* Overview of JDBC:

JDBC is an API (Application Programming Interface) that allows programmers to develop database-driven applications in the Java programming language. Using JDBC, you can write code that connects to a database to query and update data using SQL.

JDBC is a part of core Java (Java SE). Its API (classes, interfaces,…) is contained in two Java packages `java.sql` (main) and `javax.sql` (extension). The current version of JDBC is 4.3, which is packed with Java SE 9.

JDBC is one of the most commonly used APIs in the Java library. You can use JDBC to develop Java applications that talk with almost any database, with an appropriate JDBC driver is provided.

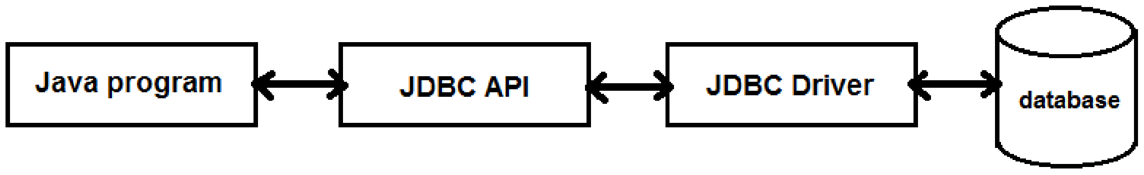
Next, let's understand some key terms in JDBC.

* JDBC Driver:

JDBC driver is a kind of software that translates JDBC calls into native database calls. For example, a JDBC driver for MySQL translates queries in Java to a protocol which MySQL database server understands.

A JDBC driver acts as a bridge between a Java program and the database, allowing programmers to write database-independent code using the API provided by JDBC. That means you write Java code only once, and your program can work with any database that provides the appropriate JDBC driver. Then if you want to change to another database, just switch the JDBC driver and your Java code remains unchanged.

The following picture illustrates the concept of JDBC driver:



From this picture, you can see that JDBC API provides a standard interface for Java programs to talk with any database. In turn, JDBC talks with a specific JDBC driver provided by the database vendor or third-parties. Whether the database is MySQL, Oracle or Microsoft SQL Server, your Java code is the same, only the JDBC driver needs to change according to the actual database.

* JDBC Driver Types:

The JDBC specification classifies JDBC drivers into 4 types, but I think you don't need to know all of them. Because most of the time, you use only one type - the driver type written in pure Java, translating JDBC requests directly to database-specific protocol. It is the most popular and commonly used JDBC driver type.

* JDBC Driver JAR Files:

A JDBC driver software is supplied as a JAR file by its database vendor or third parties. For example, MySQL provides its own JDBC driver called MySQL Connector/J - its JAR file name is `mysql-connector-java-VERSION.jar` (VERSION is a specific version number).

That means you have to find and download a JDBC driver JAR file for your database. Almost common databases such as MySQL, Oracle, Microsoft SQL Server, PostgreSQL… provide their own JDBC driver JAR files.

The JDBC driver JAR file must be present in the classpath, for the program to run. That means you have to specify the JAR file when running the program from command line like this (on Windows):

```
java -classpath .;driverJARFile MainClass
```

Note that the period symbol . denotes the current directory, and the semicolon ; is the path separator. On Linux, the path separator is colon:

```
java -classpath .:driverJARFile MainClass
```

For example, the following command runs a program with a MySQL Connector/J driver JAR file:

```
java -classpath .:mysql-connector-java-8.0.jar StudentProgram
```

Here, the JAR file `mysql-connector-java-8.0.jar` is placed in the same directory with the main class `StudentProgram.class`.

You don't need to specify the JDBC driver JAR file when compiling Java code. And when distributing your Java application, you need to pack the JDBC driver JAR file along with the application's JAR file

Transactions with JDBC

1. Why Transactions?

In database systems, transactions are designed to preserve data integrity by grouping multiple statements to be executed as a single unit. In a transaction, either all of the statements are executed, or none of the statements is executed. If any statement failed to execute, the whole transaction is aborted and the database is rolled back to the previous state. This assures the data is kept consistence in the events of network problems, software errors, etc.

Let’s see an example.

Imagine in a sales application, the manager saves a new order and also updates total sales to date in the current month. The order details and the total sales should be updated at the same time, otherwise the data will be inconsistence. Here, the application should group the save order details statement and update total sales statement in a transaction. Both these statements must be executed. If either one statement failed to execute, all changes are discarded.

Transactions also provide protection against conflicts that might arise when multiple users access the same data at the same time, by using locking mechanisms to block access by others to the data that is being accessed by the current transaction.

2. Typical Transactions Handling Workflow in JDBC

Now, let’s see how to use transactions with JDBC. Given `connection` is an active database connection, the following code skeleton illustrates a typical workflow of handling a database transaction with JDBC in Java:

```
try {
      // begin the transaction:
      connection.setAutoCommit(false);
      // execute statement #1
      // execute statement #2
      // execute statement #3
      // ...
      // commit the transaction
      connection.commit();
} catch (SQLException ex) {
      // abort the transaction
      connection.rollback();
} finally {
      // close statements
      connection.setAutoCommit(true);
}
```

Let’s explore this workflow in details.

Disabling Auto Commit mode

By default, a new connection is in auto-commit mode. This means each SQL statement is treated as a transaction and is automatically committed right after it is executed. So we have to disable the auto commit mode to enable two or more statements to be grouped into a transaction:

```
connection.setAutoCommit(false);
```

Committing the transaction

After the auto commit mode is disabled, all subsequent SQL statements are included in the current transaction, and they are committed as a single unit until we call the method `commit()`:

```
connection.commit();
```

So a transaction begins right after the auto commit is disabled and ends right after the connection is committed. Remember to execute SQL statements between these calls to ensure they are in the same transaction.

Rolling back the transaction

If any statement failed to execute, a `SQLException` is thrown, and in the catch block, we invoke the method `rollback()` to abort the transaction:

```
connection.rollback();
```

Any changes made by the successful statements are discarded and the database is rolled back to the previous state before the transaction.

Enabling Auto Commit mode

Finally, we enable the auto commit mode to get the connection back to the default state:

```
connection.setAutoCommit(true);
```

In the default state (auto commit is enabled), each SQL is treated as a transaction and we don’t need to call the `commit()` method manually.

3. A JDBC Transaction Example

Now, let’s look at a real code example. Suppose that we are working on a database called **sales** with the following tables:

When a new order is saved (a new row inserted into the table **orders**), the monthly sales also must be updated (a corresponding row gets updated in the table **monthly_sales**). So these two statements (save order and update sales) should be grouped into a transaction.

The following method shows how to execute these two statements in a transaction with JDBC:

```
public void saveOrder(int productId, Date orderDate, float amount, int
reportMonth) {
      PreparedStatement orderStatement = null;
      PreparedStatement saleStatement = null;

      try {
            conn.setAutoCommit(false);
            String sqlSaveOrder = "insert into orders (product_id, order_date,
amount)";
            sqlSaveOrder += " values (?, ?, ?)";
            String sqlUpdateTotal = "update monthly_sales set total_amount =
total_amount + ?";
            sqlUpdateTotal += " where product_id = ? and report_month = ?";

            orderStatement = conn.prepareStatement(sqlSaveOrder);
            saleStatement = conn.prepareStatement(sqlUpdateTotal);

            orderStatement.setInt(1, productId);
            orderStatement.setDate(2, orderDate);
            orderStatement.setFloat(3, amount);
            saleStatement.setFloat(1, amount);
            saleStatement.setInt(2, productId);
            saleStatement.setInt(3, reportMonth);

            orderStatement.executeUpdate();
            saleStatement.executeUpdate();

            conn.commit();
      } catch (SQLException ex) {
            if (conn != null) {
                  try {
                        conn.rollback();
                        System.out.println("Rolled back.");
                  } catch (SQLException exrb) {
                        exrb.printStackTrace();
                  }
            }
        } finally {
            try {
                if (orderStatement != null ) {
                    orderStatement.close();
                }
                if (saleStatement != null ) {
                    saleStatement.close();
                }
                conn.setAutoCommit(true);
            } catch (SQLException excs) {
                excs.printStackTrace();
            }
        }
    }
```

