate updater methods are called to update the values of the columns.

```
rs.moveToInsertRow();
```

public void insertRow() throws SQLException This method writes the insert row buffer to the database. Note that the cursor must be on the insert row when this method is called. Also, note that each column must be set to a value before the row is inserted in the database or a `SQLException` will be thrown. The `insertRow()` method can be called more than once—however, the `insertRow` follows the same rules as a SQL INSERT command. Unless the primary key is auto-generated, two inserts of the same data will result in a `SQLException` (duplicate primary key).

```
rs.insertRow();
```

public void moveToCurrentRow() throws SQLException This method returns the result set cursor to the row the cursor was on before the `moveToInsertRow()` method was called.

Let's look at a simple example, where we will add a new row in the Author table:

```
// We have a connection and we are in a try-catch block...
Statement stmt = conn.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
                                      ResultSet.CONCUR_UPDATABLE);
ResultSet rs = stmt.executeQuery("SELECT AuthorID, FirstName, LastName
                                  FROM Author");
rs.next();
rs.moveToInsertRow();                 // Move the special insert row
rs.updateInt("AuthorID", 1055);       // Create an author ID
rs.updateString("FirstName", "Tom");  // Set the first name
rs.updateString("LastName", "McGinn"); // Set the last name
rs.insertRow();                       // Insert the row into the database
rs.moveToCurrentRow();                // Move back to the current row in
                                      // ResultSet
```

Getting Information about a Database Using DatabaseMetaData

Note: On the real exam, you might not get any questions about `DatabaseMetaData`.

In the example we are using in this chapter, Bob's Books, we know quite a lot about the tables, columns, and relationships between the tables because we had that nifty data model earlier. But what if that were not the case? This section covers `DatabaseMetaData`, an interface that provides a significant amount of information about the database itself. This topic is fairly advanced stuff and is not on the exam, but it is provided here to give you an idea about how you can use metadata to build a model of a database without having to know anything about the database in advance.

Recall that the `Connection` object we obtained from `DriverManager` is an object that represents an actual connection with the database. And while the `Connection` object is primarily used to create `Statement` objects, there are a couple of important methods to study in the `Connection` interface. A

`Connection` can be used to obtain information *about* the database as well. This data is called "metadata," or "data about data."

One of `Connection`'s methods returns a `DatabaseMetaData` object instance, through which we can get information about the database, about the driver, and about transaction semantics that the database and JDBC driver support.

To obtain an instance of a `DatabaseMetaData` object, we use `Connection`'s `getMetaData()` method:

```
String url = "jdbc:derby://localhost:1521/BookSellerDB";
String user = "bookguy";
String pwd = "$3lleR";
try {
  Connection conn = DriverManager.getConnection(url, user, pwd);
  DatabaseMetaData dbmd = conn.getMetaData(); // Get the database
                                              // metadata
} catch (SQLException se) { }
```

`DatabaseMetaData` is a comprehensive interface, and through an object instance, we can determine a great deal about the database and the supporting driver. Most of the time, as a developer, you aren't coding against a database blindly and know the capabilities of the database and the driver before you write any code. Still, it is helpful to know that you can use `getObject` to return the value of the column, regardless of its type—very useful when all you want to do is create a report. We'll look at an example.

Here are a few methods we will highlight:

- **`getColumns()`** Returns a description of columns in a specified catalog and schema
- **`getProcedures()`** Returns a description of the stored procedures in a given catalog and schema
- **`getDriverName()`** Returns the name of the JDBC driver
- **`getDriverVersion()`** Returns the version number of the JDBC driver

as a string

- **`supportsANSI92EntryLevelSQL()`** Returns a boolean true if this database supports ANSI92 entry-level grammar

It is interesting to note that `DatabaseMetaData` methods also use `ResultSet` objects to return data about the database. Let's look at these methods in more detail.

public ResultSet getColumns(String catalog, String schemaPattern, String tableNamePattern, String columnNamePattern) throws SQLException This method is one of the best all-purpose data retrieval methods for details about the tables and columns in your database. Before we look at a code sample, it might be helpful to define catalogs and schemas. In a database, a schema is an object that enforces the integrity of the tables in the database. The schema name is generally the name of the person who created the database. In our examples, the BookGuy database holds the collection of tables and is the name of the schema. Databases may have multiple schemas stored in a catalog.

