CHAPTER 1

What Is JDBC Programming?

It is the theory that decides what we can observe.

Albert Einstein

This chapter explains JDBC programming by using a set of questions and answers. Java and JDBC are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries. According to Sun Microsystems, JDBC is not an acronym and does not stand for Java Database Connectivity (but the fact of the matter is that most Java engineers believe that JDBC stands for **J**ava **D**ata**B**ase **C**onnectivity).

JDBC is a platform-independent interface between relational databases and Java. In today's Java world, JDBC is a standard API for accessing enterprise data in relational databases (such as Oracle, MySQL, Sybase, PostgreSQL, and DB2) using SQL (Structured Query Language). In this chapter, we will examine the basic aspects of JDBC, and save the details about JDBC metadata for upcoming chapters. Data and metadata (data about data/information) are at the heart of most business applications, and JDBC deals with data and metadata stored and manipulated in relational database systems (RDBMSs). Note that each RDBMS has a lot of metadata, and JDBC maps some of those metadata in a uniform and consistent fashion by its API.

Note In using the word *metadata*, we must use the exact term when it is a Java API (since Java is a case-sensitive language)—for example, `DatabaseMetaData`, `RowSetMetaData`, and `ResultSetMetaData`—but in our discussion and descriptions, we will use *metadata* (and not *MetaData*).

This book takes an examples-based approach to describing the metadata features available in JDBC (such as getting a list of tables or views, or getting a signature of a stored procedure). Whether you are a new or an experienced database or JDBC developer, you should find the examples and accompanying text a valuable and accessible knowledge base for creating your own database solutions. Using JDBC's database metadata, you can generate GUI/web-based applications (for example, see `http://dev2dev.bea.com/lpt/a/257`). Also, you can develop web entry forms based on metadata (for example, see `http://www.elet.polimi.it/conferences/wq04/final/paper03.pdf`).

In this book, we use some basic Java/JDBC utility classes (such as the `DatabaseUtil` class), which are available for download from the Source Code section of the Apress website. The `DatabaseUtil` class provides methods for closing JDBC objects (such as `Connection`, `ResultSet`,

Statement, and PreparedStatement). The reason for using the DatabaseUtil class is to make the code compact and more readable (for example, closing a ResultSet object by DatabaseUtil takes one line of code versus a couple of lines without using DatabaseUtil).

VeryBasicConnectionManager is a very simple class that provides Connection objects for Oracle and MySQL by using getConnection(dbVendor). In real production applications, the VeryBasicConnectionManager class is not an acceptable solution and should be replaced by a connection pool manager (such as the Excalibur from http://excalibur.apache.org/ and the commons-dbcp package from http://jakarta.apache.org/commons/dbcp/). We use these classes to demonstrate JDBC concepts for different vendors such as Oracle and MySQL. Connection pooling is a technique used for reusing and sharing Connection objects among requesting clients.

The remaining chapters in this book will deal with JDBC metadata and nothing but JDBC metadata.

1.1. What Is JDBC?

JDBC is a set of programming APIs that allows easy connection to a wide range of databases (especially relational databases) through Java programs. In this book, we will be using JDBC 2.0 and 3.0 versions (JDBC 4.0 is just a specification and has not been implemented extensively yet.) In Java 2 Platform Standard Edition (J2SE) 5.0 (which supports JDBC 3.0), the JDBC API is defined by two packages:

- java.sql provides the API for accessing and processing data stored in a data source (usually a relational database) using the Java programming language. This package provides the foundation and most commonly used objects (such as Connection, ResultSet, Statement, and PreparedStatement). Also, this package provides classes and interfaces to get both database and result set metadata from the database server. This package has a set of classes and interfaces (such as DatabaseMetaData and ResultSetMetaData) that deal with database metadata, which will be one of the focuses of this book.

- javax.sql provides the API for server-side data source access. According to the Java Development Kit (JDK) documentation, "This package supplements the java.sql package and, as of the version 1.4 release, is included in the JDK. It remains an essential part of the Java 2 SDK, Enterprise Edition (J2EE)." This package provides services for J2EE (such as DataSource and RowSets). Also, the package has a set of classes and interfaces (such as RowSetMetaData) that deal with row set metadata. In this book we focus on the metadata components of this package.