And here is the full source code of a test program:

```
    import java.sql.*;

    public class JDBCTransactionsDemo {
        private String dbURL = "jdbc:mysql://localhost:3306/sales";
        private String user = "root";
        private String password = "password";
        private Connection conn;

        public void connect() {
            try {
                conn = DriverManager.getConnection(dbURL, user, password);
                System.out.println("Connected.");
            } catch (SQLException ex) {
                ex.printStackTrace();
            }
        }

        public void disconnect() {
            try {
                conn.close();
                System.out.println("Closed.");
            } catch (SQLException ex) {
                ex.printStackTrace();
            }
        }

        public void saveOrder(int productId, Date orderDate, float amount, int
    reportMonth) {
            PreparedStatement orderStatement = null;
            PreparedStatement saleStatement = null;

            try {
           conn.setAutoCommit(false);
```

```
                 String sqlSaveOrder = "insert into orders (product_id,
order_date, amount)";
                  sqlSaveOrder += " values (?, ?, ?)";

                 String sqlUpdateTotal = "update monthly_sales set
total_amount = total_amount + ?";
           sqlUpdateTotal += " where product_id = ? and report_month = ?";

                 orderStatement = conn.prepareStatement(sqlSaveOrder);
                 saleStatement = conn.prepareStatement(sqlUpdateTotal);
           orderStatement.setInt(1, productId);
                 orderStatement.setDate(2, orderDate);
                 orderStatement.setFloat(3, amount);

                  saleStatement.setFloat(1, amount);
                  saleStatement.setInt(2, productId);
                  saleStatement.setInt(3, reportMonth);

                  orderStatement.executeUpdate();
                  saleStatement.executeUpdate();

                  conn.commit();
            } catch (SQLException ex) {
                  if (conn != null) {
                        try {
                              conn.rollback();
                              System.out.println("Rolled back.");
                        } catch (SQLException exrb) {
                              exrb.printStackTrace();
                        }
                  }
            } finally {
                  try {
                        if (orderStatement != null ) {
                              orderStatement.close();
                        }
                        if (saleStatement != null ) {
                              saleStatement.close();
                        }
                        conn.setAutoCommit(true);
                  } catch (SQLException excs) {
                        excs.printStackTrace();
                  }
            }
      }

      public static void main(String[] args) {
            JDBCTransactionsDemo demo = new JDBCTransactionsDemo();

            int productId = 1;
            int reportMonth = 7;
            Date date = new Date(System.currentTimeMillis());
            float amount = 580;

            demo.connect();
            demo.saveOrder(productId, date, amount, reportMonth);

            demo.disconnect();
        }
    }
```

Here, you understand how the `setAutoCommit()`, `commit()` and `rollback()` methods are used in real code.

4. Using Save Points in a Transaction

The JDBC API provides the **`Connection.setSavepoint()`** method that marks a point to which the transaction can be rolled back. The `rollback()` method is overloaded to takes a save point as its argument:

```
    connection.rollback(savepoint)
```

This allows us to undo only changes after the save point in case something wrong happen. The changes before the save point are still committed. The following workflow helps you understand how save points are used in a transaction:

```
    try {
        // begin the transaction:
        connection.setAutoCommit(false);

        // execute statement #1
        // execute statement #2
        // statements #1 & #2 are executed successfully till this point:

        Savepoint savepoint = connection.setSavepoint();
        // execute statement #3
        // execute statement #4

        if (/* something wrong */) {
            // roll back the transaction to the savepoint:
            connection.rollback(savepoint);
        }

        // execute statement #5
        // ...
        // commit the transaction
        connection.commit();
    } catch (SQLException ex) {
        // abort the transaction
        connection.rollback();
    } finally {
        // close statements
        connection.setAutoCommit(true);
    }
```

Now, let’s see a real code example. In the following program, the transaction consists of the following statement:

- insert a new product to the table products.

- insert a new order to the table orders.

- update total amount in the monthly sales.

In case the amount of the new order cannot help monthly sales > 10,000, the transaction is rolled back to the point where the new product was inserted.

Here’s the code of the program that shows how to use save point in a transaction with JDBC:

```
import java.sql.*;

public class JDBCTransactionSavePointDemo {
      private String dbURL = "jdbc:mysql://localhost:3306/sales";
      private String user = "root";
      private String password = "password";
      private Connection conn;

      public void connect() {
            try {
                  conn = DriverManager.getConnection(dbURL, user, password);
                  System.out.println("Connected.");
            } catch (SQLException ex) {
                  ex.printStackTrace();
            }
      }

      public void disconnect() {
            try {
                  conn.close();
                  System.out.println("Closed.");
            } catch (SQLException ex) {
                  ex.printStackTrace();
            }
      }

      public void saveOrder(String newProductName, float newProductPrice,
                        int productId, Date orderDate, float orderAmount, int
reportMonth) {
            PreparedStatement productStatement = null;
            PreparedStatement orderStatement = null;
            PreparedStatement saleStatement = null;
            PreparedStatement getTotalStatement = null;

            try {
                  conn.setAutoCommit(false);

                  String sqlSaveProduct = "insert into products (product_name,
price)";
                  sqlSaveProduct += " values (?, ?)";

                  productStatement = conn.prepareStatement(sqlSaveProduct);
                  productStatement.setString(1, newProductName);
                  productStatement.setFloat(2, newProductPrice);
                  productStatement.executeUpdate();
                  Savepoint savepoint = conn.setSavepoint();

                  String sqlSaveOrder = "insert into orders (product_id,
order_date, amount)";
                  sqlSaveOrder += " values (?, ?, ?)";
                  orderStatement = conn.prepareStatement(sqlSaveOrder);

                  orderStatement.setInt(1, productId);
                  orderStatement.setDate(2, orderDate);
                  orderStatement.setFloat(3, orderAmount);

                  orderStatement.executeUpdate();
                  String sqlGetTotal = "select total_amount from
monthly_sales";
                  sqlGetTotal += " where product_id = ? and report_month = ?";

                  getTotalStatement = conn.prepareStatement(sqlGetTotal);
                  getTotalStatement.setInt(1, productId);
                  getTotalStatement.setInt(2, reportMonth);

                  ResultSet rs = getTotalStatement.executeQuery();
                  rs.next();
                  float totalAmount = rs.getFloat("total_amount");
                  rs.close();

                  if (totalAmount + orderAmount < 10000) {
                        conn.rollback(savepoint);
                  }

                  String sqlUpdateTotal = "update monthly_sales set
total_amount = total_amount + ?";
                  sqlUpdateTotal += " where product_id = ? and report_month =
?";

                  saleStatement = conn.prepareStatement(sqlUpdateTotal);

                  saleStatement.setFloat(1, orderAmount);
                  saleStatement.setInt(2, productId);
                  saleStatement.setInt(3, reportMonth);
                  saleStatement.executeUpdate();
                  conn.commit();
            } catch (SQLException ex) {
                  if (conn != null) {
                        try {
                              conn.rollback();
                              System.out.println("Rolled back.");
                        } catch (SQLException exrb) {
                              exrb.printStackTrace();
                        }
                  }
            } finally {
                  try {
                        if (productStatement != null ) {
                              productStatement.close();
                        }
                        if (orderStatement != null ) {
                              orderStatement.close();
                        }
                        if (saleStatement != null ) {
                              saleStatement.close();
                        }
                        if (getTotalStatement != null ) {
                              getTotalStatement.close();
                        }
                        conn.setAutoCommit(true);
                  } catch (SQLException excs) {
                        excs.printStackTrace();
                  }
            }
      }

      public static void main(String[] args) {
            JDBCTransactionSavePointDemo demo = new
JDBCTransactionSavePointDemo();

            String newProductName = "iPod";
            float newProductPrice = 399;
            int productId = 1;
            int reportMonth = 7;
            Date date = new Date(System.currentTimeMillis());
            float orderAmount = 580;

            demo.connect();
            demo.saveOrder(newProductName, newProductPrice, productId, date,
orderAmount, reportMonth);
            demo.disconnect();
      }
}
```

NOTE:

- The JDBC API provides the **`Connection.releaseSavepoint(savepoint)`** method that removes the specified save point from the current transaction. A save point has been released become invalid and cannot be rolled back to. Any attempt to roll back the transaction to a released save point causes a `SQLException`.

- A save point is automatically released and becomes invalid when the transaction is committed or when the entire transaction is rolled back.

Executing Stored Procedures

* What is a Stored Procedure?

A stored procedure is executable code that is stored in the database. Stored procedures usually collect and customize common operations like inserting a row into a table, gathering statistical information or encapsulating complex business logic and calculations.

A complex series of SQL statements is often saved into a stored procedure to save time, memory, avoid network traffic and increase security.

In MySQL, we can use the **`CREATE PROCEDURE`** statement to create a stored procedure.

In this lesson, you will learn how to:

- Calling a simple stored procedure which has only IN parameters.

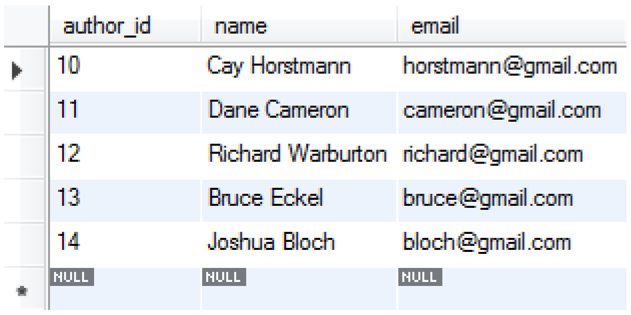
- Creating a stored procedure from Java.

- Calling a stored procedure which has IN, OUT and INOUT parameters and retrieve the values of these parameters.

- Calling a stored procedure which returns a result set and process this result set.

