Unit-2

Database Programming

2.1 The Design of JDBC

Database vendors could provide their own drivers to plug in to the driver manager. There would then be a simple mechanism for registering third-party drivers with the driver manager.

This organization follows the very successful model of Microsoft's ODBC which provided a C programming language interface for database access. Both JDBC and ODBC are based on the same idea: Programs written according to the API talk to the driver manager, which, in turn, uses a driver to talk to the actual database.

1. JDBC-ODBC bridge driver
2. Native-API driver (partially java driver)
3. Network Protocol driver (fully java driver)
4. Thin driver (fully java driver)

2.1.1 JDBC Driver Types

1. A *type 1 driver* translates JDBC to ODBC and relies on an ODBC driver to communicate with the database.

2. A *type 2 driver* is written partly in Java and partly in native code; it communicates with the client API of a database.

3. A *type 3 driver* is a pure Java client library that uses a database-independent protocol to communicate database requests to a server component, which then translates the requests into a database-specific protocol.

4. A *type 4 driver* is a pure Java library that translates JDBC requests directly to a database-specific protocol.

Type 1 JDBC driver: (JDBC to ODBC Bridge)

The JDBC-ODBC bridge driver uses ODBC driver to connect to the database. The JDBC-ODBC bridge driver converts JDBC method calls into the ODBC function calls. This is now discouraged because of thin driver. Oracle does not support the JDBC-ODBC Bridge from Java 8.

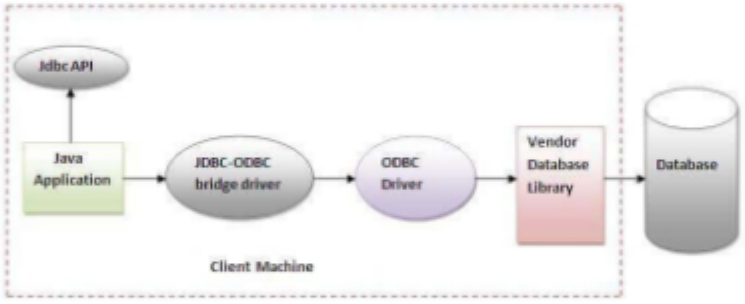

Figure- JDBC-ODBC Bridge Driver

Advantages:

- easy to use.
- can be easily connected to any database.

Disadvantages:

- Performance degraded because JDBC method call is converted into the ODBC function calls.
- The ODBC driver needs to be installed on the client machine.

Type 2 : Native-API driver

The Native API driver uses the client-side libraries of the database. The driver converts JDBC method calls into native calls of the database API. It is not written entirely in java.

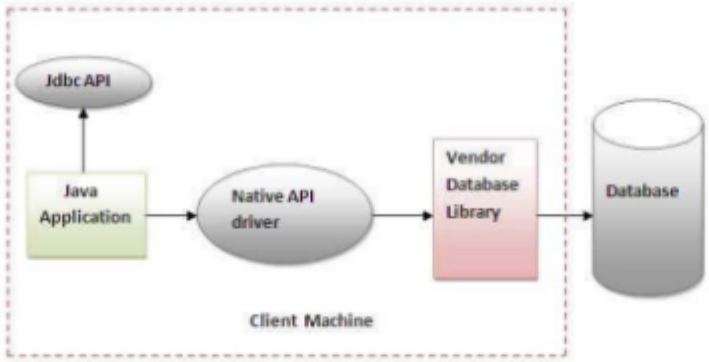

Figure- Native API Driver

Advantage:

- performance upgraded than JDBC-ODBC bridge driver.

Disadvantage:

- The Native driver needs to be installed on the each client machine.
- The Vendor client library needs to be installed on client machine.

3) Network Protocol driver

The Network Protocol driver uses middleware (application server) that converts JDBC calls directly or indirectly into the vendor-specific database protocol. It is fully written in java.

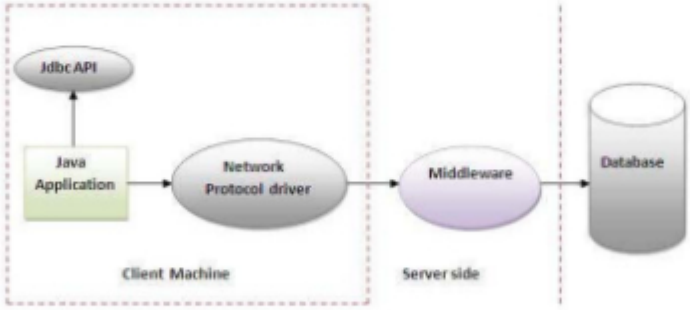

Figure- Network Protocol Driver

Advantage:

- No client side library is required because of application server that can perform many tasks like auditing, load balancing, logging etc.

Disadvantages:

- Network support is required on client machine.

- Requires database-specific coding to be done in the middle tier.
- Maintenance of Network Protocol driver becomes costly because it requires database-specific coding to be done in the middle tier.

4) Thin driver

The thin driver converts JDBC calls directly into the vendor-specific database protocol. That is why it is known as thin driver. It is fully written in Java language.