In a nutshell, JDBC is a database-independent API for accessing a relational database. You pass SQL to Java methods in the JDBC classes (the packages java.sql and javax.sql) and get back JDBC objects (such as ResultSet, DatabaseMetaData, and ResultSetMetaData) that represent the results of your query. JDBC is designed so simply that most database programmers need learn only a few methods to accomplish most of what they need to do.

Figure 1-1 shows how a database application (such as a Java application/applet/servlet) uses JDBC to interact with one or more databases.

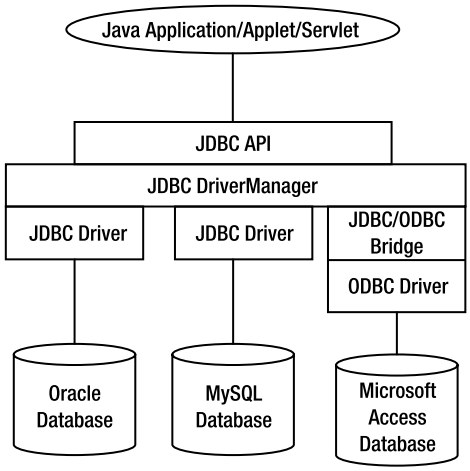

Figure 1-1. *Java database application using JDBC*

Figure 1-1 presents the basic outline of the JDBC architecture. JDBC's `DriverManager` class provides the basic service for managing a set of JDBC drivers. The `DriverManager` loads JDBC drivers in memory, and can also be used to create `java.sql.Connection` objects to data sources (such as Oracle and MySQL). For more details, refer to *JDBC Recipes: A Problem-Solution Approach* (Apress, 2005) and Sun's official site on JDBC: `http://java.sun.com/products/jdbc/index.jsp`.

Note that you can have more than one driver and therefore more than one database. Figure 1-2 illustrates how a Java application uses JDBC to interact with one or more relational databases (such as Oracle and MySQL) without knowing about the underlying JDBC driver implementations. Figure 1-2 illustrates the core JDBC classes and interfaces that interact with Java and JDBC applications. This figure also shows the basic relationships of the `DatabaseMetaData` and `ResultSetMetaData` interfaces with other JDBC objects.

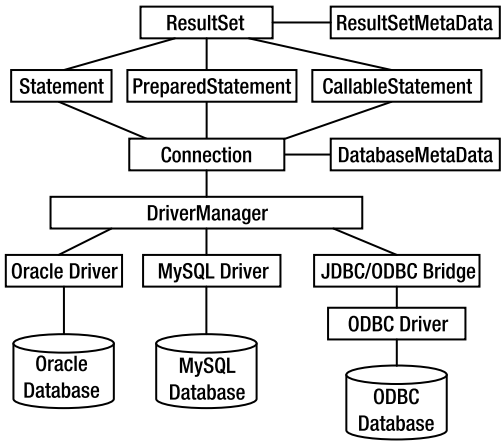

Figure 1-2. *Using JDBC database metadata*

The following are core JDBC classes, interfaces, and exceptions in the `java.sql` package:

- `DriverManager`: This class loads JDBC drivers in memory. It is a "factory" class and can also be used to create `java.sql.Connection` objects to data sources (such as Oracle, MySQL, etc.).
- `Connection`: This interface represents a connection with a data source. The `Connection` object is used for creating `Statement`, `PreparedStatement`, and `CallableStatement` objects.
- `DatabaseMetaData`: This interface provides detailed information about the database as a whole. The `Connection` object is used for creating `DatabaseMetaData` objects.
- `Statement`: This interface represents a static SQL statement. It can be used to retrieve `ResultSet` objects.
- `PreparedStatement`: This interface extends `Statement` and represents a precompiled SQL statement. It can be used to retrieve `ResultSet` objects.
- `CallableStatement`: This interface represents a database stored procedure. It can execute stored procedures in a database server.
- `ResultSet`: This interface represents a database result set generated by using SQL's `SELECT` statement. `Statement`, `PreparedStatement`, `CallableStatement`, and other JDBC objects can create `ResultSet` objects.
- `ResultSetMetaData`: This interface provides information about the types and properties of the columns in a `ResultSet` object.
- `SQLException`: This class is an exception class that provides information on a database access error or other errors.