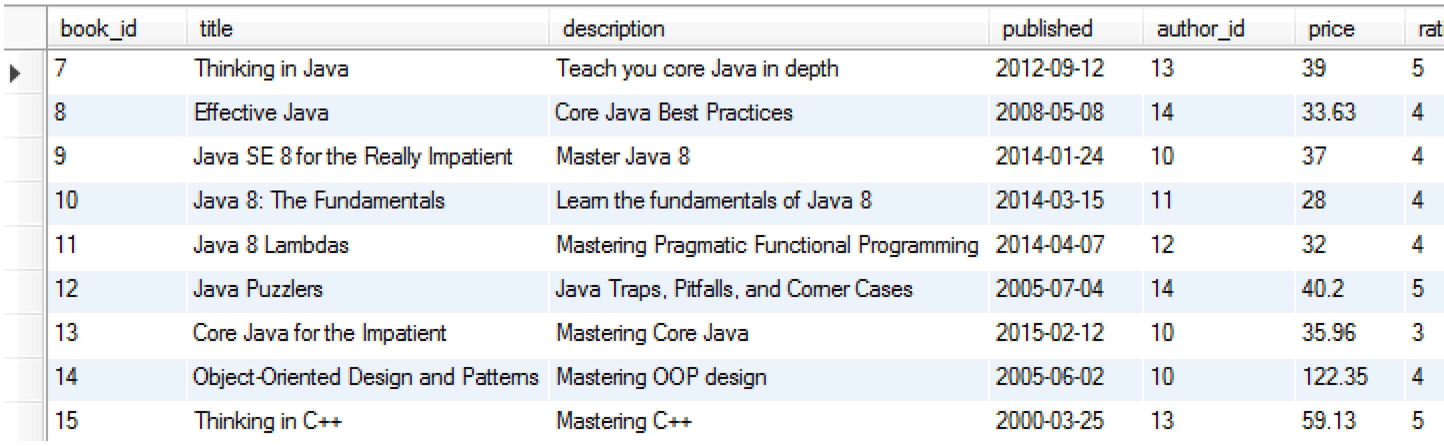
We will use MySQL database for the examples, and suppose that you know how to create a stored procedure using MySQL Workbench tool. Here's the database structure:

Imagine we have some dummy data for the table **author** as following:



author_id	name	email
10	Cay Horstmann	horstmann@gmail.com
11	Dane Cameron	cameron@gmail.com
12	Richard Warburton	richard@gmail.com
13	Bruce Eckel	bruce@gmail.com
14	Joshua Bloch	bloch@gmail.com
NULL	NULL	NULL

And dummy data for the table **book** as following:



book_id	title	description	published	author_id	price	rat
7	Thinking in Java	Teach you core Java in depth	2012-09-12	13	39	5
8	Effective Java	Core Java Best Practices	2008-05-08	14	33.63	4
9	Java SE 8 for the Really Impatient	Master Java 8	2014-01-24	10	37	4
10	Java 8: The Fundamentals	Learn the fundamentals of Java 8	2014-03-15	11	28	4
11	Java 8 Lambdas	Mastering Pragmatic Functional Programming	2014-04-07	12	32	4
12	Java Puzzlers	Java Traps, Pitfalls, and Corner Cases	2005-07-04	14	40.2	5
13	Core Java for the Impatient	Mastering Core Java	2015-02-12	10	35.96	3
14	Object-Oriented Design and Patterns	Mastering OOP design	2005-06-02	10	122.35	4
15	Thinking in C++	Mastering C++	2000-03-25	13	59.13	5

* Calling a Simple Stored Procedure from Java

In MySQL Workbench, create a new routine (expand the database and you see a node called **Routines**. Right click and select **Create Routine…)** and paste the following code:

```
CREATE PROCEDURE `booksdb`.`create_author` (IN name VARCHAR(45), email
VARCHAR(45))
BEGIN
   DECLARE newAuthorID INT;

   INSERT INTO author (name, email) VALUES (name, email);

   SET newAuthorID = (SELECT author_id FROM author a WHERE a.name = name);

   INSERT INTO book (title, description, published, author_id, price, rating)
       VALUES (CONCAT('Life Story of ', name),
               CONCAT('Personal Stories of ', name),
               date('2016-12-30'), newAuthorID, 10.00, 0);
END
```

As you can see, this stored procedure is named as ***create_author***. It has two input parameters ***name*** and ***email***.

In the body of the procedure (code between **BEGIN** and **END**), we insert a row into the table **author**. Then select the ID value of this recently inserted row (**author_id**), store it into a variable named **newAuthorID**. Then we insert a new row into the table **book**, in which we use the author name for the title and description of the book. Notice that the variable **newAuthorID** is used in the second INSERT statement to set foreign key for the new row.

Within the workbench, you can call this stored procedure by executing the following query:

```
call create_author('Patrick Maka', 'patrick@gmail.com')
```

Now, let's see how to call this stored procedure using JDBC.

Here are the steps to call a simple stored procedure from Java code with JDBC:

```
CallableStatement statement = connection.prepareCall("{call procedure_name(?,
?, ?)}");
// setting input parameters on the statement object
// statement.setString(parameterIndex, parameterValue);
statement.execute();
statement.close();
```

Notice the **`CALL`** statement syntax:

```
"{call procedure_name(?, ?, ?)}
```

The procedure's parameters are denoted by the question marks, separated by comma. Then we use the `setXXX()`methods on the statement object to set value for the parameters, just like setting parameters for a `PreparedStatement`.

Invoking `execute()` method on the statement object will run the specified stored procedure. This method returns true if the stored procedure returns a result set, false if not, and throw `SQLException` in cases of an error occurred.

The following is a test Java program that calls the stored procedure **create_author** which we created previously:

```
import java.sql.*;

public class StoredProcedureCallExample1 {
      public static void main(String[] args) {
            String dbURL = "jdbc:mysql://localhost:3306/booksdb";
            String user = "root";
            String password = "password";

            try (
                  Connection conn = DriverManager.getConnection(dbURL, user, 
password);
                  CallableStatement statement = conn.prepareCall("{call 
create_author(?, ?)}");
            ) {
                  statement.setString(1, "Bill Gates");
                  statement.setString(2, "bill@microsoft.com");
                  statement.execute();
                  statement.close();
                  System.out.println("Stored procedure called successfully!");
            } catch (SQLException ex) {
                  ex.printStackTrace();
            }
      }
}
```

Compile and run this program. You should see the following output:

```
Stored procedure called successfully!
```

Let’s verify the database. Querying all rows from the table **author** we see a new row was added:

And checking the table **book** also lets us see a new row added:

* Creating a Stored Procedure from Java

Besides using a database tool like MySQL Workbench, we can create a stored procedure from within a Java program by executing the “`CREATE PROCEDURE`” SQL statement, just like executing a normal SQL statement.

The following Java program creates a simple MySQL stored procedure called **delete_book** which removes a row from the table book based on the specified book ID:

```
import java.sql.*;

public class StoredProcedureCreateExample {
      public static void main(String[] args) {
            String dbURL = "jdbc:mysql://localhost:3306/booksdb";
            String user = "root";
            String password = "password";

            try (
                  Connection conn = DriverManager.getConnection(dbURL, user,
password);
                  Statement statement = conn.createStatement();
            ) {
                  String queryDrop = "DROP PROCEDURE IF EXISTS delete_book";
                  String queryCreate = "CREATE PROCEDURE delete_book (IN
bookID INT) ";

                  queryCreate += "BEGIN ";
                  queryCreate += "DELETE FROM book WHERE book_id = bookID; ";
                  queryCreate += "END";

                  // drops the existing procedure if exists
                  statement.execute(queryDrop);

                  // then creates a new stored procedure
                  statement.execute(queryCreate);
                  statement.close();

                  System.out.println("Stored procedure created
successfully!");
            } catch (SQLException ex) {
                  ex.printStackTrace();
            }
      }
}
```

Note that we have to execute two queries: the first one is to drop the stored procedure if exists; and the second actually creates the stored procedure.

Running this program would produce the following output:

```
Stored procedure created successfully!
```

Switch to MySQL Workbench and refresh the *Object Browser* pane, you should see the newly created stored procedure appears there.

* Calling a Stored Procedure Having OUT and INOUT parameters from Java

Consider the following stored procedure:

```
CREATE PROCEDURE `summary_report`(
       IN title VARCHAR(45),
       OUT totalBooks INT,
       OUT totalValue DOUBLE,
       INOUT highPrice DOUBLE
   )
BEGIN
   DECLARE maxPrice DOUBLE;

   SELECT COUNT(*) AS bookCount, SUM(price) as total
       FROM book b JOIN author a ON b.author_id = a.author_id
       AND b.title LIKE CONCAT('%', title, '%')
   INTO totalBooks, totalValue;




   SELECT MAX(price) FROM book WHERE price INTO maxPrice;



   IF (maxPrice > highPrice) THEN

       SET highPrice = maxPrice;

   END IF;

END
```

This stored procedure has 4 parameters:

- **IN** `title VARCHAR(45)`: input parameter. The procedure searches for books whose titles contain the words specified by this parameter.

- **OUT** `totalBooks INT`: The procedure counts total of the matching books and stores the value into this output parameter.

- **OUT** `totalValue DOUBLE`: The procedure counts total value of the matching books and stores the value into this output parameter.

- **INOUT** `highPrice DOUBLE`: This is both input/output parameter. The procedure selects the max price in all books and if it is greater than the parameter value, assigns it to the parameter.

Now, let's see how to execute this stored procedure using JDBC code.

