14

Java

Java is one of the simplest languages in which you can write MySQL applications. Its database access API JDBC (Java DataBase Connectivity) is one of the more mature database-independent database access APIs in common use. Most of what we cover in this chapter can be applied to Oracle, Sybase, MS SQL Server, mSQL, and any other database engine as well as MySQL. In fact, nearly none of the MySQL-specific information in this chapter has anything to do with coding. Instead, the "proprietary" information relates only to downloading MySQL support for JDBC and configuring the runtime environment. Everything else is largely independent of MySQL, excepting for features not supported by MySQL like transactions.

In this chapter, we assume a basic understanding of the Java programming language and Java concepts. If you do not already have this background, we strongly recommend taking a look at *Learning Java* (O'Reilly & Associates, Inc.). For more details on how to build the sort of three-tier database applications we discussed in Chapter 6, *Database Applications*, take a look at *Database Programming with JDBC and Java, 2nd Edition* (O'Reilly & Associates, Inc.).

The JDBC API

Like all Java APIs, JDBC is a set of classes and interfaces that work together to support a specific set of functionality. In the case of JDBC, this functionality is naturally database access. The classes and interfaces that make up the JDBC API are thus abstractions from concepts common to database access for any kind of database. A `Connection`, for example, is a Java interface representing a database connection. Similarly, a `ResultSet` represents a result set of data returned from a SQL `SELECT` statement. Java puts the classes that form the JDBC API together in the `java.sql` package which Sun introduced in JDK 1.1.

The underlying details of database access naturally differ from vendor to vendor. JDBC does not actually deal with those details. Most of the classes in the `java.sql` package are in fact interfaces—and thus no implementation details. Individual database vendors provide implementations of these interfaces in the form of something called a JDBC driver. As a database programmer, however, you need to know only a few details about the driver you are using—the rest you manage via the JDBC interfaces.

The first database-dependent thing you need to know is what drivers exist for your database. Different people provide different JDBC implementations for a variety of databases. As a database programmer, you want to select a JDBC implementation that will provide the greatest stability and performance for your application. Though it may seem counterintuitive, JDBC implementations provided by the database vendors generally sit at the bottom of the pack when it comes to stability and flexibility. As an Open Source project, however, MySQL relies on drivers provided by other developers in the community.

Sun has created four classifications that divide JDBC drivers based on their architectures. Each JDBC driver classification represents a trade-off between performance and flexibility.

Type 1

These drivers use a bridging technology to access a database. The JDBC-ODBC bridge that comes with JDK 1.2 is the most common example of this kind of driver. It provides a gateway to the ODBC API. Implementations of the ODBC API in turn perform the actual database access. Though useful for learning JDBC and quick testing, bridging solutions are rarely appropriate for production environments.

Type 2

Type 2 drivers are native API drivers. "Native API" means that the driver contains Java code that calls native C or C++ methods provided by the database vendor. In the context of MySQL, a Type 2 driver would be one that used MySQL's C API under the covers to talk to MySQL on behalf of your application. Type 2 drivers generally provide the best performance, but they do require the installation of native libraries on clients that need to access the database. Applications using Type 2 drivers have a limited degree of portability.

Type 3

Type 3 drivers provide a client with a pure Java implementation of the JDBC API where the driver uses a network protocol to talk to middleware on the server. This middleware, in turn, performs the actual database access. The middleware may or may not use JDBC for its database access. The Type 3 architecture is actually more of a benefit to driver vendors than application

architects since it enables the vendor to write a single implementation and claim support for any database that has a JDBC driver. Unfortunately, it has weak performance and unpredictable stability.

Type 4

Using network protocols built into the database engine, Type 4 drivers talk directly to the database using Java sockets. This is the most direct pure Java solution. Because these network protocols are almost never documented, most Type 4 drivers come from the database vendors. The Open Source nature of MySQL, however, has enabled several independent developers to write different Type 4 MySQL drivers.

Practically speaking, Type 2 and Type 4 drivers are the only viable choices for a production application. At an abstract level, the choice between Type 2 and Type 4 comes down to a single issue: Is platform independence critical? By platform independence, we mean that the application can be bundled up into a single jar and run on any platform. Type 2 drivers have a hard time with platform independence since you need to package platform-specific libraries with the application. If the database access API has not been ported to a client platform, then your application will not run on the platform. On the other hand, Type 2 drivers tend to perform better than Type 4 drivers.