1.2. What Is JDBC Programming?

JDBC programming can be explained in the following simple steps:

- Importing required packages
- Registering the JDBC drivers
- Opening a connection to a database
- Creating a `Statement/PreparedStatement/CallableStatement` object
- Executing a SQL query and returning a `ResultSet` object
- Processing the `ResultSet` object
- Closing the `ResultSet` and `Statement` objects
- Closing the connection

The first hands-on experience with JDBC in this book involves a basic and complete example to illustrate the overall concepts related to creating and accessing data in a database. Here, we assume that a database (MySQL or Oracle) is created and available for use. Database creation is DBMS-specific. This means that each vendor has a specific set of commands for

creating a database. For this obvious reason, database creation commands are not portable (this is the main reason why JDBC has stayed away from database creation commands).

Step 1: Import the Required Packages

Before you can use the JDBC driver to get and use a database connection, you must import the following packages: `java.sql` and `javax.sql`.

```
import java.sql.*;
import javax.sql.*;
```

To import classes and interfaces properly, refer to the Java Language Specification (`http://java.sun.com/docs/books/jls/`).

Step 2: Register the JDBC Drivers

A JDBC driver is a set of database-specific implementations for the interfaces defined by JDBC. These driver classes come into being through a bootstrap process. This is best shown by stepping through the process of using JDBC to connect to a database,

Let's use Oracle's type 4 JDBC driver as an example. First, the main driver class must be loaded into the Java virtual machine (VM):

```
Class.forName("oracle.jdbc.driver.OracleDriver");
```

The specified driver (i.e., the `oracle.jdbc.driver.OracleDriver` class) must implement the `java.sql.Driver` interface. A class initializer (a static code block) within the `oracle.jdbc.driver.OracleDriver` class registers the driver with the `java.sql.DriverManager`.

Now let's use MySQL's JDBC driver as an example. First, the main driver class must be loaded into the Java VM:

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

The specified driver (i.e., the `com.mysql.jdbc.Driver` class) must implement the `java.sql.Driver` interface. A class initializer (a static code block) within the `com.mysql.jdbc.Driver` class registers the driver with the `java.sql.DriverManager`. Note that when loading a JDBC driver, you must make sure that the database-specific driver API (usually a JAR file) is in your CLASSPATH environment variable. Each database vendor has its own specific implementation of the JDBC driver.

Step 3: Opening a Connection to a Database

Next, we need to obtain a connection to the database. To get a connection object to a database, you need at least three pieces of information: the database URL, the database username, and the database user's password.

```
import java.sql.Connection;
import java.sql. DriverManager;
…
String dbURL = "jdbc:oracle:thin:@localhost:1521:kitty";
String dbUsername = "scott";
String dbPassword = "tiger";
Connection conn = DriverManager.getConnection(dbURL, dbUsername, dbPassword);
```

DriverManager determines which registered driver to use by invoking the acceptsURL(String url) method of each driver, passing each the JDBC URL. The first driver to return true in response will be used for this connection. In this example, OracleDriver will return true, so DriverManager then invokes the connect() method of OracleDriver to obtain an instance of OracleConnection. It is this database-specific connection instance implementing the java.sql.Connection interface that is passed back from the java.sql.DriverManager.getConnection() call.

There is an alternate method for creating a database connection: first get a JDBC driver, then use that driver to get a connection:

```
import java.sql.Connection;
import java.sql. Driver;
import java.sql. DriverManager;
import java.util.Properties;
…
String dbURL = "jdbc:oracle:thin:@localhost:1521:kitty";
String dbUsername = "scott";
String dbPassword = "tiger";
Properties dbProps = new Properties();
String driverName = "oracle.jdbc.driver.OracleDriver";
Driver jdbcDriver = (Driver) Class.forName(driverName).newInstance();
dbProps.put("user", dbUsername);
dbProps.put("password", dbPassword);
Connection conn = jdbcDriver.connect(databaseURL, dbProps);
```

Step 4: Creating a Statement/PreparedStatement/CallableStatement Object

Once you have a valid java.sql.Connection object, you can create statement objects (such as Statement, PreparedStatement, and CallableStatement). The bootstrap process continues when you create a statement:

```
Connection conn = <get-a-valid-Connection-object>;
Statement stmt = conn.createStatement();
```