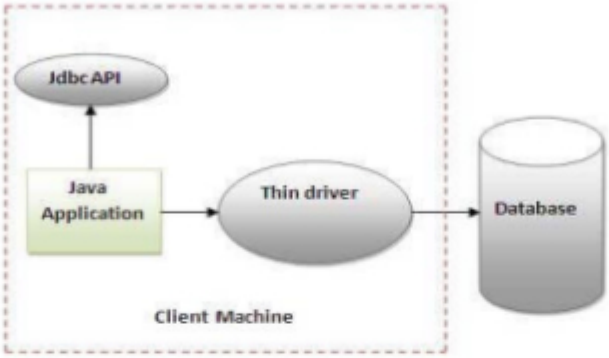

Figure- Thin Driver

Advantage:

- Better performance than all other drivers.
- No software is required at client side or server side.

Disadvantage:

- Drivers depend on the Database.

The ultimate goal of JDBC is to make possible the following:

• Programmers can write applications in the Java programming language to access any database using standard SQL statements (or even specialized extensions of SQL) while still following Java language conventions.

• Database vendors and database tool vendors can supply the low-level drivers. Thus, they can optimize their drivers for their specific products.

1.1.2 Typical Uses of JDBC

The traditional client/server model has a rich GUI on the client and a database on the server (see Figure 2.1). In this model, a JDBC driver is deployed on the client.

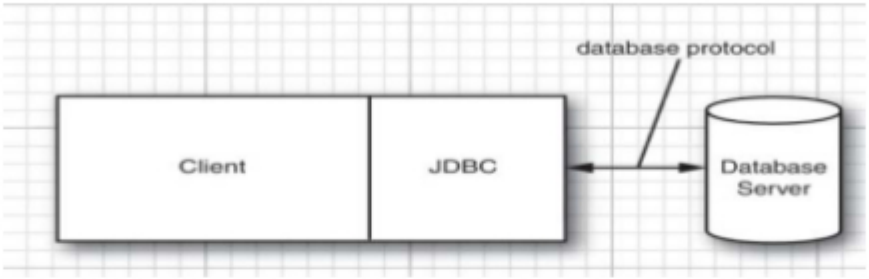

Figure 2.1 A traditional client/server application

However, nowadays it is far more common to have a three-tier model where the client application does not make database calls.

Instead, it calls on a middleware layer on the server that in turn makes the database queries.

The three-tier model has a couple of advantages. It separates *visual presentation* (on the client) from the *business logic* (in the middle tier) and the raw data (in the database).

Communication between the client and the middle tier typically occurs through HTTP. JDBC manages the communication between the middle tier and the back-end database. Figure 2.2 shows the basic architecture.

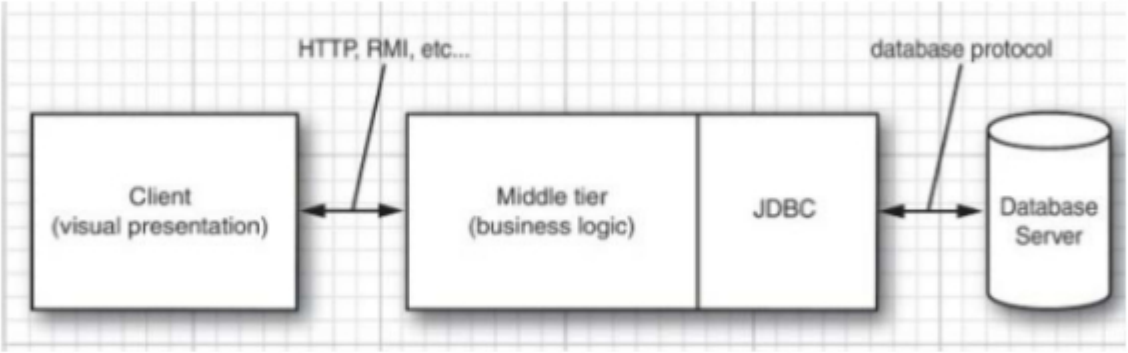

Figure 2.2 A three-tier

2.2 The Structured Query Language

JDBC lets you communicate with databases using SQL, which is the command language for essentially all modern relational databases.

JDBC lets you communicate with databases using SQL, which is the command language for essentially all modern relational databases.

Author_ID	Name	Fname
ALEX	Alexander	Christopher
BROO	Brooks	Frederick P.
. . .	. . .	. . .

Table 5.1 The `Authors` Table

Title	ISBN	Publisher_ID	Price
A Guide to the SQL Standard	0-201-96426-0	0201	47.95
A Pattern Language: Towns, Buildings, Construction	0-19-501919-9	019	65.00
. . .	. . .	. . .	. . .

Table 5.2 The `Books` Table

ISBN	Author_ID	Seq_No
0-201-96426-0	DATE	1
0-201-96426-0	DARW	2
0-19-501919-9	ALEX	1
. . .	. . .	. . .