To retrieve the values of the **OUT** and **INOUT** parameters, JDBC requires these parameters must be registered before calling the procedure, by invoking the following method on `CallableStatement`object:

```
void registerOutParameter(int parameterIndex, int sqlType)
```

For example, the following code registers 3 output parameters for the procedure **summary_report** above:

```
CallableStatement statement = conn.prepareCall("{call summary_report(?, ?, ?,
?)}");
statement.registerOutParameter(2, Types.INTEGER);
statement.registerOutParameter(3, Types.DOUBLE);
statement.registerOutParameter(4, Types.DOUBLE);
```

After the procedure has been called, we can use the `getXXX()` method on the `CallableStatement` object to retrieve the values of the output parameters. For example, the following code gets values of the 3 output parameters returned by the procedure **summary_report**:

```
Integer totalBook = (Integer) statement.getObject(2, Integer.class);
Double totalValue = statement.getDouble(3);
Double highPrice = statement.getDouble("highPrice");
```

As you can see, there are three ways to retrieve the values: by index and type; by index; and by parameter name.

And following is full source code of a test program:

```
import java.sql.*;
```

```
public class StoredProcedureCallExample2 {
      public static void main(String[] args) {
            String dbURL = "jdbc:mysql://localhost:3306/booksdb";
            String user = "root";
            String password = "password";

            try (
                  Connection conn = DriverManager.getConnection(dbURL, user,
password);
                  CallableStatement statement = conn.prepareCall("{call
summary_report(?, ?, ?, ?)}");
            ) {
                  statement.registerOutParameter(2, Types.INTEGER);
                  statement.registerOutParameter(3, Types.DOUBLE);
                  statement.registerOutParameter(4, Types.DOUBLE);

                  statement.setString(1, "Java");
                  statement.setDouble(4, 50);

                  statement.execute();

                  Integer totalBook = (Integer) statement.getObject(2,
Integer.class);
                  Double totalValue = statement.getDouble(3);
                  Double highPrice = statement.getDouble("highPrice");

                  System.out.println("Total books: " + totalBook);
                  System.out.println("Total value: " + totalValue);
                  System.out.println("High price: " + highPrice);

                  statement.close();
            } catch (SQLException ex) {
                  ex.printStackTrace();
            }
      }

}
```

Running this program would give the following output:

```
Total books: 7
Total value: 245.79000091552734
High price: 122.3499984741211
```

* Calling a Stored Procedure Returning a Result Set from Java

A stored procedure can returns a result set. Consider the following procedure:

```
CREATE PROCEDURE `get_books`(IN rate INT)
BEGIN
   SELECT * FROM book WHERE rating >= rate;
END
```

Let’s see how to retrieve this result set in Java. The following code snippet shows you how to retrieve and process a result set returned from a stored procedure using JDBC code:

```
CallableStatement statement = conn.prepareCall("{call get_books(?)}");
statement.setInt(1, 5);
boolean hadResults = statement.execute();

while (hadResults) {
       ResultSet resultSet = statement.getResultSet();
       // process result set
       while (resultSet.next()) {
              // retrieve values of fields
             String title = resultSet.getString("title");

       }
       hadResults = statement.getMoreResults();
}
```

And here is the full source code of a demo program:

```
import java.sql.*;

public class StoredProcedureCallExample3 {
      public static void main(String[] args) {
            String dbURL = "jdbc:mysql://localhost:3306/booksdb";
            String user = "root";
            String password = "password";

            try (
                  Connection conn = DriverManager.getConnection(dbURL, user,
password);
                  CallableStatement statement = conn.prepareCall("{call
get_books(?)}");
            ) {
                  statement.setInt(1, 5);
                  boolean hadResults = statement.execute();

                  // print headings
                  System.out.println("| Title | Description | Rating |");
                  System.out.println("================================");

                  while (hadResults) {
                        ResultSet resultSet = statement.getResultSet();
                        // process result set
                        while (resultSet.next()) {
                              String title = resultSet.getString("title");
                              String description =
resultSet.getString("description");
                              int rating = resultSet.getInt("rating");
                              System.out.println(
                                          "| " + title + " | " + description +
" | " + rating + " |");
                        }
                        hadResults = statement.getMoreResults();
                  }
                  statement.close();
            } catch (SQLException ex) {
                  ex.printStackTrace();
            }
      }
}
```

Running this program would print the following output:

```
| Title | Description | Rating |
================================
| Thinking in Java | Teach you core Java in depth | 5 |
| Java Puzzlers | Java Traps, Pitfalls, and Corner Cases | 5 |
| Thinking in C++ | Mastering C++ | 5 |
```

Scrollable Result Sets

In the previous lessons, you learned how to execute a SQL SELECT query that returns a result set represented by the `java.sql.ResultSet` object. By default, you can iterate over rows in the result set in forward direction only, from the first row to the last row, and you can't update data in the result set.

What if you need to move forth and back in the result set, jumping on any row when needed? Also, wouldn't it be more naturally to retrieve a result set and update its data against the database?

The JDBC API allows you to use such flexible result sets by making them ***scrollable*** and ***updatable*** - That's the topic you will learn today.

Let's start with scrollable result sets.

* Understanding Scrollable Result Sets:

By default, result sets are not scrollable or updatable. If you use the following code:

```
String sql = "SELECT * FROM student";
Statement statement = connection.createStatement();
ResultSet result = statement.executeQuery(sql);
```

The `ResultSet` object returned is not scrollable or updatable. To obtain a scrollable result set, you must create a different `Statement` object with the following method:

```
Statement statement = connection.createStatement(int resultSetType, int
resultSetConcurrency);
```

For a prepared statement, us the following method:

```
PreparedStatement statement = connection.prepareStatement(String sql, int
resultSetType, int resultSetConcurrency);
```

The possible values for `resultSetType` and `resultSetConcurrency` are defined by some constants in the `ResultSet` interface, which are described below.

* ResultSet Type Values:

You can create a `Statement` that returns result sets in one of the following types:

- **`TYPE_FORWARD_ONLY`**: the result set is not scrollable (default).

- **`TYPE_SCROLL_INSENSITIVE`**: the result set is scrollable but not sensitive to database changes.

- **`TYPE_SCROLL_SENSITIVE`**: the result set is scrollable and sensitive to database changes.

* ResultSet Concurrency Values:

A `Statement` can return result sets which are read-only or updatable, specified by one of the following constants defined in the `ResultSet` interface:

- **`CONCUR_READ_ONLY`**: the result set cannot be used to update the database (default).

- **`CONCUR_UPDATABLE`**: the result set can be used to update the database.

For example, if you want to scroll through the result set but don't want to update its data, create `Statement` a like this:

```
Statement statement =
connection.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
ResultSet.CONCUR_READ_ONLY);
```

Then the result set returned is now scrollable (but it doesn't reflect to database changes once it is loaded with the data):

```
ResultSet result = statement.executeQuery(sql);
```

You can use the following methods to scroll through the result set:

- **`first()`**: moves the cursor to the first row.

- **`next()`**: moves the cursor forward one row from its current position.

- **`previous()`**: moves the cursor to the previous row.

- **`relative(int rows)`**: moves the cursor a relative number of rows from its current position. The value of `rows` can be positive (move forward) or negative (move backward).

- **`absolute(int row)`**: moves the cursor to the given row number. The value of `row` can be positive or negative. A positive number indicates the row number counting from the beginning of the result set. A negative number indicates the row number counting from the end of the result set.

Let's see a complete example program.

The following program executes a SQL SELECT statement to retrieve all rows from the table `student` in the database `college`:

```
import java.sql.*;

public class ScrollableResultSetExample {
      public static void main(String[] args) {
            String url = "jdbc:mysql://localhost:3306/college";
            String username = "root";
            String password = "password";
            try (Connection conn = DriverManager.getConnection(url, username,
password)) {
                  String sql = "SELECT * FROM student";
                  Statement statement =
conn.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
ResultSet.CONCUR_READ_ONLY);
                  ResultSet result = statement.executeQuery(sql);

                  result.first();
                  readStudentInfo("first", result);

                  result.relative(3);
                  readStudentInfo("relative(3)", result);

                  result.previous();
                  readStudentInfo("previous", result);

                  result.absolute(4);
                  readStudentInfo("absolute(4)", result);

                  result.last();
                  readStudentInfo("last", result);

                  result.relative(-2);
                  readStudentInfo("relative(-2)", result);
            } catch (SQLException ex) {
                  ex.printStackTrace();
            }
      }

      private static void readStudentInfo(String position, ResultSet result)
throws SQLException {
            String name = result.getString("name");
            String email = result.getString("email");
            String major = result.getString("major");
            String studentInfo = "%s: %s - %s - %s\n";

            System.out.format(studentInfo, position, name, email, major);
      }
}
```

As you can see, this program creates a Statement that returns scrollable, insensitive and read-only result sets. It demonstrates how to scroll through the result set using the navigation methods described above.

* Checking ResultSet Types and ResultSet Concurrency Support:

Note that not all databases support scrollable, sensitive and updatable result sets. So you should check whether the database supports these behaviors or not before processing the result sets.

The `DatabaseMetaData` interface provides methods for checking if the underlying database supports a certain result set type and concurrency. You can obtain a `DatabaseMetaData` object from the connection using the following statement:

```
DatabaseMetaData metadata = conn.getMetaData();
```

And use the following method to check if the database supports a certain result set type:

```
metadata.supportsResultSetType(int resultSetType)
```

Use the following method to check if the database supports a certain result set concurrency:

```
metadata.supportsResultSetConcurrency(int resultSetType, int
resultSetConcurrency)
```

These methods return `true` if the specified type is supported, or `false` otherwise.

