Introduction to Java™

Database connectivity with JDBC

Introduction

- Working with JDBC involves these steps:
  - Install the needed JDBC driver
    - ✓Just download the driver and put it in your classpath
  - Establish a connection to the DB
  - Send statements to the DB
    - ✓These can be either SQL statement or calls to stored procedures
  - Get their result
    - ✓Utilizing the notion of result sets

Establishing a Connection

- First, you need to establish a connection with the DBMS you want to use. Typically, a JDBC application connects to a target data source using one of two mechanisms:
  - **DriverManager**: This fully implemented class requires an application to load a specific driver, using a hardcoded URL.
    - ✓ As part of its initialization, the DriverManager class attempts to load the driver classes referenced in the jdbc.drivers system property.
    - ✓ This allows you to customize the JDBC Drivers used by your applications.
  - **DataSource**: This interface is preferred over DriverManager because it allows details about the underlying data source to be transparent to your application.
    - ✓ A DataSource object's properties are set so that it represents a particular data source.
- Establishing a connection involves two steps: Loading the driver, and making the connection.

Loading the Driver

- Loading the driver you want to use is very simple. It involves just one line of code in your program. To use the Java DB driver, add the following line of code:

```
Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver");
```

  - In this example, we load the driver for MS SQL *http://msdn.microsoft.com/library/ms378672.aspx*

Loading the Driver

- Calling the *Class.forName* automatically creates an instance of a driver and registers it with the DriverManager.
  - You don't need to create an instance of the class.
  - If you were to create your own instance, you would be creating an unnecessary duplicate, but it would do no harm.
- After you have loaded a driver, it can make a connection with a DBMS.
- The second step in establishing a connection is to have the appropriate driver connect to the DBMS.

Connecting to a DB: Using the DriverManager Class

- Connection URLs have the following form:
  - *jdbc:<subprotocol>:<dbName>[propertyList]*
  - The *subprotocol* portion of the URL identifies a the DBMS to use.
    - ✓ For example, if the driver developer has registered the name *sqlserver* as the subprotocol, the first and second parts of the JDBC URL will be jdbc:sqlserver .
  - The *dbName* portion of the URL identifies a specific database.
  - A database can be in one of many locations: in the current working directory, on the classpath, in a JAR file, in a specific Java DB database home directory, or in an absolute location on your file system.
  - The driver documentation will also give you guidelines for the rest of the JDBC URL.
  - This last part of the JDBC URL supplies information for identifying the data source.

Connecting to a DB: Using the DriverManager Class

- The getConnection method establishes a connection:

```
Connection conn =
DriverManager.getConnection("jdbc:sqlserver://[serverName
[\instanceName][:portNumber]][;property=value[;property=v
alue]]");
```

- For more details for MS SQL connection URL refer to *http://msdn.microsoft.com/library/ms378672.aspx*
- So, if you log in to your MS SQL (local) server with a login name of "Fernanda" and a password of "J8", just these two lines of code will establish a connection:

```
String url =
"jdbc:sqlserver://localhost:1433;databaseName=Northwind;
user=Fernanda;password=J8;"
Connection con = DriverManager.getConnection(url);
```

Retrieving Values from Result Sets

- The ResultSet interface provides methods for retrieving and manipulating the results of executed queries.
- ResultSet objects can have different functionality and characteristics.
  - These characteristics are result set type, result set concurrency, and cursor holdability.
  - A table of data representing a database result set is usually generated by executing a statement that queries the database.
- The type of a ResultSet object determines the level of its functionality in two areas:
  - the ways in which the cursor can be manipulated.
  - how concurrent changes made to the underlying data source are reflected by the ResultSet object.

Retrieving Values from Result Sets

- The sensitivity of the ResultSet object is determined by one of three different ResultSet types:
  - TYPE_FORWARD_ONLY — The result set is not scrollable; its cursor moves forward only, from before the first row to after the last row. The rows contained in the result set depend on how the underlying database materializes the results. That is, it contains the rows that satisfy the query at either the time the query is executed or as the rows are retrieved.
  - TYPE_SCROLL_INSENSITIVE — The result set is scrollable; its cursor can move both forward and backward relative to the current position, and it can move to an absolute position.
  - TYPE_SCROLL_SENSITIVE — The result set is scrollable; its cursor can move both forward and backward relative to the current position, and it can move to an absolute position.

Sending SELECT statements from a Java program to the DB

- JDBC returns results in a ResultSet object, so we need to declare an instance of the class ResultSet to hold our results.
- In addition, the Statement methods executeQuery and getResultSet both return a ResultSet object, as do various DatabaseMetaData methods.