Table 5.3 The `BooksAuthors` Table

Publisher_ID	Name	URL
0201	Addison-Wesley	www.aw-bc.com
0407	John Wiley & Sons	www.wiley.com
. . .	. . .	. . .

Table 5.4 The `Publishers` Table

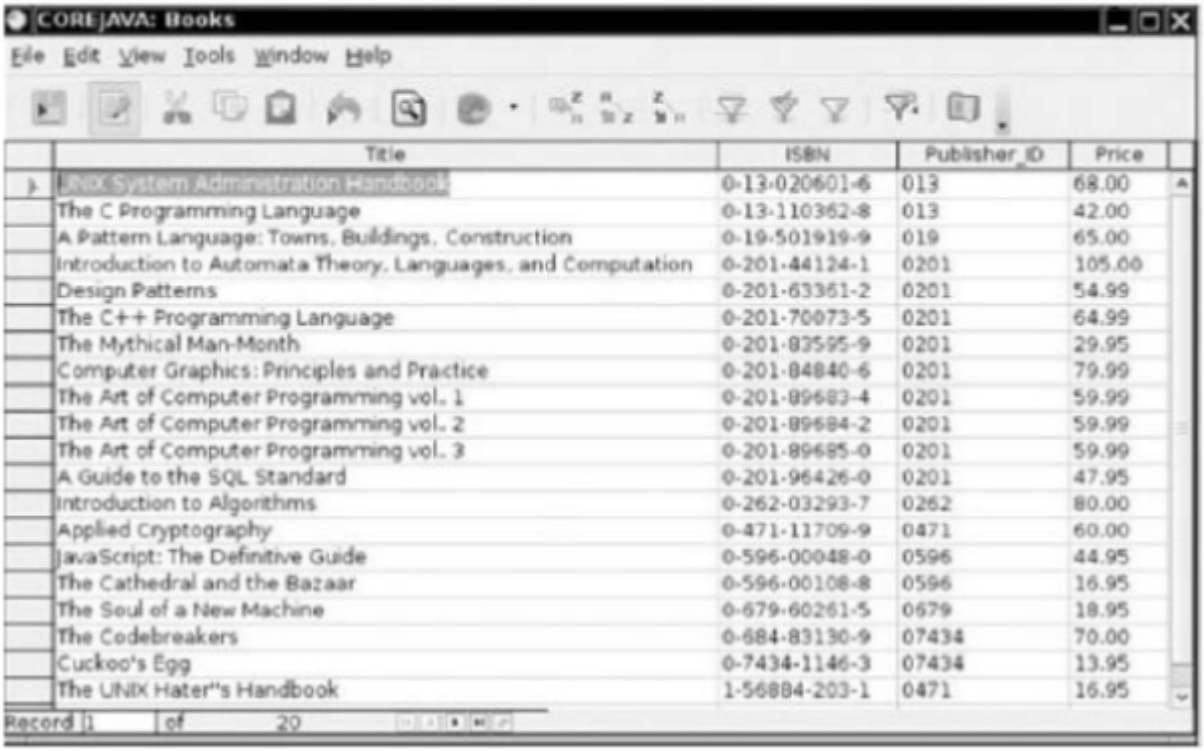

Figure 5.3 Sample table containing books

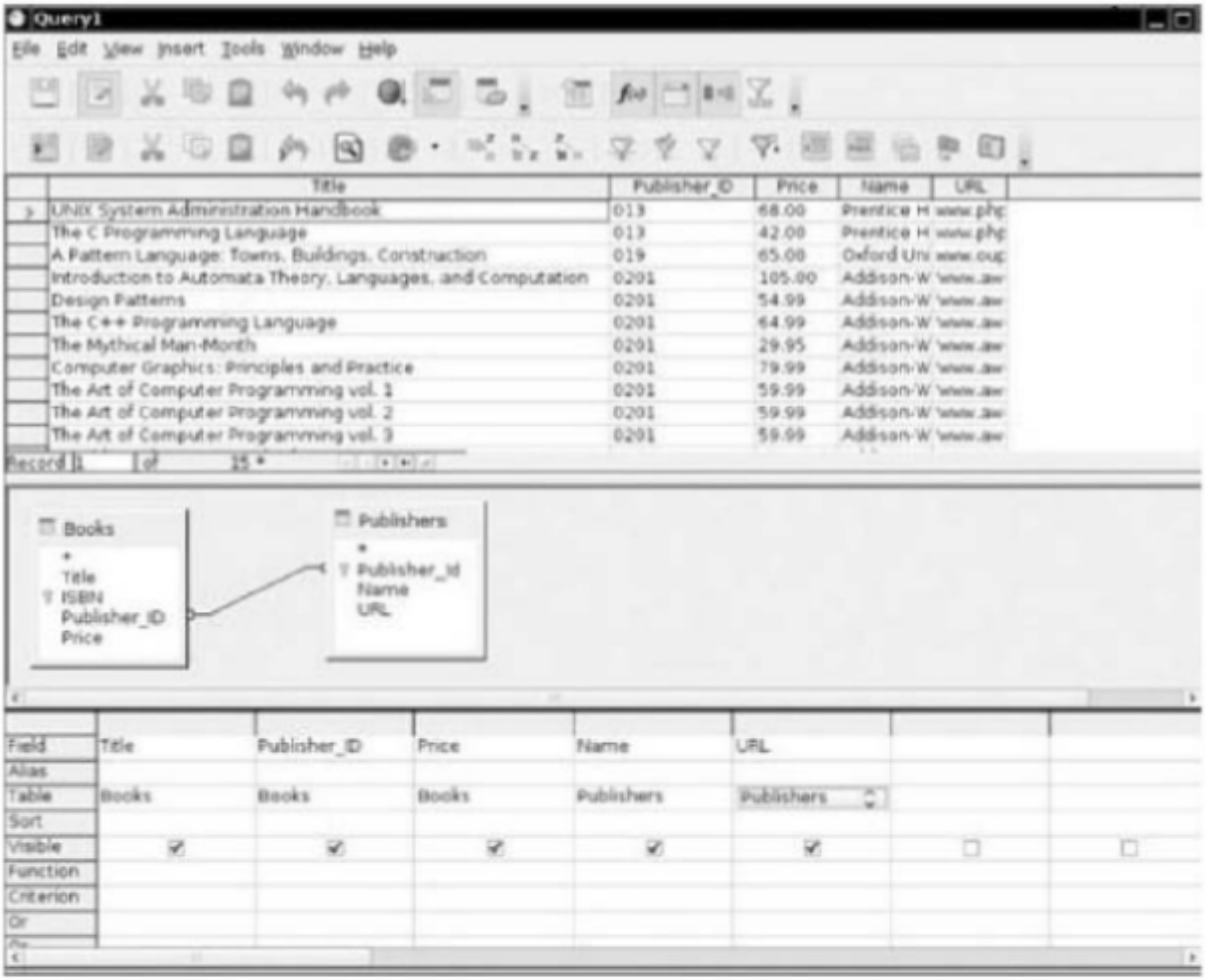

Figure 5.4 Two tables joined together

```
1. SELECT Books.Title, Books.Publisher_Id, Books.Price, Publishers.Name,
   Publishers.URL FROM Books, Publishers WHERE Books.Publisher_Id =
   Publishers.Publisher_Id

2. SELECT * FROM Books

3. SELECT ISBN, Price, Title FROM Books

4. SELECT ISBN, Price, Title FROM Books WHERE Price <= 29.95


5. SELECT ISBN, Price, Title FROM Books WHERE Title NOT LIKE '%n_x%'

6. SELECT Title FROM Books WHERE Title LIKE '%''%'

7. SELECT * FROM Books, Publishers

8. SELECT * FROM Books, Publishers
   WHERE Books.Publisher_Id = Publishers.Publisher_Id
```

```
9. UPDATE Books SET Price = Price - 5.00
   WHERE Title LIKE '%C++%

10.     DELETE FROM Books
        WHERE Title LIKE '%C++%'

11.     INSERT INTO Books
        VALUES ('A Guide to the SQL Standard', '0-201-96426-0', '0201',
        47.95)
```

```
CREATE TABLE Books
(
Title CHAR(60),
ISBN CHAR(13),
Publisher_Id CHAR(6),
Price DECIMAL(10,2)
)
```

Data Types	Description
INTEGER or INT	Typically, a 32-bit integer
SMALLINT	Typically, a 16-bit integer
