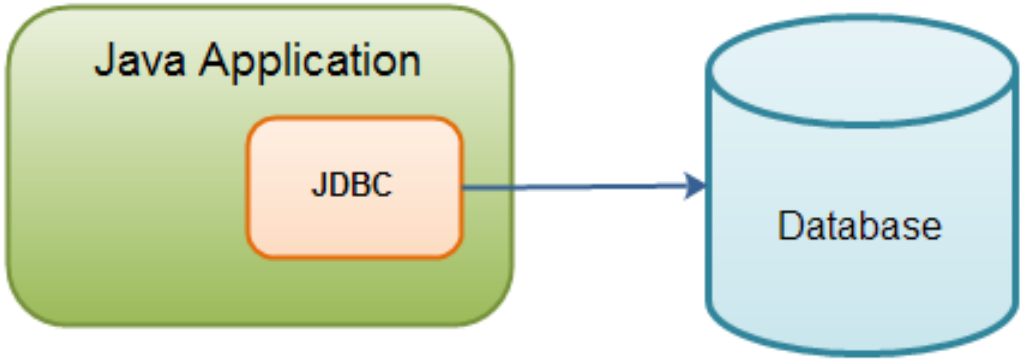
Programmation Orientée Objet
Partie 4:Connexion aux Bases de données
ISIL -- ESTE



Introduction to JDBC

Pr. Said BENKIRANE

2017/2018

Road Map

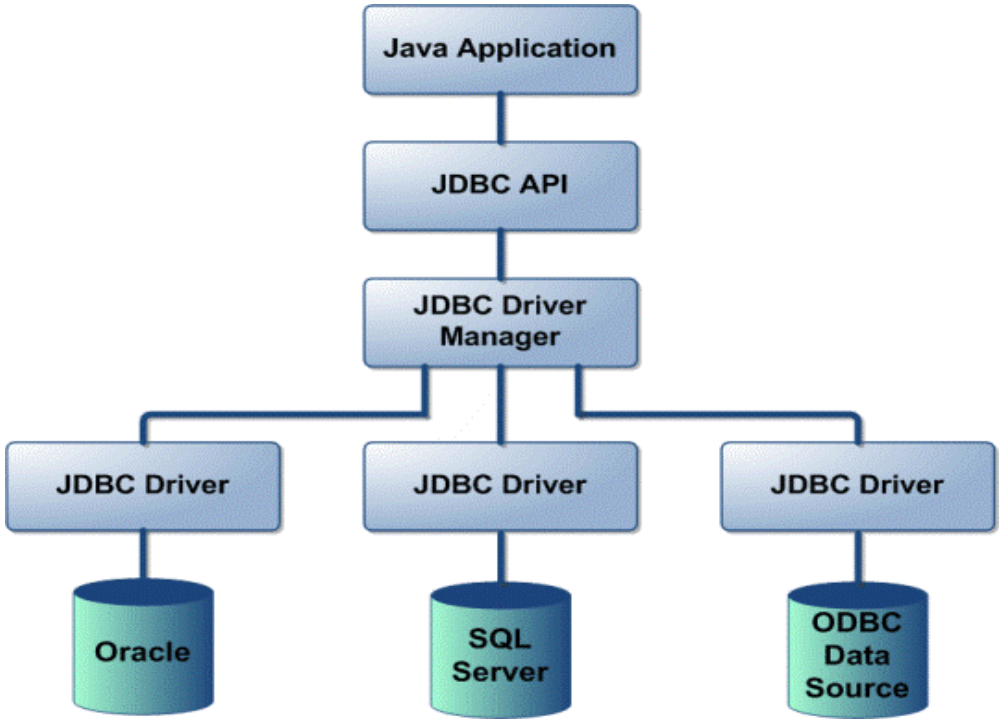
- Introduction to JDBC/JDBC Drivers
- Overview: Six Steps to using JDBC
- Example 1: Setting up Tables via JDBC
- Example 2: Inserting Data via JDBC
- Example 3: Querying Data via JDBC
- Exception Handling Overview
- Advanced Techniques

Introduction to JDBC

- JDBC is a simple API for connecting from Java applications to multiple databases.
- Lets you smoothly translate between the world of the database, and the world of the Java application.
- The idea of a universal database access API is not a new one. For example, Open Database Connectivity (ODBC) was developed to create a single standard for database access in the Windows environment.
- JDBC API aims to be as simple as possible while providing developers with maximum flexibility.