Knowing the driver type provides only a starting point for making a decision about which JDBC driver to use in your application. The decision really comes down to knowing the drivers that exist for your database of choice and how they compare to each other. Table 11-1 lists the JDBC drivers available for MySQL. Of course, you are also able to use any sort of ODBC bridge to talk to MySQL as well—but we do not recommend it under any circumstance for MySQL developers.

Table 0-1. . JDBC Drivers for MySQL

Driver Name	OSI[a] License	JDBC Version	Home Page
mm (GNU)	LGPL	1.x and 2.x	http://mmmysql.sourceforge.net/
twz	no	1.x	http://www.voicenet.com/~zellert/tjFM/
Caucho	QPL	2.x	http://www.caucho.com/projects/jdbc-mysql/index.xtp

[a] Open Source Initiative (http://www.opensource.org). For drivers released under an OSI-approved license, the specific license is referenced.

Of the three MySQL JDBC drivers, twz sees the least amount of development and thus likely does not serve the interests of most programmers these days. The GNU driver (also known as mm MySQL), on the other hand, has seen constant develop-

ment and is the most mature of the three JDBC drivers. Not to be outdone, Caucho claims significant performance benefits over the GNU driver.

The JDBC Architecture

We have already mentioned that JDBC is a set of interfaces implemented by different vendors. Figure 11-1 shows how database access works from an application's perspective. In short, the application simply makes method calls to the JDBC interfaces. Under the covers, the implementation being used by that application performs the actual database calls.

FIGURE14-1.BMP

Figure 0-1. . The JDBC architecture

JDBC is divided up into two Java packages:

- `java.sql`
- `javax.sql`

The `java.sql` package was the original package that contained all of the JDBC classes and interfaces. JDBC 2.0, however, introduced something called the JDBC Optional Package—the `javax.sql` package—with interfaces that a driver does not have to implement. In fact, the interfaces themselves are not even part of the J2SE as of JDK 1.3 (though it always has been part of the J2EE).

As it turns out, some of the functionality in the JDBC Optional Package is so important that it has been decided that it is no longer "optional" and should instead be part of the J2SE with the release of JDK 1.4. For backwards compatibility, the Optional Package classes remain in `javax.sql`.

Connecting to MySQL

JDBC represents a connection to a database through the `Connection` interface. Connecting to MySQL thus requires you to get an instance of the `Connection` interface from your JDBC driver. JDBC supports two ways of getting access to a database connection:

1. Through a JDBC Data Source
2. Using the JDBC Driver Manager

The first method—the data source—is the preferred method for connecting to a database. Data sources come from the Optional Package and thus support for them is still spotty. No matter what environment you are in, you can rely on driver manager connectivity.

Data Source Connectivity

Data source connectivity is very simple. In fact, the following code makes a connection to any database—it is not specific to MySQL:

```
Context ctx = new InitialContext();
DataSource ds = (DataSource)ctx.lookup("jdbc/myds");
Connection conn = ds.getConnection("userid", "password");
```

The first line in this example actually comes from the Java Naming and Directory Interface (JNDI*) API. JNDI is an API that provides access to naming and directory services. Naming and directory services are specialized data stores that enable you to associate related data under a familiar name. In a Windows environment, for example, network printers are stored in Microsoft ActiveDirectory under a name. In order to print to the networked color printer, a user does not need to know all of the technical details about the printer. Those details are stored in the directory. The user simply needs to know the name of the printer. The directory, in other words, stored all of the details about the printer in a directory where an application could access those details by name.

Though data source connectivity does not require a data source be stored in a directory, you will find that a directory is the most common place you will want to store data source configuration details. As a result, you can simply ask the directory for the data source by name. In the above example, the name of the data source is "jdbc/myds". JNDI enables your application to grab the data source from the directory by its name without worrying about all of the configuration details.

Though this sounds simple enough, you are probably wondering how the data source got in the directory in the first place. Someone has to put it there. Programmatically, putting the data source in the directory can be as simple as the following code:

```
SomeDataSourceClass ds = new SomeDataSourceClass();
Context ctx = new InitialContext();

// configure the DS by setting configuration attributes
ctx.bind("jdbc/myds", ds);
```

We have two bits of "magic" in this code. The first bit of magic is the `SomeDataSourceClass` class. In short, it is an implementation of the `javax.sql.DataSource` interface. In some cases, this implementation may come from the JDBC vendor—but not always. In fact, none of the MySQL drivers currently ship with a

* A full discussion of JNDI is way beyond the scope of this chapter. You minimally need a JNDI service provider (analogous to a JDBC driver) and to set some environment variables to support that service provider. You also need a directory service to talk to. If you do not have access to a directory service, you can always practice using the file system service provider available on the JNDI home page at http://java.sun.com/products/jndi or use the driver manager approach.