In this example, using the Java DB database as our sample database, the catalog is null and our schema is "BOOKGUY", and we are using a SQL catch-all pattern "`%`" for the table and column name patterns, like the "`*`" character you are probably used to with file systems like Windows. Thus, we are going to retrieve all of the tables and columns in the schema. Specifically, we are going to print out the table name, column name, the SQL data type for the column, and the size of the column. Note that here we used uppercase column identifiers. These are the column names verbatim from the JavaDoc, but in truth, they are not case-sensitive either, so "`Table_Name`" would have worked just as well. Also, the JavaDoc specifies the column index for these column headings, so we could have also used `rs.getString(3)` to get the table name.

```
String url = "jdbc:derby://localhost:1521/BookSellerDB";
String user = "bookguy";
String pwd = "$3lleR";
try {
  Connection conn = DriverManager.getConnection(url, user, pwd);
  DatabaseMetaData dbmd = conn.getMetaData();
  ResultSet rs
   = dbmd.getColumns(null, "BOOKGUY", "%", "%"); // Get a ResultSet
                                                 // for any catalog (null)
                                                 // in the BOOKGUY schema
                                                 // for all tables (%)
                                                 // for all columns (%)
  while (rs.next()) {
    out.print("Table Name:  " + rs.getString("TABLE_NAME") + " ");
    out.print("Column_Name: " + rs.getString("COLUMN_NAME") + " ");
    out.print("Type_Name:   " + rs.getString("TYPE_NAME") + " ");
    out.println("Column Size  " + rs.getString("COLUMN_SIZE"));
  }
} catch (SQLException se) {
   out.println("SQLException: " + se);
}
```

Running this code produces output something like this:

```
Table Name:  AUTHOR Column_Name: AUTHORID Type_Name:   INTEGER Column Size
10 Primary Key
Table Name:  AUTHOR Column_Name: FIRSTNAME Type_Name:   VARCHAR Column Size
20
Table Name:  AUTHOR Column_Name: LASTNAME Type_Name:   VARCHAR Column Size
20
Table Name:  BOOK Column_Name: ISBN Type_Name:   VARCHAR Column Size  10
Primary Key
Table Name:  BOOK Column_Name: TITLE Type_Name:   VARCHAR Column Size  100
Table Name:  BOOK Column_Name: PUBDATE Type_Name:   DATE Column Size  10
Table Name:  BOOK Column_Name: FORMAT Type_Name:   VARCHAR Column Size  30
Table Name:  BOOK Column_Name: UNITPRICE Type_Name:   DOUBLE Column Size  52
Table Name:  BOOKS_BY_AUTHOR Column_Name: AUTHORID Type_Name:   INTEGER
Column Size  10
Table Name:  BOOKS_BY_AUTHOR Column_Name: ISBN Type_Name:   VARCHAR Column
Size  10
Table Name:  CUSTOMER Column_Name: CUSTOMERID Type_Name:   INTEGER Column
Size  10 Primary Key
Table Name:  CUSTOMER Column_Name: FIRSTNAME Type_Name:   VARCHAR Column Size
30
Table Name:  CUSTOMER Column_Name: LASTNAME Type_Name:   VARCHAR Column Size
30
Table Name:  CUSTOMER Column_Name: EMAIL Type_Name:   VARCHAR Column Size  40
Table Name:  CUSTOMER Column_Name: PHONE Type_Name:   VARCHAR Column Size  15
```

public ResultSet getProcedures(String catalog, String schemaPattern, String procedureNamePattern) throws SQLException Stored procedures are functions that are sometimes built into a database and often defined by a database developer or database admin. These functions can range from data cleanup to complex queries. This method returns a result set that contains descriptive information about the stored procedures for a catalog and schema. In the example code, we will use null for the catalog name and schema pattern. The null indicates that we do not wish to narrow the search (effectively, the same as using a catch-all "%" search). Note that this example is returning the name of every stored procedure in the database. A little later, we'll look at how to actually call a stored procedure.

```
try {
  Connection conn = ...
  DatabaseMetaData dbmd = conn.getMetaData();
  ResultSet rs =
       dbmd.getProcedures(null, null, "%");     // Get a ResultSet of all
                                                // the stored procedures
                                                // in any catalog (null)
                                                // in any schema (null)
                                                // with wildcard name (%)
  while(rs.next()) {
    out.println("Procedure Name: " + rs.getString("PROCEDURE_NAME"));
  }
} catch (SQLException se) { }
```

Note that the output from this code fragment is highly database dependent. Here is sample output from the Derby (JavaDB) database that ships with the JDK:

```
Procedure Name: INSTALL_JAR
Procedure Name: REMOVE_JAR
Procedure Name: REPLACE_JAR
Procedure Name: SYSCS_BACKUP_DATABASE
Procedure Name: SYSCS_BACKUP_DATABASE_AND_ENABLE_LOG_ARCHIVE_MODE
Procedure Name: SYSCS_BACKUP_DATABASE_AND_ENABLE_LOG_ARCHIVE_MODE_NOWAIT
Procedure Name: SYSCS_BACKUP_DATABASE_NOWAIT
Procedure Name: SYSCS_BULK_INSERT
```

public String getDriverName() throws SQLException This method simply returns the name of the JDBC driver as a string. This method would be useful to log at the start of the application, as you'll see in the next section.

```
System.out.println("getDriverName: " + dbmd.getDriverName());
```