Understanding JDBC Drivers

- To connect to a database, you first need a JDBC Driver.
- JDBC Driver: set of classes that interface with a specific database engine.



JDBC Drivers

- JDBC drivers exist for every major database including: Oracle, SQL Server, Sybase, and MySQL.
- For MySQL, we will be using the open source MySQL Connector/J.
- http://www.mysql.com/downloads/api-jdbc.html.

Installing the MySQL Driver

- To use the MySQL Connector/J Driver, you need to download the complete distribution; and
- **Add the following JAR to your CLASSPATH:**
  - **mysql-connector-java-3.0.11-stable-bin.jar**
- **To use the driver within Tomcat, copy the jar file above to:**
  - **[TOMCAT_HOME]\ROOT\WEB-INF\lib**

Six Steps to Using JDBC

1. Load the JDBC Driver
2. Establish the Database Connection
3. Create a Statement Object
4. Execute a Query
5. Process the Results
6. Close the Connection

1) Loading the JDBC Driver

- To use a JDBC driver, you must load the driver via the `Class.forName()` method.
- In general, the code looks like this:

```
Class.forName("jdbc.DriverXYZ");
```

  - where jbdc.DriverXYZ is the JDBC Driver you want to load.
- If you are using a JDBC-ODBC Driver, your code will look like this:

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
```

Loading the MySQL Driver

- If you are using the MySQL Driver, your code will look like this:

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

- `Class.forName()` will throw a `ClassNotFoundException` if your CLASSPATH is not set up properly.
- Hence, it's a good idea to surround the `forName()` with a try/catch block.

2) Establish the Connection

- Once you have loaded your JDBC driver, the next step is to establish a database connection.
- The following line of code illustrates the basic idea:

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

Creating a Connection URL

- The only difficulty in establishing a connection is specifying the correct URL.
- In general, the URL has the following format: *jdbc:subprotocol:subname*.
  - JDBC indicates that this is a JDBC Connection (no mystery there!)
  - The subprotocol identifies the driver you want to use.
  - The subname identifies the database name/location.

Connection URL: ODBC

- For example, the following code uses a JDBC-ODBC bridge to connect to the local database:
  - `String url = "jdbc:odbc:Fred";`
  - `Connection con = DriverManager.getConnection(url, “username", "password");`

Connection URL: MySQL

- Here's how you might connect to MySQL:
  - `String url = "jdbc:mysql://localhost/webdb";`
  - `Connection con = DriverManager.getConnection(url);`
- In this case, we are using the MySQL JDBC Driver to connect to the *webdb* database, located on the *localhost* machine.
- If this code executes successfully, we will have a Connection object for communicating directly with the database.

3) Create a Statement Object

- The JDBC `Statement` object sends SQL statements to the database.
- Statement objects are created from active `Connection` objects.
- For example:
  - `Statement stmt = con.createStatement();`
- With a Statement object, you can issue SQL calls directly to the database.

4) Execute a Query

- `executeQuery()`
  - Executes the SQL query and returns the data in a table (ResultSet)
  - The resulting table may be empty but never null