Using the executeQuery method

- First you need to create a scrollable ResultSet object.

```
Statement stmt =
     con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
     ResultSet.CONCUR_READ_ONLY);
ResultSet srs = stmt.executeQuery("SELECT * FROM COFFEES");
```

- 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.

Using the executeQuery method

- The point to remember here is that 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.
- 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.
- The variable *srs*, which is an instance of ResultSet, contains the rows of coffees and prices stored in the table COFFEES.
- In order to access the names and prices a ResultSet object maintains a cursor, which points to its current row of data.
- When a ResultSet object is first created, the cursor is positioned before the first row.

Using the ResultSet Methods

- To move the cursor, you can use the following methods:
  - next() - moves the cursor forward one row. Returns true if the cursor is now positioned on a row and false if the cursor is positioned after the last row.
  - previous() - moves the cursor backwards one row. Returns true if the cursor is now positioned on a row and false if the cursor is positioned before the first row.
  - first() - moves the cursor to the first row in the ResultSet object. Returns true if the cursor is now positioned on the first row and false if the ResultSet object does not contain any rows.
  - last() - moves the cursor to the last row in the ResultSet object. Returns true if the cursor is now positioned on the last row and false if the ResultSet object does not contain any rows.
  - beforeFirst() - positions the cursor at the start of the ResultSet object, before the first row. If the ResultSet object does not contain any rows, this method has no effect.
  - afterLast() - positions the cursor at the end of the ResultSet object, after the last row. If the ResultSet object does not contain any rows, this method has no effect.
  - relative(int rows) - moves the cursor relative to its current position.
  - absolute(int row) - positions the cursor on the row-th row of the ResultSet object.

Using the ResultSet Methods

- 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.
- 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.

Using the get*XXX* Methods

- The ResultSet interface declares getter methods (getBoolean, getLong, and so on) for retrieving column values from the current row.
- Your application can retrieve values using either the index number of the column or the name of the column.
  - The column index is usually more efficient.
  - Columns are numbered from 1.
  - For maximum portability, result set columns within each row should be read in left-to-right order, and each column should be read only once.
  - Column names used as input to getter methods are case insensitive.

Using the get*XXX* Methods

- When a getter method is called with a column name and several columns have the same name, the value of the first matching column will be returned.
  - The column name option is designed to be used when column names are used in the SQL query that generated the result set.
  - For columns that are NOT explicitly named in the query, it is best to use column numbers.
  - If column names are used, the programmer should take care to guarantee that they uniquely refer to the intended columns, which can be assured with the SQL AS clause.

Using the get*XXX* Methods

- The get*XXX* method of the appropriate type retrieves the value in each column.
  - For example, the first column in each row of srs 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.
  - The following code accesses the values stored in the current row of srs and prints a line with the name followed by three spaces and the price.

Using the get*XXX* Methods

- Each time the method next is invoked, the next row becomes the current row, and the loop continues until there are no more rows in rs.
- The method getString is invoked on the ResultSet object srs, so getString retrieves (gets) the value stored in the column COF_NAME in the current row of srs.
  - 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.
  - Note that 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.
    - ✓ You cannot, however, retrieve the new SQL3 datatypes with it.
- Getting all values with getString can be very useful, but it also has its limitations.
  - If it is used to retrieve a numeric type, getString converts the numeric value to a Java String object, and the value has to be converted back to a numeric type before it can be operated on as a number.

Example

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

ResultSet srs =
       stmt.executeQuery( "SELECT COF_NAME, PRICE FROM COFFEES");

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

- The output will look something like this:
  Colombian 7.99
  Colombian_Decaf 8.99
  French_Roast_Decaf 9.99

Example

- You can process all of the rows is srs going backward, but to do this, the cursor must start out located after the last row.
  - You can move the cursor explicitly to the position after the last row with the method afterLast.
  - Then the method previous() moves the cursor from the position after the last row to the last row, and then to the previous row with each iteration through the while loop.
  - The loop ends when the cursor reaches the position before the first row, where the method previous() returns false .

Example

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

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

- The printout will look similar to this:
  French_Roast_Decaf 9.99
  Colombian_Decaf 8.99
  Colombian 7.99

Using the get*XXX* Methods

- JDBC offers two ways to identify the column from which a get*XXX* 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).
    - ✓ Using the column number instead of the column name looks like this:
      ```
      String s = srs.getString(1);
      float n = srs.getFloat(2);
      ```
    - ✓ The first line of code gets the value in the first column of the current row of rs (column COF_NAME), converts it to a Java String object, and assigns it to s.
    - ✓ The second line of code gets the value stored in the second column of the current row of rs , converts it to a Java float, and assigns it to n.
    - ✓ Note that the column number refers to the column number in the result set, not in the original table.