Obviously, the name of the driver depends on the JDBC driver you are using. Again, with the Derby database and JDBC driver, the output from this method looks something like this:

```
getDriverName: Apache Derby Network Client JDBC Driver
```

public String getDriverVersion() throws SQLException This method returns the JDBC driver version number as a string. This information and the driver name would be good to log in at startup of an application.

```
Logger logger = Logger.getLogger("com.cert.DatabaseMetaDataTest");
Connection conn = ...
DatabaseMetaData dbmd = conn.getMetaData();
logger.log(Level.INFO, "Driver Version: {0}", dbmd.getDriverVersion());
logger.log(Level.INFO, "Driver Name: {0}", dbmd.getDriverName());
```

Statements written to the log are generally recorded in a log file, but

depending on the IDE, they can also be written to the console. In NetBeans, for example, the log statements look something like this in the console:

```
Sep 23, 2012 3:55:39 PM com.cert.DatabaseMetaDataTest main
INFO: Driver Version: 10.8.2.2 - (1181258)
Sep 23, 2012 3:55:39 PM com.cert.DatabaseMetaDataTest main
INFO: Driver Name: Apache Derby Network Client JDBC Driver
```

public boolean supportsANSI92EntryLevelSQL() throws SQLException This method returns true if the database and JDBC driver support ANSI SQL-92 entry-level grammar. Support for this level (at a minimum) is a requirement for JDBC drivers (and, therefore, the database).

```
Connection conn = ...
DatabaseMetaData dbmd = conn.getMetaData();
if (!dbmd.supportsANSI92EntryLevelSQL()) {
    logger.log(Level.WARNING, "JDBC Driver does not meet minimum
            requirements for SQL-92 support");
}
```

When Things Go Wrong—Exceptions and Warnings

Whenever you are working with a database using JDBC, there is a possibility that something can go wrong. A JDBC connection is typically through a socket to a database resource on the network. So already we have at least two possible points of failure—the network can be down and/or the database can be down. And that assumes everything else you are doing with your database is correct, that all your queries are perfect! Like other Java exceptions, `SQLException` is a way for your application to determine what the problem is and take action if necessary.

Let's look at the type of data you get from a `SQLException` through its methods.

public String getMessage() This method is actually inherited from

`java.lang.Exception`, which `SQLException` extends from. This method returns the detailed reason why the exception was thrown. Often, the message contents `SQLState` and error code provide specific information about what went wrong.

public String getSQLState() The `String` returned by `getSQLState` provides a specific code and related message. `SQLState` messages are defined by the X/Open and SQL:2003 standards; however, it is up to the implementation to use these values. You can determine which standard your JDBC driver uses (or if it does not) through the `DatabaseMetaData.getSQLStateType()` method. Your implementation may also define additional codes specific to the implementation, so in either case, it is a good idea to consult your JDBC driver and database documentation. Because the `SQLState` messages and codes tend to be specific to the driver and database, the typical use of these in an application is limited to either logging messages or debugging information.

public int getErrorCode() Error codes are not defined by a standard and are thus implementation specific. They can be used to pass an actual error code or severity level, depending on the implementation.

public SQLException getNextException() One of the interesting aspects of `SQLException` is that the exception thrown could be the result of more than one issue. Fortunately, JDBC simply tacks each exception onto the next in a process called chaining. Typically, the most severe exception is thrown last, so it is the first exception in the chain.

You can get a list of all of the exceptions in the chain using the `getNextException()` method to iterate through the list. When the end of the list is reached, `getNextException()` returns a null. In this example, the `SQLExceptions`, `SQLState`, and vendor error codes are logged:

```
Logger logger = Logger.getLogger("com.example.MyClass");
try {
  // some JDBC code in a try block
  // ...
} catch (SQLException se) {
  while (se != null) {
    logger.log(Level.SEVERE, "------ SQLException ------");
    logger.log(Level.SEVERE, "SQLState: " + se.getSQLState());
    logger.log(Level.SEVERE, "Vendor Error code: " +
                             se.getErrorCode());
    logger.log(Level.SEVERE,"Message: " + se.getMessage());
    se = se.getNextException();
  }
}
```

Warnings

Although `SQLWarning` is a subclass of `SQLException`, warnings are silently chained to the JDBC object that reported them. This is probably one of the few times in Java where an object that is part of an exception hierarchy is not thrown as an exception. The reason is that a warning is not an exception per se. Warnings can be reported on `Connection`, `Statement`, and `ResultSet` objects.

For example, suppose that we mistakenly set the result-set type to TYPE_SCROLL_SENSITIVE when creating a `Statement` object. This does not create an exception; instead, the database will handle the situation by chaining a `SQLWarning` to the `Connection` object and resetting the type to TYPE_FORWARD_ONLY (the default) and continue on. Everything would be fine, of course, until we tried to position the cursor, at which point a `SQLException` would be thrown. And, like `SQLException`, you can retrieve

warnings from the SQLWarning object using the getNextWarning() method.

```
Connection conn =
  DriverManager.getConnection("jdbc:derby://localhost:1527/BookSellerDB",
 "bookguy", "$3lleR");
Statement stmt =
  conn.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
                       ResultSet.CONCUR_UPDATABLE);
String query = "SELECT * from Book WHERE Book.Format = 'Hardcover'";
ResultSet rs = stmt.executeQuery(query);
SQLWarning warn = conn.getWarnings();    // Get any SQLWarnings
while (warn != null) {                   // If there is a SQLWarning, print it
  out.println("SQLState: " + warn.getSQLState());
  out.println("Message: " + warn.getMessage());
  warn = warn.getNextWarning();          // Get the next warning
}
```

Connection objects will add warnings (if necessary) until the Connection is closed or until the clearWarnings() method is called on the Connection instance. The clearWarnings() method sets the list of warnings to null until another warning is reported for this Connection object.

