JAVA JDBC

Introduction to Databases

Month	Gas	EatingOut	Utilities	Phone
January	$200.25	$109.87	$97.00	$45.08
February	$225.34	$121.08	$97.00	$23.36
March	$254.78	$130.45	$97.00	$56.09

- Assuming you drove the same number of miles per month, gas is getting pricey - maybe it is time to get a Prius.
- You are eating out more month to month (or the price of eating out is going up) - maybe it's time to start doing some meal planning.
- And maybe you need to be a little less social - that phone bill is high.

Talking to a Database

There are three important concepts when working with a database:

- Creating a connection to the database
- Creating a statement to execute in the database
- Getting back a set of data that represents the results

SQL Queries

- **INSERT** Add a row to the table Expenses, and set each of the columns in the table to the values expressed in the parentheses.
- **SELECT with WHERE** this SQL statement returns a single row identified by the primary key–the Month column. Think of this statement as a refinement to Read–more like a Find or Find by primary key.

SQL Queries

- **SELECT** When the SELECT clause does not have a WHERE clause, we are asking the database to return every row. Further, because we are using an asterisk (*) following the SELECT, we are asking for every column. Basically, it is a dump of the data. Think of this statement as a Read All.

SQL Queries

- **UPDATE** Change the data in the Phone and EatingOut cells to the new data provided for February.
- **DELETE** Remove a row altogether from the database where the Month is April.

Example SQL CRUD Commands

"CRUD"	SQL Command	Example SQL Query	Expressed in English
Create	INSERT	`INSERT INTO Expenses VALUES ('April', 231.21, 29.87, 97.00, 45.08)`	Add a new row (April) to expenses with the following values....
Read (or Find)	SELECT	`SELECT * FROM Expenses WHERE Month="February"`	Get me all of the columns in the Expenses table for February.
Read All	SELECT	`SELECT * FROM Expenses`	Get me all of the columns in the Expenses table.
Update	UPDATE	`UPDATE Expenses SET Phone=32.36, EatingOut=111.08 WHERE Month='February'`	Change my phone expense and EatingOut expense for February to....
Delete	DELETE	`DELETE FROM Expenses WHERE Month='April'`	Remove the row of expenses for April.

Core Interfaces of the JDBC API

- Fully implement the interfaces: java.sql.Driver, java.sql.DatabaseMetaData, java.sql.ResultSetMetaData.
- Implement the java.sql.Connection interface. (Note that some methods are optional depending upon the SQL version the database.)
- Implement the java.sql.Statement, java.sql.PreparedStatement.
- Implement the java.sql.CallableStatement interfaces if the database supports stored procedures.
- Implement the java.sql.ResultSet interface.

Connect to a DB using DriverManager

- Not all of the types defined in the JDBC API are interfaces. One important class for JDBC is the java.sql.DriverManager class. This concrete class is used to interact with a JDBC driver and return instances of Connection objects to you. Conceptually, the way this works is by using a design pattern called Factory.

How JDBC Drivers Register with the DriverManager

First, one or more JDBC drivers, in a JAR or ZIP file, are included in the classpath of your application.

The DriverManager class uses a service provider mechanism to search the classpath for any JAR or ZIP files that contain a file named java.sql.Driver in the META-INF/services folder of the driver jar or zip.

How JDBC Drivers Register with the DriverManager

The DriverManager will then attempt to load the class it found in the java.sql.Driver file using the class loader:

```
Class.forName("org.apache.derby.jdbc.ClientDriver");
```

How JDBC Drivers Register with the DriverManager

```
try {
        Class.forName("connect.microsoft.MicrosoftDriver");
        Class.forName("oracle.jdbc.driver.OracleDriver");
        Class.forName("com.sybase.jdbc.SybDriver");
        Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
} catch(ClassNotFoundException cnfe) {
        System.err.println("Error loading driver: " + cnfe);
}
```

How JDBC Drivers Register with the DriverManager

When the driver class is loaded, its static initialization block is executed. Per the JDBC specification, one of the first activities of a driver instance is to "self-register,, with the DriverManager class by invoking a static method on DriverManager.

```
public class ClientDriver implements java.sql.Driver{
  static {
    ClientDriver driver = new ClientDriver();
    DriverManager.registerDriver(driver);
 }
//...
}
```

How JDBC Drivers Register with the DriverManager

Now, when your application invokes the DriverManager.getConnection() method and passes a JDBC URL, username, and password to the method, the DriverManager simply invokes the connect() method on the registered Driver.

