JDBC

Taken from
http://java.sun.com/products/jdbc/learning.html

see also the book: Fisher, Ellis, Bruce, “JDBC API Tutorial and Reference”, Third Edition, Addison Wesley

Getting Started

- Install Java on your machine: The JDBC library is included in the J2SE distribution
- Install a driver on your machine: the driver is provided by the database vendor or by a third party.
  - the installation consists of just copying the driver onto your machine; there is no special configuration needed.

Types of JDBC Drivers

- According to the JDBC specification, there are four types of JDBC driver architectures:
- Type 1
  - Drivers that implement the JDBC API as a mapping to another data access API, such as Open Database Connectivity (ODBC). Drivers of this type are generally dependent on a native library, which limits their portability.
  - If you download either the Solaris or Windows versions of JDK1.1 or higher, you will automatically get the JDBC-ODBC Bridge driver, which does not itself require any special configuration.
- Type 2
  - Drivers that are written partly in the Java™ programming language and partly in native code. The drivers use a native client library specific to the data source to which they connect. Because of the native code, their portability is limited.

Types of JDBC Drivers

- Type 3
  - Drivers that use a pure Java client and communicate with a database using a database-independent protocol. The database then communicates the client's requests to the data source.
- Type 4
  - Drivers that are pure Java and implement the network protocol for a specific data source. The client connects directly to the data source.

Scenario

- Small coffee house called The Coffee Break, where coffee beans are sold by the pound
- The database contains only two tables, one for types of coffee and one for coffee suppliers.

JDBC API

- To use the JDBC API is necessary to include in your file the command

  import java.sql.*;

- For some functionalities you need to include also the JDBC Optional Package

  import javax.sql.*;

- Make sure the .jar file containing the JDBC driver is in the classpath or use an IDE such as Eclipse

Establishing a Connection

- Two steps:
  1. loading the driver
  2. making the connection.

Loading the Driver

- Your driver documentation will give you the class name to use. For instance, if the class name is jdbc.DriverXYZ , you would load the driver with the following line of code:
  - Class.forName("jdbc.DriverXYZ");
- Calling Class.forName locates the driver class, loads, and links it, and registers it with the DriverManager

Loading the Driver

- For the JDBC-ODBC Bridge driver
  - Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
- For the SQL Server driver
  - Class.forName("com.microsoft.sqlserver.jdbc.SQL ServerDriver");
- For the DB2 driver

  Class.forName("com.ibm.db2.jcc.DB2Driver");

Making the Connection

Connection con = DriverManager.getConnection(url);

Connection con = DriverManager.getConnection(url, "myLogin", "myPassword");

- The url follows the syntax specified by the driver.
- JDBC-ODBC Bridge driver: the JDBC URL will start with jdbc:odbc: . The rest of the URL is the ODBC data source name.
- If you are using ODBC to access an ODBC data source called "Fred“ with username “Fernanda” and password “J8”, for example, the code would be

String url = "jdbc:odbc:Fred";

Connection con = DriverManager.getConnection(url, "Fernanda", "J8");

SQL Server Driver Url

Type 4 driver:

jdbc:sqlserver://serverName\instance:port;property=value [;property=value]

- **jdbc:sqlserver://** (Required) is the protocol and the sub-protocol and is constant.
- **serverName** (Optional) is the address of the server to connect to. This could be a DNS or IP address, or it could be localhost or 127.0.0.1 for the local computer. If not specified in the connection URL, the server name must be specified in the properties collection.
- **instanceName** (Optional) is the instance to connect to on serverName. If not specified, a connection to the default instance is made.
- **port** (Optional) is the port to connect to on serverName. The default is 1433. If you are using the default, you do not have to specify the port, nor its preceding ':', in the URL.

SQL Server driver Url

- **property** (Optional) is one or more option connection properties. Any property from a list can be specified. Properties can only be delimited by using the semi-colon (';'), and they cannot be duplicated.
- Most important properties:
  - user or userName: the username of the user connecting to the database
  - password: the password

SQL Server Driver Url Examples

- Connect to the default database on the local computer:

jdbc:sqlserver://localhost;user=MyUserName;password=prova

- Connect to a named instance on the local machine:

jdbc:sqlserver://localhost\si2006;user=MyUserName;password=prova

- Connect on the non-default port 4000 to the local machine:

jdbc:sqlserver://localhost:4000;user=MyUserName;password=prova

SQL Server Driver Connection Examples

- Note that if you specify an instance name, you should use double \ in the url

```
Connection con = DriverManager.getConnection(
"jdbc:sqlserver://localhost\\si2006", "myLogin",
    "myPassword");
Connection con = DriverManager.getConnection(
"jdbc:sqlserver://192.168.0.252", "utente", "Infonew1");
```

DB2 Driver URLs

Type 4 Driver

jdbc:db2://serverName[:port]/database:property=value;[…n]

- **jdbc:db2://** (Required) is the protocol and the sub-protocol and is constant.
- **serverName** is the address of the server to connect to. This could be a DNS or IP address, (in our case 192.168.0.252).
- **port** is the port to connect to on serverName. In our case 50000
- **database** is the database to connect to on serverName. In our case PROVA

DB2 Driver URLs

- **property** (Optional) is one or more option connection properties. Any property from a list can be specified. Properties can only be delimited by using the semi-colon (';'), and they cannot be duplicated.
- Most important properties:
  - user: the username of the user connecting to the database
  - password: the password

DB2 Driver URLs Examples

- jdbc:db2://10.17.2.91:50000/PROVA
- jdbc:db2://192.168.0.252:50000/PROVA

DriverManager.getConnection

- Making the Connection
- Connection con = DriverManager.getConnection(url, “utente", “Infonew1");
- The connection returned by the method DriverManager.getConnection is an open connection you can use to create JDBC statements that pass your SQL statements to the DBMS.
- In the previous example, con is an open connection, and we will use it in the examples that follow.

COFFES Table

- It contains the essential information about the coffees sold at The Coffee Break, including the coffee names, the ID of their supplier their prices, the number of pounds sold the current week, and the number of pounds sold to date.