Moving in a ResultSet

- You can move the cursor to a particular row in a ResultSet object.
  - 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.
    - ✓ The following code moves the cursor to the fourth row of srs:

```
srs.absolute(4);
```

    - ✓ If srs has 500 rows, the following moves the cursor to row 497:

```
srs.absolute(-4);
```

Moving in a ResultSet

- Three methods move the cursor to a position relative to its current position.
  - The method next moves the cursor forward one row, and the method previous moves the cursor backward one row.
  - With the method relative, you can specify how many rows to move from the current row and also the direction in which to move.
    - A positive number moves the cursor forward the given number of rows; a negative number moves the cursor backward the given number of rows.

```
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
```

Other ResultSet methods

- The method getRow lets you check the number of the row where the cursor is positioned.
  - For example, you can use getRow to verify the current position of the cursor in the previous example as follows:

```
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
```

Other ResultSet methods

- 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.
  - For example

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

Using the get*XXX* Methods

- 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.

Updating Tables

- Updating a row in a ResultSet object is a two-phase process.
  - First, the new value for each column being updated is set.
  - Then the change is applied to the row.
  - The row in the underlying data source is not updated until the second phase is completed.
- The ResultSet interface contains two update methods for each JDBC type, one specifying the column to be updated as an index and one specifying the column name as it appears in the select list.
  - Column names supplied to updater methods are case insensitive.
  - If a select list contains the same column more than once, the first instance of the column will be updated.

Updating Tables

- First, you need to create a ResultSet object that is updatable.
- To do this, supply the ResultSet constant CONCUR_UPDATABLE to the createStatement method
  - The Statement object it creates produces an updatable ResultSet object each time it executes a query
  - 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

- The method updateRow applies all column changes to the current row.
  - The changes are not made to the row until updateRow has been called.
  - You can use the cancelUpdates method to back out changes made to the row before the updateRow method is called.

```
Statement stmt =
       conn.createStatement(ResultSet.TYPE_FORWARD_ONLY,
       ResultSet.CONCUR_UPDATABLE);
ResultSet srs =
       stmt.executeQuery("select COF_Name from COFFEES
       where price = 7.99");

srs.next();
srs.updateString("COF_NAME", "Foldgers");
srs.updateRow();
```

Using the update*XXX* Methods

- An update is the modification of a column value in the current row.
  - Update operations 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. update*XXX* methods take two parameters: the column to update and the new value to put in that column.
- As with the ResultSet.getXXX methods, the parameter designating the column may be either the column name or the column number.
- There is a different update*XXX* method for updating each datatype (updateString, updateBigDecimal, updateInt, and so on) just as there are different getXXX methods for retrieving different datatypes.

Using the update*XXX* Methods

- To make the update take effect in the database and not just the result set, we must call the ResultSet method updateRow.
  - E.g.

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

- If you had moved the cursor to a different row before calling the method updateRow, the update would have been lost.

Using the update*XXX* Methods

- You can cancel the update by calling the method cancelRowUpdates.
  - You have to invoke cancelRowUpdates before invoking the method updateRow.
    - ✓ Once updateRow is called, calling the method cancelRowUpdates does nothing.
  - Note that cancelRowUpdates cancels all of the updates in a row.
    - ✓ If there are many invocations of the update*XXX* methods on the same row, you cannot cancel just one of them.
- The concept to remember is that updates and related operations apply to the row where the cursor is positioned.
  - Even if there are many calls to update*XXX* methods, it takes only one call to the method updateRow to update the database with all of the changes made in the current row.

ResultSet movement

- All cursor movements refer to rows in a ResultSet object, not rows in the underlying database.
  - If a query selects five rows from a database table, there will be five rows in the result set, with the first row being row 1, the second row being row 2, and so on.
  - Row 1 can also be identified as the first, and, in a result set with five rows, row 5 is the last.
  - 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 DBMS keeps track of which rows were selected, and it makes updates to the proper rows, but they may be located anywhere in the table.
    - ✓ When a row is inserted, for example, there is no way to know where in the table it has been inserted.