If the connection was successful, the method returns a Connection object instance to DriverManager, which, in turn, passes that back to you.

Start your application:

```
java-classpath ... MyDBApp
```

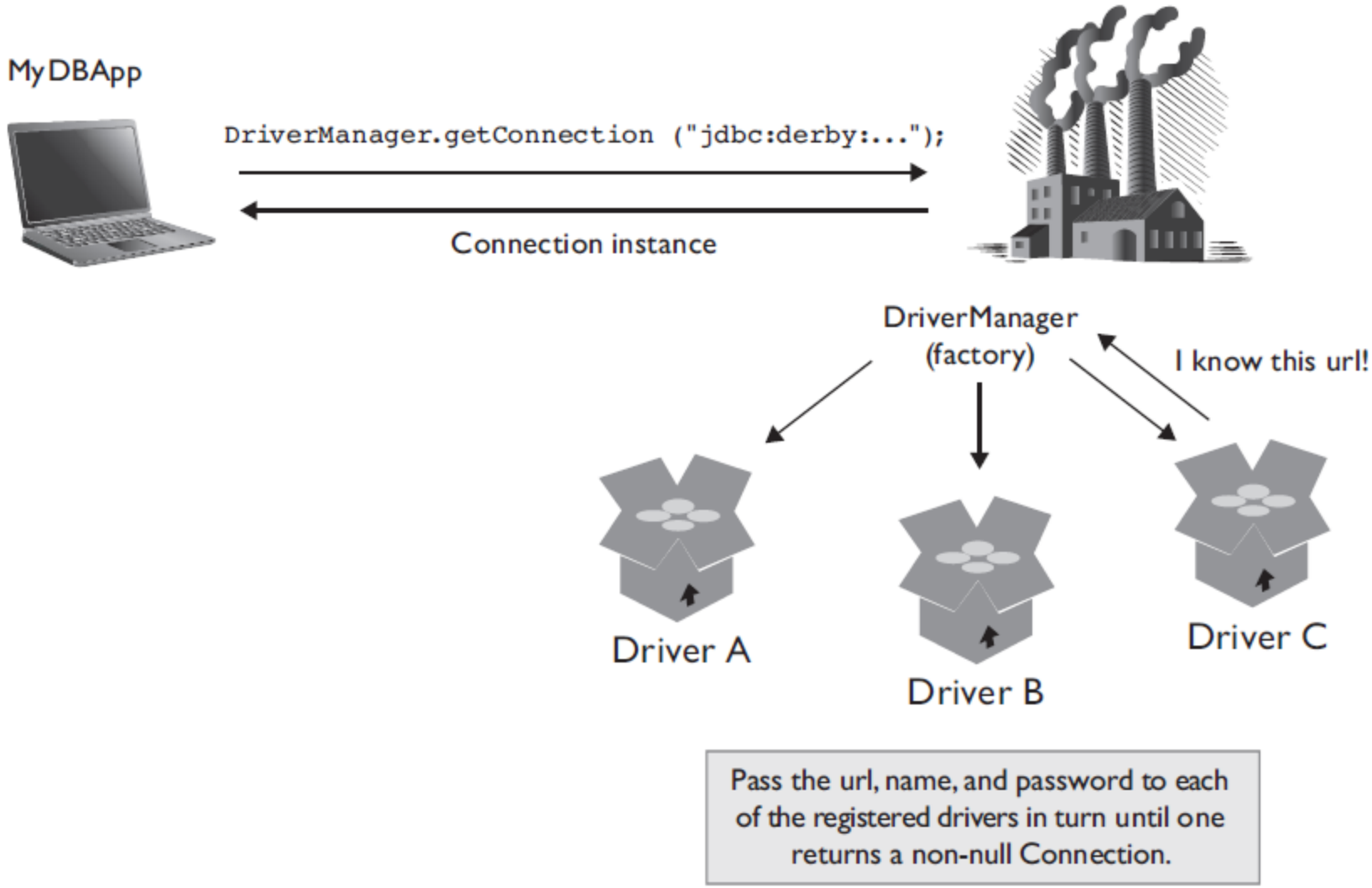
DriverManager
(factory)

Classload the class defined in the
`META-INF/services/java.sql.Driver` file.