COF_NAME	SUP_ID	PRICE	SALES	TOTAL
Colombian	101	7.99	0	0
French_Roast	49	8.99	0	0
Espresso	150	9.99	0	0
Colombian_Decaf	101	8.99	0	0
French_Roast_Decaf	49	9.99	0	0

SUPPLIERS Table

- SUPPLIERS gives information about each of the suppliers:

SUP_ID	SUP_NAME	STREET	CITY	STATE	ZIP
101	Acme, Inc.	99 Market Street	Groundsville	CA	95199
49	Superior Coffee	1 Party Place	Mendocino	CA	95460
150	The High Ground	100 Coffee Lane	Meadows	CA	93966

CREATE TABLE COFFES

```
CREATE TABLE COFFEES<mat>
(COF_NAME VARCHAR(32),
SUP_ID INTEGER,
PRICE FLOAT,
SALES INTEGER,
TOTAL INTEGER)
```

Table Creation

```
String createTableCoffees = "CREATE TABLE”+
“ COFFEES<mat> (COF_NAME VARCHAR(32), “+
“SUP_ID INTEGER, PRICE FLOAT, " +
"SALES INTEGER, TOTAL INTEGER)";
```

Creating JDBC Statements

- A Statement object is what sends your SQL statement to the DBMS.
- You simply create a Statement object and then execute it, supplying the appropriate execute method together with the SQL statement you want to send.
- For a SELECT statement, the method to use is executeQuery . For statements that create or modify tables, the method to use is executeUpdate.

Creating JDBC Statements

- It takes an instance of an active connection to create a Statement object. In the following example, we use our Connection object con to create the Statement object stmt :

```
Statement stmt = con.createStatement();
```

Executing Statements

- At this point stmt exists, but it does not have an SQL statement to pass on to the DBMS. We need to supply that to the method we use to execute stmt. For example, in the following code fragment, we supply executeUpdate with the SQL statement from the example above:

```
stmt.executeUpdate(createTableCoffees);
```

- We used the method executeUpdate because the SQL statement contained in createTableCoffees is a DDL (data definition language) statement. executeUpdate is also used to execute SQL statements that update a table.

Inserting Data in COFFEES

```
Statement stmt = con.createStatement();
    stmt.executeUpdate( "INSERT INTO
    COFFEES<mat> " + "VALUES ('Colombian', 101,
    7.99, 0, 0)");
```

- The code that follows inserts a second row into the table COFFEES. Note that we can just reuse the Statement object stmt rather than having to create a new one for each execution.

```
stmt.executeUpdate("INSERT INTO COFFEES<mat> "
    + "VALUES ('French_Roast', 49, 8.99, 0, 0)");
```

Inserting the Remaining Data in COFFEES

```
stmt.executeUpdate("INSERT INTO COFFEES<mat> "
    +
    "VALUES ('Espresso', 150, 9.99, 0, 0)");
stmt.executeUpdate("INSERT INTO COFFEES<mat> "
    +
    "VALUES ('Colombian_Decaf', 101, 8.99, 0, 0)");
stmt.executeUpdate("INSERT INTO COFFEES<mat> "
    +
    "VALUES ('French_Roast_Decaf', 49, 9.99, 0, 0)");
```

Selecting Data from COFFEES

ResultSet rs = stmt.executeQuery("SELECT“ +

” COF_NAME, PRICE FROM COFFEES<mat>");

- JDBC returns results in a ResultSet object, so we need to declare a ResultSet variable to hold our results.

Retrieving Results

- The variable rs, which is an instance of ResultSet, contains the rows of coffees and prices shown above.
- In order to access the names and prices, we will go to each row and retrieve the values according to their types. The method next moves what is called a **cursor** to the next row and makes that row (called the current row) the one upon which we can operate.
- Since the cursor is initially positioned just above the first row of a ResultSet object, the first call to the method next moves the cursor to the first row and makes it the current row.
- Successive invocations of the method next move the cursor down one row at a time from top to bottom.

next

- next returns true if the successive row is a valid row otherwise it returns false
- If it is called when the cursor is on the last row it returns false
- It can be used for cycles

Retrieving Results

- We use the getXXX method of the appropriate type to retrieve the value in each column. For example, the first column in each row of rs is COF_NAME, which stores a value of SQL type VARCHAR .
- The method for retrieving a value of SQL type VARCHAR is getString.
- The second column in each row stores a value of SQL type FLOAT, and the method for retrieving values of that type is getFloat.

Retrieving Results

```
String query = "SELECT COF_NAME, PRICE “+
“ FROM COFFEES<mat>";
ResultSet rs = stmt.executeQuery(query);
while (rs.next()) {
    String s = rs.getString("COF_NAME");
    float n = rs.getFloat("PRICE");
    System.out.println(s + " " + n);
}
```

Details

String s = rs.getString("COF_NAME");

- The method getString will retrieve (get) the value stored in the column COF_NAME in the current row of rs. The value that getString retrieves has been converted from an SQL VARCHAR to a String in the Java programming language, and it is assigned to the String object s.
- The situation is similar with the method getFloat except that it retrieves the value stored in the column PRICE, which is an SQL FLOAT, and converts it to a Java float before assigning it to the variable n .

getXXX

- JDBC offers two ways to identify the column from which a getXXX method gets a value.
- One way is to give the column name, as was done in the example above.
- The second way is to give the column index (number of the column), with 1 indicating the first column, 2 , the second, and so on:

```
String s = rs.getString(1);
float n = rs.getFloat(2);
```

getXXX

- Using the column number is slightly more efficient, and there are some cases where the column number is required.
- In general, though, supplying the column name is essentially equivalent to supplying the column number.

getXXX

- JDBC allows a lot of latitude as far as which getXXX methods you can use to retrieve the different SQL types.
- For example, the method getInt can be used to retrieve any of the numeric or character types. The data it retrieves will be converted to an int; that is, if the SQL type is VARCHAR, JDBC will attempt to parse an integer out of the VARCHAR.
- The method getInt is recommended for retrieving only SQL INTEGER types, however, and it cannot be used for the SQL types BINARY, VARBINARY, LONGVARBINARY, DATE, TIME, or TIMESTAMP.

getXXX

- Although the method getString is recommended for retrieving the SQL types CHAR and VARCHAR , it is possible to retrieve any of the basic SQL types with it.
- For instance, if it is used to retrieve a numeric type, getString will convert the numeric value to a Java String object.

Updating Tables

- Suppose that after a successful first week, the owner of The Coffee Break wants to update the SALES column in the table COFFEES by entering the number of pounds sold for each type of coffee. The SQL statement to update one row might look like this:

Updating Tables

String updateString = "UPDATE COFFEES<mat> " +

"SET SALES = 75 " +

"WHERE COF_NAME LIKE 'Colombian'";

- Using the Statement object stmt , this JDBC code executes the SQL statement contained in updateString :

stmt.executeUpdate(updateString);

Seeing the Results

```
String query = "SELECT COF_NAME, SALES FROM”+
“COFFEES<mat> WHERE COF_NAME LIKE
    'Colombian'";
ResultSet rs = stmt.executeQuery(query);
while (rs.next()) {
    String s = rs.getString("COF_NAME");
    int n = rs.getInt("SALES");
    System.out.println(n + " pounds of " + s +
    " sold this week.");
}
```

Seeing the Results

- Prints the following:

75 pounds of Colombian sold this week.