`DataSource` implementation. If you are using some sort of application server like Orion or WebLogic, then those application servers will provide a `DataSource` implementation for you that will work with MySQL.

Configuring your data source depends on the properties demanded by the data source implementation class. In most cases, a data source implementation will want to know the JDBC URL and name of the `java.sql.Driver` interface implementation for the driver. We will cover these two things in the next section on driver manager connectivity.

Though we have been very vague about configuring a JDBC data source programmatically, you should not despair. You should never have to configure a JDBC data source programmatically. The vendor that provides your data source implementation should provide you with a configuration tool capable for publishing the configuration for a data source to a directory. All application servers come with such a tool. A tool of this sort will prompt you for the values it needs in order to enter a new data source in a directory and then allow you to save that configuration to the directory. Your application can then access the data source by name as shown earlier in the chapter.

Driver Manager Connectivity

One of the few implementation classes in the `java.sql`.package is the `DriverManager` class. It maintains a list of implementations of the JDBC `java.sql.Driver` class and provides you with database connections based on JDBC URLs you provide it. A JDBC URL comes in the form of *jdbc:protocol:subprotocol*. It tells a `DriverManager` which database engine you wish to connect to and it provides the `DriverManager` with enough information to make a connection.

JDBC uses the word “driver” in multiple contexts. In the lower-case sense, a JDBC driver is the collection of classes that together implement all of the JDBC interfaces and provide an application with access to at least one database. In the upper-case sense, the `Driver` is the class that implements `java.sql.Driver`. Finally, JDBC provides a `DriverManager` that can be used to keep track of all of the different `Driver` implementations.

The protocol part of the URL refers to a given JDBC driver. The protocol for the Caucho MySQL driver, for example, is *mysql-caucho* while the GNU driver uses *mysql*. The subprotocol provides the implementation-specific connection data. All MySQL drivers require a host name and database name in order to make a connection. Optionally, they may require a port if your database engine is not run-

ning as root. Table 11-2 shows the configuration information for the MySQL JDBC drivers.

Table 0-2. . Configuration Information for MySQL JDBC Drivers

Driver	Implementation	URL
Caucho	`com.caucho.jdbc.mysql.Driver`	jdbc:mysql-caucho://HOST[:PORT]/DB
GNU	`org.gjt.mm.mysql.Driver`	jdbc:mysql://[HOST][:PORT]/DB[?PROP1=VAL1][&PROP2=VAL2]...
twz	`twz1.jdbc.mysql.jdbc-MysqlDriver`	jdbc:z1MySQL://HOST[:PORT]/DB[?PROP1=VAL1][&PROP2=VAL2]...

As you can see, the URLs for the GNU driver and twz driver are very different from the Caucho driver. As a general rule, the format of the Caucho driver is actually the preferred format since the you can specify properties separately.

Your first task is to register the driver implementation with the JDBC `DriverManager`. There are two key ways to register a driver:

1. You can specify the name of the drivers you want to have registered on the command line of your application using the jdbc.drivers property: *java -Djdbc.drivers=com.caucho.jdbc.mysql.Driver MyAppClass*.
2. You can explicitly load the class in your program by doing a `new` or a `Class.forName()`: `Class.forName("twz1.jdbc.mysql.jdbc-MysqlDriver").newInstance()`.

For portability's sake, we recommend that you put all configuration information in some sort of configuration file like a properties file and then load the configuration data from that configuration file. By taking this approach, your application will have no dependencies on MySQL or the JDBC driver you are using. You can simply change the values in the configuration file to move from the GNU driver to Caucho or from MySQL to Oracle.