In order to do something useful with a database, we create the following table:

```
create table MyEmployees (
   id INT PRIMARY KEY,
   firstName VARCHAR(20),
   lastName VARCHAR(20),
   title VARCHAR(20),
   salary INT
);
```

Then we insert two records:

```
insert into MyEmployees(id, firstName, lastName, title, salary)
   values(60,  'Bill',  'Russel',   'CTO',    980000);

insert into MyEmployees(id, firstName, lastName, title, salary)
   values(70, 'Alex',  'Baldwin',   'Software Engineer', 88000);
```

The connection reference points to an instance of `OracleConnection`. This database-specific implementation of `Connection` returns a database-specific implementation of `Statement`, namely `OracleStatement`.

Step 5: Executing a Query and Returning a ResultSet Object

Invoking the `execute()` method of this statement object will execute the database-specific code necessary to issue a SQL statement against the Oracle database and retrieve the results (as a table):

```
String query = "SELECT id, lastName FROM MyEmployees";
ResultSet result = stmt.executeQuery(query);
```

The result is a table (as a `ResultSet` object) returned by executing the `SELECT` statement. Again, what is actually returned is an instance of `OracleResultSet`, which is an Oracle-specific implementation of the `java.sql.ResultSet` interface. By iterating the result, we can get all of the selected records.

So the purpose of a JDBC driver is to provide these implementations that hide all the database-specific details behind standard Java interfaces.

Step 6: Processing the ResultSet Object

`ResultSet.next()` returns a boolean: `true` if there is a next row or record and `false` if not (meaning the end of the data or set has been reached). Conceptually, a pointer or cursor is positioned just before the first row when the `ResultSet` is obtained. Invoking the `next()` method moves to the first row, then the second, and so on.

Once positioned at a row, the application can get the data on a column-by-column basis using the appropriate `ResultSet.getXXX()` method. Here are the methods used in the example to collect the data:

```
if (rs.next()) {
   String firstName = rs.getString(1);
   String lastName = rs.getString(2);
   String title = rs.getString(3);
   int salary = rs.getInt(4);
}
```

or we may use the column names (instead of column positions):

```
if (rs.next()) {
   String firstName = rs.getString("firstName");
   String lastName = rs.getString("lastName");
   String title = rs.getString("title");
   int salary = rs.getInt("salary");
}
```

The order of the `getString(columnNumber)` should be the same as the order of columns selected in the SQL `SELECT` statement; otherwise, we could run into an error.

Step 7: Closing JDBC Objects

Releasing or closing JDBC resources (such as `ResultSet`, `Statement`, `PreparedStatement`, and `Connection` objects) immediately instead of waiting for it to happen on its own can improve the overall performance of your application. From a good software engineering point of view, you should put `close()` statements in a `finally` clause, because it guarantees that the statements in the `finally` clause will be executed as the last step regardless of whether an exception has occurred.

Closing ResultSet

`ResultSet` has a `close()` method that releases the `ResultSet` object's database and JDBC resources immediately instead of waiting for that to happen when it is automatically closed. Another major reason to close the `ResultSet` objects immediately after they are done is that we increase concurrency; as long as the `ResultSet` object is open, the DBMS internally holds a lock.

Here is some sample code for closing a `ResultSet` object. It is always a good idea to have utility classes to close these JDBC resources, and the following method can do the job:

```
/**
 * Close the ResultSet object.  Releases the
 * ResultSet object's database and JDBC resources
 * immediately instead of waiting for them to be
 * automatically released.
 * @param rs a ResultSet object.
 */
public static  void close(java.sql.ResultSet rs) {
   if (rs == null) {
      return;
   }

   try {
      rs.close();
      // result set is closed now
   }
   catch(Exception ignore) {
      // ignore the exception
      // could not close the result set
      // cannot do much here
   }
}
```