- Since the WHERE clause limited the selection to only one row, there was just one row in the ResultSet rs and one line was printed as output. Accordingly, it is possible to write the code without a while loop:

```
rs.next();
String s = rs.getString(1);
int n = rs.getInt(2);
System.out.println(n + " pounds of " + s +
" sold this week.");
```

Updating the TOTAL

```
String updateString = "UPDATE COFFEES<mat> " +
"SET TOTAL = TOTAL + 75 " +
"WHERE COF_NAME LIKE 'Colombian'";
stmt.executeUpdate(updateString);
String query = "SELECT COF_NAME, TOTAL “
+”FROM COFFEES<mat> WHERE COF_NAME LIKE “+
    “'Colombian'";
ResultSet rs = stmt.executeQuery(query);
while (rs.next()) {
      String s = rs.getString(1);
      int n = rs.getInt(2);
      System.out.println(n + " pounds of " + s + " sold to date.");
}
```

Prepared Statements

- PreparedStatement is a subclass of Statement.
- It is given an SQL statement when it is created.
- Two advantages:
  - In most cases, this SQL statement will be sent to the DBMS right away, where it is compiled. This means that when the PreparedStatement is executed, the DBMS can just run the PreparedStatement 's SQL statement without having to compile it first.
  - The SQL statement can have parameters

Creating a PreparedStatement Object

- With the prepareStatement method of a Connection object

```
PreparedStatement updateSales =
    con.prepareStatement(
"UPDATE COFFEES<mat> SET SALES = ? WHERE “+
    “COF_NAME LIKE ?");
```

- The question marks are placeholders for parameters

Supplying Values for PreparedStatement Parameters

- You will need to supply values to be used in place of the question mark placeholders, if there are any, before you can execute a PreparedStatement object.
- You do this by calling one of the setXXX methods defined in the class PreparedStatement .
- In general, there is a setXXX method for each type in the Java programming language.

```
updateSales.setInt(1, 75);
updateSales.setString(2, "Colombian");
```

- First argument: parameter position
- Second argument: parameter value

Executing a Prepared Statement

```
PreparedStatement updateSales =
    con.prepareStatement(
"UPDATE COFFEES<mat> SET SALES = ? WHERE “+
    “COF_NAME LIKE ?");
updateSales.setInt(1, 75);
updateSales.setString(2, "Colombian");
updateSales.executeUpdate():
```

executeUpdate takes no argument because the SQL statement is already stored in PreparedStatement

Changing Parameters

- Once a parameter has been set with a value, it will retain that value until it is reset to another value or the method clearParameters is called

Changing Parameters

```
updateSales.setInt(1, 100);
updateSales.setString(2, "French_Roast");
updateSales.executeUpdate();
// changes SALES column of French Roast row to 100
updateSales.setString(2, "Espresso");
updateSales.executeUpdate();
// changes SALES column of Espresso row to 100 (the
//  first  parameter stayed 100, and the second
// parameter was set to "Espresso")
```

Using a Loop to Set Values

```
PreparedStatement updateSales;
String updateString = "update COFFEES<mat> " +
                      "set SALES = ? where COF_NAME like ?";
updateSales = con.prepareStatement(updateString);
int [] salesForWeek = {175, 150, 60, 155, 90};
String [] coffees = {"Colombian", "French_Roast", "Espresso",
                     "Colombian_Decaf", "French_Roast_Decaf"};
int len = coffees.length;
for(int i = 0; i < len; i++) {
            updateSales.setInt(1, salesForWeek[i]);
            updateSales.setString(2, coffees[i]);
            updateSales.executeUpdate();
    }
```

Return Values for the Method executeUpdate

- The return value for executeUpdate is an int that indicates how many rows of a table were updated.

```
updateSales.setInt(1, 50);
updateSales.setString(2, "Espresso");
int n = updateSales.executeUpdate();
// n = 1 because one row had a change in it
```

- When the method executeUpdate is used to execute a DDL statement, such as in creating a table, it returns the int 0.

```
int n = executeUpdate(createTableCoffees); // n = 0
```

Return Values for the Method executeUpdate

- When the return value for executeUpdate is 0, it can mean one of two things:
  1. the statement executed was an update statement that affected zero rows,
  2. the statement executed was a DDL statement.

Creating SUPPLIERS

- We need to create the table SUPPLIERS and populate it with values.

```
String createSUPPLIERS = "create table SUPPLIERS<mat> "+
"(SUP_ID INTEGER, SUP_NAME VARCHAR(40), " +
     "STREET VARCHAR(40), CITY VARCHAR(20), " +
 "STATE CHAR(2), ZIP CHAR(5))";
stmt.executeUpdate(createSUPPLIERS);
```

Populating SUPPLIERS with Values

```
stmt.executeUpdate("insert into SUPPLIERS<mat> values (101, " +
 "'Acme, Inc.', '99 Market Street', 'Groundsville', " +
"'CA', '95199'");
stmt.executeUpdate("Insert into SUPPLIERS<mat> values (49," +
 "'Superior Coffee', '1 Party Place', 'Mendocino', 'CA', '95460'");
stmt.executeUpdate("Insert into SUPPLIERS<mat> values (150, " +
 "'The High Ground', '100 Coffee Lane', 'Meadows', 'CA', " +
"'93966'");
ResultSet rs = stmt.executeQuery("select * from SUPPLIERS<mat>");
```

SUP_ID	SUP_NAME	STREET	CITY	STATE	ZIP
101	Acme, Inc.	99 Market Street	Groundsville	CA	95199
49	Superior Coffee	1 Party Place	Mendocino	CA	95460
150	The High Ground	100 Coffee Lane	Meadows	CA	93966

Query with a Join

```
String query = "SELECT COFFEES<mat>.COF_NAME " +
   "FROM COFFEES<mat>, SUPPLIERS<mat> " +
   "WHERE SUPPLIERS<mat>.SUP_NAME LIKE 'Acme, Inc.' " +
   "and SUPPLIERS<mat>.SUP_ID = COFFEES<mat>.SUP_ID";
ResultSet rs = stmt.executeQuery(query);
System.out.println("Coffees bought from Acme, Inc.: ");
while (rs.next()) {
    String coffeeName = rs.getString("COF_NAME");
    System.out.println("     " + coffeeName);
}
```