Once you have registered your driver, you can then ask the DriverManager for a Connection. You do this by calling the getConnection() method in the driver with the information identifying the desired connection. This information minimally includes a JDBC URL, user ID, and password. You may optionally include a set of parameters:

```
Connection conn = DriverManager.getConnection("jdbc:mysql-caucho://carthage/Web", "someuser", "somepass");
```

This code returns a connection associated with the database "Web" on the MySQL server on the machine carthage using the Caucho driver under the user ID "someuser" and authenticated with "somepass". Though the Caucho driver has the simplest URL, connecting with the other drivers is not much more difficult. They just ask that you specify connection properties such as the user ID and password as part of the JDBC URL. Table 11-3 lists the URL properties for the GNU driver and Table 11-4 lists them for the twz driver.

Table 0-3. . URL Properties for the GNU (mm) JDBC Driver

Name	Default	Description
autoReconnect	false	Causes the driver to attempt a reconnect when the connection dies.
characterEncoding	none	The Unicode encoding to use when Unicode is the character set.
initialTimeout	2	The initial time between reconnects in seconds when autoReconnect is set.
maxReconnects	3	The maximum number of times the driver should attempt a reconnect.
maxRows	0	The maximum number of rows to return for queries. 0 means return all rows.
password	none	The password to use in connecting to MySQL
useUnicode	false	Unicode is the character set to be used for the connection.
user	none	The user to use for the MySQL connection.

Table 0-4. . URL Properties for the twz JDBC Driver

Name	Default	Description
autoReX	true	Manages automatic reconnect for data update statements.
cacheMode	memory	Dictates where query results are cached.
cachePath	.	The directory to which result sets are cached if cacheMode is set t "disk".
connectionTimeout	120	The amount of time, in seconds, that a thread will wait on action by a connection before throwing an exception.
db	mysql	The MySQL database to which the driver is connected.
dbmdDB	<connection>	The MySQL database to use for database meta-data operations.
dbmdMaxRows	66536	The maximum number of rows returned by a database meta-data operation.
dbmdPassword	<connection>	The password to use for database meta-data operations.

Table 0-4. . URL Properties for the twz JDBC Driver

Name	Default	Description
dbmdUser	<connection>	The user ID to use for database meta-data operations.
dbmdXcept	false	Exceptions will be thrown on unsupported database meta-data operations instead of the JDBC-compliant behavior of returning an empty result.
debugFile	none	Enables debugging to the specified file.
debugRead	false	When debugging is enabled, data read from MySQL is dumped to the debug file. This will severely degrade the performance of the driver.
debugWrite	false	When debugging is enabled, data written to MySQL is dumped to the debug file. This will severely degrade the performance of the driver.
host	localhost	The host machine on which MySQL is running.
maxField	*65535*	The maximum field size for data returned by MySQL. Any extra data is silently truncated.
maxRows	Integer.MAX_ VALUE	The maximum number of rows that can be returned by a MySQL query.
moreProperties	none	Tells the driver to look for more properties in the named file.
multipleQuery	true	Will force the caching of the result set allowing multiple queries to be open at once.
password	none	The password used to connect to MySQL.
port	3306	The port on which MySQL is listening.
socketTimeout	none	The time in seconds that a socket connection will block before throwing an exception.
user	none	The user used to connect to MySQL.
RSLock	false	Enables locking of result sets for a statement for use in multiple threads.

As a result, connections for these two drivers commonly look like:

```
Connection conn = DriverManager.getConnection("jdbc:mysql://carthage/
Web?user=someuser&password=somepass");
```

or for twz:

```
Connection conn =
 DriverManager.getConnection("jdbc:z1MySQL://carthage/Web?user=someuser&password="somepass");
```

Instead of passing the basic connection properties of "user" and "password" as a second and third argument to getConnection(), GNU and twz instead pass them as part of the URL. In fact, you can pass any of the properties as part of the URL. JDBC, however, has a standard mechanism for passing driver-specific connection properties to getConnect():

```
Properties p = new Properties();
Connection conn;

p.put("user", "someuser");
p.put("password", "somepass");
p.put("useUnicode", "true");
p.put("characterEncoding", "UTF-8");
conn = DriverManager.getConnection(url, p);
```

Unfortunately, the way in which MySQL supports these optional properties is a bit inconsistent. It is thus best to go with the preferred manner for your driver, however unwieldy it makes the URLs.

Example 11-1 shows how to make a connection to MySQL using the GNU driver.