Closing Statement

`Statement` has a `close()` method, which releases this `Statement` object's database and JDBC resources immediately instead of waiting for this to happen when it is automatically closed. Here is some sample code for closing a `Statement` object. It is always a good idea to have utility classes to close these JDBC resources, and the following method can do the job:

```
/**
 * Close the Statement object.  Releases the Statement
 * object's database and JDBC resources immediately instead
 * of waiting for them to be automatically released.
 * @param stmt a Statement object.
 */
public static  void close(java.sql.Statement stmt) {
   if (stmt == null) {
      return;
   }

   try {
      stmt.close();
      // result set is closed now
   }
   catch(Exception ignore) {
      // ignore the exception
      // could not close the statement
      // can not do much here
   }
}
```

Closing PreparedStatement

`PreparedStatement` does not have a direct `close()` method, but since `PreparedStatement` extends `Statement`, then you may use the `Statement.close()` method for `PreparedStatement` objects. It is always a good idea to have utility classes to close these JDBC resources, and the following method can do the job:

```
/**
 * Close the PreparedStatement object.  Releases the
 * PreparedStatement object's database and JDBC
 * resources immediately instead of waiting for them
 * to be automatically released.
 * @param pstmt a PreparedStatement object.
 */
public static  void close(java.sql.PreparedStatement pstmt) {
   if (pstmt == null) {
      return;
   }

   try {
      pstmt.close();
      // PreparedStatement object is closed now
   }
```

```
    catch(Exception ignore) {
       // ignore the exception
       // could not close the PreparedStatement
       // can not do much here
    }
}
```

Closing Connection

If you are using a connection pool manager to manage a set of database connection objects, then you need to release the `Connection` object to the connection pool manager (this is called a "soft" close). Alternatively, you can use the `close()` method, which releases the `Connection` object's database and JDBC resources immediately instead of waiting for them to be automatically released. Here is some sample code for closing a `Connection` object. It is always a good idea to have utility classes to close these JDBC resources, and the following method can do the job:

```
/**
 * Close the Connection object.  Releases the Connection
 * object's database and JDBC resources immediately instead
 * of waiting for them to be automatically released.
 * @param conn a Connection object.
 */
public static  void close(java.sql.Connection conn) {
   if (conn == null) {
      return;
   }

   try {
      if (!conn.isClosed()) {
         // close the connection-object
         conn.close();
      }
      // connection object is closed now
   }
   catch(Exception ignore) {
      // ignore the exception
      // could not close the connection-object
      // can not do much here
   }
}
```

1.3. How Do You Handle JDBC Errors/Exceptions?

In JDBC, errors and exceptions are identified by the `java.sql.SQLException` class (which extends the `java.lang.Exception` class). `SQLException` is a *checked* exception. There are two types of exceptions in Java: checked and unchecked, or runtime, exceptions. A checked exception is a subclass of `java.lang.Throwable` (the `Throwable` class is the superclass of all errors and

exceptions in the Java language) but not of RunTimeException (RuntimeException is the superclass of those exceptions that can be thrown during the normal operation of the Java VM). Checked exceptions have to be caught (and handled properly) or appear in a method that specifies in its signature that it throws that kind of exception.

When a JDBC object (such as Connection, Statement, or ResultSet) encounters a serious error, it throws a SQLException. For example, an invalid database URL, an invalid database username or password, database connection errors, malformed SQL statements, an attempt to access a nonexistent table or view, and insufficient database privileges all throw SQLException objects.

The client (the database application program) accessing a database server needs to be aware of any errors returned from the server. JDBC give access to such information by providing several levels of error conditions:

- SQLException: SQLExceptions are Java exceptions that, if not handled, will terminate the client application. SQLException is an exception that provides information on a database access error or other errors.
- SQLWarning: SQLWarnings are subclasses of SQLException, but they represent nonfatal errors or unexpected conditions, and as such, can be ignored. SQLWarning is an exception that provides information on database access warnings. Warnings are silently chained to the object whose method caused it to be reported.
- BatchUpdateException: BatchUpdateException is an exception thrown when an error occurs during a batch update operation. In addition to the information provided by SQLException, a BatchUpdateException provides the update counts for all commands that were executed successfully during the batch update, that is, all commands that were executed before the error occurred. The order of elements in an array of update counts corresponds to the order in which commands were added to the batch.
- DataTruncation: DataTruncation is an exception that reports a DataTruncation warning (on reads) or throws a DataTruncation exception (on writes) when JDBC unexpectedly truncates a data value.