NUMERIC(*m*, *n*), DECIMAL(*m*, *n*) or DEC(*m*, *n*)	Fixed-point decimal number with *m* total digits and *n* digits after the decimal point
FLOAT(*n*)	A floating-point number with *n* binary digits of precision
REAL	Typically, a 32-bit floating-point number
DOUBLE	Typically, a 64-bit floating-point number
CHARACTER(*n*) or CHAR(*n*)	Fixed-length string of length *n*
VARCHAR(*n*)	Variable-length strings of maximum length *n*
BOOLEAN	A Boolean value
DATE	Calendar date, implementation-dependent
TIME	Time of day, implementation-dependent
TIMESTAMP	Date and time of day, implementation-dependent
BLOB	A binary large object
CLOB	A character large object

Table 5.5 Common SQL Data Types

5.3 JDBC Configuration

There are many excellent choices, such as IBM DB2, Microsoft SQL Server, MySQL, Oracle, and PostgreSQL.

5.3.1 Database URLs

When connecting to a database, you must use various database-specific parameters such as host names, port numbers, and database names.

```
jdbc:derby://localhost:1527/COREJAVA;create=true
jdbc:postgresql:COREJAVA
```

These JDBC URLs specify a Derby database and a PostgreSQL database named COREJAVA.

The general syntax is:

`jdbc:`*`subprotocol`*`:`*`other stuff`*

where a subprotocol selects the specific driver for connecting to the database. The format for the *other stuff* parameter depends on the subprotocol used.

5.3.2 Driver JAR Files

You need to obtain the JAR file in which the driver for your database is located.