Example 0-1. A Complete Sample of Making a JDBC Connection

```
import java.sql.*;

public class Connect {
  public static void main(String argv[]) {
    Connection con = null;

    try {
      // here is the JDBC URL for this database
      String url = "jdbc:mysql://athens.imaginary.com/Web?user=someuser&password=somepass";
      // more on what the Statement and ResultSet classes do later
      Statement stmt;
      ResultSet rs;

      // either pass this as a property, i.e.
      // -Djdbc.drivers=org.gjt.mm.mysql.Driver
      // or load it here like we are doing in this example
      Class.forName("org.gjt.mm.mysql.Driver");
      // here is where the connection is made
      con = DriverManager.getConnection(url);
    }
    catch( SQLException e ) {
      e.printStackTrace();
    }
    finally {
      if( con != null ) {
        try { con.close(); }
        catch( Exception e ) { }
      }
    }
```

Example 0-1. A Complete Sample of Making a JDBC Connection (continued)

```
  }
}
```

The line `con = DriverManager.getConnection(url)` makes the database connection in this example. In this case, the JDBC URL and `Driver` implementation class names are actually hard coded into this application. The only reason this is acceptable is because this application is an example driver. As we mentioned earlier, you want to get this information from a properties file or the command line in real applications.

Maintaining Portability Using Properties Files

Though our focus is on MySQL, it is good Java programming practice to make your applications completely portable. To most people, portability means that you do not write code that will run on only one platform. In the Java world, however, the word "portable" is a much stronger term. It means no hardware resource dependencies, and that means no database dependencies.

We discussed how the JDBC URL and `Driver` name are implementation dependent, but we did not discuss the details of how to avoid hard coding them. Because both are simple strings, you can pass them on the command line as runtime arguments or as parameters to applets. While that solution works, it is hardly elegant since it requires command line users to remember long command lines. A similar solution might be to prompt the user for this information; but again, you are requiring that the user remember a JDBC URL and a Java class name each time they run an application.

Properties Files

A more elegant solution than either of the above solutions would be to use a properties file. Properties files are supported by the `java.util.ResourceBundle` and its subclasses to enable an application to extract runtime specific information from a text file. For a JDBC application, you can stick the URL and `Driver` name in the properties file, leaving the details of the connectivity up to an application administrator. Example 11-2 shows a properties file that provides connection information.

Example 0-2. The SelectResource.properties File with Connection Details for a Connection

```
Driver=org.gjt.mm.mysql.Driver
URL=jdbc:mysql://athens.imaginary.com/Web?user=someuser&password=somepass
```

Example 11-3 shows the portable `Connect` class.

Example 0-3. Using a Properties File to Maintain Portability

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

public class Connect {
  public static void main(String argv[]) {
    Connection con = null;
    ResourceBundle bundle = ResourceBundle.getBundle("SelectResource");

    try {
      String url = bundle.getString("URL");
      Statement stmt;
      ResultSet rs;

       Class.forName(bundle.getString("Driver"));
      // here is where the connection is made
      con = DriverManager.getConnection(url);
    }
    catch( SQLException e ) {
      e.printStackTrace();
    }
    finally {
      if( con != null ) {
        try { con.close(); }
        catch( Exception e ) { }
      }
    }
  }
}
```

We have gotten rid of anything specific to MySQL or the GNU driver in the sample connection code. One important issue still faces portable JDBC developers. JDBC requires any driver to support SQL2 entry level. This is an ANSI standard for minimum SQL support. As long as you use SQL2 entry level SQL in your JDBC calls, your application will be 100% portable to other database engines. Fortunately, MySQL is SQL2 entry level, even though it does not support many of the advanced SQL2 features.

Data Sources Revisited

Earlier in the chapter, we fudged a bit on how data sources were configured. Specifically, we stated that you can configure a data source either using a tool or through Java code. In most cases you will do so using a tool. The way in which you go about configuring a data source is very dependent on the vendor providing the data source. Now that you have a greater appreciation of connection properties, you should have a good idea of what you will need to configure a data source to support MySQL.

In order to better illustrate the way in which a data source can be set up for an application, it helps to look at a real world application environment. Orion is a J2EE compliant application server that is free for non-commercial use. In this application, it is serving up Java Server Pages (JSPs) that go against a MySQL database. The JSP makes the following JDBC call in order to do its database work:

```
InitialContext ctx = new InitialContext();
DataSource ds = (DataSource)ctx.lookup("jdbc/AddressBook");
Connection = ds.getConnection();
```

This looks familiar so far? Of course, it begs the question: how exactly does "jdbc/AddressBook" book get configured? In Orion, you configure the data source by editing a file called data-sources.xml. Here is the entry for "jdbc/AddressBook":