Statements and ResultSets also generate SQLWarnings, and these objects have their own clearWarnings() methods. Statement warnings are cleared automatically when a statement is reexecuted, and ResultSet warnings are cleared each time a new row is read from the result set.

The following sections summarize the methods associated with SQLWarnings.

SQLWarning getWarnings() throws SQLException This method gets the first SQLWarning object or returns null if there are no warnings for this Connection, Statement, or ResultSet object. A SQLException is thrown if

the method is called on a closed object.

void clearWarnings() throws SQLException This method clears and resets the current set of warnings for this `Connection`, `Statement`, or `ResultSet` object. A `SQLException` is thrown if the method is called on a closed object.

Properly Closing SQL Resources

In this chapter, we have looked at some very simple examples where we create a `Connection` and `Statement` and a `ResultSet` all within a single `try` block and catch any `SQLException`s thrown. What we have not done so far is properly close these resources. The reality is that it is probably less important for such small examples, but for any code that uses a resource, like a socket, or a file, or a JDBC database connection, closing the open resources is a good practice.

It is also important to know when a resource is closed automatically. Each of the three major JDBC objects—`Connection`, `Statement`, and `ResultSet`—has a `close()` method to explicitly close the resource associated with the object and explicitly release the resource. We hope by now you also realize that the objects have a relationship with each other, so if one object executes `close()`, it will have an impact on the other objects. The following table should help explain this.

Method Call	Has the Following Action(s)
`Connection.close()`	Releases the connection to the database. Closes any `Statement` created from this `Connection`.
`Statement.close()`	Releases this `Statement` resource. Closes any open `ResultSet` associated with this `Statement`.
`ResultSet.close()`	Releases this `ResultSet` resource. Note that any `ResultSetMetaData` objects created from the `ResultSet` are still accessible.
`Statement.executeXXXX()`	Any `ResultSet` associated with a previous `Statement` execution is automatically closed.

It is also a good practice to minimize the number of times you close and re-create `Connection` objects. As a rule, creating the connection to the database and passing the username and password credentials for authentication is a relatively expensive process, so performing the activity once for every SQL query can cause code to execute slowly. In fact, typically, database connections are created in a pool and connection instances are handed out to applications as needed, rather than allowing or requiring individual applications to create them.

`Statement` objects are less expensive to create. There are ways to precompile SQL statements using a `PreparedStatement`, which reduces the overhead associated with creating SQL query strings and sending those strings to the database for execution, but understanding `PreparedStatement` is no longer on the exam.

`ResultSets` are the least expensive of the objects to create, and as you saw in the section “ResultSets,” for results from a single table, you can use the `ResultSet` to update, insert, and delete rows, so it can be very efficient to use a `ResultSet`.

Let’s look at one of our previous examples, where we used a `Connection`,

a `Statement`, and a `ResultSet`, and rewrite this code to close the resources properly.

```
Connection conn = null;
String url, user, pwd;     // These are populated somewhere else
try {
  conn = DriverManager.getConnection(url, user, pwd);
  Statement stmt = conn.createStatement();
  ResultSet rs = stmt.executeQuery("SELECT * FROM Customer");
  // ... process the results
  // ...
  if (rs != null && stmt != null) {
    rs.close();           // Attempt to close the ResultSet
    stmt.close();         // Attempt to close the Statement
  }
} catch (SQLException se) {
  out.println("SQLException: " + se);
} finally {
  try {
    if (conn != null) {
      conn.close();       // Close the Connection
    }
  } catch (SQLException sec) {
    out.println("Exception closing connection!");
  }
}
```

Notice all the work we have to go through to close the `Connection`—we first need to make sure we actually got an object and not a null, and then we need to try the `close()` method inside of another `try` inside of the `finally` block! Fortunately, there is an easier way….

Using try-with-resources to Close Connections, Statements, and ResultSets

One of the most useful changes in Java SE 7 (JDK 7) was a number of small modifications to the language, including a new `try` statement to support automatic resource management. This language change is called `try`-with-resources, and its longer name belies how much simpler it makes writing code with resources that should be closed. The `try`-with-resources statement will automatically call the `close()` method on any resource declared in the parentheses at the end of the `try` block.