For example, the following code checks if the database supports scrollable and sensitive result sets:

```
boolean isScrollSensitive =
metadata.supportsResultSetType(ResultSet.TYPE_SCROLL_SENSITIVE);
System.out.println("Support Scroll Sensitive: " + isScrollSensitive);
```

To check if the database supports scrollable, changes-sensitive and updatable result sets, use the following code:

```
boolean isUpdatable =
metadata.supportsResultSetConcurrency(ResultSet.TYPE_SCROLL_SENSITIVE,
ResultSet.CONCUR_UPDATABLE);
System.out.println("Support Updatable: " + isUpdatable);
```

Let’s see another example. The following program uses a scrollable result set that is sensitive to database changes. It allows the user to enter a row number to jump on:

```
import java.sql.*;
import java.io.*;

public class ScrollableResultSetSensitiveExample {
      public static void main(String[] args) {
            String url = "jdbc:mysql://localhost:3306/college";
            String username = "root";
            String password = "password";

            Console console = System.console();

            try (Connection conn = DriverManager.getConnection(url, username,
password)) {
                  DatabaseMetaData metadata = conn.getMetaData();
                  boolean isScrollSensitive =
metadata.supportsResultSetType(ResultSet.TYPE_SCROLL_SENSITIVE);

                  if (!isScrollSensitive) {
                        System.out.println("The database doesn't support
scrollable and sensitive result sets.");
                        return;
                  }

                  String sql = "SELECT * FROM student";
                  Statement statement =
conn.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
ResultSet.CONCUR_READ_ONLY);
                  ResultSet result = statement.executeQuery(sql);

                  int row = -1;
                  while (row != 0) {
                        row = Integer.parseInt(console.readLine("Enter row
number: "));
                        if (result.absolute(row)) {
                              readStudentInfo("Student at row " + row + ": ",
result);
                        } else {
                              System.out.println("There's no student at row "
+ row);
                        }
                  }
            } catch (SQLException ex) {
                  ex.printStackTrace();
            }
      }
      private static void readStudentInfo(String position, ResultSet result)
throws SQLException {
            String name = result.getString("name");
            String email = result.getString("email");
            String major = result.getString("major");
            String studentInfo = "%s: %s - %s - %s\n";

            System.out.format(studentInfo, position, name, email, major);
      }
}
```

As you can see, this program allows the user to print data of any row in the result set. It repeats until the user types 0 to quit.

You can test the sensitive to database changes feature by running this program to query data at a given row number. Then use MySQL Command Line Client program to updates that row, and then come back to the program to query data from that row again, you will see the changes are reflected automatically.

Scrollable result sets that are sensitive to database changes is a useful feature that helps programmers write code easily with minimum effort (reducing the code that executes SQL statements).

Imagine you are developing database application which is used concurrently by multiple users. Besides the ability to scroll through the result set in any direction, changes to the database made by one user can be reflected to other users automatically. But make sure the database supports scrollable and sensitive result sets first.

Updatable Result Sets

In this lesson you will learn how to use updatable result sets in JDBC. By default, result sets are not updatable if you create a `Statement`object with no-argument constructor like this:

```
Statement statement = connection.createStatement();
```

or if you create a `PreparedStatement` object like this:

```
PreparedStatement statement = connection.prepareStatement(sql);
```

So to use updatable result sets, you must specify the appropriate result set type and result set concurrency values when create a `Statement` object:

```
Statement statement = connection.createStatement(int resultSetType, int
resultSetConcurrency);
```

and for a prepared statement:

```
PreparedStatement statement = connection.prepareStatement(String sql, int
resultSetType, int resultSetConcurrency);
```

Remember the possible values for result set type are defined by the following constants in the `ResultSet` interface:

- **`TYPE_FORWARD_ONLY`**: the result set is not scrollable (default).

- **`TYPE_SCROLL_INSENSITIVE`**: the result set is scrollable but not sensitive to database changes.

- **`TYPE_SCROLL_SENSITIVE`**: the result set is scrollable and sensitive to database changes.

and the possible values for the result set concurrency:

- **`CONCUR_READ_ONLY`**: the result set cannot be used to update the database (default).

- **`CONCUR_UPDATABLE`**: the result set can be used to update the database.

For example, the following code creates a `Statement` object that will produce updatable result sets which are scrollable and insensitive to database changes:

```
Statement statement = connection.createStatement(
                  ResultSet.TYPE_SCROLL_INSENSITIVE,
ResultSet.CONCUR_UPDATABLE);
```

and for a `PreparedStatement` object:

```
PreparedStatement statement = connection.prepareStatement(sql,
                  ResultSet.TYPE_SCROLL_INSENSITIVE,
ResultSet.CONCUR_UPDATABLE);
```

Then the result set returned is updatable:

```
ResultSet result = statement.executeQuery(sql);
```

It’s also recommended to check if the underlying database supports updatable result sets or not, for example:

```
DatabaseMetaData metadata = connection.getMetaData();

boolean isUpdatable = metadata.supportsResultSetConcurrency(
            ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_UPDATABLE);
if (!isUpdatable) {
      // exit
}
// continue
```

* Updating the Current Row in the Result Set:

You can update data of the current row in the result set by calling **`updateXXX()`** methods:

```
updateString(int columnIndex, String x)
updateString(String columnLabel, String x)
updateInt(int columnIndex, int x)
updateInt(String columnLabel, int x)
....
```

As you can see, you can specify the column in the current row by using either column index or column label. Note that `columnIndex` is the order of the column in the result set, which may be different than the order in the actual database.

It's recommended to specify columns by their names (labels) instead of indexes for code readability and correctness.

After calling `updateXXX()` methods, call the **`updateRow()`** method to commit the changes to the database.

For example, the following code snippet executes a SQL SELECT query to get all rows from the `student` table, and then update the 3rd row:

```
String sql = "SELECT * FROM student";
Statement statement = conn.createStatement(
            ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_UPDATABLE);

ResultSet result = statement.executeQuery(sql);
result.absolute(3);

result.updateString("name", "New Name");
result.updateString("email", "newemail@gmail.com");
result.updateString("major", "New Major");
```

```
        result.updateRow();
```

* Inserting a New Row in the Result Set:

The following code snippet shows you how to insert a new row in the result set:

```
        result.moveToInsertRow();

        result.updateString("name", "New Name");
        result.updateString("email", "newemail@gmail.com");
        result.updateString("major", "New Major");

        result.insertRow();
        result.moveToCurrentRow();
```

You see, first you need to move the cursor to the appropriate position by calling `moveToInsertRow()`. Use `updateXXX()` methods to specify values for columns in the row, and then call `insertRow()` to save changes to the database.

The method `moveToCurrentRow()` moves the cursor back to the remembered position before the new row was inserted.

* Deleting the Current Row in the Result Set:

To remove the current row in the result set, simply call `deleteRow()`. For example:

```
        result.deleteRow();
```

That’s how to perform changes on an updatable result set. Let’s see a complete example program below.

NOTE: The `updateRow()`, `insertRow()` and `deleteRow()` methods throw `SQLException` if a database access error occurs, or the result set is in read-only mode, or they are called on a closed result set.

* Updatable ResultSet Example Program:

The following program retrieves all rows from the `student` table in a MySQL database schema named `college`. It allows the user to enter a row number to see the details of that row. Then it asks if the user wishes to update, delete or insert row (type ‘y’ to confirm).

Here’s the complete code of the program:

```
import java.sql.*;
import java.io.*;

public class UpdatableResultSetExample {
      public static void main(String[] args) {
            String url = "jdbc:mysql://localhost:3306/college";
            String username = "root";
            String password = "password";

            Console console = System.console();

            try (Connection conn = DriverManager.getConnection(url, username,
password)) {
                  DatabaseMetaData metadata = conn.getMetaData();

                  boolean isUpdatable = metadata.supportsResultSetConcurrency(
                              ResultSet.TYPE_SCROLL_INSENSITIVE,
ResultSet.CONCUR_UPDATABLE);
                  if (!isUpdatable) {
                        System.out.println("The database does not support
updatable result sets.");
                        return;
                  }

                  String sql = "SELECT * FROM student";
                  Statement statement = conn.createStatement(
                              ResultSet.TYPE_SCROLL_INSENSITIVE,
ResultSet.CONCUR_UPDATABLE);

                  ResultSet result = statement.executeQuery(sql);

                  int row = -1;
                  while (row != 0) {
                        row = Integer.parseInt(console.readLine("Enter row
number: "));
                        if (result.absolute(row)) {
                              readStudentInfo("Student at row " + row + ": ",
result);
                              String answer = console.readLine("Do you want to
update this row (Y/N)?: ");

                              if (answer.equalsIgnoreCase("Y")) {
                                    String name = console.readLine("\tUpdate
name: ");
                                    String email = console.readLine("\tUpdate
email: ");
```

```
                                    String major = console.readLine("\tUpdate
major: ");

                                    result.updateString("name", name);
                                    result.updateString("email", email);
                                    result.updateString("major", major);

                                    result.updateRow();
                                    System.out.println("The student at row " +
row + " has been updated.");
                              }

                              answer = console.readLine("Do you want to delete
this row (Y/N)?: ");

                              if (answer.equalsIgnoreCase("Y")) {
                                    result.deleteRow();
                                    System.out.println("The student at row " +
row + " has been deleted.");
                              }

                              answer = console.readLine("Do you want to insert
new row (Y/N)?: ");

                              if (answer.equalsIgnoreCase("Y")) {
                                    result.moveToInsertRow();

                                    String name = console.readLine("\tUpdate
name: ");
                                    String email = console.readLine("\tUpdate
email: ");
                                    String major = console.readLine("\tUpdate
major: ");

                                    result.updateString("name", name);
                                    result.updateString("email", email);
                                    result.updateString("major", major);

                                    result.insertRow();
                                    result.moveToCurrentRow();
                                    System.out.println("The new student has
been inserted.");
                              }
                        } else {
                              System.out.println("There's no student at row "
+ row);
                        }
                  }
            } catch (SQLException ex) {
                  ex.printStackTrace();
            }
      }
      private static void readStudentInfo(String position, ResultSet result)
                  throws SQLException {
            String name = result.getString("name");
```

```
                String email = result.getString("email");
                String major = result.getString("major");
                String studentInfo = "%s: %s - %s - %s\n";

                System.out.format(studentInfo, position, name, email, major);
            }
        }
```

This program runs in an infinite loop until the user enters row number 0 to quit.