Transactions

- When a connection is created, it is in auto-commit mode.
- This means that each individual SQL statement is treated as a transaction and will be automatically committed right after it is executed.
- To disable auto-commit mode:

```
con.setAutoCommit(false);
```

- where con is an active connection

Transactions

- To commit a transaction use

con.commit();

- To roll back a transaction use

con.rollback();

- To set a transaction isolation level use the Connection method setTransactionIsolation that takes an int
- To find out what transaction isolation level your DBMS is set to, use the Connection method getTransactionIsolation that returns an int

Isolation Levels

- Five constants defined in the Connection interface:
  - TRANSACTION_NONE
  - TRANSACTION_READ_UNCOMMITTED
  - TRANSACTION_READ_COMMITTED
  - TRANSACTION_REPEATABLE_READ
  - TRANSACTION_SERIALIZABLE
- Example

```
con.setTransactionIsolation(
    Connection.TRANSACTION_READ_COMMITTED);
```

Isolation Levels

- Even though JDBC allows you to set a transaction isolation level, doing so will have no effect unless the driver and DBMS you are using support it.

Example

```
PreparedStatement updateSales;
PreparedStatement updateTotal;
String updateString="UPDATE COFFEES<mat> SET SALES = ?
    WHERE"+" COF_NAME = ?";
String updateStatement="UPDATE COFFEES<mat> SET TOTAL =
    TOTAL + ? WHERE COF_NAME = ?";
String query = "select COF_NAME, SALES, TOTAL from "+
    "COFFEES<mat>";
try {
    con = DriverManager.getConnection(url,"myLogin", "myPassword");
    updateSales = con.prepareStatement(updateString);
    updateTotal = con.prepareStatement(updateStatement);
    int [] salesForWeek = {175, 150, 60, 155, 90};
    String [] coffees = {"Colombian", "French_Roast", "Espresso",
    "Colombian_Decaf","French_Roast_Decaf"};
```

Example

```
int len = coffees.length;
con.setAutoCommit(false);
for (int i = 0; i < len; i++) {
        updateSales.setInt(1, salesForWeek[i]);
        updateSales.setString(2, coffees[i]);
        updateSales.executeUpdate();
        updateTotal.setInt(1, salesForWeek[i]);
        updateTotal.setString(2, coffees[i]);
        updateTotal.executeUpdate();
        con.commit();
    }
```

Example

```
    con.setAutoCommit(true);
    updateSales.close(); updateTotal.close();
    stmt = con.createStatement();
    ResultSet rs = stmt.executeQuery(query);
    while (rs.next()) {
        String c = rs.getString("COF_NAME");
        int s = rs.getInt("SALES");
        int t = rs.getInt("TOTAL");
        System.out.println(c + "     " +  s + "    " + t);
    }
    stmt.close();
    con.close();
}
```

Example

```
catch(SQLException ex) {
    System.err.println("SQLException: " + ex.getMessage());
    if (con != null) {
    try {
            System.err.print("Transaction is being ");
            System.err.println("rolled back");
            con.rollback();
    }
    catch(SQLException excep) {
            System.err.print("SQLException: ");
            System.err.println(excep.getMessage());
    }
}
```

Stored Procedures

```
String createProcedure =
"create procedure SHOW_SUPPLIERS " +
 "as " +
 "select SUPPLIERS<mat>.SUP_NAME, “+
“COFFEES<mat>.COF_NAME " +
 "from SUPPLIERS<mat>, COFFEES<mat> " +
 "where SUPPLIERS<mat>.SUP_ID = COFFEES<mat>.SUP_ID "
    +
 "order by SUP_NAME";
Statement stmt = con.createStatement();
stmt.executeUpdate(createProcedure);
```

Stored Procedures

- The procedure SHOW_SUPPLIERS will be compiled and stored in the database as a database object that can be called, similar to the way you would call a method.

Calling a Stored Procedure from JDBC

- The first step is to create a CallableStatement object.
- As with Statement and PreparedStatement objects, this is done with an open Connection object.
- A CallableStatement object contains a call to a stored procedure; it does not contain the stored procedure itself.

```
CallableStatement cs = con.prepareCall(
    "{call SHOW_SUPPLIERS}");
ResultSet rs = cs.executeQuery();
```

Escape Syntax

- {call SHOW_SUPPLIERS} is the escape syntax for stored procedures.
- When the driver encounters "{call SHOW_SUPPLIERS}" , it will translate this escape syntax into the native SQL used by the database to call the stored procedure named SHOW_SUPPLIERS .

Execution of the CallableStatement

- Note that the method used to execute cs is executeQuery because cs calls a stored procedure that contains one query and thus produces one result set.
- If the procedure had contained one update or one DDL statement, the method executeUpdate would have been the one to use.
- It is sometimes the case, however, that a stored procedure contains more than one SQL statement, in which case it will produce more than one result set, more than one update count, or some combination of result sets and update counts. In this case, where there are multiple results, the method execute should be used to execute the CallableStatement .

Parameters

- The class CallableStatement is a subclass of PreparedStatement , so a CallableStatement object can take input parameters just as a PreparedStatement object can.

Exceptions

```
try {
    Class.forName("myDriverClassName");
} catch(java.lang.ClassNotFoundException e) {
    System.err.print("ClassNotFoundException: ");
    System.err.println(e.getMessage());
}
```

Exceptions

```
try {
    // Code that could generate an exception goes here.
    // If an exception is generated, the catch block below
    // will print out information about it.
} catch(SQLException ex) {
    System.err.println("SQLException: " +
    ex.getMessage());
}
```

Exceptions

- If you were to run CreateCOFFEES.java twice, you would get an error message similar to this:

SQLException: There is already an object named

'COFFEES' in the database.

Severity 16, State 1, Line 1

- This example illustrates printing out the message component of an SQLException object, which is sufficient for most situations.

Exceptions

- The SQLException object contains three parts:
  - the message (a string that describes the error),
  - the SQL state (a string identifying the error according to the X/Open SQLState conventions),
  - the vendor error code (a number that is the driver vendor's error code number)

Example

```
try {// Code that could generate an exception goes here.
    // If an exception is generated, the catch block below
    // will print out information about it.
} catch(SQLException ex) {
    System.out.println("\n--- SQLException caught ---\n");
    while (ex != null) {
            System.out.println("Message:   "+ ex.getMessage ());
            System.out.println("SQLState:  "+ ex.getSQLState ());
            System.out.println("ErrorCode: "+ ex.getErrorCode ());
            ex = ex.getNextException();
            System.out.println("");
    }
}
```

Example

- If you try to create the table COFFEES twice, you would get the following printout:

--- SQLException caught ---

Message: There is already an object named 'COFFEES'

in the database. Severity 16, State 1, Line 1

SQLState: 42501

ErrorCode: 2714

- SQLState is a code defined in X/Open and ANSI-92 that identifies the exception. Two examples of SQLState code numbers and their meanings follow:
  - 08001 -- No suitable driver
  - HY011 -- Operation invalid at this time
- The vendor error code is specific to each driver, so you need to check your driver documentation for a list of error codes and what they mean.