The SQLException class extends the java.lang.Exception class and defines an additional method called getNextException(). This allows JDBC classes to chain a series of SQLException objects together. In addition, the SQLException class defines the getMessage(), getSQLState(), and getErrorCode() methods to provide additional information about an error or exception. In general, a JDBC client application might have a catch block that looks something like this:

```
String dbURL = ...;
String dbUser = ...;
String dbPassword = ...;
Connection conn = null;
try {
   conn = DriverManager.getConnection(dbURL, dbUser, dbPassword);
   //
   // when you are here, it means that an exception has not
   // happened and you can use the connection object
   // (i.e., conn) to do something useful with the database
   ...
}
```

```
catch (SQLException e) {
   // something went wrong: maybe dbUser/dbPassword is not defined
   // maybe te dbURL is malformed, and other possible reasons.
   // now handle the exception, maybe print the error code
   // and maybe log the error, ...
   while(e != null) {
      System.out.println("SQL Exception/Error:");
      System.out.println("error message=" + e.getMessage());
      System.out.println("SQL State= " + e.getSQLState());
      System.out.println("Vendor Error Code= " + e.getErrorCode());
      // it is possible to chain the errors and find the most
      // detailed errors about the exception
      e = e.getNextException(  );
   }
}
catch (Exception e2) {
   // handle non-SQL exception …
}
```

To understand transaction management, you need to understand the `Connection.setAutoCommit()` method. Its signature is

```
void setAutoCommit(boolean autoCommit) throws SQLException
```

According to J2SE 1.5, `setAutoCommit()` sets this connection's autocommit mode to the given state. If a connection is in autocommit mode, then all its SQL statements will be executed and committed as individual transactions. Otherwise, its SQL statements are grouped into transactions that are terminated by a call to either the `commit()` or the `rollback()` method. By default, new connections are in autocommit mode.

The following example shows how to handle `commit()` and `rollback()` when an exception happens:

```
String dbURL = ...;
String dbUser = ...;
String dbPassword = ...;
Connection conn = null;
try {
   conn = DriverManager.getConnection(dbURL, dbUser, dbPassword);
   conn.setAutoCommit(false);  // begin transaction
   stmt.executeUpdate("CREATE TABLE cats_tricks(" +
      "name VARCHAR(30), trick VARHAR(30))") ;
   stmt.executeUpdate("INSERT INTO cats_tricks(name, trick) " +
      "VALUES('mono', 'rollover')") ;
   conn.commit() ;  // commit/end transaction
   conn.setAutoCommit(true) ;
}
```

```
catch(SQLException e) {
   // print some useful error messages
   System.out.println("SQL Exception/Error:");
   System.out.println("error message=" + e.getMessage());
   System.out.println("SQL State= " + e.getSQLState());
   System.out.println("Vendor Error Code= " + e.getErrorCode());
   // rollback the transaction
   // note that this rollback will not undo the creation of
   // DDL statements (such as CREATE TABLE statement)
   conn.rollback();
   // optionally, you may set the auto commit to "true"
   conn.setAutoCommit(true) ;
}
```

In the following example we force the exception to happen: instead of VARCHAR (when creating the cats_tricks table), we type VARZCHAR (the database server will not understand VARZCHAR and therefore it will throw an exception):

```
String dbURL = ...;
String dbUser = ...;
String dbPassword = ...;
Connection conn = null;
try {
   conn = DriverManager.getConnection(dbURL, dbUser, dbPassword);
   conn.setAutoCommit(false);
   stmt.executeUpdate("CREATE TABLE cats_tricks("+
      "name VARZCHAR(30), trick VARHAR(30))") ;
   stmt.executeUpdate("INSERT INTO cats_tricks(name, trick) "+
      "VALUES('mono', 'rollover')") ;
   conn.commit() ;
   conn.setAutoCommit(true) ;
}
catch(SQLException e) {
   // print some useful error messages
   System.out.println("SQL Exception/Error:");
   System.out.println("error message=" + e.getMessage());
   System.out.println("SQL State= " + e.getSQLState());
   System.out.println("Vendor Error Code= " + e.getErrorCode());
   // rollback the transaction
   conn.rollback();
   // optionally, you may set the auto commit to "true"
   conn.setAutoCommit(true) ;
}
```

1.4. What Is JDBC Metadata Programming?

JDBC metadata programming is very similar to JDBC programming with one exception: you deal with metadata instead of data. For example, in JDBC programming you are interested in getting employee data and then processing it, but in JDBC metadata programming, you are not interested in getting actual data, but you want to get metadata (data about data, such as the table names in a database).