```
ResultSet results =
  statement.executeQuery("SELECT a, b FROM table");
```

- `executeUpdate()`
  - Used to execute for INSERT, UPDATE, or DELETE SQL statements
  - The return is the number of rows that were affected in the database
  - Supports Data Definition Language (DDL) statements CREATE TABLE, DROP TABLE and ALTER TABLE

Useful Statement Methods

- getMaxRows/setMaxRows
  - Determines the number of rows a `ResultSet` may contain
  - Unless explicitly set, the number of rows are unlimited (return value of 0)
- getQueryTimeout/setQueryTimeout
  - Specifies the amount of a time a driver will wait for a STATEMENT to complete before throwing a `SQLException`

5) Process the Results

- A `ResultSet` contains the results of the SQL query.
- Useful Methods
  - All methods can throw a `SQLException`
  - close
    - Releases the JDBC and database resources
    - The result set is automatically closed when the associated `Statement` object executes a new query
  - getMetaDataObject
    - Returns a `ResultSetMetaData` object containing information about the columns in the `ResultSet`

ResultSet (Continued)

- Useful Methods
  - next
    - Attempts to move to the next row in the ResultSet
      - If successful true is returned; otherwise, false
      - The first call to next positions the cursor a the first row

ResultSet (Continued)

- Useful Methods
  - findColumn
    - Returns the corresponding integer value corresponding to the specified column name
    - Column numbers in the result set do not necessarily map to the same column numbers in the database
  - get*Xxx*
    - Returns the value from the column specified by column name or column index as an *`Xxx`* Java type
    - Returns 0 or `null`, if the value is a SQL NULL
    - Legal **`getXxx`** types:

```
double      byte        int
Date        String      float
short       long        Time
Object
```

6) Close the Connection

- To close the database connection:
  - stmt.close();
  - connection.close();
- Note: Some application servers, such as BEA WebLogic maintain a pool of database connections.
  - This is much more efficient, as applications do not have the overhead of constantly opening and closing database connections.

Example 1:
Setting Up Tables via JDBC

The Coffee Tables

- To get started, we will first examine JDBC code for creating new tables.
- This java code creates a table for storing coffee data:
- Here's the SQL Statement:

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

The Coffee Table

- You could create this table via MySQL, but you can also create it via JDBC.
- A few things to note about the table:
  - The column named SUP_ID contains an integer value indicating a Supplier ID.
    - Suppliers will be stored in a separate table. In this case, SUP_ID is referred to as a *foreign key*.
  - The column named SALES stores values of SQL type INTEGER and indicates the number of pounds of coffee sold during the current week.
  - The final column, TOTAL, contains a SQL INTEGER which gives the total number of pounds of coffee sold to date.

```
import java.sql.*;

public class CreateCoffees {
    public static void main(String args[]) {
        String url = "jdbc:mysql://localhost/webdb";
        Connection con;
        String createString;
        createString = "create table COFFEES " +
                       "(COF_NAME VARCHAR(32), " +
                       "SUP_ID INTEGER, " +
                       "PRICE FLOAT, " +
                       "SALES INTEGER, " +
                       "TOTAL INTEGER)";
        Statement stmt;
```

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

        try {
            con = DriverManager.getConnection(url);
            stmt = con.createStatement();
            stmt.executeUpdate(createString);
            stmt.close();
            con.close();

        } catch(SQLException ex) {
            System.err.println("SQLException: " + ex.getMessage());
        }
    }
}
```

1 2 3 4 6

Example 2:

Inserting Data via JDBC

```
import java.sql.*;

public class InsertCoffees {

    public static void main(String args[]) throws SQLException {
        System.out.println ("Adding Coffee Data");
        ResultSet rs = null;
        PreparedStatement ps = null;
        String url = "jdbc:mysql://localhost/cerami";
        Connection con;
        Statement stmt;
        try {
            Class.forName("com.mysql.jdbc.Driver");
        } catch(java.lang.ClassNotFoundException e) {
            System.err.print("ClassNotFoundException: ");
            System.err.println(e.getMessage());
        }
```

1

```
try {
            con = DriverManager.getConnection(url);
3           stmt = con.createStatement();
            stmt.executeUpdate ("INSERT INTO COFFEES " +
                  "VALUES('Amaretto', 49, 9.99, 0, 0)");
4           stmt.executeUpdate ("INSERT INTO COFFEES " +
                  "VALUES('Hazelnut', 49, 9.99, 0, 0)");
            stmt.executeUpdate ("INSERT INTO COFFEES " +
                  "VALUES('Amaretto_decaf', 49, 10.99, 0, 0)");
            stmt.executeUpdate ("INSERT INTO COFFEES " +
                  "VALUES('Hazelnut_decaf', 49, 10.99, 0, 0)");
6           stmt.close();
            con.close();
            System.out.println ("Done");
          } catch(SQLException ex) {
            System.err.println("-----SQLException-----");
            System.err.println("SQLState:  " + ex.getSQLState());
            System.err.println("Message:  " + ex.getMessage());
            System.err.println("Vendor:  " + ex.getErrorCode());
          }
      }
}
```