Moving the Cursor in Scrollable Result Set

- Up to now we have seen the features in the JDBC 1.0 API
- One of the new features in the JDBC 2.0 API is the ability to move a result set's cursor backward as well as forward.
- There are also methods that let you move the cursor to a particular row and check the position of the cursor.
- Scrollable result sets make it possible to create a GUI (graphical user interface) tool for browsing result sets.
- Another use is moving to a row in order to update it.

Scrollable ResultSet

- The following line of code illustrates one way to create a scrollable ResultSet object:

```
Statement stmt = con.createStatement(
    ResultSet.TYPE_SCROLL_SENSITIVE,
    ResultSet.CONCUR_READ_ONLY);
ResultSet srs = stmt.executeQuery(
  "SELECT COF_NAME, PRICE FROM COFFEES<mat>");
```

Scrollable ResultSet

- Two arguments to the method createStatement.
  - The first argument is one of three constants added to the ResultSet API to indicate the type of a ResultSet object: TYPE_FORWARD_ONLY , TYPE_SCROLL_INSENSITIVE , and TYPE_SCROLL_SENSITIVE .
  - The second argument is one of two ResultSet constants for specifying whether a result set is read-only or updatable: CONCUR_READ_ONLY and CONCUR_UPDATABLE .
- If you specify a type, you must also specify whether it is read-only or updatable.
- Also, you must specify the type first, and because both parameters are of type int , the compiler will not complain if you switch the order.

TYPE_FORWARD_ONLY

- Specifying the constant TYPE_FORWARD_ONLY creates a nonscrollable result set, that is, one in which the cursor moves only forward.
- If you do not specify any constants for the type and updatability of a ResultSet object, you will automatically get one that is TYPE_FORWARD_ONLY and CONCUR_READ_ONLY (as is the case when you are using only the JDBC 1.0 API).

TYPE_SCROLL_INSENSITIVE or TYPE_SCROLL_SENSITIVE

- You will get a scrollable ResultSet object if you specify one of the following ResultSet constants: TYPE_SCROLL_INSENSITIVE or TYPE_SCROLL_SENSITIVE The difference between the two has to do with whether a result set reflects changes that are made to it while it is open and whether certain methods can be called to detect these changes.
- Generally speaking,
  - a result set that is TYPE_SCROLL_INSENSITIVE does not reflect changes made by others while it is still open and
  - one that is TYPE_SCROLL_SENSITIVE does.
- All three types of result sets will make changes visible if they are closed and then reopened.
- No matter what type of result set you specify, you are always limited by what your DBMS and driver actually provide

Scrollable ResultSet

- Once you have a scrollable ResultSet object, srs in the previous example, you can use it to move the cursor around in the result set.
- Even when a result set is scrollable, the cursor is initially positioned before the first row.

next and previous

- The counterpart to the method next, is the new method previous, which moves the cursor backward (one row towards the beginning of the result set).
- next and previous return false when the cursor goes beyond the result set (to the position after the last row or before the first row), which makes it possible to use them in a while loop.

Using previous

```
srs.afterLast();
while (srs.previous()) {
    String name = srs.getString("COF_NAME");
    float price = srs.getFloat("PRICE");
    System.out.println(name + "     " + price);
}
```

afterLast moves the cursor explicitly to the position after the last row

Moving the Cursor

- The methods first, last, beforeFirst, and afterLast move the cursor to the row indicated in their names.
- The method absolute will move the cursor to the row number indicated in the argument passed to it.
  - If the number is positive, the cursor moves the given number from the beginning, so calling absolute(1) puts the cursor on the first row.
  - If the number is negative, the cursor moves the given number from the end, so calling absolute(-1) puts the cursor on the last row.

absolute

- The following line of code moves the cursor to the fourth row of srs :

srs.absolute(4);

- If srs has 500 rows, the following line of code will move the cursor to row 497:

srs.absolute(-4);

Relative Moves

- Three methods move the cursor to a position relative to its current position.
  - next
  - previous
  - relative: you can specify how many rows to move from the current row. A positive number moves the cursor forward the given number of rows; a negative number moves the cursor backward the given number of rows.

relative Example

```
srs.absolute(4); // cursor is on the fourth row
. . .
srs.relative(-3); // cursor is on the first row
. . .
srs.relative(2); // cursor is on the third row
```

getRow

- The method getRow lets you check the number of the row where the cursor is positioned.

```
srs.absolute(4);
int rowNum = srs.getRow(); // rowNum should be 4
srs.relative(-3);
int rowNum = srs.getRow(); // rowNum should be 1
srs.relative(2);
int rowNum = srs.getRow(); // rowNum should be 3
```

Position Tests

- Four additional methods let you verify whether the cursor is at a particular position.
- The position is stated in their names: isFirst , isLast , isBeforeFirst , isAfterLast .
- These methods all return a boolean and can therefore be used in a conditional statement.

isAfterLast Example

```
if (!srs.isAfterLast()) {
    srs.afterLast();
}
while (srs.previous()) {
    String name = srs.getString("COF_NAME");
    float price = srs.getFloat("PRICE");
    System.out.println(name + "     " + price);
}
```

Making Updates to Updatable Result Sets

- With the JDBC 2.0 API it is possible to update rows in a result set using methods in the Java programming language rather than having to send an SQL command.
- You need to create a ResultSet object that is updatable. In order to do this, you supply the ResultSet constant CONCUR_UPDATABLE to the createStatement method.

Making Updates to Updatable Result Sets

- An updatable ResultSet object does not necessarily have to be scrollable, but when you are making changes to a result set, you generally want to be able to move around in it.
- With a scrollable result set, you can move to rows you want to change, and if the type is TYPE_SCROLL_SENSITIVE, you can get the new value in a row after you have changed it.