```
<data-source connection-driver="org.gjt.mm.mysql.Driver"
        class="com.evermind.sql.DriverManagerDataSource"
        name="AddressBook"
        url="jdbc:mysql://carthage/Address?user=test&amp;password=test"
        location="jdbc/AddressBook"/>
```

Simple Database Access

The `Connect` example did not do much. It simply showed you how to connect to MySQL. A database connection is useless unless you actually talk to the database. The simplest forms of database access are SELECT, INSERT, UPDATE, and DELETE statements. Under the JDBC API, you use your database `Connection` instance to create `Statement` instances. A `Statement` represents any kind of SQL statement. Example 11-4 shows how to insert a row into a database using a `Statement`.

Example 0-4. Inserting a Row into MySQL Using a JDBC Statement Object

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

public class Insert {
  // We are inserting into a table that has two columns: TEST_ID (int)
  // and TEST_VAL (char(55))
  // args[0] is the TEST_ID and args[1] the TEST_VAL
  public static void main(String argv[]) {
    Connection con = null;
    ResourceBundle bundle = ResourceBundle.getBundle("SelectResource");

    try {
      String url = bundle.getString("URL");
      Statement stmt;

      Class.forName(bundle.getString("Driver"));
      // here is where the connection is made
      con = DriverManager.getConnection(url, "user", "pass");
      stmt = con.createStatement();
      stmt.executeUpdate("INSERT INTO TEST (TEST_ID, TEST_VAL) " +
```

Example 0-4. Inserting a Row into MySQL Using a JDBC Statement Object

```
                    "VALUES(" + args[0] + ", '" + args[1] + "')");
    }
    catch( SQLException e ) {
      e.printStackTrace();
    }
    finally {
      if( con != null ) {
        try { con.close(); }
        catch( Exception e ) { }
      }
    }
  }
}
```

If this were a real application, we would of course verified that the user entered an INT for the TEST_ID, that it was not a duplicate key, and that the TEST_VAL entry did not exceed 55 characters. This example nevertheless shows how simple performing an insert is. The `createStatement()` method does just what it says: it creates an empty SQL statement associated with the `Connection` in question. The `executeUpdate()` method then passes the specified SQL on to the database for execution. As its name implies, `executeUpdate()` expects SQL that will be modifying the database in some way. You can use it to insert new rows as shown earlier, or instead to delete rows, update rows, create new tables, or do any other sort of database modification.

Queries and Result Sets

Queries are a bit more complicated than updates because queries return information from the database in the form of a `ResultSet`. A `ResultSet` is an interface that represents zero or more rows matching a database query. A JDBC `Statement` has an `executeQuery()` method that works like the `executeUpdate()` method—except it returns a `ResultSet` from the database. Exactly one `ResultSet` is returned by `executeQuery()`, however, you should be aware that JDBC supports the retrieval of multiple result sets for databases that support multiple result sets. MySQL, however, does not support multiple result sets. It is nevertheless important for you to be aware of this issue in case you are ever looking at someone else's code written against another database engine. Example 11-5 shows a simple query. Figure 11-2 shows the data model behind the test table.

Example 0-5. A Simple Query

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

public class Select {
  public static void main(String argv[]) {
    Connection con = null;
```

Example 0-5. A Simple Query (continued)

```
    ResourceBundle bundle =
        ResourceBundle.getBundle("SelectResource");

    try {
      String url = bundle.getString("URL");
      Statement stmt;
      ResultSet rs;

       Class.forName(bundle.getString("Driver"));
      // here is where the connection is made
      con = DriverManager.getConnection(url, "user", "pass");
      stmt = con.createStatement();
      rs = stmt.executeQuery("SELECT * from TEST ORDER BY TEST_ID");
      System.out.println("Got results:");
      while(rs.next()) {
        int a= rs.getInt("TEST_ID");
        String str = rs.getString("TEST_VAL");