2

Example 3:
Querying Data via JDBC

```
import java.sql.*;

public class SelectCoffees {

    public static void main(String args[]) throws SQLException {
        ResultSet rs = null;
        PreparedStatement ps = null;
        String url = "jdbc:mysql://localhost/cerami";
        Connection con;
        Statement stmt;
        try {
            Class.forName(" com.mysql.jdbc.Driver ");
        } catch(java.lang.ClassNotFoundException e) {
            System.err.print("ClassNotFoundException: ");
            System.err.println(e.getMessage());
        }

        try {
            con = DriverManager.getConnection(url);
            stmt = con.createStatement();
```

1

2

3

```
4   ResultSet uprs = stmt.executeQuery("SELECT * FROM COFFEES");
    System.out.println("Table COFFEES:");
5   while (uprs.next()) {
        String name = uprs.getString("COF_NAME");
        int id = uprs.getInt("SUP_ID");
        float price = uprs.getFloat("PRICE");
        int sales = uprs.getInt("SALES");
        int total = uprs.getInt("TOTAL");
        System.out.print(name + "   " + id + "   " + price);
        System.out.println("   " + sales + "   " + total);
    }
6   uprs.close();
    stmt.close();
    con.close();

  } catch(SQLException ex) {
        System.err.println("-----SQLException-----");
        System.err.println("SQLState:  " + ex.getSQLState());
        System.err.println("Message:  " + ex.getMessage());
        System.err.println("Vendor:  " + ex.getErrorCode());
  }
}
}
```

JDBC Exception Handling

Exception Handling

- SQL Exceptions
  - Nearly every JDBC method can throw a `SQLException` in response to a data access error
  - If more than one error occurs, they are chained together
  - SQL exceptions contain:
    - Description of the error, `getMessage`
    - The SQLState (Open Group SQL specification) identifying the exception, `getSQLState`
    - A vendor-specific integer, error code, `getErrorCode`
    - A chain to the next `SQLException`, `getNextException`

SQL Exception Example

```
try {

      ... // JDBC statement.

} catch (SQLException sqle) {
  while (sqle != null) {
    System.out.println("Message:  " +
   sqle.getMessage());
    System.out.println("SQLState: " +
   sqle.getSQLState());
    System.out.println("Vendor Error: " +
                       sqle.getErrorCode());
    sqle.printStrackTrace(System.out);
    sqle = sqle.getNextException();
  }
}
```

Using Prepared Statements

Using Prepared Statements

- So far we know how to use JDBC **`Statement`** objects for querying/updating tables.
- The **`PreparedStatement`** object provides similar functionality and provides two additional benefits:
  - Faster execution
  - Parameterized SQL Statements

Prepared Statements are Faster

- Unlike a regular **`Statement`** object, a **`PreparedStatement`** object is given a SQL statement when it is created.
- The advantage: the SQL statement will be sent to the database directly, where it will be pre-compiled.
- As a result, **`PreparedStatements`** are generally faster to execute that regular **`Statements`**, especially if you execute the same PreparedStatement multiple times.

Prepared Statements can be Parameterized

- **`PreparedStatements`** are generally more convenient that regular **`Statements`** because they can easily be parameterized.
- For example, you can create a **`PreparedStatement`** SQL template, and then specify parameters for the your SQL query (examples to follow.)

Creating a PreparedStatement Object

- As with **`Statement`** objects, you create a **`PreparedStatement`** object with a **`Connection`** method.
- For example:

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

- In this example, the ? indicates a parameter placeholder which can be set via the JDBC API.

Setting Parameters

- Once you have your **`PreparedStatement`**, you need to supply parameter values for each of the question mark placeholders.
- You do this by calling one of the **`setXXX`** methods defined in the PreparedStatement API.
  - If the value you want to substitute for a question mark is a Java **`int`**, you call the **`setInt()`** method.
  - If the value you want to substitute for a question mark is a Java **`String`**, you call the **`setString()`** method.
  - In general, there is a **`setXXX`** method for each type in the Java programming language.