Row Set and Cached Row Set

In the previous lessons you learned about scrollable and updatable result sets which are powerful and flexible, but they have a drawback: it needs to keep the database connection open during the entire user interaction. This behavior may affect your application’s performance in case there are many concurrent connections to the database.

In addition, being tied to database connection make `ResultSet` objects unable to be transferred between components or tiers in an application.

In this situation, you can use a ***row set*** which doesn’t need to keep the database connection always open, and is leaner than a result set.

In this lesson, I will help you understand row set and how to use one of its implementations - cached row set.

* Understanding RowSet:

A row set contains all data from a result set, but it can be disconnected from the database. A row set may make a connection with a database and keep the connection open during its life cycle, in which case it is called *connected row set*.

A row set may also make connection with a database, get data from it, and then close the connection. Such a row set is called *disconnected row set*.

You can make changes to data in a disconnected row set, and commit changes to the database later (the row set must re-establish the connection with the database).

In JDBC, a row set is represented by the `RowSet` interface which is defined in the `javax.sql` package. The `javax.sql package` is an extension of JDBC, besides the primary package `java.sql`.

The `RowSet` interface extends the `java.sql.ResultSet` interface, which means you can use a row set just like a result set.

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

- A **`CachedRowSet`** stores data in memory so you can work on the data without keeping the connection open all the time. `CachedRowSet` is the super interface of the ones below.

- A **`FilteredRowSet`** allows filtering data without having to write SQL SELECT queries.

- A **`JoinRowSet`** combines data from different `RowSet` objects, which is equivalent to SQL JOIN queries.

- A **`JdbcRowSet`** is a thin wrapper around a `ResultSet` that makes it possible to use the result set as a JavaBeans component.

- A **`WebRowSet`** can read and write data in XML format, making it possible to transfer the data through tiers in a web application.

In this lesson, I will guide you how to use `CachedRowSet` as a reference. For other types of row sets, please refer to its Javadocs.

* Understanding CachedRowSet:

A `CachedRowSet` object is a container for rows of data that caches its rows in memory, which makes it possible to operate (scroll and update) without keeping the database connection open all the time.

A `CachedRowSet` object makes use of a connection to the database only briefly: while it is reading data to populate itself with rows, and again while it is committing changes to the underlying database. So the rest of the time, a `CachedRowSet` object is disconnected, even while its data is being modified. Hence it is called disconnected row set.

Being disconnected, a `CachedRowSet` object is much leaner than a `ResultSet` object, making it easier to pass a `CachedRowSet` object to another component in an application.

You can modify data in a `CachedRowSet` object, but the modifications are not immediately reflected in the database. You need to make an explicit request to accept accumulated changes (insert, update and delete rows). The `CachedRowSet` then reconnects to the database and issues SQL statements to commit the changes.

* Creating a CachedRowSet Object:

You can create a `CachedRowSet` object either from a reference implementation provided by JDK (default), or from an implementation of database vendor (if available).

The following code snippet creates a `CachedRowSet` object by using a `RowSetFactory` which is created by the `RowSetProvider`:

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

This creates a `CachedRowSet` object from the implementation class `com.sun.rowset.CachedRowSetImpl`. It's equivalent to the following statement:

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

However, it's recommended to create a `CachedRowSet` object from a `RowSetFactory` because the reference implementation may be changed in future.

* Populating Data to a CachedRowSet:

There are two ways for populating data from the database to a `CachedRowSet` object:

- Populate data from an existing `ResultSet` object.

- Populate data by executing a SQL command.

Let's see each way in details.

Populate data to a `CachedRowSet` object from a ResultSet object:

Given a `ResultSet` object which is created from a `Statement`, the following code populates data from the result set to the cached row set:

```
ResultSet result = statement.executeQuery(sql);
RowSetFactory factory = RowSetProvider.newFactory();
CachedRowSet rowset = factory.createCachedRowSet();
rowset.populate(result);
```

Now you can close the connection and still be able to scroll through rows in the rowset.

Populate data to a `CachedRowSet` object by executing SQL command:

In this case, you need to set database connection properties and SQL statement for the `CachedRowSet` object, and then call the `execute()` method. For example:

```
String url = "jdbc:mysql://localhost:3306/college";
String username = "root";
String password = "password";
RowSetFactory factory = RowSetProvider.newFactory();
CachedRowSet rowset = factory.createCachedRowSet();
rowset.setUrl(url);
rowset.setUsername(username);
rowset.setPassword(password);
rowset.setCommand(sql);
rowset.execute();
```

After the `CachedRowSet` object is populated , you can iterate over its rows by using `ResultSet`'s methods because `CachedRowSet` extends `ResultSet`. For example, the following code snippet iterates all rows in the row set and print details of each row:

```
rowset.populate(result);

while (rowset.next()) {
      String name = rowset.getString("name");
      String email = rowset.getString("email");
      String major = rowset.getString("major");
      System.out.printf("%s - %s - %s\n", name, email, major);
}
```

* Modifying Data in a CachedRowSet:

You can make changes to data (insert, update and delete rows) in a `CachedRowSet`object just like you do with an updatable `ResultSet`. But the changes are not immediately reflected in the database until you explicitly request to accept changes.

Before making any changes, you must set table name for the row set so it knows the table needs to be updated:

```
rowset.setTableName("student");
```

For example, the following code snippet updates the 5th row in the row set:

```
rowset.absolute(5);
rowset.updateString("name", name);

rowset.updateString("email", email);

rowset.updateString("major", major);
rowset.updateRow();
```

The following code inserts a new row to the row set:

```
rowset.moveToInsertRow();

rowset.updateNull("student_id");
rowset.updateString("name", name);
rowset.updateString("email", email);
rowset.updateString("major", major);
rowset.insertRow();

rowset.moveToCurrentRow();
```

Note that you must call `updateNull(column_name)` for the primary key column of the table if that column's values are auto-generated. Otherwise an exception throws.

And the following code removes the current row in the row set:

```
rowset.deleteRow();
```

* Committing Changes to the Database:

To actually save the accumulated changes (update, insert and delete rows) to the underlying database, call:

```
rowset.acceptChanges();
```

If the `CachedRowSet` object is populated from a `ResultSet` object, pass the `Connection` object to the method:

```
rowset.acceptChanges(connection);
```

Note that the `acceptChanges()` method throws `SyncProviderException` if it found conflicts when trying to synchronize with the database. So you must handle this exception. Also make sure to disable auto commit mode:

```
connection.setAutoCommit(false);
```

* A Complete CachedRowSet Example Program:

Let’s see a complete program that demonstrates how to use `CachedRowSet`. The following program populates rows from `student` table in a MySQL database named `college`. Then it asks the user to update, delete and insert rows interactively.