In JDBC metadata programming, we're interested in database metadata and result set metadata. For example, for metadata, we want answers to such questions as

- What is the list of tables or views available in the database?
- What are the names and types of the columns in tables or views?
- What is the signature of a specific stored procedure?

In JDBC, several key interfaces comprise the metadata portion:

- `DatabaseMetaData`: Provides information about the database as a whole.
- `ResultSetMetaData`: Used to identify the types and properties of the columns in a `ResultSet` object.
- `RowSetMetaData`: An object that contains information about the columns in a `RowSet` object. This interface is an extension of the `ResultSetMetaData` interface with methods for setting the values in a `RowSetMetaData` object.
- `ParameterMetaData`: An object that can be used to get information about the types and properties of the parameters in a `PreparedStatement` object.
- `DriverPropertyInfo`: `Driver` properties for making a connection. The `DriverPropertyInfo` class is of interest only to advanced programmers who need to interact with a `Driver` via the method `getDriverProperties()` to discover and supply properties for connections.

The remaining chapters will dissect these metadata classes and interfaces and will provide detailed information about using them. Figure 1-3 shows the creation of metadata interfaces and classes.

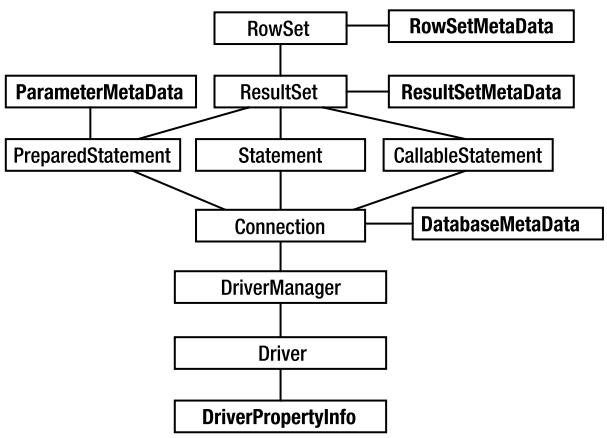

Figure 1-3. *JDBC's metadata classes and interfaces*

Database metadata is most often used by tools or developers who write programs that use advanced, nonstandard database features and by programs that dynamically discover database objects such as schemas, catalogs, tables, views, and stored procedures.

Some of the `DatabaseMetaData` methods return results in the form of a `ResultSet` object. The question is how should we handle `ResultSet` objects as return values? Metadata is retrieved from these `ResultSet` objects using the normal `ResultSet.getXXX()` methods, such as `getString()` and `getInt()`. To make the metadata useful for any kind of clients, I have mapped these `ResultSet` objects into XML.

1.5. What Is an Example of JDBC Metadata Programming?

Suppose you want to get the major and minor JDBC version number for the driver you are using. You can use the following methods from `DatabaseMetaData`:

```
int  getDriverMajorVersion()
 // Retrieves this JDBC driver's major version number.

int  getDriverMinorVersion()
 // Retrieves this JDBC driver's minor version number.
```

As you can see, these methods do not return the employee or inventory data, but they return metadata.

Our solution combines these into a single method and returns the result as an XML `String` object, which any client can use. The result has the following syntax:

```
<?xml version='1.0'>
<DatabaseInformation>
    <majorVersion>database-major-version</majorVersion>
    <minorVersion>database-minor-version</minorVersion>
    <productName>database-product-name</productName>
    <productVersion>database-product-version</productVersion>
</DatabaseInformation>
```