`DriverManager.registerDriver(this);`

A JDBC driver
(jar file)

Repeat this process for every jar file in the classpath that has a `java.sql.Driver` file.


MyDBApp
DriverManager.getConnection ("jdbc:derby:...");
Connection instance
DriverManager
(factory)
I know this url!
Driver A
Driver B
Driver C
Pass the url, name, and password to each of the registered drivers in turn until one returns a non-null Connection.

To summarize:

- The JVM loads the DriverManager class, a concrete class in the JDBC API.
- The DriverManager class loads any instances of classes it finds in the META-INF/services/java.sql.Driver file of JAR/ZIP files on the classpath.
- Driver classes call DriverManager.register(this) to self-register with the DriverManager.

To summarize:

- When the DriverManager.getConnection(String url) method is invoked, DriverManager invokes the connect() method of each of these registered Driver instances with the URL string.
- The first Driver that successfully creates a connection with the URL returns an instance of a Connection object to the DriverManager.getConnection method invocation.

Submit Queries and Read Results from the Database

Get connection We are creating a Connection object instance using the information we need to access Bob's Books Database (stored on a Java DB Relational database, BookSellerDB, and accessed via the credentials).

Create statement We are using the Connection to create a Statement object. The Statement object handles passing Strings to the database as queries for the database to execute.

Execute query We are executing the query string on the database and returning a ResultSet object.

Process results We are iterating through the result set rows - each call to next() moves us to the next row of results.

Statements

Method (Each Throws SQLException)	Description
`ResultSet executeQuery(String sql)`	Execute a SQL query and return a `ResultSet` object, i.e., SELECT commands.
`int executeUpdate(String sql)`	Execute a SQL query that will only modify a number of rows, i.e., INSERT, DELETE, or UPDATE commands.
`boolean execute(String sql)`	Execute a SQL query that may return a result set OR modify a number of rows (or do neither). The method will return true if there is a result set, or false if there may be a row count of affected rows.
`ResultSet getResultSet()`	If the return value from the `execute()` method was true, you can use this method to retrieve the result set from the query.
`int getUpdateCount()`	If the return value from the `execute()` method was false, you can use this method to get the number of rows affected by the SQL command.

public boolean execute(String sql)

```
ResultSet rs;
int numRows;
boolean status = stmt.execute("");  // True if there is a ResultSet
if (status) { // True
   rs = stmt.getResultSet(); // Get the ResultSet
   // Process the result set...
} else { // False
   numRows = stmt.getUpdateCount(); // Get the update count
   if (numRows == -1) { // If -1, there are no results
      out.println("No results");
   } else { // else, print the number of
      // rows affected
      out.println(numRows + " rows affected.");
   }
}
```

SQL Injection

```
String s = System.console().readLine(
                    "Enter your e-mail address: ");
ResultSet rs = stmt.executeQuery(
      "SELECT * FROM Customer WHERE EMail='" + s + "'");
```

SELECT * FROM Customer WHERE Email='tom@trouble.com' OR 'x'='x'

ResultSet

The **ResultSet** object represents the results of the query – all of the data in each row on a per-column basis. Again, as a reminder, how data in a **ResultSet** are stored is entirely up to the JDBC driver vendor.

Moving Forward in a ResultSet

```
String query = "SELECT First_Name, Last_Name,
EMail FROM Customer WHERE Last_Name LIKE 'C%'";
```

```
ResultSet rs=
stmt.executeQuery(query);
```

cursor

ResultSet

```
rs.next()=true;
```

```
rs.next()=true;
```

```
rs.next()=true;
```

```
rs.next()=false;
```

Bob	Collins	bob.collins@yahoo.com
Rebecca	Cabeca	rebecca.cabeca@gmail.com
Anthony	Clark	anthony.clark@gmail.com

Reading Data from a ResultSet

```
while (rs.next()){
    System.out.print(rs.getInt("CustomerID") + " ");
    System.out.print(rs.getString("FirstName") + " ");
    System.out.print(rs.getString("LastName") + " ");
    System.out.print(rs.getString("EMail") + " ");
    System.out.println(rs.getString("Phone"));
}
```