Example

```
Connection con = DriverManager.getConnection(
    "jdbc:mySubprotocol:mySubName");
Statement stmt = con.createStatement(
    ResultSet.TYPE_SCROLL_SENSITIVE,
    ResultSet.CONCUR_UPDATABLE);
ResultSet uprs = stmt.executeQuery(
    "SELECT COF_NAME, PRICE FROM COFFEES<mat>");
```

Notes

- Just specifying that a result set be updatable does not guarantee that the result set you get is updatable.
- If a driver does not support updatable result sets, it will return one that is readonly.
- The query you send can also make a difference. In order to get an updatable result set, the query must generally specify the primary key as one of the columns selected, and it should select columns from only one table.

Notes

- The following line of code checks whether the ResultSet object *uprs* is updatable.
- int concurrency = uprs.getConcurrency();
- The variable *concurrency* will be one of the following:
  - 1007 to indicate ResultSet.CONCUR_READ_ONLY
  - 1008 to indicate ResultSet.CONCUR_UPDATABLE

uprs Content

COF_NAME	PRICE
Colombian	7.99
French_Roast	8.99
Espresso	9.99
Colombian_Decaf	8.99
French_Roast_Decaf	9.99

Updating a Result Set Programmatically

- Suppose that we want to raise the price of French Roast Decaf coffee to 10.99. Using the JDBC 1.0 API, the update would look something like this:

```
stmt.executeUpdate(
    "UPDATE COFFEES<mat> SET PRICE = 10.99 " +
    "WHERE COF_NAME = 'French_Roast_Decaf'");
```

- In JDBC 2.0

```
uprs.last();
uprs.updateFloat("PRICE", 10.99f);
```

Update Operations

- Update operations in the JDBC 2.0 API affect column values in the row where the cursor is positioned,
- All of the update methods you call will operate on that row until you move the cursor to another row.
- The ResultSet.updateXXX methods generally take two parameters:
  - the column to update, either by column name or by column number.
  - the new value to put in that column.
- There is a different updateXXX method for updating each data type (updateString, updateBigDecimal, updateInt, and so on)

Update Operations

- At this point, the price in *uprs* for French Roast Decaf will be 10.99, but the price in the table COFFEES in the database will still be 9.99.
- To make the update take effect in the database, we must call the ResultSet method updateRow.

```
uprs.updateRow();
```

- Note that you must call the method updateRow before moving the cursor. If you move the cursor to another row before calling updateRow, the updates are lost, that is, the row will revert to its previous column values.

Update Operations

- Suppose that you realize that the update you made is incorrect.
- You can restore the previous value by calling the cancelRowUpdates method if you call it before you have called the method updateRow.
- Once you have called updateRow, the method cancelRowUpdates will no longer work.

```
uprs.last();
  uprs.updateFloat("PRICE", 10.99f);
  . . .
  uprs.cancelRowUpdates();
```

Update Operations

- If you want to update the price for Colombian_Decaf, you have to move the cursor to the row containing that variety of coffee.
- Because the row for Colombian_Decaf immediately precedes the row for French_Roast_Decaf, you can call the method previous to position the cursor on the row for Colombian_Decaf.

```
uprs.previous();
uprs.updateFloat("PRICE", 9.79f);
uprs.updateRow();
```

Notes

- All cursor movements refer to rows in a ResultSet object, not rows in the underlying database.
- The ordering of the rows in the result set has nothing at all to do with the order of the rows in the base table.
- In fact, the order of the rows in a database table is indeterminate. The driver keeps track of which rows were selected, and it makes updates to the proper rows, but they may be located anywhere in the table.

Inserting and Deleting Rows Programmatically

- With the JDBC 1.0 API

```
stmt.executeUpdate("INSERT INTO COFFEES<mat> "
    +
    "VALUES ('Kona', 150, 10.99, 0, 0)");
```

Inserting and Deleting Rows Programmatically

- In the JDBC 2.0 API every ResultSet object has a row called the *insert row*, a special row in which you can build a new row.
- Steps:
  1. move the cursor to the insert row, which you do by invoking the method moveToInsertRow.
  2. set a value for each column in the row. You do this by calling the appropriate updateXXX method for each value.
  3. call the method insertRow to insert the row you have just populated with values into the result set. This method simultaneously inserts the row into both the ResultSet object and the database table from which the result set was selected.

Example

```
Statement stmt = con.createStatement(
                    ResultSet.TYPE_SCROLL_SENSITIVE,
                    ResultSet.CONCUR_UPDATABLE);
 ResultSet uprs = stmt.executeQuery(
                  "SELECT * FROM COFFEES<mat>");
```

Example

```
uprs.moveToInsertRow();
uprs.updateString("COF_NAME", "Kona");
uprs.updateInt("SUP_ID", 150);
uprs.updateFloat("PRICE", 10.99f);
uprs.updateInt("SALES", 0);
uprs.updateInt("TOTAL", 0);

uprs.insertRow();
```

Example (Alternative Solution)

```
uprs.moveToInsertRow();
uprs.updateString(1, "Kona");
uprs.updateInt(2, 150);
uprs.updateFloat(3, 10.99f);
uprs.updateInt(4, 0);
uprs.updateInt(5, 0);

uprs.insertRow();
```

updateXXX

- In both updates and insertions, calling an updateXXX method does not affect the underlying database table.
- The method updateRow must be called to have updates occur in the database.
- For insertions, the method insertRow inserts the new row into the result set and the database at the same time.

Inserting

- What happens if you insert a row without supplying a value for every column in the row?
- If a column has a default value or accepts SQL NULL values, you can get by with not supplying a value.
- If a column does not have a default value and does not accept NULL, you will get an SQLException if you fail to set a value for it.
- You will also get an SQLException if a required table column is missing in your ResultSet object.
- In the example above, the query was SELECT * FROM COFFEES<mat>, which produced a result set with all the columns of all the rows. When you want to insert one or more rows, your query does not have to select all rows, but you should generally select all columns.

Inserting

- After you have called the method insertRow, you can start building another row to be inserted,
- Note that you if you move the cursor from the insert row before calling the method insertRow, you will lose all of the values you have added to the insert row.

Moving from the Insert Row

- When you call the method moveToInsertRow, the result set keeps track of which row the cursor is sitting on, which is, by definition, the current row.
- The method moveToCurrentRow, which you can invoke only when the cursor is on the insert row, moves the cursor from the insert row back to the row that was previously the current row.
- To move the cursor from the insert row back to the result set, you can also invoke any of the methods that move the cursor: first, last, beforeFirst, afterLast, absolute, previous, relative.

Deleting a Row Programmatically

- You simply move the cursor to the row you want to delete and then call the method deleteRow.
- Example:

```
uprs.absolute(4);
uprs.deleteRow();
```

- These two lines of code remove the fourth row from *uprs* and also from the database.