For example, the PostgreSQL drivers are available at

`http://jdbc.postgresql.org.`

Include the driver JAR file on the class path when running a program that accesses the database.

When you launch programs from the command line, simply use the command

`java -classpath `*`driverPath`*`:. `*`ProgramName`*

On Windows, use a semicolon to separate the current directory (denoted by the . character) from the driver JAR location.

5.3.3 Starting the Database

The database server needs to be started before you can connect to it. The details depend on your database.

With the *Derby* database, follow these steps:

1. Open a command shell and change to a directory that will hold the database files.
2. Locate the file derbyrun.jar. With some versions of the JDK, it is contained in the *jdk*/db/lib directory. If it's not there, install Apache Derby and locate the JAR file in the installation directory. We will denote the directory containing lib/derbyrun.jar with *derby*.
3. Run the command

```
java -jar derby/lib/derbyrun.jar server start
```

4. Double-check that the database is working correctly. Create a file ij.properties that contains these lines:

```
ij.driver=org.apache.derby.jdbc.ClientDriver
ij.protocol=jdbc:derby://localhost:1527/
ij.database=COREJAVA;create=true
```

From another command shell, run Derby's interactive scripting tool (called ij) by executing

```
java -jar derby/lib/derbyrun.jar ij -p ij.properties
```

Now you can issue SQL commands such as

```
CREATE TABLE Greetings (Message CHAR(20));
INSERT INTO Greetings VALUES ('Hello, World!');
SELECT * FROM Greetings;
DROP TABLE Greetings;
```

Note that each command must be terminated by a semicolon. To exit, type
EXIT;

5. When you are done using the database, stop the server with the command
java -jar *derby*/lib/derbyrun.jar server shutdown

5.3.4 Registering the Driver Class

Many JDBC JAR files (such as the Derby driver included with Java SE 8) automatically register the driver class.

A JAR file can automatically register the driver class if it contains a file META-INF/services/java.sql.Driver. We can simply unzip your driver 's JAR file to check.

If your driver 's JAR file doesn't support automatic registration, you need to find out the name of the JDBC driver classes used by your vendor. Typical driver names are:

```
org.apache.derby.jdbc.ClientDriver
org.postgresql.Driver
```

There are two ways to register the driver with the DriverManager. One way is to load the driver class in your Java program. For example,

```
Class.forName("org.postgresql.Driver"); // force loading of driver class
```

This statement causes the driver class to be loaded, thereby executing a static initializer that registers the driver.

Alternatively, you can set the jdbc.drivers property. You can specify the property with a command-line argument, such as

```
java -Djdbc.drivers=org.postgresql.Driver ProgramName
```

Or, your application can set the system property with a call such as

```
System.setProperty("jdbc.drivers", "org.postgresql.Driver");
```

You can also supply multiple drivers; separate them with colons, for example
org.postgresql.Driver:org.apache.derby.jdbc.ClientDriver

5.3.5 Connecting to the Database

In your Java program, open a database connection like this:

```
String url = "jdbc:postgresql:COREJAVA";
String username = "dbuser";
String password = "secret";
Connection conn = DriverManager.getConnection(url, username,
password);
```

The driver manager iterates through the registered drivers to find a driver that can use the subprotocol specified in the database URL.
The getConnection method returns a Connection object. In the following sections, you will see how to use the Connection object to execute SQL statements

Here is an example for connecting to a PostgreSQL database:

```
jdbc.drivers=org.postgresql.Driver
jdbc.url=jdbc:postgresql:COREJAVA
jdbc.username=dbuser
jdbc.password=secret
```

After connecting to the database, the test program executes the following SQL statements:

```
CREATE TABLE Greetings (Message CHAR(20))
INSERT INTO Greetings VALUES ('Hello, World!')
SELECT * FROM Greetings
```

The result of the SELECT statement is printed, and you should see an output of Hello, World!

Then the table is removed by executing the statement

```
DROP TABLE Greetings
```

To run this test, start your database, as described previously, and launch the program as

```
java -classpath .:driverJAR test.TestDB
```

5.4 Working with JDBC Statements

how to use the JDBC Statement to execute SQL statements, obtain results, and deal with errors.

5.4.1 Executing SQL Statements

To execute a SQL statement, you first create a Statement object. To create statement objects, use the Connection object that you obtained from the call to DriverManager.getConnection.

```
Statement stat = conn.createStatement();
```

Next, place the statement that you want to execute into a string, for example

```
String command = "UPDATE Books"
+ " SET Price = Price - 5.00"
+ " WHERE Title NOT LIKE '%Introduction%'";
```

Then call the executeUpdate method of the Statement interface:

```
stat.executeUpdate(command);
```

The executeUpdate method returns a count of the rows that were affected by the SQL statement, or zero for statements that do not return a row count.

When you execute a query, you are interested in the result. The executeQuery object returns an object of type ResultSet that you can use to walk through the result one row at a time.

```
ResultSet rs = stat.executeQuery("SELECT * FROM Books");
```

The basic loop for analyzing a result set looks like this:

```
while (rs.next())
{
```

```
look at a row of the result set
}
```

A large number of accessor methods give you this information.

```
String isbn = rs.getString(1);
double price = rs.getDouble("Price");
```

5.4.2 Managing Connections, Statements, and Result Sets

Every Connection object can create one or more Statement objects.