Reading Data from a ResultSet

```
while (rs.next()){
    System.out.print(rs.getInt(1)+ " ");
    System.out.print(rs.getString(2) + " ");
    System.out.print(rs.getString(3) + " ");
    System.out.print(rs.getString(4) + " ");
    System.out.println(rs.getString(5));
}
```

Reading Data from a ResultSet

Column indexes start with 1. It is important to keep in mind that when you are accessing columns using integer index values, the column indexes always start with 1, not 0 as in traditional arrays. If you attempt to access a column with an index of less than 1 or greater than the number of columns returned, a SQLException will be thrown.

SQL Types and JDBC Types

SQL Type	Java Type	ResultSet get methods
BOOLEAN	`boolean`	`getBoolean(String columnName)` `getBoolean(int columnIndex)`
INTEGER	`int`	`getInt(String columnName)` `getInt(int columnIndex)`
DOUBLE, FLOAT	`double`	`getDouble(String columnName)` `getDouble(int columnIndex)`
REAL	`float`	`getFloat(String columnName)` `getFloat(int columnIndex)`
BIGINT	`long`	`getLong(String columnName)` `getLong(int columnIndex)`
CHAR, VARCHAR, LONGVARCHAR	`String`	`getString(String columnName)` `getString(int columnIndex)`
DATE	`java.sql.Date`	`getDate(String columnName)` `getDate(int columnIndex)`
TIME	`java.sql.Time`	`getTime(String columnName)` `getTime(int columnIndex)`
TIMESTAMP	`java.sql.Timestamp`	`getTimestamp(String columnName)` `getTimestamp(int columnIndex)`
Any of the above	`java.lang.Object`	`getObject(String columnName)` `getObject(int columnIndex)`

Getting Information about a ResultSet

TYPE_FORWARD_ONLY The default value for a ResultSet – the cursor moves forward only through a set of results.

TYPE_SCROLL_INSENSITIVE A cursor position can be moved in the result forward or backward, or positioned to a particular cursor location. Any changes made to the underlying data – the database itself - are not reflected in the result set. In other words, the result set does not have to „keep state" with the database. This type is generally supported by databases.

Getting Information about a ResultSet

TYPE_SCROLL_SENSITIVE A cursor can be changed in the results forward or backward, or positioned to a particular cursor location. Any changes made to the underlying data are reflected in the open result set. As you can imagine, this is difficult to implement, and is therefore not implemented in a database or JDBC driver very often.

ResultSet Cursor Positioning Methods

Method	Effect on the Cursor and Return Value
`boolean next()`	Moves the cursor to the next row in the `ResultSet`. Returns `false` if the cursor is positioned beyond the last row.
`boolean previous()`	Moves the cursor backward one row. Returns `false` if the cursor is positioned before the first row.
`boolean absolute(int row)`	Moves the cursor to an absolute position in the `ResultSet`. Rows are numbered from 1. Moving to row 0 moves the cursor to before the first row. Moving to negative row numbers starts from the last row and works backward. Returns `false` if the cursor is positioned beyond the last row or before the first row.
`boolean relative(int row)`	Moves the cursor to a position relative to the current position. Invoking `relative(1)` moves forward one row; invoking `relative(-1)` moves backward one row. Returns `false` if the cursor is positioned beyond the last row or before the first row.
`boolean first()`	Moves the cursor to the first row in the `ResultSet`. Returns `false` if there are no rows in the `ResultSet` (empty result set).

ResultSet Cursor Positioning Methods

Method	Effect on the Cursor and Return Value
`boolean last()`	Moves the cursor to the last row in the `ResultSet`. Returns `false` if there are no rows in the `ResultSet` (empty result set).
`void beforeFirst()`	Moves the cursor to before the first row in the `ResultSet`.
`void afterLast()`	Moves the cursor to after the last row in the `ResultSet`.