Issue

- With some JDBC drivers, a deleted row is removed and is no longer visible in a result set.
- Some JDBC drivers use a blank row as a placeholder (a "hole") where the deleted row used to be.
- If there is a blank row in place of the deleted row, you can use the method absolute with the original row positions to move the cursor because the row numbers in the result set are not changed by the deletion.
- You can use methods in the DatabaseMetaData interface to discover the exact behavior of your driver.

Seeing Changes in Result Sets

- Result sets vary greatly in their ability to reflect changes made in their underlying data.
- If you modify data in a ResultSet object, the change will always be visible if you close it and then reopen it during a transaction.
- You will also see changes made by others when you reopen a result set if your transaction isolation level makes them visible.

Seeing Changes in Result Sets

- So when can you see visible changes you or others made while the ResultSet object is still open? (Generally, you will be most interested in the changes made by others because you know what changes you made yourself.)
- The answer depends on the type of ResultSet object you have.
- With a ResultSet object that is TYPE_SCROLL_SENSITIVE, you can always see **visible** updates made by you and others to existing column values. You may see inserted and deleted rows, but the only way to be sure is to use DatabaseMetaData methods that return this information.

Seeing Changes in Result Sets

- Visible updates depend on the transaction isolation level.
- With the isolation level READ COMMITTED, a TYPE_SCROLL_SENSITIVE result set will not show any changes before they are committed, but it can show changes that may have other consistency problems.

Seeing Changes in Result Sets

- In a ResultSet object that is TYPE_SCROLL_INSENSITIVE, you cannot see changes made to it by others while it is still open, but you may be able to see your own changes with some implementations.
- This is the type of ResultSet object to use if you want a consistent view of data and do not want to see changes made by others.

Getting the Most Recent Data

- You can do this using the method refreshRow, which gets the latest values for a row straight from the database.
- This method can be relatively expensive, especially if the DBMS returns multiple rows each time you call refreshRow. Nevertheless, its use can be valuable if it is critical to have the latest data.
- Even when a result set is sensitive and changes are visible, an application may not always see the very latest changes that have been made to a row if the driver retrieves several rows at a time and caches them.

Getting the Most Recent Data

- Note that the result set should be sensitive; if you use the method refreshRow with a ResultSet object that is TYPE_SCROLL_INSENSITIVE, refreshRow does nothing.

Example

```
Statement stmt = con.createStatement(
            ResultSet.TYPE_SCROLL_SENSITIVE,
            ResultSet.CONCUR_READ_ONLY);
 ResultSet srs = stmt.executeQuery(“SELECT COF_NAME,
            PRICE FROM COFFEES<mat> “);
 srs.absolute(4);
 float price1 = srs.getFloat("PRICE");
 // do something. . .
 srs.absolute(4);
 srs.refreshRow();
 float price2 = srs.getFloat("PRICE");
 if (price2 > price1) {
   // do something. . .
 }
```

Making Batch Updates

- A batch update is a set of multiple update statements that is submitted to the database for processing as a batch.
- Sending batch updates can, in some situations, be much more efficient than sending update statements separately.
- It is a JDBC 2.0 feature

Using Statement Objects for Batch Updates

- Statement, PreparedStatement and CallableStatement objects have a list of commands that is associated with them.
- This list may contain statements for updating, inserting, or deleting a row; and it may also contain DDL statements such as CREATE TABLE and DROP TABLE.
- It cannot, however, contain a statement that would produce a ResultSet object, such as a SELECT statement.
- In other words, the list can contain only statements that produce an update count.

Using Statement Objects for Batch Updates

- The list, which is associated with a Statement object at its creation, is initially empty.
- You can add SQL commands to this list with the method addBatch and empty it with the method clearBatch.
- When you have finished adding statements to the list, you call the method executeBatch to send them all to the database to be executed as a unit, or batch.

Example

```
con.setAutoCommit(false);
Statement stmt = con.createStatement();
stmt.addBatch("INSERT INTO COFFEES<mat> " +
              "VALUES('Amaretto', 49, 9.99, 0, 0)");
stmt.addBatch("INSERT INTO COFFEES<mat> " +
              "VALUES('Hazelnut', 49, 9.99, 0, 0)");
stmt.addBatch("INSERT INTO COFFEES<mat> " +
              "VALUES('Amaretto_decaf', 49, 10.99, 0, 0)");
stmt.addBatch("INSERT INTO COFFEES<mat> " +
              "VALUES('Hazelnut_decaf', 49, 10.99, 0, 0)");
  int [] updateCounts = stmt.executeBatch();
  con.commit();
  con.setAutoCommit(true);
```

Notes

- To allow for correct error handling, you should always disable auto-commit mode before beginning a batch update.
- executeBatch: the DBMS will execute the commands in the order in which they were added to the list of commands.
- If all four commands execute successfully, the DBMS will return an update count for each command in the order in which it was executed.
- The update counts, int values indicating how many rows were affected by each command, are stored in the array *updateCounts*.

Parameterized Batch Update

```
con.setAutoCommit(false);
PreparedStatement pstmt = con.prepareStatement(
            "INSERT INTO COFFEES<mat> VALUES(?, ?, ?, ?, ?)");
pstmt.setString(1, "Amaretto");      pstmt.setInt(2, 49);
pstmt.setFloat(3, 9.99);             pstmt.setInt(4, 0);
pstmt.setInt(5, 0);
pstmt.addBatch();

pstmt.setString(1, "Hazelnut");      pstmt.setInt(2, 49);
pstmt.setFloat(3, 9.99);             pstmt.setInt(4, 0);
pstmt.setInt(5, 0);
pstmt.addBatch();

int [] updateCounts = pstmt.executeBatch();
con.commit();
con.setAutoCommit(true);
```

Batch Update Exceptions

- You will get a BatchUpdateException when you call the method executeBatch if
  1. one of the SQL statements you added to the batch produces a result set (usually a query) or
  2. one of the SQL statements in the batch does not execute successfully for some other reason.

Batch Update Exceptions

- A BatchUpdateException contains an array of update counts that is similar to the array returned by the method executeBatch.
- In both cases, the update counts are in the same order as the commands that produced them.
- This tells you how many commands in the batch executed successfully and which ones they are.

Batch Update Exceptions

- BatchUpdateException is derived from SQLException.
  This means that you can use all of the methods available to an SQLException object with it.