There is a caveat: A resource declared in the `try`-with-resource statement must implement the `AutoCloseable` interface. One of the changes for JDBC in Java SE 7 (JDBC 4.1) was the modification of the API so that `Connection`, `Statement`, and `ResultSet` all extend the `AutoCloseable` interface and support automatic resource management. So we can rewrite our previous code example using `try`-with-resources:

```
String url, user, pwd;  // These are populated somewhere else
try (Connection conn = DriverManager.getConnection(url, user, pwd)){
  Statement stmt = conn.createStatement();
  ResultSet rs = stmt.executeQuery("SELECT * FROM Customer");
  // ... process the results
  // ...
  if (rs != null && stmt != null) {
    rs.close();       // Attempt to close the ResultSet
    stmt.close();     // Attempt to close the Statement
  }
} catch (SQLException se) {
  out.println("SQLException: " + se);
}
```

Notice that we must include the object type in the declaration inside of the parentheses. The following will throw a compilation error:

```
try (conn = DriverManager.getConnection(url, user, pwd);) {
```

The `try`-with-resources can also be used with multiple resources, so you could include the `Statement` declaration in the `try` as well:

```
try (Connection conn = DriverManager.getConnection(url, user, pwd);
     Statement stmt = conn.createStatement()) {
```

Note that when more than one resource is declared in the `try`-with-resources statement, the resources are closed in the reverse order of their declaration—so `stmt.close()` will be called first, followed by `conn.close()`.

It probably makes sense that, if an exception is thrown from the `try` block, the exception will be caught by the `catch` statement, but what happens to exceptions thrown as a result of closing the resources in the `try`-with-

resources statement? Any exceptions thrown as a result of closing resources at the end of the `try` block are suppressed, if there was also an exception thrown in the `try` block. These exceptions can be retrieved from the exception thrown by calling the `getSuppressed()` method on the exception thrown.

For example:

```
} catch (SQLException se) {
  out.println("SQLException: " + se);
  Throwable[] suppressed = se.getSuppressed(); // Get an array of
                                               // suppressed
                                               // exceptions
  for (Throwable t: suppressed) {  // Iterate through the array
    out.println("Suppressed exception: " + t);
  }
}
```

CERTIFICATION SUMMARY

Core JDBC API

Remember that the JDBC API is a set of interfaces with one important concrete class, the `DriverManager` class. You write code using the well-defined set of JDBC interfaces, and the provider of your JDBC driver writes code implementations of those interfaces. The key (and, therefore, required) interfaces a JDBC driver must implement include `Driver`, `Connection`, `Statement`, and `ResultSet`.

The driver provider will also implement an instance of `DatabaseMetaData`, which you use to invoke a method to query the driver for information about the database and JDBC driver. One important piece of information is if the database is SQL-92 compliant, and there are a number of methods that begin

with “supports” to determine the capabilities of the driver. One important method is `supportsResultSetType()`, which is used to determine if the driver supports scrolling result sets.

DriverManager

The `DriverManager` is one of the few concrete classes in the JDBC API, and you will recall that the `DriverManager` is a factory class—using the `DriverManager`, you construct instances of `Connection` objects. In reality, the `DriverManager` simply holds references to registered JDBC drivers, and when you invoke the `getConnection()` method with a JDBC URL, the `DriverManager` passes the URL to each driver in turn. If the URL matches a valid driver, host, port number, username, and password, then that driver returns an instance of a `Connection` object. Remember that the JDBC URL is simply a string that encodes the information required to make a connection to a database.

How a JDBC driver is registered with the `DriverManager` is also important. In JDBC 4.0 and later, the driver jar file simply needs to be on the classpath, and the `DriverManager` will take care of finding the driver’s `Driver` class implementation and load that. JDBC 3.0 and earlier, require that the driver’s `Driver` class implementation be manually loaded using the `Class.forName()` method with the fully qualified class name of the class.

Statements and ResultSets

The most important use of a database is clearly using SQL statements and queries to create, read, update, and delete database records. The `Statement` interface provides the methods needed to create SQL statements and execute them. Remember that there are three different `Statement` methods to execute SQL queries: one that returns a result set, `executeQuery()`; one that returns an affected row count, `executeUpdate()`; and one general-purpose method that returns a boolean to indicate if the query produced a result set, `execute()`.

`ResultSet` is the interface used to read columns of data returned from a query, one row at a time. `ResultSet` objects represent a snapshot (a copy) of the data returned from a query, and there is a cursor that points to just above the first row when the results are returned. Unless you created a `Statement`

object using the `Connection.createStatement(int, int)` method that takes `resultSetType` and `resultSetConcurrency` parameters, `ResultSets` are not updatable and only allow the cursor to move forward through the results. However, if your database supports it, you can create a `Statement` object with a type of `ResultSet.TYPE_SCROLL_INSENSITIVE` and/or a concurrency of `ResultSet.CONCUR_UPDATABLE`, which allows any result set created with the `Statement` object to position the cursor anywhere in the results (scrollable) and allows you to change the value of any column in any row in the result set (updatable). Finally, when using a `ResultSet` that is scrollable, you can determine the number of rows returned from a query—and this is the only way to determine the row count because there is no "`rowCount`" method.