Setting Parameters: Example

- setXXX arguments:
  - The first argument indicates which question mark placeholder is to be set.
  - The second argument indicates the replacement value.
- For example:
  - updateSales.setInt(1, 75);
  - updateSales.setString(2, "Colombian");

Setting Parameters: Example

- These two code fragments accomplish the same thing:
- Code Fragment 1:

```
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():
```

Executing a Prepared Statement

- To execute a PreparedStatement:
  - executeUpdate()
  - executeQuery()
- Same as a regular Statement, except that no SQL String parameter is specified (because it has already been specified.)

More on Parameters

- Once a parameter has been set with a value, it will retain that value until it is reset to another value or the **`clearParameters()`** method is called.
- You can therefore create one **`PreparedStatement`** and:
  - set two parameters, then execute.
  - change just one parameter, then re-execute.
  - change another parameter, then re-execute, etc.

Changing Parameters

- An example:

```
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")
```

Using a Loop to Set Values

- You can often make coding easier by using a for loop or a while loop to set values for input parameters.
- The next code fragment illustrates the basic idea:
  - One PreparedStatement is created.
  - A for loop runs 5 times. Each time through, the code sets a new value and executes the SQL statement.
  - Updates sales for 5 different coffees.

Return Values for executeUpdate()

- executeQuery() always returns a ResultSet object.
- executeUpdate() returns an int that indicates how many rows of the table were updated.
- For example:

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

- In this case, only 1 row is affected. Hence, executeUpdate() returns 1.
- When the method executeUpdate() is used to execute a table creation/alteration statement, it always return 0.

For Loop Example

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

Using Joins

- Sometimes you need to use two or more tables to get the data you want.
- For example:
  - Proprietor of the Coffee Break wants a list of the coffees he buys from Acme, Inc.
  - This involves data from two tables: COFFEES and SUPPLIERS.
  - To do this, you must perform a SQL Join.
- A join is a database operation that relates two or more tables by means of values that they share in common.
  - In our example, the tables COFFEES and SUPPLIERS both have a column SUP_ID, which can be used to join them.

SUPPLIER Table

- Before going any further, we need to create the SUPPLIERS table and populate it with values.
- The code below create the table:

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

SUPPLIER Data

- The code below inserts data for three suppliers:

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

Verifying the new data

- The following code selects the whole table and lets us see what the table SUPPLIERS looks like:
- ResultSet rs = stmt.executeQuery("select * from SUPPLIERS");
- The result set will look similar to this:

```
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
```

Creating a Join

- Now that we have both tables, we can proceed with the Join.
- The goal is the find coffees that are purchased from a particular supplier.
- Since both tables have a SUP_ID, we can use this ID to perform the Join.
- Since you are using two tables within one SQL statement, you usually indicate each field with a TableName.FieldName. For example: COFFEES.SUP_ID or SUPPLIERS.SUP_ID.

Creating a Join

- Here's the Join:

```
String query = "
SELECT COFFEES.COF_NAME " +
   "FROM COFFEES, SUPPLIERS " +
   "WHERE SUPPLIERS.SUP_NAME LIKE 'Acme, Inc.' " +
   "and SUPPLIERS.SUP_ID = COFFEES.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);
}
```

Join Results

- The code fragment on the last slide will produce the following output:

```
Coffees bought from Acme, Inc.:
        Colombian
        Colombian_Decaf
```

- Full code is available on the next few slides…

```
import java.sql.*;

public class Join {

    public static void main(String args[]) {

        String url = "jdbc:mySubprotocol:myDataSource";
        Connection con;
        String query = "select SUPPLIERS.SUP_NAME, COFFEES.COF_NAME " +
                       "from COFFEES, SUPPLIERS " +
                       "where SUPPLIERS.SUP_NAME like 'Acme, Inc.' and " +
                       "SUPPLIERS.SUP_ID = COFFEES.SUP_ID";
        Statement stmt;

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

1

```
try {
        con = DriverManager.getConnection (url,
                "myLogin", "myPassword");

        stmt = con.createStatement();