Inserting with Identity Columns

```
CREATE TABLE TestTable Col1 int IDENTITY, Col2 varchar(50), Col3 int);

String SQL = "INSERT INTO TestTable (Col2, Col3) VALUES ('S', 50)";
Statement stmt = con.createStatement();
int count = stmt.executeUpdate(SQL, Statement.RETURN_GENERATED_KEYS);
ResultSet rs = stmt.getGeneratedKeys();
ResultSetMetaData rsmd = rs.getMetaData();
int columnCount = rsmd.getColumnCount();
if (rs.next()) {
   do {
      for (int i=1; i<=columnCount; i++) {
        System.out.println("KEY " + i + " = " + rs.getString(i));
      }
   } while(rs.next());
} else {
   System.out.println("NO KEYS WERE GENERATED.");
}
rs.close();
```

Using Prepared Statements

- Sometimes it is more convenient to use a PreparedStatement object for sending SQL statements to the database.
  - This special type of statement is derived from the more general class, Statement.
  - If you want to execute a Statement object many times, it normally reduces execution time to use a PreparedStatement object instead.
- The main feature of a PreparedStatement object is that, unlike a Statement object, it is given an SQL statement when it is created.
  - The advantage to this is that in most cases, this SQL statement is sent to the DBMS right away, where it is compiled.
    - ✓ As a result, the PreparedStatement object contains an SQL statement that has been precompiled.
    - ✓ Thus the DBMS can just run the PreparedStatement SQL statement without having to compile it first.

Using Prepared Statements

- Although PreparedStatement objects can be used for SQL statements with no parameters, you probably use them most often for SQL statements that take parameters.
  - The advantage of using SQL statements that take parameters is that you can use the same statement and supply it with different values each time you execute it.

Creating a PreparedStatement Object

- As with Statement objects, you create PreparedStatement objects with a Connection method.

```
PreparedStatement updateSales =
    con.prepareStatement( "UPDATE COFFEES
        SET SALES = ? WHERE COF_NAME LIKE ?");
```

- The variable updateSales now contains the SQL statement, "UPDATE COFFEES SET SALES = ? WHERE COF_NAME LIKE ?",
  - In most cases, it has also been sent to the DBMS and been precompiled.

Supplying Values for PreparedStatement Parameters

- You 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 PreparedStatement class.
  - If the value you want to substitute for a question mark is a Java int, you call the method setInt.
  - If the value you want to substitute for a question mark is a Java String, you call the method setString, and so on.
  - In general, there is a *setXXX* method for each primitive type declared in the Java programming language.
- Example:

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

  - The first argument given to a *setXXX* method indicates which question mark placeholder is to be set, and the second argument indicates the value to which it is to be set.

Supplying Values for PreparedStatement Parameters

Code Fragment 1:

```
Statement stmt =
con.createStatement(ResultSet.TYP
E_SCROLL_SENSITIVE,
ResultSet.CONCUR_UPDATABLE);

String updateString = "UPDATE
COFFEES SET SALES = 75 " +
"WHERE COF_NAME LIKE
'Colombian'";

stmt.executeUpdate(updateString);
```

Code Fragment 2:

```
PreparedStatement
updateSales =
con.prepareStatement(
"UPDATE COFFEES SET SALES =
? WHERE COF_NAME LIKE ? ");

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

updateSales.executeUpdate();
```

Using Prepared Statements

- The method executeUpdate was used to execute both the Statement stmt and the PreparedStatement updateSales.
  - Notice, however, that no argument is supplied to executeUpdate when it is used to execute updateSales.
  - This is because updateSales already contains the SQL statement to be executed.
- Looking at these examples, you might wonder why you would choose to use a PreparedStatement object with parameters instead of just a simple statement, since the simple statement involves fewer steps.
  - If you were going to update the SALES column only once or twice, then there would be no need to use an SQL statement with input parameters.
  - If you will be updating often, on the other hand, it might be much easier to use a PreparedStatement object, especially in situations where you can use a for loop or while loop to set a parameter to a succession of values.
- Once a parameter has been set with a value, it retains that value until it is reset to another value, or the method clearParameters is called.

Reusing a Prepared Statement

```
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 reset to "Espresso")
```

- You can often make coding easier by using a for loop or a while loop to set values for input parameters.