`SQLException` is the base class for exceptions thrown by JDBC, and because one query can result in a number of exceptions, the exceptions are chained. To determine all of the reasons a method call returned a `SQLException`, you must iterate through the exception by calling the `getNextException()` method. JDBC also keeps track of warnings for methods on `Connection`, `Statement`, and `ResultSet` objects using a `SQLWarning` exception type. Like `SQLException`, `SQLWarning` is silently chained to the object that caused the warning—for example, suppose that you attempt to create a `Statement` object that supports the scrollable `ResultSet`, but the database does not support that type. A `SQLWarning` will be added to the `Connection` object (the `Connection.createStatement(int, int)` method creates a `Statement` object). The `getWarnings()` method is used to return any `SQLWarnings`.

One of the important additions to Java SE 7 was the `try`-with-resources statement, and all of the JDBC interfaces have been updated to support the new `AutoCloseable` interface. However, bear in mind that there is an order of precedence when closing `Connections`, `Statements`, and `ResultSets`. So when a `Connection` is closed, any `Statement` created from that `Connection` is also closed, and likewise, when a `Statement` is closed, any `ResultSet` created using that `Statement` is also closed. And attempting to invoke a method on a closed object will result in a `SQLException`!

TWO-MINUTE DRILL

Here are some of the key points from the certification objectives in this chapter.

Core Interfaces of the JDBC API (OCP Objective 11.1)

- To be compliant with JDBC, driver vendors must provide implementations for the key JDBC interfaces: `Driver`, `Connection`, `Statement`, and `ResultSet`.
- `DatabaseMetaData` can be used to determine which SQL-92 level your driver and database support.
- `DatabaseMetaData` provides methods to interrogate the driver for capabilities and features.

Connect to a Database Using DriverManager (OCP Objective 11.2)

- The JDBC API follows a factory pattern, where the `DriverManager` class is used to construct instances of `Connection` objects.
- The JDBC URL is passed to each registered driver, in turn, in an attempt to create a valid `Connection`.
- Identify the Java statements required to connect to a database using JDBC.
- JDBC 3.0 (and earlier) drivers must be loaded prior to their use.
- JDBC 4.0 drivers just need to be part of the classpath, and they are automatically loaded by the `DriverManager`.

Submit Queries and Read Results from the Database (OCP Objective 11.3)

- The `next()` method must be called on a `ResultSet` before reading the first row of results.
- When a `Statement execute()` method is executed, any open `ResultSets` tied to that `Statement` are automatically closed.
- When a `Statement` is closed, any related `ResultSets` are also closed.
- `ResultSet` column indexes are numbered from 1, not 0.

- The default `ResultSet` is not updatable (read-only), and the cursor moves forward only.
- A `ResultSet` that is scrollable and updatable can be modified, and the cursor can be positioned anywhere within the `ResultSet`.
- `ResultSetMetaData` can be used to dynamically discover the number of columns and their type returned in a `ResultSet`.
- `ResultSetMetaData` does not have a row count method. To determine the number of rows returned, the `ResultSet` must be scrollable.
- `ResultSet` fetch size can be controlled for large data sets; however, it is a hint to the driver and may be ignored.
- `SQLExceptions` are chained. You must iterate through the exceptions thrown to get all of the reasons why an exception was thrown.
- `SQLException` also contains database-specific error codes and status codes.
- The `executeQuery` method is used to return a `ResultSet` (SELECT).
- The `executeUpdate` method is used to update data, to modify the database, and to return the number of rows affected (INSERT, UPDATE, DELETE, and DDLs).
- The `execute` method is used to perform any SQL command. A boolean `true` is returned when the query produces a `ResultSet` and `false` when there are no results, or if the result is an update count.
- There is an order of precedence in the closing of `Connections`, `Statements`, and `ResultSets`.
- Using the `try`-with-resources statement, you can close `Connections`, `Statements`, and `ResultSets` automatically (they implement the new `AutoCloseable` interface in Java SE 7).
- When a `Connection` is closed, all of the related `Statements` and `ResultSets` are closed.

SELF TEST

1 Given:

```
String url = "jdbc:mysql://SolDBServer/soldb";
String user = "sysEntry";
String pwd = "fo0B3@r";
// INSERT CODE HERE
Connection conn = DriverManager.getConnection(url, user, pwd);
```

Assuming `"org.gjt.mm.mysql.Driver"` is a legitimate class, which line, when inserted at `// INSERT CODE HERE`, will correctly load this JDBC 3.0 driver?