        System.out.print(" key= " + a);
        System.out.print(" str= " + str);
        System.out.print("\n");
      }
      stmt.close();
    }
    catch( SQLException e ) {
      e.printStackTrace();
    }
    finally {
      if( con != null ) {
        try { con.close(); }
        catch( Exception e ) { }
      }
    }
  }
}
```

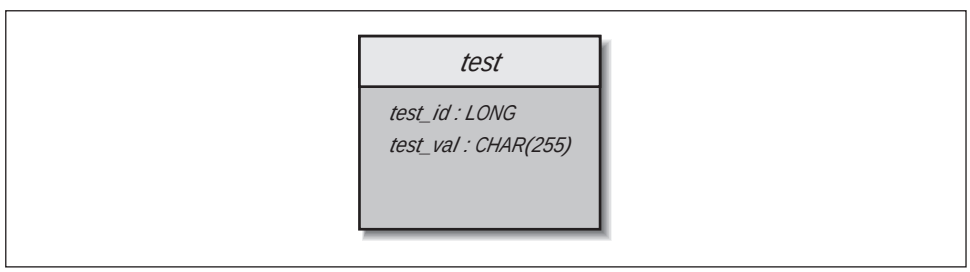

Figure 0-2. The test table from the sample database

The `Select` application executes the query and then loops through each row in the `ResultSet` using the `next()` method. Until the first call to `next()`, the `ResultSet` does not point to any row. Each call to `next()` points the `ResultSet`

to the subsequent row. You are done processing rows when `next()` returns `false`.

You can specify that you result set is scrollable, meaning that you can move around in the result set—not just forward on a row-by-row basis. The ResultSet instances generated by a Statement are scrollable if the statement was created to support scrollable result sets. Connection enables this to happen by an alternate form of the createStatement() method:

```
Statement stmt = conn.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
                                      ResultSet.CONCUR_READ_ONLY);
```

The first argument says that any result sets of the newly created statement should be scrollable. By default, a statement's result sets are not scrollable. The second argument relates to an advanced feature of JDBC, updatable result sets, that lies beyond the scope of this book.

With a scrollable result set, you can make calls to `previous()` to navigate backwards through the results and `absolute()` and `relative()` to move to arbitrary rows. Like `next()`, `previous()` moves one row through the result set, except in the opposite direction. The `previous()` method returns false when you attempt to move before the first row. Finally `absolute()` moves the result set to a specific row, whereas `relative()` moves the result set a specific number of rows before or after the current row.

Dealing with a row means getting the values for each of its columns. Whatever the value in the database, you can use the getter methods in the `ResultSet` to retrieve the column value as whatever Java datatype you like. In the `Select` application, the call to `getInt()` returned the TEST_ID column as an `int` and the call to `getString()` returned the TEST_VAL column as a `String`. These getter methods accept either the column number—starting with column 1—or the column name. You should, however, avoid retrieving values using a column name at all costs since retrieving results by column name is many, many times slower than retrieving them by column number.

One area of mismatch between Java and MySQL lies in the concept of a SQL `NULL`. SQL is specifically able to represent some data types as null that Java cannot represent as null. In particular, Java has no way of representing primitive data types as nulls. As a result, you cannot immediately determine whether a 0 returned from MySQL through `getInt()` really means a 0 is in that column or if no value is in that column. JDBC addresses this mismatch through the `wasNull()` method.

As its name implies, `wasNull()` returns `true` if the last value fetched was SQL `NULL`. For calls returning a Java object, the value will generally be `NULL` when a SQL `NULL` is read from the database. In these instances, `wasNull()` may appear somewhat redundant. For primitive datatypes, however, a valid value—like 0—

may be returned on a fetch. The `wasNull()` method gives you a way to see if that value was `NULL` in the database.

Error Handling and Clean Up

All JDBC method calls can throw SQLException or one of its subclasses if something happens during a database call. Your code should be set up to catch this exception, deal with it, and clean up any database resources that have been allocated. Each of the JDBC classes mentioned so far has a `close()` method associated with it. Practically speaking, however, you only really need to make sure you close things whose calling process might remain open for a while. In the examples we have seen so far, you only really need to close your database connections. Closing the database connection closes any statements and result sets associated with it automatically. If you intend to leave a connection open for any period of time, however, it is a good idea to go ahead and close the statements you create using that connection when you finish with them. In the JDBC examples you have seen, this clean up happens in a finally clause. You do this since you want to make sure to close the database connection no matter what happens.

Dynamic Database Access

So far we have dealt with applications where you know exactly what needs to be done at compile time. If this were the only kind of database support that JDBC provided, no one could ever write tools like the *mysql* interactive command line tool that determines SQL calls at runtime and executes them. The JDBC `Statement` class provides the `execute()` method for executing SQL that may be either a query or an update. Additionally, `ResultSet` instances provide runtime information about themselves in the form of an interface called `ResultSetMetaData` which you can access via the `getMetaData()` call in the `ResultSet`.

Meta Data

The term meta data sounds officious, but it is really nothing other than extra data about some object that would otherwise waste resources if it were actually kept in the object. For example, simple applications do not need the name of the columns associated with a `ResultSet`—the programmer probably knew that when the code was written. Embedding this extra information in the `ResultSet` class is thus not considered by JDBC's designers to be core to the functionality of a `ResultSet`. Data such as the column names, however, is very important to some database programmers—especially those writing dynamic database access. The JDBC designers provide access to this extra information—the meta data—via the `ResultSetMetaData` interface. This class specifically provides:

- The number of columns in a result set
- Whether NULL is a valid value for a column
- The label to use for a column header
- The name for a given column
- The source table for a given column
- The datatype of a given column

Example 11-6 shows some of the source code from a command line tool like mysql that accepts arbitrary user input and sends it to MySQL for execution. The rest of the code for this example can be found at the O'Reilly Web site with the rest of the examples from this book.

Example 0-6. An Application for Executing Dynamic SQL