Here’s the code of the program:

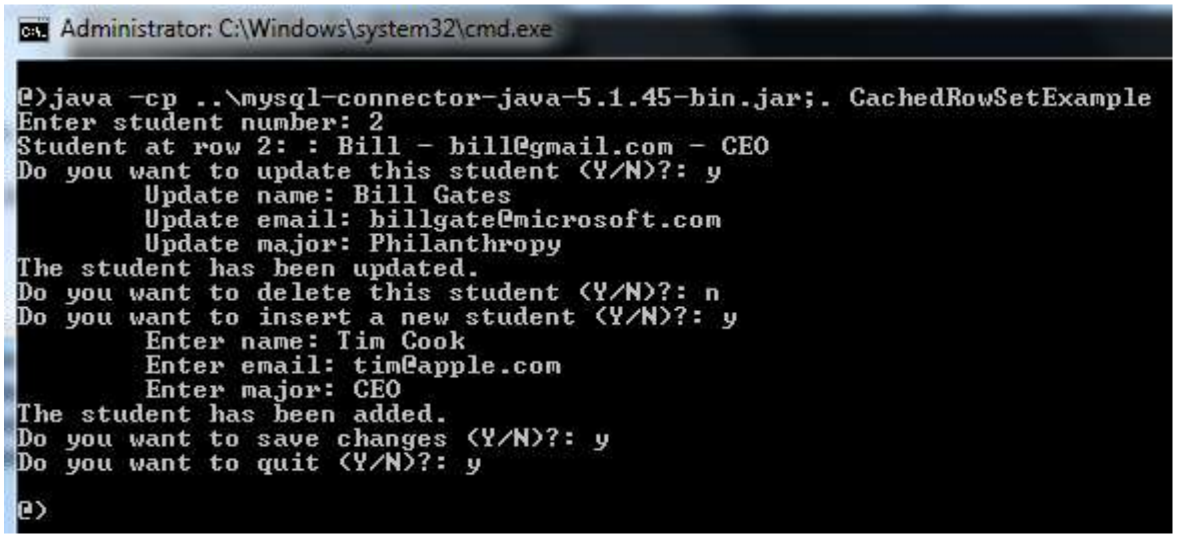
```
import java.sql.*;

import javax.sql.rowset.*;

import javax.sql.rowset.spi.*;

import java.io.*;



public class CachedRowSetExample1 {

      static Console console = System.console();

      static String answer;

      static boolean quit = false;



      public static void main(String[] args) {

            String url = "jdbc:mysql://localhost:3306/college";

            String username = "root";

            String password = "password";



            try (Connection conn = DriverManager.getConnection(url, username,
password)) {

                  conn.setAutoCommit(false);



                  String sql = "SELECT * FROM student";
            Statement statement = conn.createStatement();
            ResultSet result = statement.executeQuery(sql);

            RowSetFactory factory = RowSetProvider.newFactory();
            CachedRowSet rowset = factory.createCachedRowSet();

            rowset.setTableName("student");
            rowset.populate(result);

            while (!quit) {
                if (!readStudent(rowset)) continue;

                updateStudent(rowset);
                deleteStudent(rowset);
                insertStudent(rowset);

                saveChanges(rowset, conn);
                askToQuit();
            }
        } catch (SQLException ex) {
            System.out.println(ex.getMessage());
            ex.printStackTrace();
        }
    }
```

```
      static void readStudentInfo(String position, ResultSet result)
            throws SQLException {
            String name = result.getString("name");
            String email = result.getString("email");
            String major = result.getString("major");
            String studentInfo = "%s: %s - %s - %s\n";

            System.out.format(studentInfo, position, name, email, major);
      }

      static boolean readStudent(ResultSet result) throws SQLException {
            int row = Integer.parseInt(console.readLine("Enter student number:
"));
            if (result.absolute(row)) {
                  readStudentInfo("Student at row " + row + ": ", result);
                  return true;
            } else {
                  System.out.println("There's no student at row " + row);
                  return false;
            }
      }

      static void updateStudent(ResultSet result) throws SQLException {
            answer = console.readLine("Do you want to update this student
(Y/N)?: ");
```

```
            if (answer.equalsIgnoreCase("Y")) {

                  String name = console.readLine("\tUpdate name: ");

                  String email = console.readLine("\tUpdate email: ");

                  String major = console.readLine("\tUpdate major: ");


                  if (!name.equals("")) result.updateString("name", name);

                  if (!email.equals("")) result.updateString("email", email);

                  if (!major.equals("")) result.updateString("major", major);


                  result.updateRow();

                  System.out.println("The student has been updated.");

            }

      }


      static void deleteStudent(ResultSet result) throws SQLException {

            answer = console.readLine("Do you want to delete this student
(Y/N)?: ");


            if (answer.equalsIgnoreCase("Y")) {

                  result.deleteRow();

                  System.out.println("The student has been removed.");

            }

      }


      static void insertStudent(ResultSet result) throws SQLException {
            answer = console.readLine("Do you want to insert a new student 
(Y/N)?: ");


            if (answer.equalsIgnoreCase("Y")) {

                  String name = console.readLine("\tEnter name: ");

                  String email = console.readLine("\tEnter email: ");

                  String major = console.readLine("\tEnter major: ");


                  result.moveToInsertRow();


                  result.updateNull("student_id");

                  result.updateString("name", name);

                  result.updateString("email", email);

                  result.updateString("major", major);


                  result.insertRow();

                  result.moveToCurrentRow();


                  System.out.println("The student has been added.");

            }

      }


      static void saveChanges(CachedRowSet rowset, Connection conn) {

            answer = console.readLine("Do you want to save changes (Y/N)?: ");


            if (answer.equalsIgnoreCase("Y")) {
```

```
                        try {

                                rowset.acceptChanges(conn);

                        } catch (SyncProviderException ex) {

                                System.out.println("Error commiting changes to the
database: " + ex);

                        }

                }

        }


        static void askToQuit() {

                answer = console.readLine("Do you want to quit (Y/N)?: ");

                quit = answer.equalsIgnoreCase("Y");

        }

}
```

The following screenshot illustrates how to run and use the program:



```
Administrator: C:\Windows\system32\cmd.exe

@)java -cp ..\mysql-connector-java-5.1.45-bin.jar;. CachedRowSetExample
Enter student number: 2
Student at row 2: : Bill - bill@gmail.com - CEO
Do you want to update this student (Y/N)?: y
        Update name: Bill Gates
        Update email: billgate@microsoft.com
        Update major: Philanthropy
The student has been updated.
Do you want to delete this student (Y/N)?: n
Do you want to insert a new student (Y/N)?: y
        Enter name: Tim Cook
        Enter email: tim@apple.com
        Enter major: CEO
The student has been added.
Do you want to save changes (Y/N)?: y
Do you want to quit (Y/N)?: y

@)
```

That's about `CachedRowSet` in JDBC.

Database Meta Data

Besides retrieving the actual data stored in tables from a database, you can also read information about that data such as table names, column names, column types, column sizes, and other capabilities and features supported by the DBMS. This information about information is called ***meta data***.

In this lesson, you will learn how to read database meta data in JDBC with two interfaces `DatabaseMetaData` and `ResultSetMetaData`.

Reading database meta data is useful when you want to create database tools that allow the users to explore the structure of databases of different DBMSs, or when you have to check whether the underlying database supports some features or not, to process further accordingly. Remember you have to use `DatabaseMetaData` to check if the database supports scrollable and updatable result sets in the previous lesson?

Once a connection is established with the database, you can retrieve meta data about that database by invoking the `getMetaData()` method on the `Connection` object:

```
Connection connection = DriverManager.getConnection(url, username, password);
DatabaseMetaData meta = connection.getMetaData();
```

The `getMetaData()` method returns an object that implements the `DatabaseMetaData` interface that provides a variety of methods to read comprehensive information about the database as a whole.

* The DatabaseMetaData Interface:

This interface is implemented by JDBC driver to let users (programmers) know the capabilities of the DBMS as well as information about the JDBC driver itself.

You know, different DBMSs often support different features, implement features in different ways, and use different data types. Therefore, having access to such information is helpful for developing database tools that can work with different databases.

The `DatabaseMetaData` interface is huge, containing hundreds of methods for reading capabilities of a DBMS. So in this lesson, I show you only a small portion of its API, and you should refer to its Javadoc for the complete list of methods.

For example, the following code reads information about database product name and version:

```
DatabaseMetaData meta = connection.getMetaData();
String productName = meta.getDatabaseProductName();
String productVersion = meta.getDatabaseProductVersion();
```

You will see more examples below.

* The ResultSetMetaData Interface:

A `ResultSet` object can be used to get information about the types and properties of columns in the result set returned by a query.

The `ResultSet` interface provides the `getMetaData()` method that returns a `ResultSetMetaData` object which you can use to obtain information about columns in the result set. For example, the following code gets the number of columns in the result set:

```
String sql = "SELECT * FROM student";
Statement statement = conn.createStatement();
ResultSet result = statement.executeQuery(sql);
ResultSetMetaData rsMeta = result.getMetaData();
int numberOfColumns = rsMeta.getColumnCount();
```

Let's see more detailed code examples about `DatabaseMetaData` and `ResultSetMetaData` below.

* Reading Structural Information of Tables in a Database:

To retrieve names of all tables, call **`getTables()`** method of the `DatabaseMetaData` interface like the following code:

```
String catalog = null, schemaPattern = null, tableNamePattern = null;
String[] types = {"TABLE"};
ResultSet result = meta.getTables(catalog, schemaPattern, tableNamePattern,
types);
```

Here, the first 3 parameters are set to null indicates that we don't want to narrow the search based on those parameters. The `getTables()` method returns a `ResultSet` object in which the 3rd column stores the name of the table. The following code reads name of the first table:

```
result.next();
String tableName = result.getString(3);
```

To read meta data about columns of a particular table, call the **`getColumns()`** method `DatabaseMetaData` interface like this:

```
String catalog = null, schemaPattern = null, columnNamePattern = null;
String tableName = "student";
ResultSet rsColumns = meta.getColumns(catalog, schemaPattern, tableName,
columnNamePattern);
```

Again, the null arguments meaning that they are not used to narrow the search. The `getColumns()` method returns a `ResultSet` object which you can use its `getXXX()`methods to read properties of the columns such as name, type, size, etc.

To get information about primary keys of a table, invoke the `getPrimaryKeys()` method of the `DatabaseMetaData` interface like this:

```
String catalog = null, schemaPattern = null;
String tableName = "student";
ResultSet rsPK = meta.getPrimaryKeys(catalog, schemaPattern, tableName);
```

The returned `ResultSet` object stores the name of the primary columns.