We can use the same Statement object for multiple unrelated commands and queries. However, a statement has *at most one* open result set.

When we are done using a ResultSet, Statement, or Connection, we should call the close method immediately.

To make absolutely sure that a connection object cannot possibly remain open, use a try-with-resources statement:

```
try (Connection conn = . . .)
{
      Statement stat = conn.createStatement();
      ResultSet result = stat.executeQuery(queryString);
      process query result
}
```

5.4.3 Analyzing SQL Exceptions

Each SQLException has a chain of SQLException objects that are retrieved with the getNextException method.

We can simply use an enhanced for loop:

```
for (Throwable t : sqlException)
{
      do something with t
}
```

We can call getSQLState and getErrorCode on a SQLException to analyze it further.

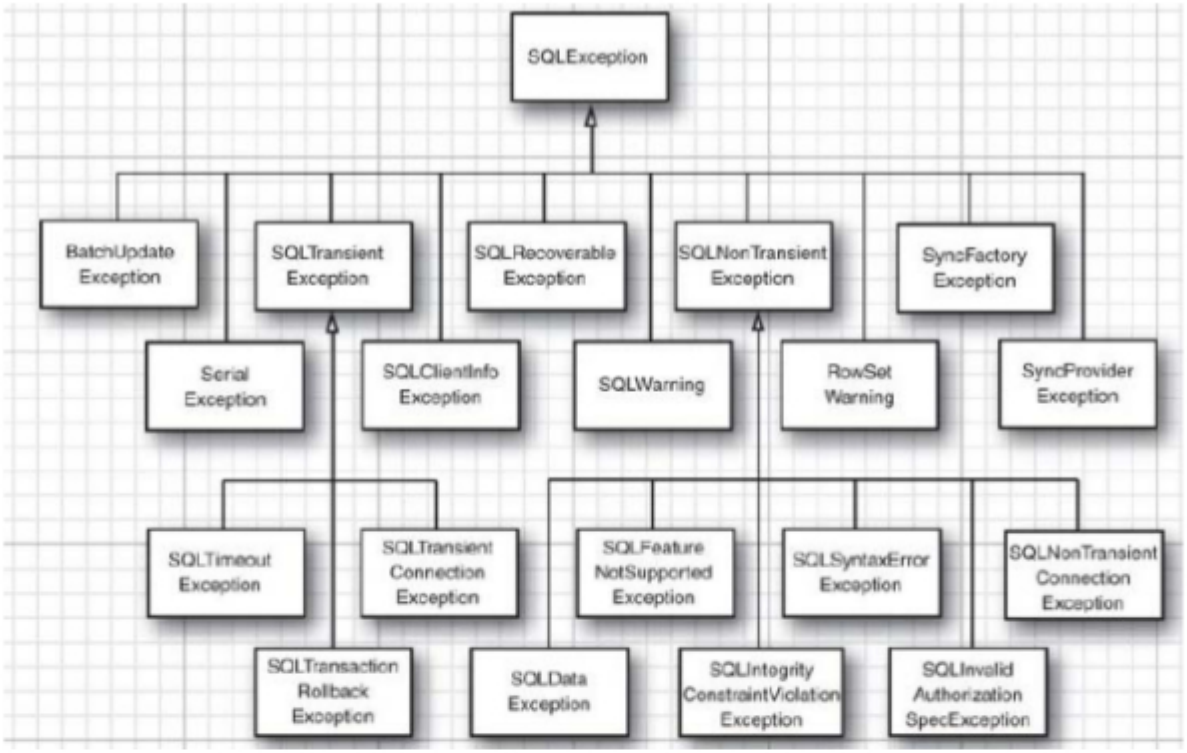

Figure 5.5 SQL exception types

We can retrieve warnings from connections, statements, and result sets.

To retrieve all warnings, use this loop:

```
SQLWarning w = stat.getWarning();
while (w != null)
{
        do something with w
        w = w.nextWarning();
}
```

5.4.4 Populating a Database

Specifically, the program reads data from a text file in a format such as

```
CREATE TABLE Publishers (Publisher_Id CHAR(6), Name CHAR(30), URL CHAR(80));
INSERT INTO Publishers VALUES ('0201', 'Addison-Wesley', 'www.aw-bc.com');
INSERT INTO Publishers VALUES ('0471', 'John Wiley & Sons', 'www.wiley.com');
. . .
```

Make sure that your database server is running, and run the program as follows:

```
java -classpath driverPath:. exec.ExecSQL Books.sql
java -classpath driverPath:. exec.ExecSQL Authors.sql
java -classpath driverPath:. exec.ExecSQL Publishers.sql
java -classpath driverPath:. exec.ExecSQL BooksAuthors.sql
```

The following steps briefly describe the ExecSQL program:

1. Connect to the database. The getConnection method reads the properties in the file database.properties and adds the jdbc.drivers property to the system properties. The getConnection method uses the jdbc.url, jdbc.username, and jdbc.password properties to open the database connection.