A. `DriverManager.registerDriver("org.gjt.mm.mysql.Driver");`

B. `Class.forName("org.gjt.mm.mysql.Driver");`

C. `DatabaseMetaData.loadDriver("org.gjt.mm.mysql.Driver");`

D. `Driver.connect("org.gjt.mm.mysql.Driver");`

E. `DriverManager.getDriver("org.gjt.mm.mysql.Driver");`

2 Given that you are working with a JDBC 4.0 driver, which three are required for this JDBC driver to be compliant?

A. Must include a `META-INF/services/java.sql.Driver` file

B. Must provide implementations of `Driver`, `Connection`, `Statement`, and `ResultSet` interfaces

C. Must support scrollable `ResultSets`

D. Must support updatable `ResultSets`

E. Must support transactions

F. Must support the SQL99 standard

G. Must support `PreparedStatement` and `CallableStatement`

3 Which three are available through an instance of `DatabaseMetaData`?

A. The number of columns returned

B. The number of rows returned

C. The name of the JDBC driver

D. The default transaction isolation level

E. The last query used

F. The names of stored procedures in the database

4 Given:

```
try {
  Statement stmt = conn.createStatement();
  String query =
         "SELECT * FROM Author WHERE LastName LIKE 'Rand%'";
  ResultSet rs = stmt.executeQuery(query);  // Line X
  if (rs == null) {                         // Line Y
    System.out.println("No results");
  } else {
    System.out.println(rs.getString("FirstName"));
  }
} catch (SQLException se) {
  System.out.println("SQLException");
}
```

Assuming a `Connection` object has already been created (conn) and that the query produces a valid result, what is the result?

A. Compiler error at line X

B. Compiler error at line Y

C. No result

D. The first name from the first row that matches ‘Rand%’

E. `SQLException`

F. A runtime exception

5 Given the SQL query:

```
String query = "UPDATE Customer SET EMail='John.Smith@comcast.net'
               WHERE CustomerID = 5000";
```

Assuming this is a valid SQL query and there is a valid `Connection` object (conn), which will compile correctly and execute this query?

A.
```
Statement stmt = conn.createStatement();
stmt.executeQuery(query);
```

B.
```
Statement stmt = conn.createStatement(query);
stmt.executeUpdate();
```

C.
```
Statement stmt = conn.createStatement();
stmt.setQuery(query);
stmt.execute();
```

D.
```
Statement stmt = conn.createStatement();
stmt.execute(query);
```

E.
```
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeUpdate(query);
```

6 Given:

```
try {
  ResultSet rs = null;
  try (Statement stmt = conn.createStatement()) {  // line X
    String query = "SELECT * from Customer";
    rs = stmt.executeQuery(query);                 // line Y
  } catch (SQLException se) {
    System.out.println("Illegal query");
  }
  while (rs.next()) {
    // print customer names
  }
} catch (SQLException se) {
  System.out.println("SQLException");
}
```

And assuming a valid `Connection` object (conn) and that the query will return results, what is the result?

A. The customer names will be printed out

B. Compiler error at line X

C. Illegal query

D. Compiler error at line Y

E. `SQLException`

F. Runtime exception

7 Which interfaces must a vendor implement to be JDBC compliant? (Choose all that apply.)

A. Query

B. Driver

C. ResultSet

D. Statement

E. Connection

F. SQLException

8 Given this code fragment:

```
Statement stmt = conn.createStatement();
ResultSet rs;
String query = "<QUERY HERE>";
stmt.execute(query);
if ((rs = stmt.getResultSet()) != null) {
  System.out.println("Results");
}
if (stmt.getUpdateCount() > -1) {
  System.out.println("Update");
}
```

Assuming each query is valid and that all tables have valid row data, which query statements entered into `<QUERY HERE>` produce the output that follows the query string (in the following answer? (Choose all that apply.)

A. `"SELECT * FROM Customer"`

Results

B. `"INSERT INTO Book VALUES ('1023456789', 'One Night in Paris', '1984-10-20', 'Hardcover', 13.95)"`

Update

C. `"UPDATE Customer SET Phone = '555-234-1021' WHERE CustomerID = 101"`

Update

D. `"SELECT Author.LastName FROM Author"`

Results

E. `"DELETE FROM Book WHERE ISBN = '1023456789'"`

Update

9 Which are true about queries that throw `SQLExceptions`? (Choose all that apply.)

A. A single query either executes correctly or throws a single exception

B. A single query can throw many exceptions

C. If a single query throws many exceptions, the exceptions can be captured by invoking `SQLException.getExceptions()`, which returns a `List`

D. If a single query encounters more than one exception-worthy problem, a `SQLException` is created for each problem encountered

E. If a single query throws many exceptions, the exceptions can be captured by iterating through the exceptions using `SQLException.getNextException()`

10 Which are true about the results of a query?

A. The results are stored in a `ResultSet` object

B. The results are stored in a `List` of type `Result`

C. By default, the results remain synchronized with the database

D. The results are accessed via iteration

E. Once you have the results, you can retrieve a given row without needing to iterate

F. The results are stored in a `List` of type `ResultSet`

A SELF TEST ANSWERS

1. ☑ **B** is correct. Prior to JDBC 4.0, JDBC drivers were required to register themselves with the `DriverManager` class by invoking `DriverManager.register(this);` after the driver was instantiated through a call from the classloader. The `Class.forName()` method calls the classloader, which, in turn, creates an instance of the class passed as a `String` to the method.