The following program demonstrates how to read structural information of a database, including table names, column names, column types, column sizes, and primary keys. Here’s the code:

```
import java.sql.*;

public class ReadDatabaseStructureExample {
      public static void main(String[] args) {
            String url = "jdbc:mysql://localhost:3306/college";
            String username = "root";
            String password = "password";

            try (Connection conn = DriverManager.getConnection(url, username,
password)) {
                  DatabaseMetaData meta = conn.getMetaData();

                  String catalog = null, schemaPattern = null,
tableNamePattern = null;
                  String[] types = {"TABLE"};
                  ResultSet rsTables = meta.getTables(catalog, schemaPattern,
tableNamePattern, types);

                  while (rsTables.next()) {
                        String tableName = rsTables.getString(3);
                        System.out.println("\n=== TABLE: " + tableName);

                        String columnNamePattern = null;
                        ResultSet rsColumns = meta.getColumns(catalog,
schemaPattern, tableName, columnNamePattern);
                        ResultSet rsPK = meta.getPrimaryKeys(catalog,
schemaPattern, tableName);

                        while (rsColumns.next()) {
                              String columnName =
rsColumns.getString("COLUMN_NAME");
                              String columnType =
rsColumns.getString("TYPE_NAME");
                              int columnSize =
rsColumns.getInt("COLUMN_SIZE");
                              System.out.println("\t" + columnName + " - " +
columnType + "(" + columnSize + ")");
                        }

                        while (rsPK.next()) {
                              String primaryKeyColumn =
rsPK.getString("COLUMN_NAME");
                              System.out.println("\tPrimary Key Column: " +
primaryKeyColumn);
                        }
                  }
            } catch (SQLException ex) {
                  System.out.println(ex.getMessage());
                  ex.printStackTrace();
            }
      }
}
```

Run this program and you may see the output something like this:

NOTE: You can also use `ResultSetMetaData` to read meta data about columns in a result set returned from a SQL SELECT query like this:

```
String sql = "SELECT * FROM student";
Statement statement = conn.createStatement();
ResultSet result = statement.executeQuery(sql);
ResultSetMetaData rsMeta = result.getMetaData();
```

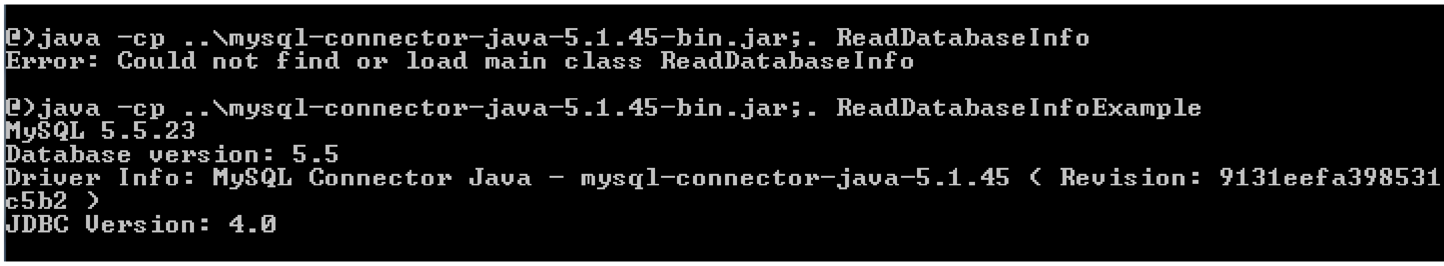
However it is not recommended because it requires getting all rows from the table first.

* Reading Information about Database Product and JDBC Driver:

The following program demonstrates how to use the `DatabaseMetaData` interface to get information about the database product and JDBC driver software such as their names and versions. Here’s the code:

```
import java.sql.*;

public class ReadDatabaseInfoExample {
      public static void main(String[] args) {
            String url = "jdbc:mysql://localhost:3306/college";
            String username = "root";
            String password = "password";

            try (Connection conn = DriverManager.getConnection(url, username, 
password)) {
                  DatabaseMetaData meta = conn.getMetaData();

                  String productName = meta.getDatabaseProductName();
                  String productVersion = meta.getDatabaseProductVersion();
                  System.out.println(productName + " " + productVersion);

                  int majorVersion = meta.getDatabaseMajorVersion();
                  int minorVersion = meta.getDatabaseMinorVersion();
                  System.out.printf("Database version: %d.%d\n", majorVersion, 
minorVersion);

                  String driverName = meta.getDriverName();
                  String driverVersion = meta.getDriverVersion();
                  System.out.println("Driver Info: " + driverName + " - " + 
driverVersion);

                  int jdbcMajorVersion = meta.getJDBCMajorVersion();
                  int jdbcMinorVersion = meta.getJDBCMinorVersion();
                  System.out.println("JDBC Version: " + jdbcMajorVersion + "." 
+ jdbcMinorVersion);
            } catch (SQLException ex) {
                  System.out.println(ex.getMessage());
                  ex.printStackTrace();
            }
      }
}
```

This program would produce the following output:



```
@)java -cp ..\mysql-connector-java-5.1.45-bin.jar;. ReadDatabaseInfo
Error: Could not find or load main class ReadDatabaseInfo

@)java -cp ..\mysql-connector-java-5.1.45-bin.jar;. ReadDatabaseInfoExample
MySQL 5.5.23
Database version: 5.5
Driver Info: MySQL Connector Java - mysql-connector-java-5.1.45 ( Revision: 9131eefa398531
c5b2 )
JDBC Version: 4.0
```

* Checking Supported Features:

Another common usage of `DatabaseMetaData` is to check whether the database support some certain features or not, as in some cases, you have to perform the check before proceeding further.

For example, the following code checks if the database supports scrollable and updatable result sets:

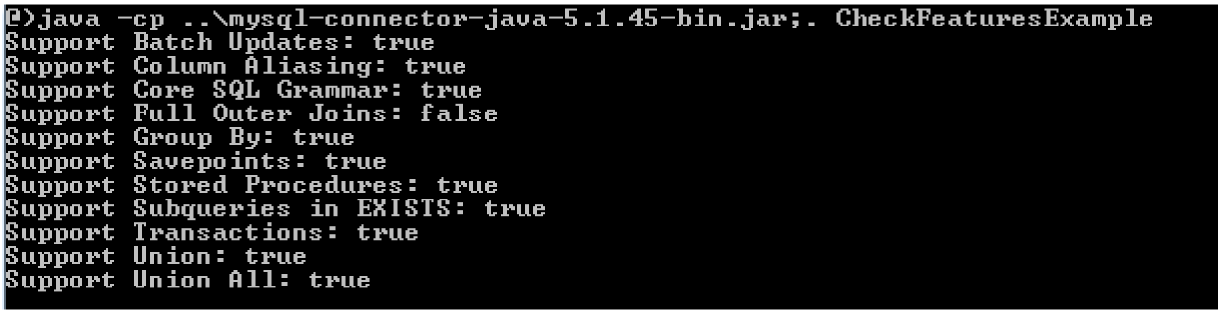
```
boolean scrollableUpdatable = meta.supportsResultSetConcurrency(
            ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_UPDATABLE);
System.out.println("Support Scrollable & Updatable Result Set: " +
scrollableUpdatable);
```

And the following program illustrates how to check various features supported by the DBMS:

```
import java.sql.*;

public class CheckFeaturesExample {
      public static void main(String[] args) {
            String url = "jdbc:mysql://localhost:3306/college";
            String username = "root";
            String password = "password";

            try (Connection conn = DriverManager.getConnection(url, username,
password)) {
                  DatabaseMetaData meta = conn.getMetaData();

                  System.out.println("Support Batch Updates: " +
meta.supportsBatchUpdates());
                  System.out.println("Support Column Aliasing: " +
meta.supportsColumnAliasing());
                  System.out.println("Support Core SQL Grammar: " +
meta.supportsCoreSQLGrammar());

                  System.out.println("Support Full Outer Joins: " +
meta.supportsFullOuterJoins());
                  System.out.println("Support Group By: " +
meta.supportsGroupBy());
                  System.out.println("Support Savepoints: " +
meta.supportsSavepoints());
```

```
                        System.out.println("Support Stored Procedures: " +
        meta.supportsStoredProcedures());
                        System.out.println("Support Subqueries in EXISTS: " +
        meta.supportsSubqueriesInExists());
                        System.out.println("Support Transactions: " +
        meta.supportsTransactions());

                        System.out.println("Support Union: " +
        meta.supportsUnion());
                        System.out.println("Support Union All: " +
        meta.supportsUnionAll());
                } catch (SQLException ex) {
                        System.out.println(ex.getMessage());
                        ex.printStackTrace();
                }
        }
}
```

Run this program, you may see the following output:



```
@>java -cp ..\mysql-connector-java-5.1.45-bin.jar;. CheckFeaturesExample
Support Batch Updates: true
Support Column Aliasing: true
Support Core SQL Grammar: true
Support Full Outer Joins: false
Support Group By: true
Support Savepoints: true
Support Stored Procedures: true
Support Subqueries in EXISTS: true
Support Transactions: true
Support Union: true
Support Union All: true
```