```
import java.sql.*;

public class Exec {
  public static void main(String args[]) {
    Connection con = null;
    String sql = "";

    for(int i=0; i<args.length; i++) {
      sql = sql + args[i];
      if( i < args.length - 1 ) {
        sql = sql + " ";
      }
    }
    System.out.println("Executing: " + sql);
    try {
      Class.forName("com.caucho.jdbc.mysql.Driver").newInstance();
      String url = "jdbc:mysql-caucho://athens.imaginary.com/TEST";
      con = DriverManager.getConnection(url, "test", "test");
      Statement s = con.createStatement();

      if( s.execute(sql) ) {
        ResultSet r = s.getResultSet();
        ResultSetMetaData meta = r.getMetaData();
        int cols = meta.getColumnCount();
        int rownum = 0;

        while( r.next() ) {
          rownum++;
          System.out.println("Row: " + rownum);
          for(int i=0; i<cols; i++) {
            System.out.print(meta.getColumnLabel(i+1) + ": "
                + r.getObject(i+1) +  ", ");
          }
          System.out.println("");
        }
      }
```

Example 0-6. An Application for Executing Dynamic SQL

```
        else {
          System.out.println(s.getUpdateCount() + " rows affected.");
        }
        s.close();
        con.close();
      }
      catch( Exception e ) {
        e.printStackTrace();
      }
      finally {
        if( con != null ) {
          try { con.close(); }
          catch( SQLException e ) { }
        }
      }
    }
}
```

Each result set provides a `ResultSetMetaData` instance via the `getMetaData()` method. In the case of dynamic database access, we need to find out the how many columns are in a result set so that we are certain to retrieve each column as well as the names of each of the columns for display to the user. The meta data for our result set provides all of this information via the `getColumnCount()` and `getColumnLabel()` methods.

Processing Dynamic SQL

The concept introduced in Example 11-6 is the dynamic SQL call. Because we do not know whether we will be processing a query or an update, we need to pass the SQL call through the `execute()` method. This method returns `true` if the statement returned a result set or `false` if none was produced. In the example, if it returns `true`, the application gets the returned `ResultSet` through a call to `getResultSet()`. The application can then go on to do normal result set processing. If, on the other hand, the statement performed some sort of database modification, you can call `getUpdateCount()` to find out how many rows were modified by the statement.

A Guest Book Servlet

You have probably heard quite a bit of talk about Java applets. We discussed in Chapter 6, however, how doing database access in the client is a really bad idea. We have packaged with the examples in this book a servlet that uses the JDBC knowledge we have discussed in this chapter to store the comments from visitors to a Web site in a database and then display the comments in the database. While servlets are not in themselves part of the three-tier solution we discussed in Chapter 6,

this example should provide a useful example of how JDBC can be used. For this example, all you need to know about servlets is that the `doPost()` method handles HTTP POST events and `doGet()` handles HTTP GET events. The rest of the code is either simple Java code or an illustration of the database concepts from this chapter. You can see the servlet in action at *http://www.imaginary.com/~george/guestbook.shtml*.