☒ **A** is incorrect because this method is meant to be invoked with an instance of a `Driver` class. **C** is incorrect because `DatabaseMetaData` does not have a `loadDriver` method, and the purpose of `DatabaseMetaData` is to return information about a database connection. **D** is incorrect because, again, while the method sounds right, the arguments are not of the right types, and this method is actually the one called by `DriverManager.getConnection` to get a `Connection` object. **E** is incorrect because although this method returns a `Driver` instance, one has to be loaded and registered with the `DriverManager` first. (OCP Objective 11.2)

2. ☑ **A, B,** and **E** are correct. To be JDBC 4.0 compliant, a JDBC driver must support the ability to autoload the driver by providing a file, `META-INF/services/java.sql.Driver`, that indicates the fully qualified class name of the `Driver` class that `DriverManager` should load on startup. The JDBC driver must implement the interfaces for `Driver`, `Connection`, `Statement`, `ResultSet`, and others. The driver must also

support transactions.

☒ **C** and **D** are incorrect. It is not a requirement to support scrollable or updatable `ResultSets`, although many drivers do. If, however, the driver reports that through `DatabaseMetaData` it supports scrollable and updatable `ResultSets`, then the driver must support all of the methods associated with cursor movement and updates. **F** is incorrect. The JDBC requires that the driver support SQL92 entry-level grammar and the SQL command DROP TABLE (from SQL92 Transitional Level). **G** is incorrect. Although JDBC 4.0 drivers must support `PreparedStatement`, `CallableStatement` is optional and only required if the driver returns true for the method `DatabaseMetaData.supportsStoredProcedures`. (OCP Objective 11.2)

3. ☑ **C, D,** and **F** are correct. `DatabaseMetaData` provides data about the database and the `Connection` object. The name, version, and other JDBC driver information are available, plus information about the database, including the names of stored procedures, functions, SQL keywords, and more. Finally, the default transaction isolation level and data about what transaction levels are supported are also available through `DatabaseMetaData`.

☒ **A** and **B** are incorrect, as they are really about the result of a query with the database. Column count is available through a `ResultSetMetaData` object, but a row count requires that you, as the developer, move the cursor to the end of a result set and then evaluate the cursor position. **E** is incorrect. There is no method defined to return the last query in JDBC. (OCP Objective 11.1)

4. ☑ **E** is correct. When the `ResultSet` returns, the cursor is pointing before the first row of the `ResultSet`. You must invoke the `next()` method to move to the next row of results *before* you can read any data from the columns. Trying to read a result using a `getXXXX` method will result in a `SQLException` when the cursor is before the first row or after the last row.

☒ **A, B, D,** and **F** are incorrect based on the above. Note about **C**: the `ResultSet` returned from `executeQuery` will never be null. (OCP Objective 11.3)

5. ☑ **D** is correct.

   ☒ Note that answer **E** is close, but will not compile because the `executeUpdate(query)` method returns an integer result. **A** will compile correctly, but throw a `SQLException` at runtime—the `executeQuery` method cannot be used on INSERT, UPDATE, DELETE, or DDL SQL queries. **B** will not compile because the `createStatement` method does not take a `String` argument for the query. **C** is incorrect because `Statement` does not have a `setQuery` method and this fragment will not compile. (OCP Objective 11.3)

6. ☑ **E** is correct. Recall that the `try`-with-resources statement on line X will automatically close the resource specified at the close of the `try` block (when the closing curly brace is reached) and closing the `Statement` object automatically closes any open `ResultSets` associated with the `Statement`. The `SQLException` thrown is that the `ResultSet` is not open. To fix this code, move the `while` statement into the `try`-with-resources block.

   ☒ **A, B, C, D,** and **F** are incorrect based on the above. (OCP Objective 11.3)

7. ☑ **B, C, D,** and **E** are correct.

   ☒ **A** and **F** are incorrect. They are not interfaces required by JDBC. To query a database, the `Statement` interface is used, not the plausibly named, but mythical, `Query` interface. (OCP Objective 11.1)

8. ☑ All of the answers are correct (**A, B, C, D, E**). SELECT statements will produce a `ResultSet` even if there are no rows. INSERT, UPDATE, and DELETE statements all produce an update count, even when the number of rows affected is 0. (OCP Objective 9.3)

9. ☑ **B, D,** and **E** are correct.

   ☒ **A** is incorrect because a single query can throw many exceptions. **C** is incorrect; when many exceptions are thrown, you must use `getNextException()` to iterate through them. (OCP Objective 11.3)

10. ☑ **A, D,** and **E** are correct. For **E** you can use, for example, `getRow()`.

    ☒ **B** is incorrect; results are stored in a `ResultSet` object. **C** is incorrect; once a `ResultSet` is created, it does NOT stay synchronized with the

database unless the `ResultSet` was created with one of the `CONCUR_XXX` cursor types. **F** is incorrect based on the above. (OCP Objective 11.3)