        ResultSet rs = stmt.executeQuery(query);
        System.out.println("Supplier, Coffee:");
        while (rs.next()) {
                String supName = rs.getString(1);
                String cofName = rs.getString(2);
                System.out.println("    " + supName + ", " + cofName);
        }

        stmt.close();
        con.close();

        } catch(SQLException ex) {
                System.err.print("SQLException: ");
                System.err.println(ex.getMessage());
        }
    }
}
```

2

3

4

5

6

Using Database Transactions

Using Transactions

- There are times when you do not want one statement to take effect unless another one also succeeds.
- For example:
  1. Take $400 out of your Checking Account.
  2. Take this $400 and transfer to your Savings Account.
- If the first statement succeeds, but the second one fails, you are out $400!
- To do with this possibility, most database support many levels of transactions.

Using Transactions

- A transaction is a set of one or more statements that are executed together as a unit.
- Hence, either all of the statements are executed, or none of the statements are executed.

Disabling Auto-Commit Mode

- 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.
- The way to allow two or more statements to be grouped into a transaction is to disable auto-commit mode.
- This is demonstrated in the following line of code, where con is an active connection:

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

Committing a Transaction

- Once auto-commit mode is disabled, no SQL statements will be committed until you call the **`commit()`** method explicitly.
- All statements executed after the previous call to the method commit will be included in the current transaction and will be committed together as a unit.
- The code on the next slide illustrates the basic idea.

Transaction Action

```
con.setAutoCommit(false);
PreparedStatement updateSales = con.prepareStatement(
    "UPDATE COFFEES SET SALES = ? WHERE COF_NAME LIKE ?");
updateSales.setInt(1, 50);
updateSales.setString(2, "Colombian");
updateSales.executeUpdate();
PreparedStatement updateTotal = con.prepareStatement(
    "UPDATE COFFEES SET TOTAL = TOTAL + ? WHERE COF_NAME
     LIKE ?");
updateTotal.setInt(1, 50);
updateTotal.setString(2, "Colombian");
updateTotal.executeUpdate();
con.commit();
con.setAutoCommit(true);
```

Rolling Back

- To cancel a transaction, call the **`rollback()`** method.
- This aborts the transaction and restores values to what they were before the attempted update.
- If you are executing multiple statements within a transaction, and one of these statements generates a **`SQLException`**, you should call the **`rollback()`** method to abort the transaction and start over again.
- Complete example is on the next few slides.

```
import java.sql.*;

public class TransactionPairs {

    public static void main(String args[]) {

        String url = "jdbc:mySubprotocol:myDataSource";
        Connection con = null;
        Statement stmt;
        PreparedStatement updateSales;
        PreparedStatement updateTotal;
        String updateString = "update COFFEES " +
                                    "set SALES = ? where COF_NAME = ?";

        String updateStatement = "update COFFEES " +
                    "set TOTAL = TOTAL + ? where COF_NAME = ?";
        String query = "select COF_NAME, SALES, TOTAL from COFFEES";
```

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

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"};
      int len = coffees.length;
```

1

2

3

```
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();
}

con.setAutoCommit(true);

updateSales.close();
updateTotal.close();
```

4

6

```
            } 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());
                    }
                }
            }
        }
    }
```

Additional Topics

- If you are curious to learn more about JDBC, check out the 2nd Part of the Sun JDBC Tutorial:
  - http://java.sun.com/docs/books/tutorial/jdbc/jdbc2dot0/index.html
  - Covers such topics as: Cursors, Connection Pools, etc.

Summary

- The JDBC Driver connections a Java application to a specific database.
- Six Steps to Using JDBC:
  1. Load the Driver
  2. Establish the Database Connection
  3. Create a Statement Object
  4. Execute the Query
  5. Process the Result Set
  6. Close the Connection
- Make sure to wrap your JDBC calls within try/catch blocks.

Summary

- PreparedStatements are just like Statements, only better!
  - Faster
  - Easier to use because of all the setXXX() methods.
- Database Joins are used to connect two or more tables together.
- Transactions are used to group two or more database calls together:
  - commit(): Commits all the statements as one unit.
  - rollback(): Aborts the transaction, and restores the database back to its original condition.

Thanks for your attention