Absolute cursor positioning

```
String query = "SELECT * FROM Author";
```

	cursor	ResultSet		
	1	1000	Rick	Riordan
rs.absolute(2);	2	1001	Nancy	Farmer
	3	1002	Ally	Condie
	4	1003	Cressida	Cowell
	5	1004	Lauren	St. John
	6	1005	Eoin	Colfer
	7	1006	Esther	Freisner
	8	1007	Chris	D'lacey
rs.absolute(9);	9	1008	Christopher	Paolini
	10	1009	Kathryn	Lasky
rs.absolute(-1);	11	1010	Nancy	Star

Properly Closing SQL Resources

Method Call	Has the Following Action(s)
`Connection.close()`	Releases the connection to the database. Closes any `Statement` created from this `Connection`.
`Statement.close()`	Releases this `Statement` resource. Closes any open `ResultSet` associated with this `Statement`.
`ResultSet.close()`	Releases this `ResultSet` resource. Note that any `ResultSetMetaData` objects created from the `ResultSet` are still accessible.
`Statement.executeXXXX()`	Any `ResultSet` associated with a previous `Statement` execution is automatically closed.

Prepared Statements

- If you are going to execute similar SQL statements multiple times, using **"prepared" statements** can be more efficient than executing a raw query each time.
- **The idea is to create a parameterized statement in a standard form that is sent to the database for compilation before actually being used.**

Prepared Statements

- You use a question mark to indicate the places where a value will be substituted into the statement.
- Each time you use the prepared statement, you replace some of the marked parameters, using a se*tXxx* call corresponding to the entry you want to set (using 1-based indexing) and the type of the parameter (e.g., setInt, setString, and so forth).
- You then use executeQuery (if you want a ResultSet back) or execute/executeUpdate as with normal statements.

Prepared Statement

```
String pQuery =
        "SELECT UnitPrice from Book WHERE Title LIKE ?";
PreparedStatement pstmt = conn.prepareStatement(pQuery);
pstmt.setString(1, "%Heroes%"); // Substitute this String
                                // for the parameter
ResultSet rs = pstmt.executeQuery();
```

Prepared Statement

- The performance advantages of prepared statements can vary , depending on how well the server supports precompiled queries and how efficiently the driver handles raw queries **(up to 50%)**.

executeQuery

executeQuery returns a ResultSet object containing the results of the query sent to the DBMS (database management system), **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 equals number of updated rows
```

Transactions

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

Transactions

```
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);
```

Rollbacking a Transaction

- Calling the method **rollback** aborts a transaction and returns any values that were modified to their previous values.
- If you are trying to execute one or more statements in a transaction and get an **SQLException** , you should call the method rollback to abort the transaction and start the transaction all over again. That is the only way to be sure of what has been committed and what has not been committed. Catching an SQLException tells you that something is wrong, but it does not tell you what was or was not committed. Since you cannot count on the fact that nothing was committed, calling the method rollback is the only way to be sure.

Rollbacking a Transaction