Reusing a Prepared Statement

```
PreparedStatement updateSales;

String updateString = "update COFFEES " + "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"};

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

Return Values for the executeUpdate Method

- Whereas executeQuery returns a ResultSet object containing the results of the query sent to the DBMS, the return value for executeUpdate is an int that indicates how many rows of a table were updated.
  - Example:

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

    - ✓ The table COFFEES was updated by having the value 50 replace the value in the column SALES in the row for Espresso.
    - ✓ That update affected one row in the table, so n is equal to 1.

Return Values for the executeUpdate Method

- When the method executeUpdate is used to execute a DDL statement, such as in creating a table, it returns 0.
- Note that when the return value for executeUpdate is 0, it can mean one of two things:
  - the statement executed was an update statement that affected zero rows
  - the statement executed was a DDL statement.

Stored Procedures

- A stored procedure is a group of SQL statements that form a logical unit and perform a particular task.
- Stored procedures are used to encapsulate a set of operations or queries to execute on a database server.
  - For example, operations on an employee database (hire, fire, promote, lookup) could be coded as stored procedures executed by application code.
  - Stored procedures can be compiled and executed with different parameters and results, and they may have any combination of input, output, and input/output parameters.
- Stored procedures are supported by most DBMSs, but there is a fair amount of variation in their syntax and capabilities.
- A simple example of like and how a MS SQL server stored procedure is invoked from JDBC follows.
  - *http://msdn.microsoft.com/library/ms378672.aspx*

Calling a Stored Procedure from JDBC (for MS SQL Server)

- 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.
- The first line of code below creates a call to the stored procedure SHOW_SUPPLIERS using the connection con.
  - The part that is enclosed in curly braces is the escape syntax for stored procedures.
  - When the driver encounters "{call dbo.SHOW_SUPPLIERS}", it will translate this escape syntax into the native SQL used by the database to call the stored procedure named SHOW_SUPPLIERS.

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

Calling a Stored Procedure from JDBC (for MS SQL Server)

- The JDBC API provides a stored procedure SQL escape syntax that allows stored procedures to be called in a standard way for all RDBMSs.
  - This escape syntax has one form that includes a result parameter and one that does not.
  - If used, the result parameter must be registered as an OUT parameter.
  - The other parameters can be used for input, output or both. Parameters are referred to sequentially, by number, with the first parameter being 1.
- Escape syntax:
  - {call <procedure-name>[(<arg1>,<arg2>, ...)]}
  - Examples: {call proc1(?,?,?) } {call proc2(?) }
- IN parameter values are set using the set methods inherited from PreparedStatement.
- The type of all OUT parameters must be registered prior to executing the stored procedure.
  - This is done with method registerOutParameter of the CallableStatement class.
  - Their values are retrieved after execution via the get methods provided here.

Calling a Stored Procedure from JDBC (for MS SQL Server) - Example

```
CREATE PROCEDURE GetManager @employeeID INT, @managerID INT OUTPUT AS
BEGIN
    SELECT @managerID = ManagerID
    FROM HumanResources.Employee
    WHERE EmployeeID = @employeeID
END
```

```
CallableStatement cstmt = con.prepareCall("{call
                                dbo.GetManager(?, ?)}");
cstmt.setInt(1, 5);
cstmt.registerOutParameter(2, java.sql.Types.INTEGER);
cstmt.execute();
System.out.println("MANAGER ID: " + cstmt.getInt(2));
cstmt.close();
```

Calling a Stored Procedure from JDBC (for MS SQL Server) - Example

```
CREATE PROCEDURE CheckContactCity (@cityName CHAR(50)) AS
BEGIN
   IF ((SELECT COUNT(*) FROM Person.Address WHERE City = @cityName) > 1)
      RETURN 1
   ELSE
      RETURN 0
END
```

```
CallableStatement cstmt = con.prepareCall("{? = call
                                    dbo.CheckContactCity(?)}");
cstmt.registerOutParameter(1, java.sql.Types.INTEGER);
cstmt.setString(2, "Atlanta");
cstmt.execute();
System.out.println("RETURN STATUS: " + cstmt.getInt(1));
cstmt.close();
```

Calling a Stored Procedure from JDBC

- 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.
  - If a stored procedure contains more than one SQL statement (thus producing more than one result set, or more than one update count, or some combination of result sets and update counts) the method *execute* should be used to execute the CallableStatement .
- INOUT parameters and the method execute are used rarely.

Calling a Function from JDBC

- This is highly dependent on the DBMS and the JDBC driver used.
  - Most vendors provide a specific escape syntax to enable this functionality.
  - MS SQL JDBC driver provides the following syntax (for built-in functions): {fn functionName}
    - ✓ functionName is a function supported by the JDBC driver. For example:

      ```
      SELECT {fn UCASE(Name)} FROM Employee
      ```

    - ✓ For MS SQL server it will work even without using the escape syntax (just use the function name).
  - User defined functions (UDF) can be called in the same way as stored procedures.
    - ✓ To call a UDF as part of a SELECT statement you should use the fully qualified name of the UDF (e.g. dbo.myFunction()).

```
String sql = "SELECT dbo.myFunction([Full Name]) AS Lname, FROM Employees";
ResultSet rs = stmt.executeQuery(sql);
while (rs.next()) {
    System.out.println(rs.getString("LName"));
}
```

A simple JDBC program

- Check out the SimpleJDBC.java file for a full example of a simple JDBC program