2. Open the file with the SQL statements. If no file name was supplied, prompt the user to enter the statements on the console.

3. Execute each statement with the generic execute method. If it returns true, the statement had a result set.

4. If there was a result set, print out the result.

5. If there is any SQL exception, print the exception and any chained exceptions that may be contained in it.
6. Close the connection to the database.

5.5 Query Execution

5.5.1 Prepared Statements

In this program, we use one new feature, *prepared statements*. Consider the query for all books by a particular publisher, independent of the author. The SQL query is:

```
SELECT Books.Price, Books.Title
FROM Books, Publishers
WHERE Books.Publisher_Id = Publishers.Publisher_Id
AND Publishers.Name = the name from the list box
```

Each host variable in a prepared query is indicated with a ?. If there is more than one variable, we must keep track of the positions of the ? when setting the values. For example, our prepared query becomes

```
String publisherQuery =
"SELECT Books.Price, Books.Title" +
" FROM Books, Publishers" +
" WHERE Books.Publisher_Id = Publishers.Publisher_Id AND Publishers.Name =
?";
```

```
PreparedStatement stat = conn.prepareStatement(publisherQuery);
```

Before executing the prepared statement, We must bind the host variables to actual values with a set method. Here, we want to set a string to a publisher name.

```
stat.setString(1, publisher);
```

The first argument is the position number of the host variable that we want to set. The position 1 denotes the first ?. The second argument is the value that we want to assign to the host variable.

Once all variables have been bound to values, you can execute the prepared statement:

```
ResultSet rs = stat.executeQuery();

int r = stat.executeUpdate();
System.out.println(r + " rows updated");
```

```
UPDATE Books
SET Price = Price + ?
WHERE Books.Publisher_Id = (SELECT Publisher_Id FROM Publishers WHERE Name = ?)
```

5.5.2 Reading and Writing LOBs

In addition to numbers, strings, and dates, many databases can store ***large objects (LOBs)*** such as images or other data. In SQL, ***binary large objects are called BLOBs,*** and ***character large objects are called CLOBs.***

To read a LOB, execute a SELECT statement and call the getBlob or getClob method on the ResultSet. We will get an object of type Blob or Clob. To get the binary data from a Blob, call the getBytes or getBinaryStream. For example, if we have a table with book cover images, we can retrieve an image like this:

```
PreparedStatement stat = conn.prepareStatement("SELECT Cover FROM BookCovers WHERE ISBN=?");
        stat.set(1, isbn);
ResultSet result = stat.executeQuery();
if (result.next())
{
        Blob coverBlob = result.getBlob(1);
```

```
        Image coverImage = ImageIO.read(coverBlob.getBinaryStream());
    }
```

here is how We store an image:

```
Blob coverBlob = connection.createBlob();
int offset = 0;
OutputStream out = coverBlob.setBinaryStream(offset);
ImageIO.write(coverImage, "PNG", out);
PreparedStatement stat = conn.prepareStatement("INSERT INTO Cover VALUES
(?, ?)");
stat.set(1, isbn);
stat.set(2, coverBlob);
stat.executeUpdate();
```

4.5.3. SQL Escapes

The "escape" syntax supports features that are commonly supported by databases but use database-specific syntax variations.

It is the job of the JDBC driver to translate the escape syntax to the syntax of a particular database.

Escapes are provided for the following features:

- Date and time literals
- Calling scalar functions
- Calling stored procedures
- Outer joins
- The escape character in LIKE clauses

Use d, t, ts for DATE, TIME, or TIMESTAMP values:

```
{d '2008-01-24'}
{t '23:59:59'}
{ts '2008-01-24 23:59:59.999'}
```

4.5.4. Multiple Results

It is possible for a query to return multiple results. This can happen when executing a stored procedure, or with databases that also allow submission of multiple SELECT statements in a single query. Here is how we retrieve all result sets.

1. Use the execute method to execute the SQL statement.
2. Retrieve the first result or update count.
3. Repeatedly call the getMoreResults method to move on to the next result set.
4. Finish when there are no more result sets or update counts.

```
boolean isResult = stat.execute(command);
boolean done = false;
while (!done)
{
      if (isResult)
      {
      ResultSet result = stat.getResultSet();
      do something with result
}
else
{
int updateCount = stat.getUpdateCount();
if (updateCount >= 0)
      do something with updateCount
else
      done = true;
}
if (!done) isResult = stat.getMoreResults();
}
```

4.5.5. Retrieving Autogenerated Keys

Most databases support some mechanism for autonumbering rows in a database.