Example

```
try {
  // make some updates
  } catch(BatchUpdateException b) {
    System.err.println("----BatchUpdateException----");
    System.err.println("SQLState:  " + b.getSQLState());
    System.err.println("Message:  " + b.getMessage());
    System.err.println("Vendor:  " + b.getErrorCode());
    System.err.print("Update counts:  ");
    int [] updateCounts = b.getUpdateCounts();
    for (int i = 0; i < updateCounts.length; i++) {
      System.err.print(updateCounts[i] + "   ");
    }
    System.err.println("");
  }
```

SQL-99 Data Types

- The JDBC 2.0 API provides interfaces that represent the mapping of the new SQL3 (SQL-99) data types into the Java programming language.
- With these new interfaces, you can work with SQL3 data types the same way you do other data types.

Using Data Sources

- Alternative way to connect to a database
- DataSource objects should be used whenever possible
- Advantages
  - code portability
  - connection pooling
  - distributed transactions.
- This functionality is integral to Enterprise JavaBeans (EJB) technology.

DataSource

- A DataSource object represents a particular DBMS or some other data source, such as a file.
- The system administrator has to deploy the DataSource objects so that the programmers can start using them.
- Deploying a DataSource object consists of three tasks:
  1. Creating an instance of the DataSource class
  2. Setting its properties
  3. Registering it with a naming service that uses the Java Naming and Directory Interface (JNDI) API

DataSource Advantages

- A DataSource object is a better alternative than the DriverManager facility for getting a connection.
  - Programmers no longer have to hard code the driver name or JDBC URL in their applications, which makes them more portable.

DataSource Advantages

- DataSource properties make maintaining code much simpler. If there is a change, the system administrator can simply update the data source's properties, and you don't have to worry about changing every application that makes a connection to the data source. For example, if the data source was moved to a different server, all the system administrator would need to do is set the serverName property to the new server name.
- Pooled connections.
- Distributed transaction.

JDBC 3.0 Functionalities

- Savepoints
- Getting the keys automatically generated by the database.

Savepoints

- Intermediate points within a transaction
- You can roll back to a savepoint, thus undoing only part of the work
  - Everything before the savepoint will be saved and everything afterwards will be rolled back
- To set a savepoint

```
Savepoint save=con.setSavepoint();
```

- To delete a savepoint

```
con.releaseSavepoint(save);
```

- To roll back to a savepoint

```
con.rollback(save);
```

Example

- Rising the prices of the most popular coffes, up to a limit
- If a price gets too high, roll back to the most recent price increase
- Two increase rounds,
  - the first for the coffes with more than 7000 pounds
  - the second for the coffes with more than 8000 pounds

Example

```
con.setAutoCommit(false);
String query = "SELECT COF_NAME, PRICE FROM
    COFFEES<mat> " + "WHERE TOTAL > ?";
String update = "UPDATE COFFEES<mat> SET PRICE = ? " +
    "WHERE COF_NAME = ?";
PreparedStatement getPrice = con.prepareStatement(query);
PreparedStatement updatePrice = con.prepareStatement(
    update);
getPrice.setInt(1, 7000);
ResultSet rs = getPrice.executeQuery();
Savepoint save1 = con.setSavepoint();
```

Example

```
while (rs.next())  {
    String cof = rs.getString("COF_NAME");
    float oldPrice = rs.getFloat("PRICE");
    float newPrice = oldPrice + (oldPrice * .05f);
    updatePrice.setFloat(1, newPrice);
    updatePrice.setString(2, cof);
    updatePrice.executeUpdate();
    System.out.println("New price of " + cof + " is " + newPrice);
    if (newPrice > 11.99) {
            con.rollback(save1);
    }
}
```

Example

```
getPrice = con.prepareStatement(query);
updatePrice = con.prepareStatement(update);
getPrice.setInt(1, 8000);
rs = getPrice.executeQuery();
System.out.println();
Savepoint save2 = con.setSavepoint();
```

Example

```
while (rs.next())  {
    String cof = rs.getString("COF_NAME");
    float oldPrice = rs.getFloat("PRICE");
    float newPrice = oldPrice + (oldPrice * .05f);
    updatePrice.setFloat(1, newPrice);
    updatePrice.setString(2, cof);
    updatePrice.executeUpdate();
    System.out.println("New price of " + cof + " is " +
    newPrice);
    if (newPrice > 11.99) {
            con.rollback(save2);
    }
}
```

Example

```
con.commit();
Statement stmt = con.createStatement();
rs = stmt.executeQuery("SELECT COF_NAME, " +
    "PRICE FROM COFFEES<mat>");
System.out.println();
while (rs.next()) {
    String name = rs.getString("COF_NAME");
    float price = rs.getFloat("PRICE");
    System.out.println("Current price of " + name +
            " is " + price);
}
con.close();
```

Getting Automatically Generated Keys

- Some DBMS automatically generate a key for a row that is inserted in a table.
- If you later want to update that row, you can use this key to identify the row
- You can find out whether your driver supports automatically generated keys with

```
DatabaseMetaData dbmd=con.getMetaData();
Boolean b = dbmd.supportsGetGeneratedKeys();
```

Getting Automatically Generated Keys

- If your driver supports them and you want to use them, you have to tell the Statement object to do it

```
stmt.executeUpdate(sqlstring,
    Statement.RETURN_GENERATED_KEYS);
```

- For a prepared statement

```
pstmt=con.prepareStatement(sqlString,
    Statement.RETURN_GENERATED_KEYS);
```

Getting Automatically Generated Keys

- After executing the statement, you retrieve the generated keys with

ResultSet keys=stmt.getGeneratedKeys();

- Each key will be a row in keys.
- It is possible for a key to be more than one column, so the row will have as many columns as the key

Getting Automatically Generated Keys

- Another way to signal the driver that it should prepare for returning generated keys is to pass it an array with the column names
- In this case

String [] keyArray={“KEY”};

Stmt.executeUpdate(sqlString,keyArray);

Metadata

- A DatabaseMetaData object provides information about a database or a DBMS
- The developers of the driver implemented the DatabaseMetaData interface so that its method return information about the driver and the database
- A ResultSetMetaData object provides information about the columns in a particular ResultSet instance
- A ParameterMetaData object provides information about the parameters in a PreparedStatement object

Rowsets

- A RowSet object is similar to a ResultSet object but it allows to work on the data also if the computer is disconnected from the database
- Example:
  - The user retrieves the data on a portable pc
  - The user disconnects from the network
  - The user works on the data
  - When the user reconnects to the network, the data is synchronized with the data on the DBMS
- Rowsets optimize the sending of data through a network

Java Persistence API

- API of Java Enterprise Edition
- Allows the persistent storage (on disk) of Java objects
- Performed by means of an object relational mapping that allow the storage in relational databases
- Contains also a special query language