```
Connection conn = DriverManager.getConnection(url, login, pass);
conn.setAutoCommit(false); // Start a transaction
Statement stmt = conn.createStatement();
int result1, result2, result3;
try {
    result1 = stmt.executeUpdate("INSERT INTO Author
            VALUES(1031, 'Rachel', 'McGinn')");
    result2 = stmt.executeUpdate("INSERT INTO Book
            VALUES('0554466789', 'My American Dolls',
                   '2012-08-31','Paperback', 7.95)");
    result3 = stmt.executeUpdate("INSERT INTO
            Books_by_Author VALUES(1031,'0554466789')");
conn.commit(); //commit the entire transaction
} catch (SQLException ex) {
    conn.rollback();
}
```

Stored Procedures

Stored procedures are supported by most DBMSs, but there is a fair amount of variation in their syntax and capabilities.

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

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.

Stored Procedures

JDBC allows you to call a database stored procedure from an application written in the Java programming language. The first step is to create a CallableStatement object. Than you should call proper execute method (depending on what is procedure created: SELECT, UPDATE and so on).

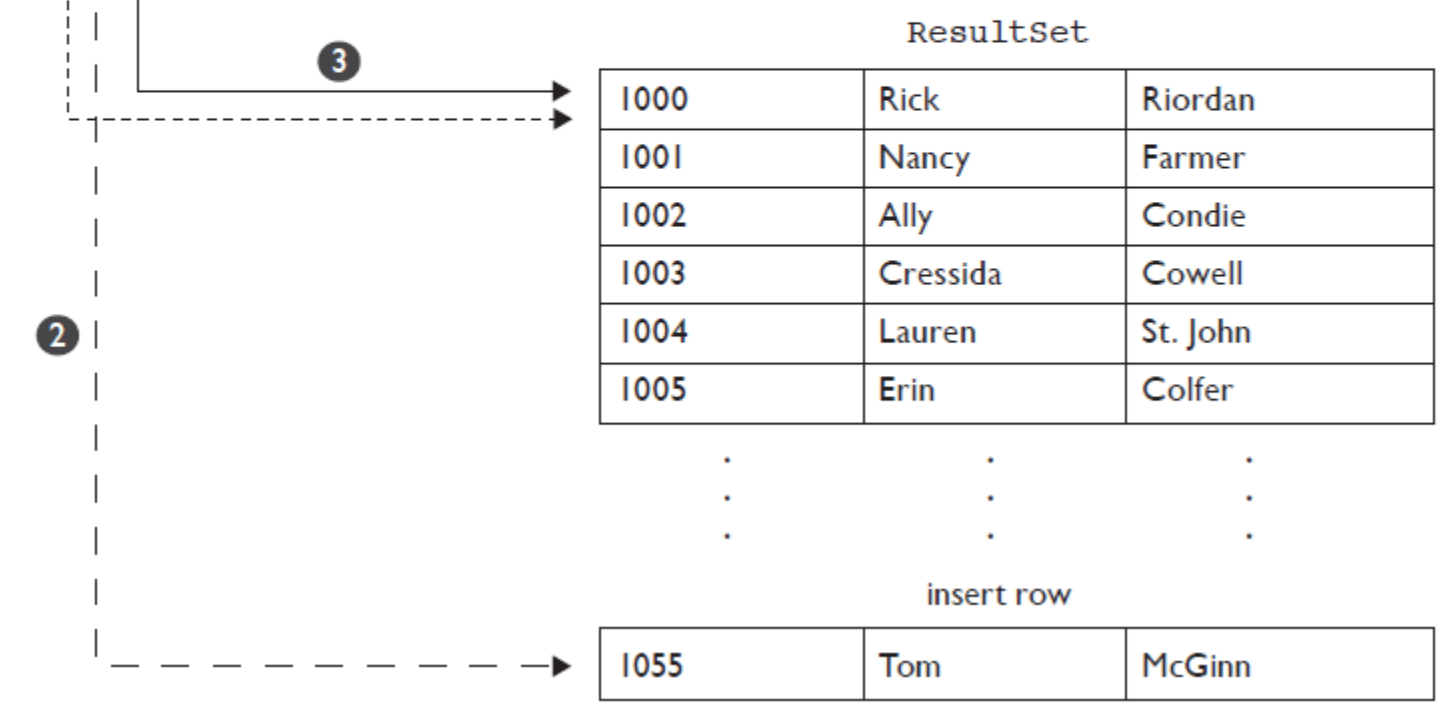
```
CallableStatement cs =
      con.prepareCall("{call SHOW_SUPPLIERS}");
ResultSet rs = cs.executeQuery();
// or executeUpdate() or execute()
```

SQLException

```
} catch(SQLException ex) {
    System.out.println("\n--- SQLException caught ---\n");
    while (ex != null) {
        System.out.println("Message:" + ex.getMessage());
        // a string that describes the error
        System.out.println("SQLState: " + ex.getSQLState ());
        // a string identifying the error according to the X/Open
        // SQLState conventions
        System.out.println("ErrorCode: " + ex.getErrorCode ());
        // a number that is the driver vendor's error code number
        ex = ex.getNextException(); // there can be more than 1
        System.out.println(" ");
    }
}
```

The ResultSet insert row

```
String query = "SELECT AuthorID, FirstName, LastName FROM Author";
ResultSet rs = stmt.executeQuery(query);
rs.next();
rs.moveToInsertRow();
rs.updateInt("AuthorID", 1055);
rs.updateString("FirstName", "Tom");
rs.updateString("LastName", "McGinn");
rsinsertRow();
rs.moveToCurrentRow();
```



Given:

```
String url = "jdbc:mysql://SolDBServer/soldb";
String user = "sysEntry";
String pwd = "fo0B3@r";
// INSERT CODE HERE
Connection conn = DriverManager.getConnection(url, user, pwd);
```

Assuming "`org.gjt.mm.mysql.Driver`" is a legitimate class, which line, when inserted at `// INSERT CODE HERE`, will correctly load this JDBC 3.0 driver?

A `DriverManager.registerDriver("org.gjt.mm.mysql.Driver");`

B. `Class.forName("org.gjt.mm.mysql.Driver");`

C. `DatabaseMetaData.loadDriver("org.gjt.mm.mysql.Driver");`

D. `Driver.connect("org.gjt.mm.mysql.Driver");`

E. `DriverManager.getDriver("org.gjt.mm.mysql.Driver");`

Given:

```
try {
  Statement stmt = conn.createStatement();
  String query =
         "SELECT * FROM Author WHERE LastName LIKE 'Rand%'";
  ResultSet rs = stmt.executeQuery(query);  // Line X
  if (rs == null) {                         // Line Y
    System.out.println("No results");
  } else {
    System.out.println(rs.getString("FirstName"));
  }
} catch (SQLException se) {
  System.out.println("SQLException");
}
```

Assuming a `Connection` object has already been created (conn) and that the query produces a valid result, what is the result?

A. Compiler error at line X

B. Compiler error at line Y

C. No result

D. The first name from the first row that matches 'Rand%'

E. `SQLException`

F. A runtime exception

Given the SQL query:

```
String query = "UPDATE Customer SET EMail='John.Smith@comcast.net'
                WHERE CustomerID = 5000";
```

Assuming this is a valid SQL query and there is a valid `Connection` object (conn), which will compile correctly and execute this query?

A.
```
Statement stmt = conn.createStatement();
stmt.executeQuery(query);
```

B.
```
Statement stmt = conn.createStatement(query);
stmt.executeUpdate();
```

C.
```
Statement stmt = conn.createStatement();
stmt.setQuery(query);
stmt.execute();
```

D.
```
Statement stmt = conn.createStatement();
stmt.execute(query);
```

E.
```
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeUpdate(query);
```

Given:

```
try {
  ResultSet rs = null;
  try (Statement stmt = conn.createStatement()) {  // line X
    String query = "SELECT * from Customer";
    rs = stmt.executeQuery(query);                 // line Y
  } catch (SQLException se) {
    System.out.println("Illegal query");
  }
  while (rs.next()) {
    // print customer names
  }
} catch (SQLException se) {
  System.out.println("SQLException");
}
```

And assuming a valid `Connection` object (conn) and that the query will return results, what is the result?

A. The customer names will be printed out

B. Compiler error at line X

C. Illegal query

D. Compiler error at line Y

E. `SQLException`

F. Runtime exception

Given this code fragment:

```
Statement stmt = conn.createStatement();
ResultSet rs;
String query = "<QUERY HERE>";
stmt.execute(query);
if ((rs = stmt.getResultSet()) != null) {
  System.out.println("Results");
}
if (stmt.getUpdateCount() > -1) {
  System.out.println("Update");
}
```

Which query statements entered into `<QUERY HERE>` produce the output that follows the query string (in the following answer), assuming each query is valid? (Choose all that apply.)

A. `"SELECT * FROM Customer"`
   `Results`

B. `"INSERT INTO Book VALUES ('1023456789', 'One Night in Paris', '1984-10-20', 'Hardcover', 13.95)"`
   `Update`

C. `"UPDATE Customer SET Phone = '555-234-1021' WHERE CustomerID = 101"`
   `Update`

D. `"SELECT Author.LastName FROM Author"`
   `Results`

E. `"DELETE FROM Book WHERE ISBN = '1023456789'"`
   `Update`

Given:

```
String q = "UPDATE Customer SET Last_name=? WHERE Customer_id=?";
try {
  PreparedStatement pstmt = conn.prepareStatement(q);
  pstmt.setString(0, "Smith");                    // Line X
  pstmt.setString(1, "5001");                     // Line Y
  int result = pstmt.executeUpdate();
  if (result != 1) System.out.println ("Error - update failed");
} catch (SQLException se) {
  System.out.println("Exception");
}
```

Assuming the table name and column names are valid, what is the result?

A. The last name of the customer with id 5001is set to `"Smith"`

B. Error – update failed

C. Exception

D. Compilation fails