These automatic numbers are often used as primary keys. When you insert a new row into a table and a key is automatically generated, We can retrieve it with the following code:

```
stmt.executeUpdate(insertStatement,
Statement.RETURN_GENERATED_KEYS);
ResultSet rs = stmt.getGeneratedKeys();
if (rs.next())
{
int key = rs.getInt(1);
. . .
}
```

4.6. Scrollable and Updatable Result Sets

4.6.1. Scrollable Result Sets

By default, result sets are not scrollable or updatable. To obtain scrollable result sets from queries, we must obtain a different Statement object with the method:

Statement stat = conn.createStatement(type, concurrency);

For a prepared statement, use the call

PreparedStatement stat = conn.prepareStatement(command, type, concurrency);

Table 4.6. ResultSet Type Values

Value	Explanation
TYPE_FORWARD_ONLY	The result set is not scrollable (default).
TYPE_SCROLL_INSENSITIVE	The result set is scrollable but not sensitive to database changes.
TYPE_SCROLL_SENSITIVE	The result set is scrollable and sensitive to database changes.

Table 4.7. ResultSet Concurrency Values

Value	Explanation
CONCUR_READ_ONLY	The result set cannot be used to update the database (default).
CONCUR_UPDATABLE	The result set can be used to update the database.

For example, if we simply want to be able to scroll through a result set but don't want to edit its data, use:

```
Statement stat = conn.createStatement(
ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_READ_ONLY);
```

All result sets that are returned by method calls

```
ResultSet rs = stat.executeQuery(query)
```

are now scrollable. A scrollable result set has a cursor that indicates the current position.

4.6.2. Updatable Result Sets

If we want to edit the result set data and have the changes automatically reflected in the database, create an updatable result set.

To obtain updatable result sets, create a statement as follows:

```
Statement stat = conn.createStatement(
ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_UPDATABLE);
```

The result sets returned by a call to executeQuery are then updatable.

For example, suppose we want to raise the prices of some books, but you don't have a simple criterion for issuing an UPDATE statement. Then, you can iterate through all books and update prices, based on arbitrary conditions.

```
String query = "SELECT * FROM Books";
ResultSet rs = stat.executeQuery(query);
while (rs.next())
{
        if (. . .)
        {
                double increase = . . .
                double price = rs.getDouble("Price");
                rs.updateDouble("Price", price + increase);
```

```
                rs.updateRow(); // make sure to call updateRow after updating fields
            }
        }
```

When you are done inserting, call moveToCurrentRow to move the cursor back to the position before the call to moveToInsertRow. Here is an example:

```
rs.moveToInsertRow();
rs.updateString("Title", title);
rs.updateString("ISBN", isbn);
rs.updateString("Publisher_Id", pubid);
rs.updateDouble("Price", price);
rs.insertRow();
rs.moveToCurrentRow();
```

We can delete the row under the cursor:

```
rs.deleteRow();
```

4.7. Row Sets

Scrollable result sets are powerful, but they have a major drawback. You need to keep the database connection open during the entire user interaction.

The RowSet interface extends the ResultSet interface, but row sets don't have to be tied to a database connection.

4.7.1. Constructing Row Sets

The javax.sql.rowset package provides the following interfaces that extend the RowSet interface:

- A CachedRowSet allows disconnected operation.
- A WebRowSet is a cached row set that can be saved to an XML file.
- The FilteredRowSet and JoinRowSet interfaces support lightweight operations on row sets that are equivalent to SQL SELECT and JOIN operations.
- A JdbcRowSet is a thin wrapper around a ResultSet.

As of Java 7, there is a standard way for obtaining a row set:

```
RowSetFactory factory = RowSetProvider.newFactory();
CachedRowSet crs = factory.createCachedRowSet();
```

If we don't use the RowSetProvider, We should use:

```
CachedRowSet crs = new com.sun.rowset.CachedRowSetImpl();
```

4.7.2. Cached Row Sets

A cached row set contains all data from a result set. Since CachedRowSet is a subinter-face of the ResultSet interface, We can use a cached row set exactly as you would use a result set.

You can populate a CachedRowSet from a result set:

```
ResultSet result = . . .;
CachedRowSet crs = new com.sun.rowset.CachedRowSetImpl();
// or use an implementation from your database vendor
crs.populate(result);
conn.close(); // now OK to close the database connection
```

Alternatively, you can let the CachedRowSet object establish a connection automatically. Set up the database parameters:

```
crs.setURL("jdbc:derby://localhost:1527/COREJAVA");
crs.setUsername("dbuser");
crs.setPassword("secret");
```

Then set the query statement and any parameters.

```
crs.setCommand("SELECT * FROM Books WHERE PUBLISHER = ?");
crs.setString(1, publisherName);
```

Finally, populate the row set with the query result:

```
crs.execute();
```

This call establishes a database connection, issues the query, populates the row set, and disconnects.

If we query result is very large, you would not want to put it into the row set in its entirety. After all, your users will probably only look at a few rows. In that case, specify a page size:

```
CachedRowSet crs = . . .;
crs.setCommand(command);
crs.setPageSize(20);
. . .
crs.execute();
```

Now we will only get 20 rows. To get the next batch of rows, call

```
crs.nextPage();
```

We can inspect and modify the row set with the same methods you use for result sets. If we modified the row set contents, we must write it back to the database by calling

```
crs.acceptChanges(conn);
```

or

```
crs.acceptChanges();
```