The Solution

The solution is generic and can support MySQL, Oracle, and other relational databases. If `Connection.getMetaData()` returns `null`, then it means that vendor's driver does not support/implement the`java.sql.DatabaseMetaData` interface. Note that the `getDatabaseMajorVersion()` method (implemented by the `oracle.jdbc.OracleDatabaseMetaData` class) is an unsupported feature; therefore, we have to use a `try-catch` block. If the method returns a `SQLException`, we return the message "unsupported feature" in the XML result:

```
import java.sql.Connection;
import java.sql.DatabaseMetaData;
…
/**
 * Get database product name and version information.
 * This method calls 4 methods (getDatabaseMajorVersion(),
 * getDatabaseMinorVersion(), getDatabaseProductName(),
 * getDatabaseProductVersion()) to get the required information
 * and it represents the information as an XML.
```

```
 *
 * @param conn the Connection object
 * @return database product name and version information
 * as an XML document (represented as a String object).
 */
public static String getDatabaseInformation(Connection conn)
    throws Exception {
        DatabaseMetaData meta = conn.getMetaData();
        if (meta == null) {
           return null;
        }

        StringBuffer sb = new StringBuffer("<?xml version='1.0'>");
        sb.append("<DatabaseInformation>");

        // Oracle (and some other vendors) do not
        // support some of the following methods
        // (such as getDatabaseMajorVersion() and
        // getDatabaseMajorVersion()); therefore,
        // we need to use try-catch block.
        try {
           int majorVersion = meta.getDatabaseMajorVersion();
           appendXMLTag(sb, "majorVersion", majorVersion);
        }
        catch(Exception e) {
           appendXMLTag(sb, "majorVersion", "unsupported feature");
        }

        try {
           int minorVersion = meta.getDatabaseMinorVersion();
           appendXMLTag(sb, "minorVersion", minorVersion);
        }
        catch(Exception e) {
           appendXMLTag(sb, "minorVersion", "unsupported feature");
        }

        String productName = meta.getDatabaseProductName();
        String productVersion = meta.getDatabaseProductVersion();
        appendXMLTag(sb, "productName", productName);
        appendXMLTag(sb, "productVersion", productVersion);
        sb.append("</DatabaseInformation>");
        return sb.toString();
    }
}
```

Client: Program for Getting Database Information

```
import java.util.*;
import java.io.*;
import java.sql.*;

import jcb.util.DatabaseUtil;
import jcb.meta.DatabaseMetaDataTool;
import jcb.db.VeryBasicConnectionManager;

public class TestDatabaseMetaDataTool_DatabaseInformation {

    public static void main(String[] args) {
        String dbVendor = args[0];  // { "mysql", "oracle" }
        Connection conn = null;
        try {
            conn = VeryBasicConnectionManager.getConnection(dbVendor);
            System.out.println("-------- getDatabaseformation -------------");
            System.out.println("conn="+conn);
            String dbInfo = DatabaseMetaDataTool.getDatabaseInformation(conn);
            System.out.println(dbInfo);
            System.out.println("-----------------------------------");
        }
        catch(Exception e){
            e.printStackTrace();
            System.exit(1);
        }
        finally {
            DatabaseUtil.close(conn);
        }
    }
}
```

Running the Solution for a MySQL Database

```
$ javac TestDatabaseMetaDataTool_DatabaseInformation.java
$ java TestDatabaseMetaDataTool_DatabaseInformation mysql
```

```
-------- getDatabaseformation -------------
conn=com.mysql.jdbc.Connection@1837697
<?xml version='1.0'>
<DatabaseInformation>
    <majorVersion>4</majorVersion>
    <minorVersion>0</minorVersion>
    <productName>MySQL</productName>
    <productVersion>4.0.4-beta-max-nt</productVersion>
</DatabaseInformation>
-----------------------------------
```

Running the Solution for an Oracle Database

The following output is formatted to fit the page:

```
$ javac TestDatabaseMetaDataTool_DatabaseInformation.java
$ java TestDatabaseMetaDataTool_DatabaseInformation oracle
```

```
-------- getDatabaseInformation -------------
conn= oracle.jdbc.driver.OracleConnection@169ca65
<?xml version='1.0'>
<DatabaseInformation>
     <majorVersion>unsupported feature</majorVersion>
     <minorVersion>unsupported feature</minorVersion>
     <productName>Oracle</productName>
     <productVersion>
         Oracle9i Enterprise Edition Release 9.2.0.1.0 - Production
         With the Partitioning, OLAP and Oracle Data Mining options
         JServer Release 9.2.0.1.0 - Production
     </productVersion>
</DatabaseInformation>
-----------------------------------
```